Blockit: Time Blocking Made Easy

Ezaan Mangalji (ezaan@seas.upenn.edu)
Griffin Fitzsimmons (gfitz@seas.upenn.edu)
Rishab Jaggi (rjaggi@seas.upenn.edu)
Young Lee (jaeyounglee@seas.upenn.edu)
External Advisors: Swapneel Seth, Sebastian Angel, Joe Devietti

Abstract

Blockit is a mobile application oriented towards students and working individuals who want to be ultra-productive. Time blocking is an extremely effective method to stay productive and efficient, but the process is a pain. Our project combines both the to-do list with the calendar in one application to productively use one's pockets of free time and manage one's workload with ease. We built a beta iOS application that integrates a user's Google Calendar and lets one block to-dos with one tap, along with a prototype automated scheduler. We evaluated our work with a feature conjoint study, Time-to-First-Block (TTFB) analysis, and an automated scheduling comparison to competitors. The results undoubtedly show that our application can provide serious benefits to users.

1 Motivations and Product High-Level Functionality

Our motivation for this project is to resolve two pain points we experience frequently as students: first, the annoyance of cross-checking our to-do lists with our calendars to schedule when to work on tasks, and second, the annoyance of emailing people back and forth to schedule a simple meeting. We also realized that there are currently no services available that solve both of these issues simultaneously. As a result, we conducted primary research to find out whether other people had similar experiences with their productivity apps and aimed to build a solution that addressed these problems. Now that we have built the prototype, we expanded our evaluation studies to draw statistical comparisons with competing apps to prove whether or not Blockit managed to offer a meaningful value proposition.

Time blocking refers to the concept of blocking or segmenting out time on one's calendar for specific tasks or work. This allows users to create a specific time zone or block to work on that task. To-dos refer to your inputted items that you want to get completed, like buying a flight ticket or working on a CIS 530 HW 8 assignment. The New York Times has done a study showing that ultra-productive people use time blocking as their primary tool for organizing their life and basically live off their calendar. However, to this date, time blocking is a tedious and annoying process that involves dragging out events on Google Calendar or Apple Calendar, and we want to help people become more effective.

1.1 Value Proposition

Blockit offers two main features: first, it seamlessly integrates a user's to-do lists with their calendar so that they can schedule when they want to work on their tasks as easily as archiving an email in their inbox. This new hybrid is simply called "The List." The app integrates with their existing calendars (e.g. Google Calendar and Apple Calendar) and users can swipe to-do list tasks to place the task into an open spot on their calendar with one tap, creating a "block" of time to work on that task. This would be especially useful in the workplace, where employees are constantly trying to block out their schedules to work on certain tasks, only to have to create each event slowly and manually. Blockit would do it all very easily and let users harness small pockets of free time to keep themselves productive.

The second feature is scheduling meetings: instead of the usual emailing back-and-forth to try to figure out when someone is free, Blockit would use the data it already knows from integrating with one's calendar to automatically schedule a time when they are guaranteed to be free. This feature is simply called "The Scheduler." We realize that plenty of services already do this (e.g. When2Meet, YouCanBook.me, Doodle, etc.), but 1) most, if not all, do not automatically integrate with one's calendar to find occupied times and 2) if both parties have registered Blockit accounts, Blockit would schedule this meeting automatically for them without the invited party having to fill out a form. This would be possible as users would have already inputted scheduling preferences when they signed up for the app.

While this is an awesome feature we wanted to implement, due to time and the COVID-19 pandemic, we were unable to build this into our iOS beta. As this was an important feature to use, we instead built out the algorithm itself in a standalone code repository. As a result, we were still able to evaluate this feature by comparing our algorithms' results to that of others. We go into much more detail on these results in the evaluation section later on.

1.2 Stakeholders

The stakeholders for this project include:

- Students trying to better manage their time
- Employees trying to manage when they will complete their tasks between a full load of meetings:
- Employees in companies trying to schedule meetings with other employees
- Gig workers (e.g. freelance photographers) who want to schedule gigs more easily into their work schedules

It is important to note that Blockit will provide value not only when it is used individually to block out free time, but also among groups of people who can benefit from the assisted scheduling functionality.

2 Related Work and Competition

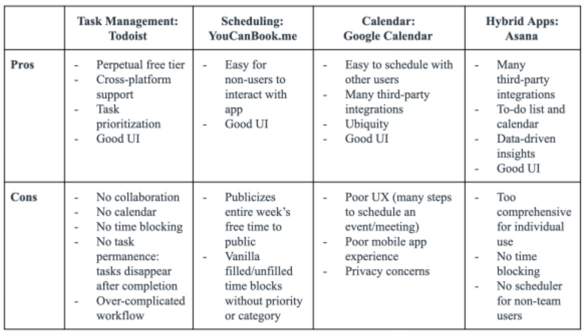
2.1 Market Research

In a 2014 report released by VisionMobile, an estimated $28 billion was spent on business and professional apps in 2013. This figure was projected to reach around $58 billion by the end of 2016. Although this is a small slice of the forecasted $6.3 trillion mobile app industry by 2021, productivity and collaboration software is a quickly growing market today.

This growth can also be indirectly measured by the growth rates of our competitors, who are described in more detail in the following section. Not only are there more competitors in each of our targeted spaces (task management, scheduling) overall, but also each of these competitors have been raising massive rounds of capital from venture capitalists with the expectation that these productivity apps will eventually become quintessential to every company in the near future.

2.2 Competition

A full list of competitors and their respective productivity sub-industries are listed in **Appendix A: Competitors** of this paper. We provide a brief explanation of the pros and cons for a select number of competitors in **Figure 1** below (a larger copy of this table can also be found in **Appendix A: Competitors**):



	Task Management: Todoist	**Scheduling: YouCanBook.me**	**Calendar: Google Calendar**	**Hybrid Apps: Asana**
Pros	- Perpetual free tier - Cross-platform support - Task prioritization - Good UI	- Easy for non-users to interact with app - Good UI	- Easy to schedule with other users - Many third-party integrations - Ubiquity - Good UI	- Many third-party integrations - To-do list and calendar - Data-driven insights - Good UI
Cons	- No collaboration - No calendar - No time blocking - No task permanence: tasks disappear after completion - Over-complicated workflow	- Publicizes entire week's free time to public - Vanilla filled/unfilled time blocks without priority or category	- Poor UX (many steps to schedule an event/meeting) - Poor mobile app experience - Privacy concerns	- Too comprehensive for individual use - No time blocking - No scheduler for non-team users

Figure 1: Pros and cons of competitors in the productivity software market.

3 Unit Cost Analysis and Revenue Model

3.1 Business Model Overview

Our go-to-market plan involves pricing our product similarly to other products already in the market. Most productivity "software-as-a-service" (SaaS) apps are offered under a freemium model: a (usually) perpetual free tier with limited features, and a paid ("premium") tier with all features. We plan to monetize Blockit in a similar fashion, creating tiers based on feature set and type of customer. In general, Blockit offers three main features: time blocking, automated scheduling, and calendar optimization. We also note three general types of customers in the SaaS market: individuals (for personal use), small teams (for business use), and enterprises. At rollout, in order to maximize traction and organic growth, Blockit would offer time blocking and automated scheduling for free for all individuals and small teams. Since we expect enterprises to comprise the bulk of our sales revenue in the long run, they will not be included in the free tier at all.

Once we reach a steady state in our business (in approximately several years), we aim to target the small teams segment as well, making the free tier only available for individuals only. Furthermore, since the free tier would now only include individuals, automated scheduling would no longer belong in the free tier (since the feature inherently requires teams). All in all, our approach would always focus on enterprise sales under a freemium model.

An analysis of our costs and revenue streams are given in the following subsections.

3.2 Cost Analysis

As a SaaS business, Blockit has the advantage of having extremely low fixed costs. All costs are strictly variable based on our usage, and include AWS EC2 and MongoDB Atlas fees, Stripe online transaction fees, and social media marketing costs. Each of these services scale automatically, so our costs will grow smoothly and predictably as Blockit gains traction. Our model assumes using a given amount of data storage and data transfer in three stages: year one, year two, and year three onwards. A detailed breakdown of each of our costs, our estimates on how much data usage we expect from users, and cost growth over time can be found in **Appendix B: Revenue Model**.

3.3 Revenue Model

Our revenue model is primarily based on paid user growth over time. Our primary source of revenue is the subscription fee users pay for the premium tier, and we expect to price Blockit at $19.99 per month, per user for enterprise customers. We thought this was a reasonable price given competitors' rates for *enterprise* customers (e.g. Notion). Our critical assumptions include the monthly growth and churn rates. We were able to minimize our assumptions to just these two metrics as a subscription model is a very predictable source of revenue, even in the long run. Putting it all together, given modest monthly growth and churn rates (converging to 3% and 2% respectively by Year 4), Blockit's profit grows to over $1.3 million per year by Year 5. A breakdown of our costs and revenue by year is shown in **Appendix B: Revenue Model**.

4 Technical Approach

Our project consisted of five technical components: the design system, mobile application, RESTful API, database, and the scheduling algorithm. We describe each of these components in detail below and is summarized in **Figure 2** below:

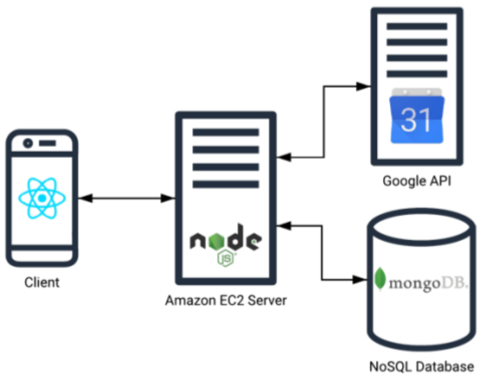


Figure 2: Diagram of our tech stack.

4.1 Design System

A design system, while not a public component, is essential to any application. We created a full guideline to the aesthetics of our app in a Figma, a collaborative interface design tool. This allowed us to plan in detail the views for most aspects in our app and create design patterns that we used throughout. It was essential that we kept our design methodology in mind at all times because a large portion of our core value proposition (aside from core functionality) was clean design and displaying only essential details. We also made use of Webflow, a prototyping tool that helps you build responsive websites and export code for implementation. These two tools allowed us to design an extremely simple, flowing, application. We provide screenshots and details of our design and website in **Appendix C: Design System**.

4.2 Mobile Application

While working through various design concepts, it became clear that a mobile application was the best form for our productivity tool to deliver the greatest value. We developed it using React Native, an open-source mobile application framework, which allowed us to leverage our team's previous experience with React.js and build a single native application for both Android and iOS. We also used Expo.io, a platform for building, deploying and quickly iterating on applications to accelerate our development. Building the mobile application basically involved two main tasks: implementing our Figma designs to make the most delightful user interface as possible and marshalling data back and forth in communicating with the back-end server.

We leveraged some great front-end libraries in implementing our designs. We also took advantage of Figma's ability to export code (mostly CSS) in order to make our implementation resemble the original designs as much as possible. Some interesting components we built included the navigation bar, swipeable to-do list tasks, and one-click calendar blocks.

Connecting the mobile app to the back-end server involved marshalling account, event and calendar data back and forth from the back-end, as well as properly representing and storing this data in the app itself. We used the Fetch API to make HTTP requests from the mobile app to the server in order to send and retrieve information and the state of our React components to store this data. We also leveraged some libraries like Moment.js for parsing, validating, manipulating, and formatting dates and times as they related to events and tasks.

Finally, we made use of the "expo-google-app-auth" API for adding Google authentication and access to our mobile application. This allowed us to seamlessly integrate a "log-in with Google" screen into our app. Upon successful authentication, a secure access token is returned, which we send back to the back-end server and store in the user session for future Google API requests. A code snippet of the Google log-in flow can be found in **Appendix F: Code Snippets**.

4.3 RESTful API

The back-end API is responsible for communication between the front-end mobile application and services like the Google API and the database where information is stored. We developed it using Node.js and the Express.js framework.

The server was able to interact with our database using the Mongoose library. This a powerful and elegant object modeling library that allows us to easily create, update and delete objects from our database. A code snippet of API endpoints for interacting with the database can be found in **Appendix F: Code Snippets**.

While the initial Google authentication mechanism is built into our mobile application front-end, all subsequent requests to the Google API are handled by our own back-end server. After retrieving the secure access token from the original authentication, the server is able to include this in every request to Google API's in order to identify/authenticate the user and retrieve, create, edit and delete events in the user's Google Calendar. The Javascript code on the server handles manipulating and editing the data before sending it to the mobile application as well as syncing the data to our own database with our own database.

For development, we deployed our API on a free-tier Heroku dyno. This allowed us to get the server up on a URL extremely quickly with only a few steps. However, this also came with some disadvantages as free-tier dynos go to sleep after 30 minutes of no activity and then take several seconds to relaunch when a new request comes in. For production, we decided to move the server to a dedicated AWS EC2 instance, which remains up and running.

4.4 Database

We decided to use a NoSQL database as it allowed us to rapidly iterate on and modify our schemas as we built out more functionality without having to worry about creating new tables or run migrations. Our database provider of choice is MongoDB and this is hosted by MongoDB's cloud product called Atlas. For production, we would port this to a collection of SQL tables to better handle scale and throughput. A code snippet of our To-do model database schema can be found in **Appendix F: Code Snippets**.

4.5 Scheduling Algorithm

Our automated scheduler attempts to find the lowest-impact free block of time in a given week. Our objective was for the algorithm to suggest meeting times that does not conflict with existing events, does not disrupt flow time, and does not extend the end of the day or beginning of the day. For more information on how we weighed and measured these metrics, see the Automatic Scheduling part of the Evaluation section of this report. We implemented our algorithm functionality in Python with an algorithm based around discretizing the day into events (from an existing calendar) and into free blocks. Below is pseudocode further detailing this subroutine.

```
getFreeBlocks(day)
$leadEvent$;
$trailEvent$;
$freeBlocks \leftarrow []$;
for $event \in day$ do
  if $leadEvent, trailEvent$ then
    $trailEnd \leftarrow endTime(leadEvent)$
    $leadStart \leftarrow startTime(leadEvent)$
    if $trailEnd < leadStart$ then
      $freeBlocks \leftarrow newBlock(trailEnd, leadStart)$
    end if
  end if
end for
```

Based on the free blocks we obtained for each day, our algorithm searched for the minimum time slot sufficiently large enough to accommodate a given event. If the block is long enough to be considered flow time (two hours) and another block is available within the time span that is smaller, our algorithm will choose this block. This example subroutine is included below.

```
getMinSlot(Weekdays)
$minWeekSlot$;
for $day \in Weekdays$ do
  if $getMinSlot(day) < minWeeksSlot$ then
    $minWeeksSlot \leftarrow day$
  end if
end for
```

Our algorithm is biased to find within-day free blocks and will only look to schedule events outside of the normal working hours. A code snippet of scheduling algorithm can be found in **Appendix E: Scheduling Data**.

4.6 Application Demo

A video demonstration of our application's functionality can be found here:

https://www.dropbox.com/s/tga2jw8s7qj6hyo/App%20Demo.mp4

5 Evaluation

When we first made our evaluation plan, we thought it would be important to test Blockit against its competitors both quantitatively and qualitatively. Ideally, the best way to evaluate our application would be to get it in the hands of as many users as possible and run thorough user studies and focused interviews; however, due to the COVID pandemic, this immediately became infeasible. As a result, we conducted analyses that we believe validate our value proposition and provide us with very useful information in improving our product.

The three analyses we conducted were:

1. Feature-based conjoint analysis: to determine which aspects of our features were most important to users,
2. Time-to-First-Block (TTFB) analysis: to measure how quick and easy it was for Blockit users to time block, compared to competitors, and
3. Automated scheduling analysis: a simulation to test the effectiveness of our standalone scheduling algorithm against other competitors.

We now go into the setup, results, and other details for each of these studies below.

5.1 Feature-based Conjoint Analysis

Conjoint analysis is a great method for understanding *implicit* user preferences as it forces people to make tradeoff decisions.

Design

We begin by creating a design for our conjoint survey. We want to have people rate different product offerings on a Likert scale from one to seven on how likely they would be to use that product. A product is composed of 6 different attributes which we selected. Each attribute has two to three levels to it, which we vary in the survey to provide different product offerings. We had the following six attributes and levels:

1. Application: web, mobile, mobile and web
2. To do list: plain list, list with categories, list with categories and subtasks
3. Task scheduling: none, task deadlines, task blocking directly into calendar
4. Automated scheduling: none, yes
5. Calendar optimization: none, yes
6. UX: separated calendar and to do list, combined and integrated single application

We then used a fractional factorial design which provides us with the levels of each attribute we need to offer in each of the survey questions to gain proper insights. The output of this can be seen in **Appendix D: Conjoint Analysis**.

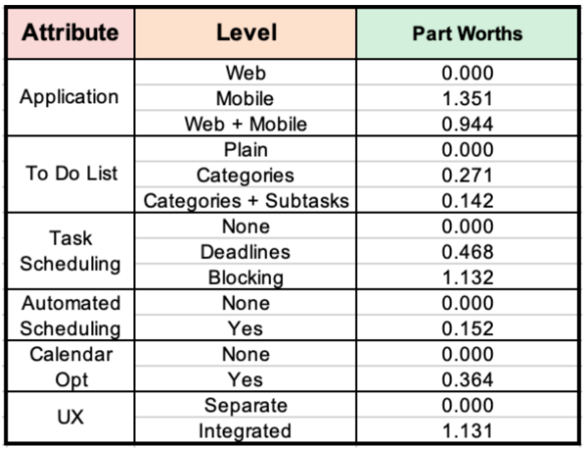
Survey

Next we created a Google Form to conduct the survey and collected results from a multitude of students and other people. The survey asks to rate 16 different product profiles on a Likert scale from one to seven, you can see a screenshot of the survey in **Appendix D: Conjoint Analysis**.

Analysis

Finally, we distill the results into a spreadsheet where we can begin to analyze them. For each person we extract their rating for each profile, this is the dependent variable. Then we create a set of indicator variables to be used as the independent variables, setting the first level of each attribute as the baseline. Indicator variable setup can be seen in **Appendix D: Conjoint Analysis**.

We then run multiple regression on this data to extract part-worths (**Figure 4**), the utility that each level provides in comparison to the baseline. You can see an example of one person's regression in **Appendix D: Conjoint Analysis**.

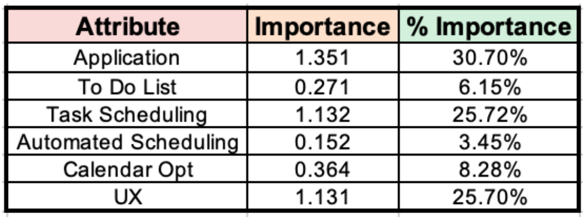


Attribute	Level	Part Worths
Application	Web	0.000
	Mobile	1.351
	Web + Mobile	0.944
To Do List	Plain	0.000
	Categories	0.271
	Categories + Subtasks	0.142
Task Scheduling	None	0.000
	Deadlines	0.468
	Blocking	1.132
Automated Scheduling	None	0.000
	Yes	0.152
Calendar Opt	None	0.000
	Yes	0.364
UX	Separate	0.000
	Integrated	1.131

Figure 4: Part-worths of our conjoint analysis.

Let us analyze these part-worths. In the Application attribute, we see that in comparison to just Web both levels that include a Mobile option are valued much higher. This confirms our belief that a mobile-first application is important and that we made a good decision at the turn of the semester in pivoting from a web app to a mobile one. We see that the type of to-do list level does not matter very much. In the Task Scheduling attribute, users want some form of scheduling, with the ability to block tasks in their calendar providing the most utility out of any level. Both the inclusion of Automated Scheduling and Calendar Optimization are good but not that important to our users. Finally, we see that having an integrated to-do list and Calendar application is very important to users.

Using the part-worths we too can find out Attribute Percent Importances using the ranges of each level within an attribute. We reach the following results in **Figure 5**:



Attribute	Importance	% Importance
Application	1.351	30.70%
To Do List	0.271	6.15%
Task Scheduling	1.132	25.72%
Automated Scheduling	0.152	3.45%
Calendar Opt	0.364	8.28%
UX	1.131	25.70%

Figure 5: Attribute Percent Importance values for each attribute.

From this we can easily see that users place the highest importance on Application, Task Scheduling, and UX attributes.

Finally, our Conjoint analysis lets us perform a market share study. To do this we used two existing products, Google Calendar and Todoist, and created product offerings using our predefined attribute levels. We also create a product profile for our application Blockit. Using the part worths of each individual we can sum product each of these three product profiles with each individual's part worths to obtain a utility value. This provides one metric for which we can compare to see a user's most valued product. Product profiles can be found in **Appendix D: Conjoint Analysis**. When Google Calendar and Todoist are the only products in the market respondents are split 62.5% to 37.5% in favor of Google Calendar. Once Blockit enters the market it takes 75.0% while both Google Calendar and Todoist are at 9.09%. This confirms that our application provides a value proposition that is unique to both Google Calendar and Todoist that users want.

5.2 Time-to-First-Block (TTFB) Analysis

Time-to-First-Block (TTFB) is a metric we defined that measures the time it takes to create one event on a specific date between two existing events. This is a key metric because it quantifies one of the most frequent use cases of Blockit: spontaneously noting and scheduling a single task. To measure TTFB across existing calendar platforms, we asked fifteen potential users (mostly friends and some working alumni) to download our three biggest calendar competitors' apps—Google Calendar, Apple's Calendar, and Spark Calendar—and follow these instructions below:

1. Start timer
2. Wake up phone
3. Unlock phone
4. Open calendar app
5. Create an event (in the fastest way possible)
6. Set time to fill the gap between two existing events
7. Name event "Work on CIS 401 pset 5"
8. Stop timer

We then asked test subjects to repeat this process five times per app (Apple Calendar, Google Calendar, Spark Calendar). We first ran a one-sided T-test with unequal variances to ensure that the differences in TTFB averages compared to our own TTFB measurements using Blockit were statistically significant: our p-value was less than 0.01 for all pairs of data (Apple vs. Blockit, Google vs. Blockit, and Spark vs. Blockit). The average TTFB values across platforms were:

- Apple: 17.8s
- Google: 18.5s
- Spark: 25.0s
- Blockit: 10.0s

Which ultimately meant that on average, Blockit was 44% faster than Apple Calendar, 46% faster than Google Calendar, and 60% faster than Spark Calendar. This is a significant improvement to the time-blocking experience. Qualitatively speaking, from surveying our test subjects, we noticed that the biggest time-saver for Blockit (and conversely, the biggest pain point of competitors' apps) was not having to manually specify the start and end times for a block as Blockit did that automatically. All in all, this TTFB study proved our original value proposition that Blockit makes time blocking quick and easy for users.

5.3 Automated Scheduling Analysis

We tested our automated scheduling algorithm against the automatic scheduling features on some of our competitors' calendar software. For this analysis, we compared Blockit to Clockwise and X.ai. We formulated our evaluation in the following way. We used a White House dataset (**Appendix E: Scheduling Data**) containing the daily schedule of the President of the United States as an example of a busy schedule. We loaded three sample weeks ($w = 3$) into Google Calendar and stress-tested the scheduling algorithms by attempting to schedule ten events ($n = 10$) into the already busy schedule. To define an objective function to evaluate the performance of the algorithms, we first define the following violation categories. These are the scheduling mistakes for which we will penalize our algorithm (**Appendix E: Scheduling Data**).

1. **Flow Time Violation (*FT*):** incurred by any scheduling action which reduced the number of two-hour blocks of within-schedule open time.
2. **Edge of Day Violation (*ED*):** incurred by any scheduling action which schedules an event before the would-be first event of the day, or after the would-be last event of the day.
3. **Event Violation (*EV*):** incurred by any scheduling action which schedules an event contemporaneously with a previously scheduled event or fails to find a viable scheduling time within the allotted date span given.

All these violations are represented in the objective function which we term the Automatic Scheduler Evaluation Metric (*ASE*) defined as the weighted average of these metrics, cumulatively summed over all m events, summed across the weekdays (Monday through Friday) of all n given weeks.

$$\mathbf{\Phi}_{ASE} = \sum_m \sum_n (0.2\, FT + 0.1\, EOD + 0.4\, EV)] * (-1)$$

The weighting was chosen ad hoc to represent the notion that an event violation would be the most egregious mistake an algorithm could make, that we value a flow time violation about half as much as an event violation, and an edge of day violation about half as much as a flow time violation. The following graphs depict our results (**Figures 6 and 7**).

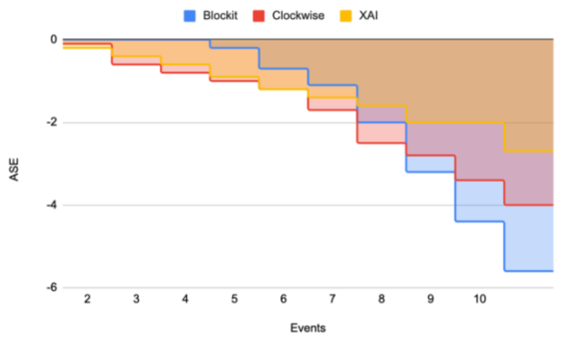


Figure 6: ASE values across Blockit and competitors.

Though our algorithm performs the worst of all the algorithms under the most stressful conditions, we are encouraged that our algorithm is effective at finding viable free space in a busy schedule before it crowds too much. To further understand what kind of mistakes all three automatic schedulers were making, we conducted the following further analysis (**Figure 7**).

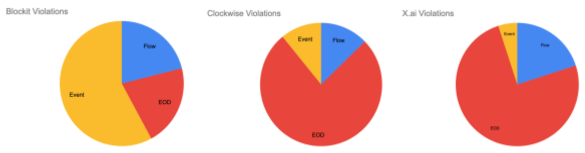


Figure 7: Kinds of mistakes by Blockit and competitors.

Clockwise makes the fewest flow time violations, which is understandable because it is a metric which they created and value in the schedule of productive people. Our algorithm's event failures are what cost us a great deal because they are highly weighted in the ASE metric. This indicates that we need to continue work on core resilience of our algorithm under various calendar types to ensure it can reliably make viable suggestions.

5.4 Future Evaluation with Users

If the COVID-19 fiasco did not strike our team would have liked to perform a series of user studies. Ideally, we would have liked to hold one-on-one interviews and focus groups with users who have our app. These types of settings are very helpful in gaining specific knowledge on user experience, intuitiveness of the application, simplicity, pain points, and other feedback. We would have also gotten users to use our application for a week and report on their productivity using some form of ratings criterion and qualitative experience information.

Additionally, as suggested by Professor Nenkova, a great long-term study that would be beneficial to run is a true experiment. Creating a test and control group and tracking their productivity for a month or even three months. This could be done by following a group of individuals for a month and tracking their productivity, then giving a random subgroup the Blockit app and continue to track both the control and test group for another month. After this we would be able to compare the productivity of those who began to use Blockit with their past performance and with the control group.

6 Societal Impact

Our project is an application that allows users to time block their to-do list into their calendar with ease, which we believe has both positive implications societally and ethically. Our product can benefit busy individuals professionally, socially and personally. It does this through a multitude of ways that help people become more productive and efficient when it comes to pain points in organizing their everyday lives. Our product also presents some potential societal and ethical pitfalls. One example we discuss is that our implementation promotes more automation of interpersonal connections. Our project also faces other, more general pitfalls about privacy concerns, cybersecurity risks, potential malicious misuse, and ethical conundrums associated with monetization down the road.

To reiterate some technical details, we created this application using React Native and Expo along with a plethora of Node Package Manager ("npm") packages. We are using the Google log-in API to protect user logins and give us access to users' Google Calendars. This provides us with a user's entire calendar information: events, meetings, locations, and invitees. Our database is a MongoDB NoSQL storage platform that helps us store user IDs and calendar information.

6.1 Positive Impacts

One of the biggest positive impacts our application provides is helping people reduce their anxiety and stress when it comes to their work week. Time blocking is a scientifically proven method to becoming more efficient. It helps users focus their time on their tasks (as it sets a distinct period to work on them), help them effectively use pockets of time in their day, and remove multitasking delays. This is great for society as it allows people to get their work done and help free up time in their day for other activities.

Additionally, our algorithm that helps people schedule events efficiently to free up chunks of flow time is great for all sorts of professions. Studies have shown that two-hour blocks of time are enough time for one to get into the zone and truly focus on a task without being interrupted. This can help users become more effective in their lives and reduce their stress when it comes to completing their tasks for school or work.

Blocking out one's to-do list helps provide users with peace of mind as it helps set aside time for all their tasks, giving them knowledge that they will be able to complete everything they need to. Finally, our application helps remove pain points regarding scheduling meetings with others. There is no need to go back and forth to figure out the best meeting time when Blockit can do it automatically, as it knows the users' scheduling preferences. All these aspects of our application provide a great deal to society, helping individuals become more productive, efficient, and less stressed.

6.2 Risks and Cybersecurity

One dilemma which our project potentially faces is the handling of user data and the issue of privacy. Our database stores the following sensitive data: standard account information (email, name, and password) as well as users' calendar information, which can be insidiously sensitive. Calendars can contain material regarding what people do every second of every day, where they are meeting, when they are meeting, and who they are meeting with. For people who use calendars as a form of task management, this private information falling into the wrong hands can be especially dangerous. Blockit would store the locations and times of many individuals which could potentially include high-ranking leaders in companies. Furthermore, since our application allows users to input their to-do list items, this information would contain what people are working on throughout the week. It is imperative that this data is protected from both developers on our end and any malicious actors. As we have learned throughout recent years, data is incredibly valuable and can tell a lot about an individual. It is important to ensure our users' privacy is protected and nothing is compromised. Currently, our application mitigates these concerns by requiring authorization for every API call made to our server, which means that users must be logged in to access any of their private information. A more detailed mitigation plan is outlined at the end of this Societal Impact section.

Additionally, our application provides read and write access to a user's calendar for creating blocks when you block an item. This is a potential risk both in terms of security and manipulation. A hacker could remove or delete events that may lead people to forgetting about important meetings or even having them attend a meeting with malicious intent. This is especially a concern for those that do not manage their own calendar (e.g. a secretary), as one would simply go along with what their calendar says.

6.3 Unintended Consequences

One unintended negative societal impact that is unique to our product is the potentially problematic automation of the process of keeping in touch. Making an effort to continually reach out and keep in touch conveys genuine caring in today's connected society. By making the process of scheduling meetings more robotic, the meetings themselves become less meaningful. Another potential unintended consequence could come with the monetization of our platform, should we continue with our plan to pursue a freemium business model. Premium users usually have access to a more advanced set of features which allows them to accomplish more. Concerning our users' interactions, however, we feel it would be unethical for the dynamic between them to reinforce existing power and economic inequalities. For example, in the scheduling of a meeting premium and a free-tier user, we want to ensure that in no way do our algorithms encode that the time of the premium user is somehow more valuable than that of the free-tier user.

6.4 Mitigation and Future Work

We have tried to do our best to mitigate these problems so far by adhering to a high level of security, but we know we must do more. We use Google OAuth 2.0 for logins instead of having to store our own login system and we use specific encryption schemes, SSL and HTTPS, using Let's Encrypt, a free SSL service, to ensure the safety of our data. Furthermore, all database requests must be authenticated, which we enforce through our RESTful API by design. We store as little sensitive data as we can in a cloud-hosted database with secure access tokens hidden in configuration files. We do all this to ensure privacy, authenticity, and integrity when it comes to data. Using the peer feedback that we received from our ethics report, we have come up with a few additional ideas we want to implement. We want to create a trusted contacts system for the automated scheduling feature of our application since a user should not be able to view anyone else's calendar without their permission. Additionally, right after creating an account, we want to display a pop-up listing all the data we will be storing that relates to the new user. Finally, in the spirit of data privacy, if a user were to delete their account, we want to ensure that all data related to that user is wiped securely and comprehensively. All of these would be great extensions to our application to help it become more secure.

In the end, our application provides a myriad of societal and ethical benefits to users to help ameliorate their lives and make them more productive. However, this does come at a cost to potential privacy threats of hackers or the misuse of personal calendar data. We believe that Blockit provides an extreme net positive impact to its users and we plan to minimize any potential negative impacts and security concerns. We all sincerely believe in the product we are building and are eager to begin using Blockit in our lives regularly.

7 Conclusion, Lessons Learned, and Future Work

We learned an immense amount technically, from becoming experts on modern design tools like Figma and Webflow, to writing organized Javascript code using state-of-the-art frameworks like React Native, to integrating distributed components with third-party APIs and deploying on cutting-edge cloud services like AWS and MongoDB. One of the most important parts of this was learning how to design, plan, build, and iterate on a product while continuously integrating feedback from users, our class, and professors. We made sure we were always building a product our users wanted by conducting rigorous testing and surveying to get detailed feedback on our prototypes. This taught us a lot, like how to build accurate conjoint analysis tests and foresee user behavior in order to create an effective UI based on user preferences rather than speculation. For many of us, these evaluation studies were the first time we applied statistical analysis techniques learned from our other classes inside a real-world scenario.

Additionally, we built a holistic business model, describing various stages of scaling our product and the associated costs and in doing so, we learned about enterprise software sales, economies of scale, revenue models, and pricing structures. It was not only interesting to discover the broad landscape of competitors Blockit was being built within, but also helpful in narrowing down our product design before we started to build.

While we were not able to truly launch this product on the Apple App Store and get it into the hands of users like we wanted to, we still built a great beta application that we feel has the potential to be an amazing productivity tool someday. Our ideas and creativity when creating a design and app were based on real needs we found in the community, but we were unfortunately not able to completely satisfy our users' needs as much as we wanted to. We found that a very difficult part of building any successful company is your ability to devote your time and mind to it. Since all four of us were full-time students, it was very difficult to find time to work on this project together, especially after schooling became remote. To create a truly good application and product, one has to both want and be able to make it his or her full-time job and not just a side hobby. Fortunately for us, we all now have a free summer of sitting at home during this pandemic and perhaps we will continue to iterate.

With the prototype built and evaluations ready to go, we were able to prove Blockit's value proposition not just qualitatively, but quantitatively as well. We truly believe in the benefits that time blocking provides us and were glad to find out others felt similarly about our product. That said, we do recognize that the evaluation we ended up conducting was relatively limited given the global context, which we discussed earlier in the Evaluation section. The longer-term studies outlined there would have been something we wish to have done instead had the pandemic not occurred. Fortunately, we were still able to get several substantial studies done remotely through our personal networks and were glad to have been able to complete this portion of our project.

We were also happy to have been able to apply many of the concepts and ideas we learned in class. For example, in the brainstorming phase of our project, we learned about scope creep in class and decided to focus on only a select number of features for our product. This ensured that we were able to divide up the building process amongst all teammates evenly and still meet realistic deadlines. Furthermore, our in-class discussions and assignments about ethics and social impact made us consider aspects of our product that we never thought about before. Our lessons from these sorts of topics have ultimately made us better engineers and we hope to continue what we have learned from Senior Design in all of our future projects.

Looking ahead, if we were to continue developing Blockit, we would first prioritize the completion and testing of our remaining features. Calendar optimization is a very data-centric feature and would require a lot of test data to get right. This was something we discussed with one of our advisors, Professor Angel, before and there were definitely many implementation issues that we would have to overcome. We must also improve upon our current scheduling algorithm in order to not only automatically schedule events, but also take user preferences (potentially from multiple parties) into account. Once we are able to achieve these milestones, our ideal next step would be to beta test our app with users in real corporate environments. While students—who comprised the majority of our evaluation user data—may find a lot of value from Blockit, our core market segment are enterprise customers and therefore our evaluation methodologies should reflect that. All in all, we are happy to be able to show that Blockit is a valuable product and we hope to inspire more people into leading more productive lives in the future through our software.

8 References

Grant, A. (2019, March 29). Productivity Isn't About Time Management. It's About Attention Management. Retrieved from https://www.nytimes.com/2019/03/28/smarter-living/productivity-isnt-about-time-management-its-about-attention-management.html

Gregg, M. (2015). 12 Getting Things Done: Productivity, Self-Management, and the Order of Things. *Networked affect*, 187.

Herrera, T. (2019, March 25). May I Have Your Attention, Please? Retrieved from https://www.nytimes.com/2019/03/25/smarter-living/time-management-productivity.html

Moment.js 2.24.0. (n.d.). Retrieved from https://momentjs.com/

Mongoose. (n.d.). Retrieved from https://mongoosejs.com/

Node.js web application framework. (n.d.). Retrieved from https://expressjs.com/

Perez, S. (2017, June 27). App economy to grow to $6.3 trillion in 2021, user base to nearly double to 6.3 billion. Retrieved from https://techcrunch.com/2017/06/27/app-economy-to-grow-to-6-3-trillion-in-2021-user-base-to-nearly-double-to-6-3-billion/

React – A JavaScript library for building user interfaces. (n.d.). Retrieved from https://reactjs.org/

9 Acknowledgements

The team would like to give a significant thank you to Professor Ani Nenkova and the TAs for helping us shape and guide our project to what it is now. We especially thank them for their ability to keep this class afloat during the pandemic—we truly appreciate it. We also want to thank our faculty advisors Professor Sheth, Professor Angel, and Professor Devietti for all their help. Finally, we would like to thank the entire CIS graduating class for continuing to persevere along with us during the pandemic; we are happy to e-graduate alongside you all.

Appendix

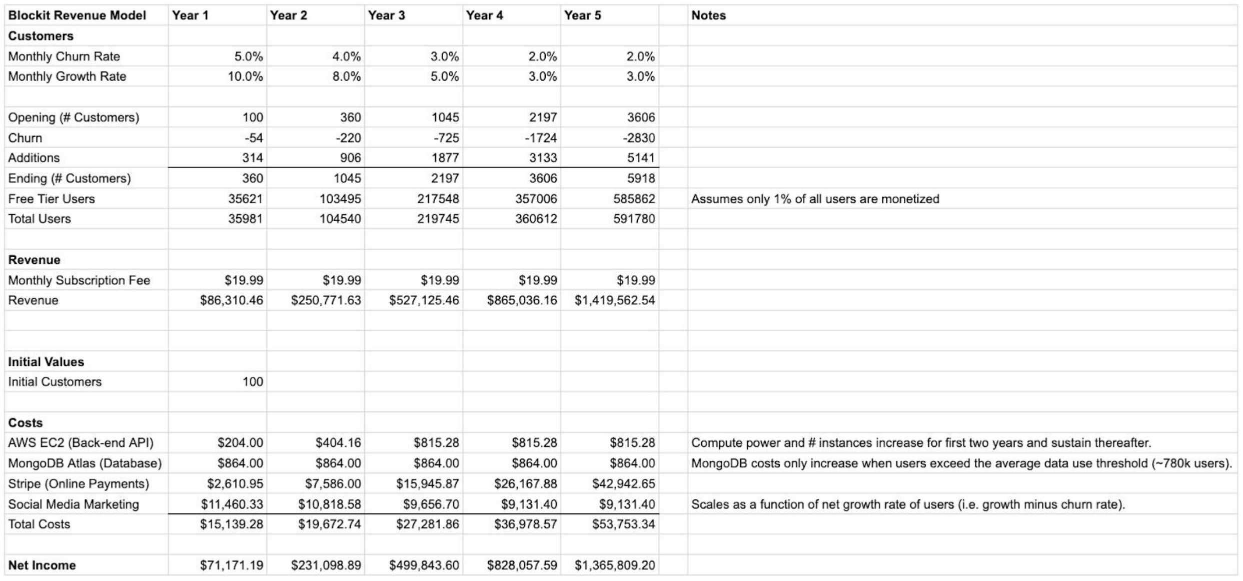
Appendix A: Competitors

Task Management	Scheduling	Calendar	Hybrid Apps
- Todoist - OmniFocus - TickTick - Wunderlist - Trello	- YouCanBook.me - When2Meet - Doodle - Calendly - Clockwise	- Google Calendar - Apple Calendar - Microsoft Outlook - Spark Mail - Fantastical	- Asana - Notion - Taskade - Any.do - Swit

Pros and Cons of Select Competitors

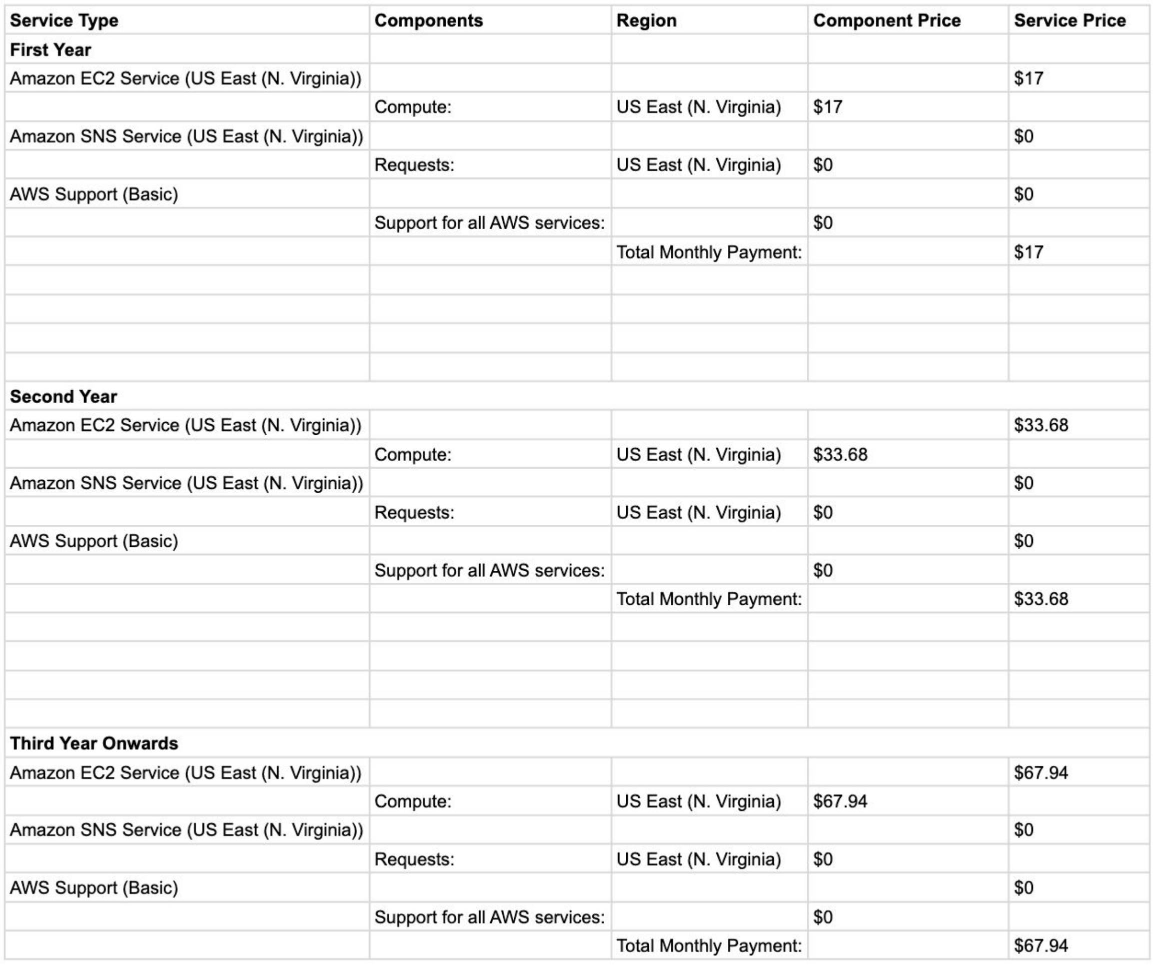
	Task Management: Todoist	Scheduling: YouCanBook.me	Calendar: Google Calendar	Hybrid Apps: Asana
Pros	- Perpetual free tier - Cross-platform support - Task prioritization - Good UI	- Easy for non-users to interact with app - Good UI	- Easy to schedule with other users - Many third-party integrations - Ubiquity - Good UI	- Many third-party integrations - To-do list and calendar - Data-driven insights - Good UI
Cons	- No collaboration - No calendar - No time blocking - No task permanence: tasks disappear after completion - Over-complicated workflow	- Publicizes entire week's free time to public - Vanilla filled/unfilled time blocks without priority or category	- Poor UX (many steps to schedule an event/meeting) - Poor mobile app experience - Privacy concerns	- Too comprehensive for individual use - No time blocking - No scheduler for non-team users

Appendix B: Revenue Model



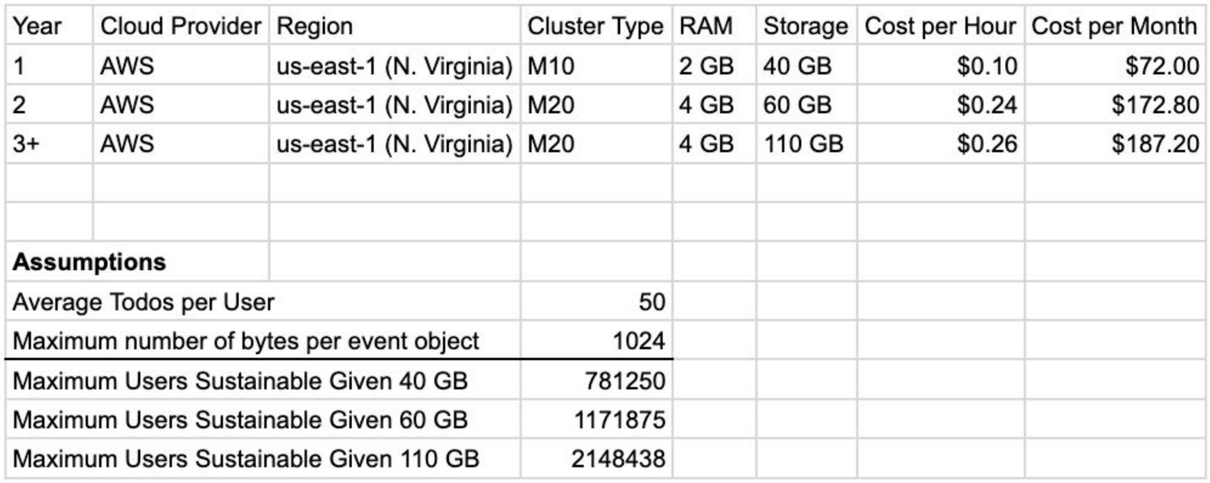
Blockit Revenue Model	Year 1	Year 2	Year 3	Year 4	Year 5		Notes
Customers							
Monthly Churn Rate	5.0%	4.0%	3.0%	2.0%	2.0%		
Monthly Growth Rate	10.0%	8.0%	5.0%	3.0%	3.0%		
Opening (# Customers)	100	360	1045	2197	3606		
Churn	-54	-220	-725	-1724	-2830		
Additions	314	906	1877	3133	5141		
Ending (# Customers)	360	1045	2197	3606	5918		
Free Tier Users	35621	103495	217548	357006	585862		Assumes only 1% of all users are monetized
Total Users	35981	104540	219745	360612	591780		
Revenue							
Monthly Subscription Fee	$19.99	$19.99	$19.99	$19.99	$19.99		
Revenue	$86,310.46	$250,771.63	$527,125.46	$865,036.16	$1,419,562.54		
Initial Values							
Initial Customers	100						
Costs							
AWS EC2 (Back-end API)	$204.00	$404.16	$815.28	$815.28	$815.28		Compute power and # instances increase for first two years and sustain thereafter.
MongoDB Atlas (Database)	$864.00	$864.00	$864.00	$864.00	$864.00		MongoDB costs only increase when users exceed the average data use threshold (~780k users).
Stripe (Online Payments)	$2,610.95	$7,586.00	$15,945.87	$26,167.88	$42,942.65		
Social Media Marketing	$11,460.33	$10,818.58	$9,656.70	$9,131.40	$9,131.40		Scales as a function of net growth rate of users (i.e. growth minus churn rate).
Total Costs	$15,139.28	$19,672.74	$27,281.86	$36,978.57	$53,753.34		
Net Income	$71,171.19	$231,098.89	$499,843.60	$828,057.59	$1,365,809.20		

AWS EC2 Costs



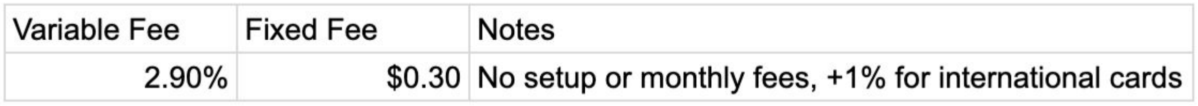
Service Type	Components	Region	Component Price	Service Price
First Year				
Amazon EC2 Service (US East (N. Virginia))				$17
	Compute:	US East (N. Virginia)	$17	
Amazon SNS Service (US East (N. Virginia))				$0
	Requests:	US East (N. Virginia)	$0	
AWS Support (Basic)				$0
	Support for all AWS services:		$0	
		Total Monthly Payment:		$17
Second Year				
Amazon EC2 Service (US East (N. Virginia))				$33.68
	Compute:	US East (N. Virginia)	$33.68	
Amazon SNS Service (US East (N. Virginia))				$0
	Requests:	US East (N. Virginia)	$0	
AWS Support (Basic)				$0
	Support for all AWS services:		$0	
		Total Monthly Payment:		$33.68
Third Year Onwards				
Amazon EC2 Service (US East (N. Virginia))				$67.94
	Compute:	US East (N. Virginia)	$67.94	
Amazon SNS Service (US East (N. Virginia))				$0
	Requests:	US East (N. Virginia)	$0	
AWS Support (Basic)				$0
	Support for all AWS services:		$0	
		Total Monthly Payment:		$67.94

MongoDB Atlas Costs



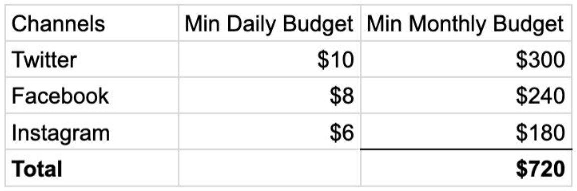
Year	Cloud Provider	Region	Cluster Type	RAM	Storage	Cost per Hour	Cost per Month
1	AWS	us-east-1 (N. Virginia)	M10	2 GB	40 GB	$0.10	$72.00
2	AWS	us-east-1 (N. Virginia)	M20	4 GB	60 GB	$0.24	$172.80
3+	AWS	us-east-1 (N. Virginia)	M20	4 GB	110 GB	$0.26	$187.20
Assumptions							
Average Todos per User			50				
Maximum number of bytes per event object			1024				
Maximum Users Sustainable Given 40 GB			781250				
Maximum Users Sustainable Given 60 GB			1171875				
Maximum Users Sustainable Given 110 GB			2148438				

Stripe Transaction Fees



Variable Fee	Fixed Fee	Notes
2.90%	$0.30	No setup or monthly fees, +1% for international cards

Social Media Marketing Costs

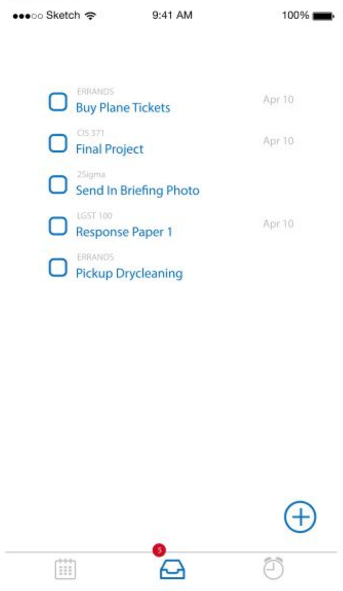


Channels	Min Daily Budget	Min Monthly Budget
Twitter	$10	$300
Facebook	$8	$240
Instagram	$6	$180
Total		**$720**

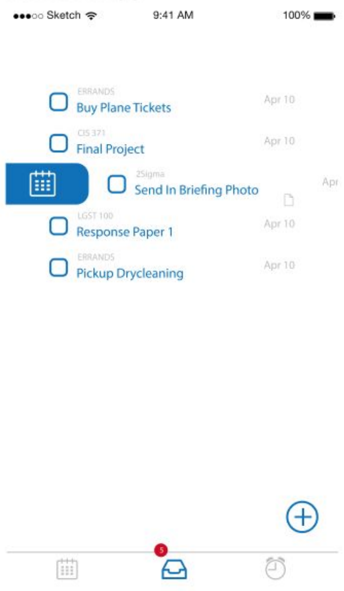
Appendix C: Design System

Figma Design System

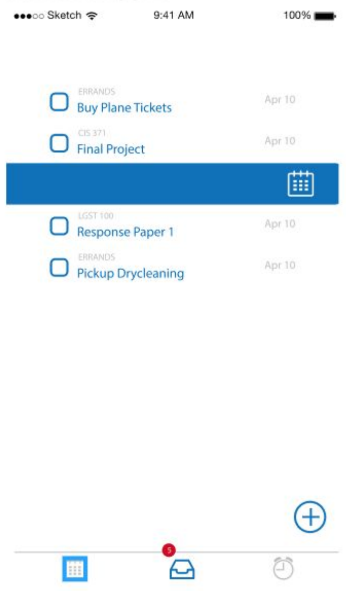
Inbox 1



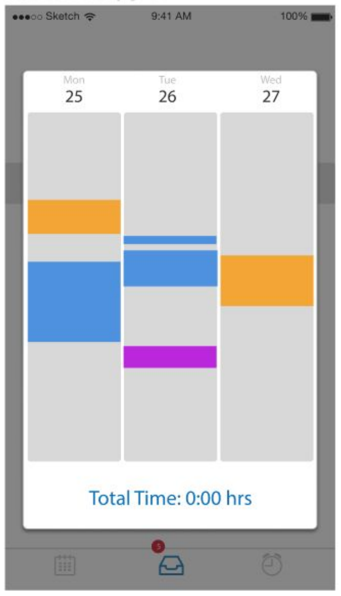
Inbox 1 Copy



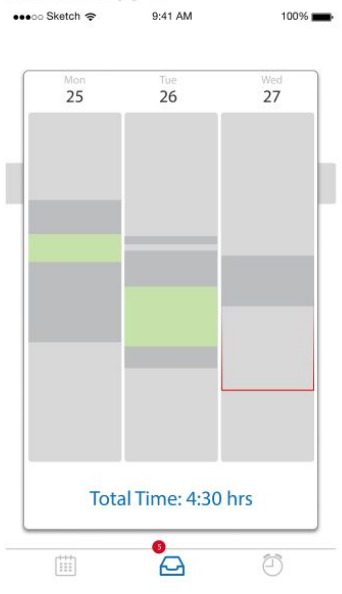
Inbox 1 Copy 2



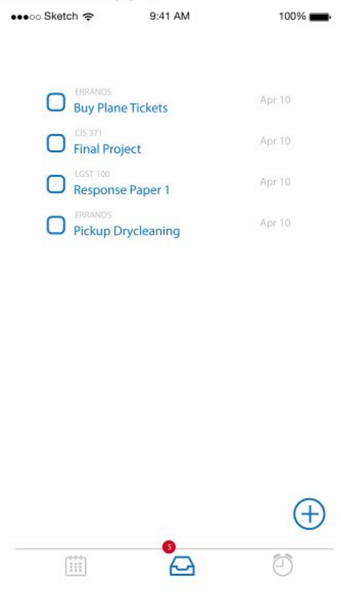
Inbox 1 Copy 3



Inbox 1 Copy 4

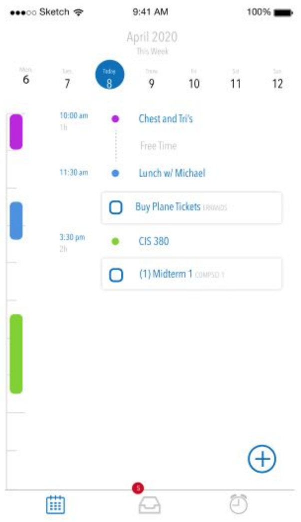


Inbox 1 Copy 5

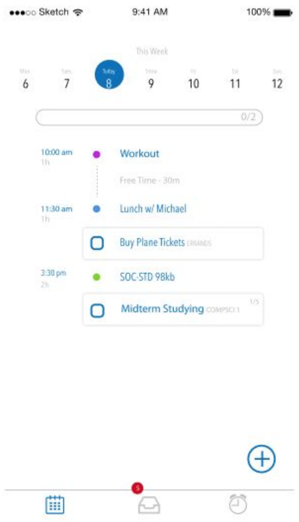


CIS 401 Final Progress Report (Team 27)

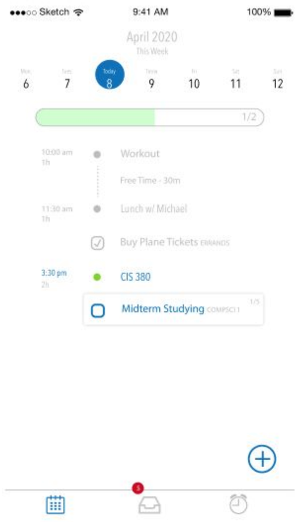
Calendar 1.1



Calendar 1.4



Calendar 1.3



Webflow Landing Page

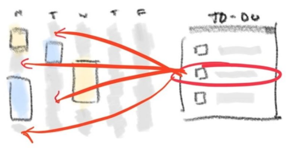
Blockit

Home

Time blocking, made easy.

Blockit is a tool to turn your calendar into a working document.

Start



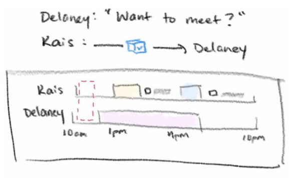
1. Time block your to-do list

Blockit integrates with your calendar so that when you can easiily select all the open spots on your calendar to work on something

2. Visualize inline

Blockit turns your calendar into a working document by uniting it with your tasks into an all-in-one view

MON
1pm
4pm
TUES
3pm

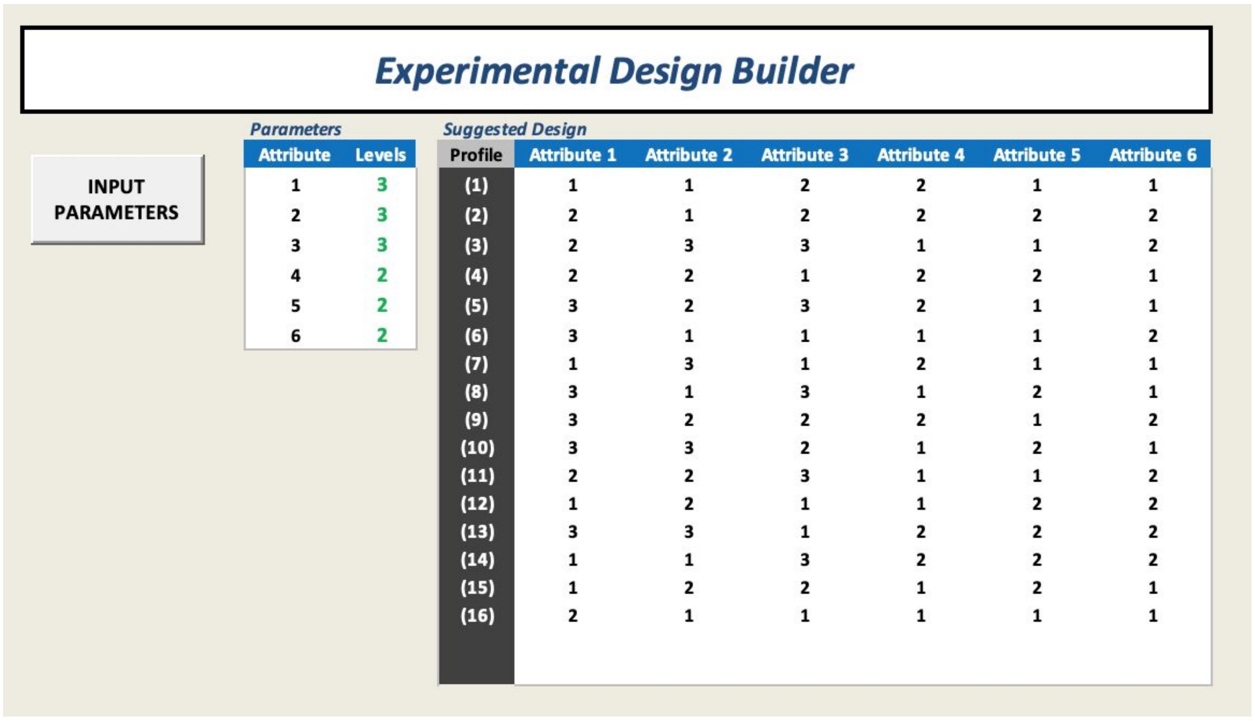


3. Automate scheduling

Use Blockit to auto-magically find the best time to meet with anyone using smart invites that look at both your calendars

Appendix D: Conjoint Analysis

Fractional Factorial Design Output



Experimental Design Builder

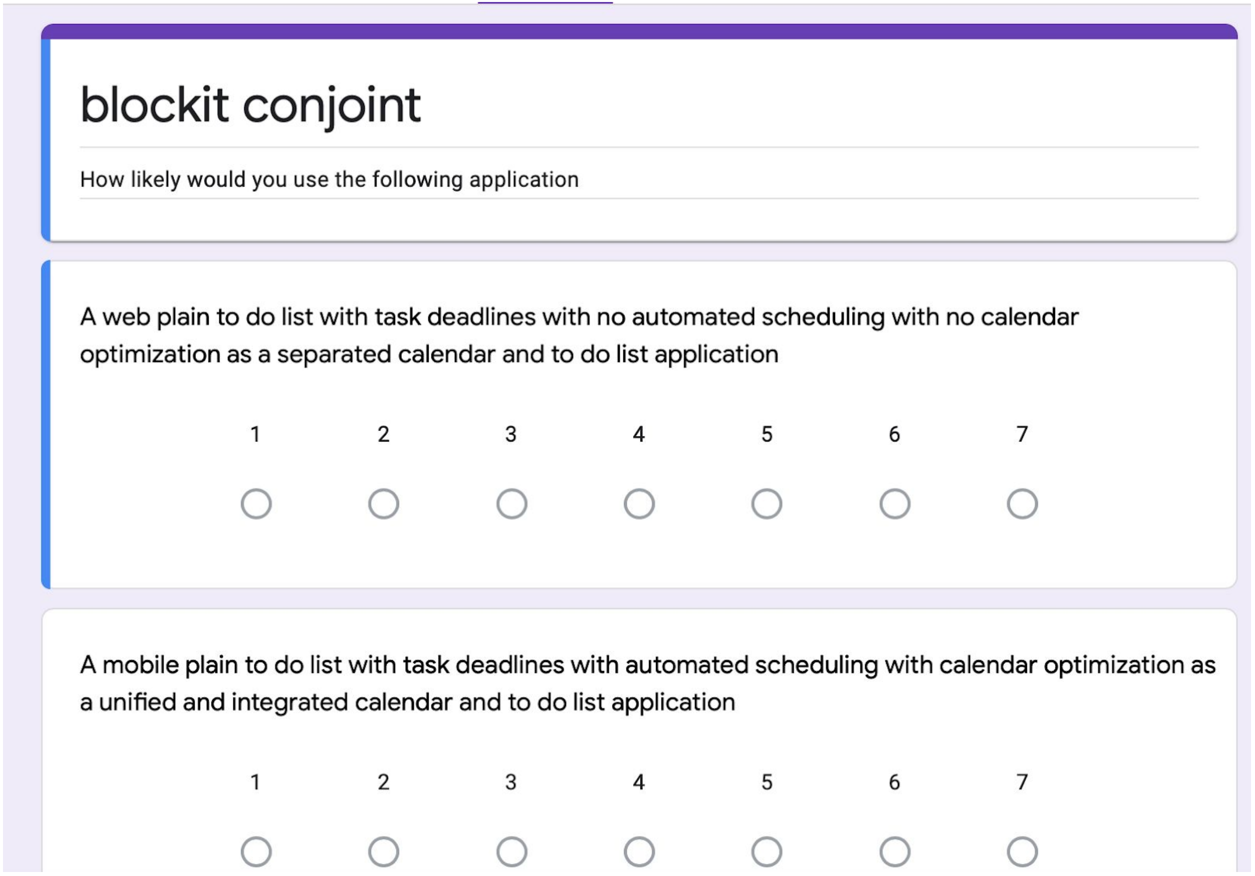
Parameters

Attribute	Levels
1	3
2	3
3	3
4	2
5	2
6	2

Suggested Design

Profile	Attribute 1	Attribute 2	Attribute 3	Attribute 4	Attribute 5	Attribute 6
(1)	1	1	2	2	1	1
(2)	2	1	2	2	2	2
(3)	2	3	3	1	1	2
(4)	2	2	1	2	2	1
(5)	3	2	3	2	1	1
(6)	3	1	1	1	1	2
(7)	1	3	1	2	1	1
(8)	3	1	3	1	2	1
(9)	3	2	2	2	1	2
(10)	3	3	2	1	2	1
(11)	2	2	3	1	1	2
(12)	1	2	1	1	2	2
(13)	3	3	1	2	2	2
(14)	1	1	3	2	2	2
(15)	1	2	2	1	2	1
(16)	2	1	1	1	1	1

Conjoint Survey



blockit conjoint

How likely would you use the following application

A web plain to do list with task deadlines with no automated scheduling with no calendar optimization as a separated calendar and to do list application

1 2 3 4 5 6 7

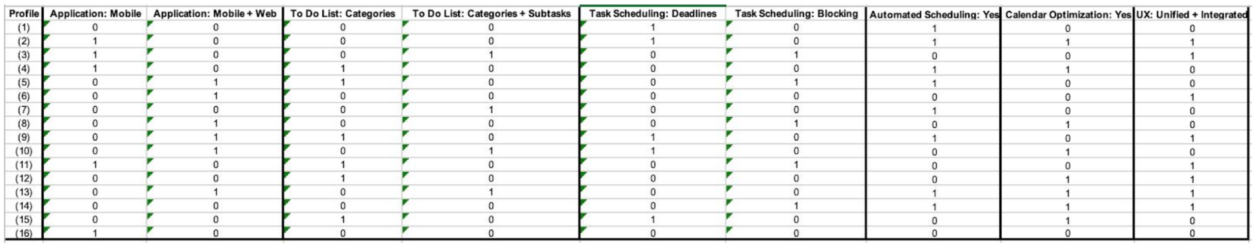
○ ○ ○ ○ ○ ○ ○

A mobile plain to do list with task deadlines with automated scheduling with calendar optimization as a unified and integrated calendar and to do list application

1 2 3 4 5 6 7

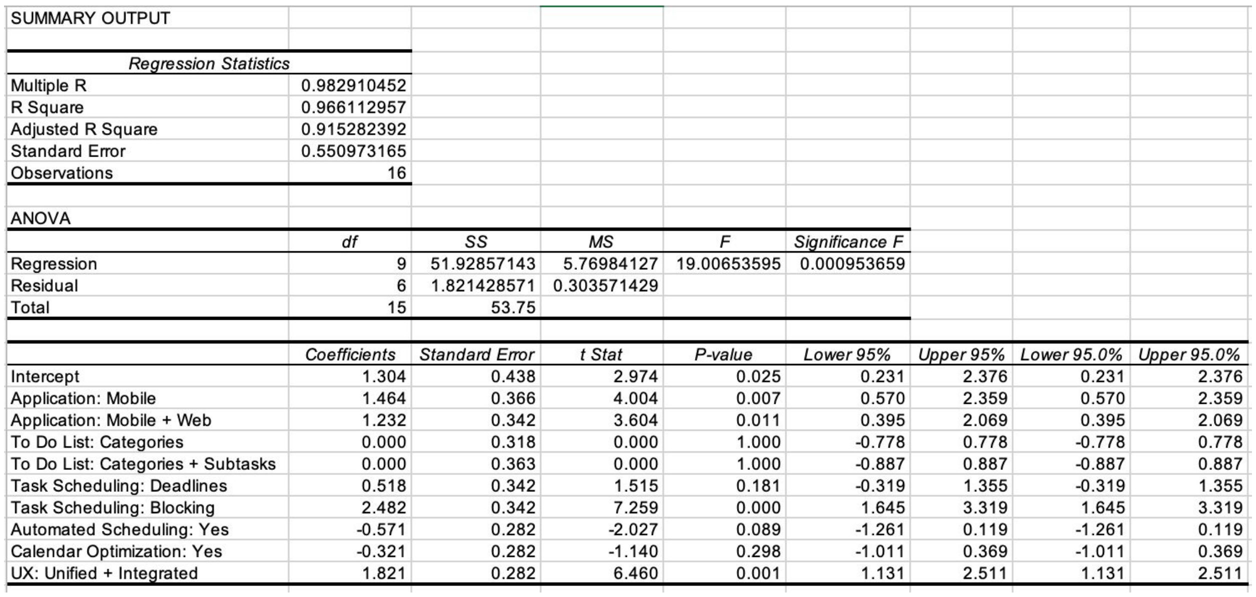
○ ○ ○ ○ ○ ○ ○

Indicator Variables



Profile	Application: Mobile	Application: Mobile + Web	To Do List: Categories	To Do List: Categories + Subtasks	Task Scheduling: Deadlines	Task Scheduling: Blocking	Automated Scheduling: Yes	Calendar Optimization: Yes	UX: Unified + Integrated
(1)	0	0	0	0	1	0	1	0	0
(2)	1	0	0	0	1	0	1	1	1
(3)	1	0	0	1	0	1	0	0	1
(4)	1	0	1	0	0	0	1	1	0
(5)	0	1	1	0	0	1	1	0	0
(6)	0	1	0	0	0	0	0	0	1
(7)	0	0	0	1	0	0	1	0	0
(8)	0	1	0	0	0	1	0	1	0
(9)	0	1	1	0	1	0	1	0	1
(10)	0	1	0	1	1	0	0	1	0
(11)	1	0	1	0	0	1	0	0	1
(12)	0	0	1	0	0	0	0	1	1
(13)	0	1	0	1	0	0	1	1	1
(14)	0	0	0	0	0	1	1	1	1
(15)	0	0	1	0	1	0	0	1	0
(16)	1	0	0	0	0	0	0	0	0

Sample Regression Output



SUMMARY OUTPUT

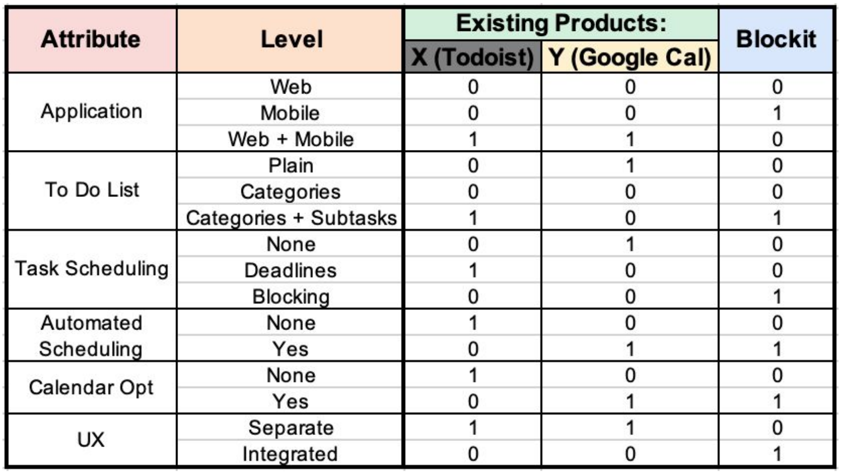
Regression Statistics	
Multiple R	0.982910452
R Square	0.966112957
Adjusted R Square	0.915282392
Standard Error	0.550973165
Observations	16

ANOVA

	df	SS	MS	F	Significance F
Regression	9	51.92857143	5.76984127	19.00653595	0.000953659
Residual	6	1.821428571	0.303571429		
Total	15	53.75			

	Coefficients	Standard Error	t Stat	P-value	Lower 95%	Upper 95%	Lower 95.0%	Upper 95.0%
Intercept	1.304	0.438	2.974	0.025	0.231	2.376	0.231	2.376
Application: Mobile	1.464	0.366	4.004	0.007	0.570	2.359	0.570	2.359
Application: Mobile + Web	1.232	0.342	3.604	0.011	0.395	2.069	0.395	2.069
To Do List: Categories	0.000	0.318	0.000	1.000	-0.778	0.778	-0.778	0.778
To Do List: Categories + Subtasks	0.000	0.363	0.000	1.000	-0.887	0.887	-0.887	0.887
Task Scheduling: Deadlines	0.518	0.342	1.515	0.181	-0.319	1.355	-0.319	1.355
Task Scheduling: Blocking	2.482	0.342	7.259	0.000	1.645	3.319	1.645	3.319
Automated Scheduling: Yes	-0.571	0.282	-2.027	0.089	-1.261	0.119	-1.261	0.119
Calendar Optimization: Yes	-0.321	0.282	-1.140	0.298	-1.011	0.369	-1.011	0.369
UX: Unified + Integrated	1.821	0.282	6.460	0.001	1.131	2.511	1.131	2.511

Product Profiles



Attribute	Level	Existing Products:		Blockit
		X (Todoist)	Y (Google Cal)	
Application	Web	0	0	0
	Mobile	0	0	1
	Web + Mobile	1	1	0
To Do List	Plain	0	1	0
	Categories	0	0	0
	Categories + Subtasks	1	0	1
Task Scheduling	None	0	1	0
	Deadlines	1	0	0
	Blocking	0	0	1
Automated Scheduling	None	1	0	0
	Yes	0	1	1
Calendar Opt	None	1	0	0
	Yes	0	1	1
UX	Separate	1	1	0
	Integrated	0	0	1

Appendix E: Scheduling Data

Scheduling Algorithm: Sample Calendar Day Events (Wednesday, November 28th, 2018)

2018-11-28	10:00	10:30	Meeting with chief of staff
2018-11-28	10:45	11:15	Media engagement
2018-11-28	12:15	12:30	White House military office departure photos
2018-11-28	12:45	13:45	Lunch with the governor of new york state
2018-11-28	14:00	14:30	Meeting with presidential personnel
2018-11-28	14:30	14:45	Policy time
2018-11-28	14:45	15:15	Policy time
2018-11-28	16:35	16:40	Depart White House en route The Ellipse
2018-11-28	16:45	16:55	Photo opportunity with national christmas tree lighting ceremony participants
2018-11-28	17:00	18:00	National Christmas tree lighting ceremony
2018-11-28	18:05	18:10	Depart The Ellipse en route the White House

Scheduling Algorithm: Sample Violation Data (Week of November 26th, 2018)

App	**Violation**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Blockit	Flow	0	0	0	1	1	1	0	0	0	0
	EOD	0	0	0	0	0	0	1	0	0	0
	Event	0	0	0	0	0	0	0	1	1	1
Clockwise	Flow	0	0	0	0	0	1	2.5	0	0	0
	EOD	0	0	0	0	0	1	1	1	0	0
	Event	0	0	0	0	0	0	0	0	1	1
XAI	Flow	0	0	0	0	0	0	0	0	0	0
	EOD	0	0	0	1	0	1	1	0	0	0
	Event	0	0	0	0	0	0	0	0	0	1

Appendix F: Code Snippets

There are four code snippets attached to this appendix:

1. **Google Log-in**
2. **Todo API Endpoints**
3. **Todo Database Schema**
4. **Scheduling Algorithm**

```
async signInWithGoogleAsync() {
  const config = {
      iosClientId: GOOGLE_IOS_CLIENTID,
      scopes: [PROFILE, EMAIL, GCAL_SCOPE],
  };
  try {
      const { type, accessToken, user } = await Google.logInAsync(config);
      if (type === 'success') {
        console.log('Google auth successful!')
        this.setState({loggedInUserEmail: user.email})
        this.setState({loggedInUserAccessToken: accessToken})
        this.sendGoogleAccessTokenToServer()
      } else {
        console.log('Google auth not successful!')
      }
  } catch (e) {
      return { error: true };
  }
}
```

PDF document made with CodePrint using Prism

```javascript
const express = require('express')
router = express.Router()

const Todo = require('../models/todo')

router.get('/test', (req, res) => {
    res.json('Todo description from API')
})

router.get('/', (req, res) => {
    Todo.find()
    .then(todos => res.json(todos))
    .catch(err => res.status(404).json({ no_todos_found: 'No Todos found' }))
})

router.get('/:id', (req, res) => {
    Todo.findById(req.params.id)
    .then(todo => res.json(todo))
    .catch(err => res.status(404).json({ no_todo_found: 'No Todo found' }))
})

router.get('/byauthor/:userid', (req, res) => {
    Todo.find({author: req.params.userid})
    .then(todo => res.json(todo))
    .catch(err => res.status(404).json({ no_todo_found: 'No Todo found' }))
})

router.post('/', (req, res) => {
    Todo.create(req.body)
    .then(todo => res.json({ msg: 'Todo created successfully' }))
    .catch(err => res.status(400).json({ error: 'Unable to add this todo' }))
    // TODO: return todoID for potential deletes
})

router.put('/:id', (req, res) => {
    Todo.findByIdAndUpdate(req.params.id, req.body)
    .then(todo => res.json({ msg: 'Updated successfully' }))
    .catch(err => res.status(400).json({ error: 'Unable to update the database' }))
})

router.delete('/:id', (req, res) => {
    Todo.findByIdAndRemove(req.params.id, req.body)
    .then(todo => res.json({ msg: 'Todo entry deleted successfully' }))
    .catch(err => res.status(404).json({ error: 'Todo not found' }))
})

module.exports = router
```

PDF document made with CodePrint using Prism

```
const mongoose = require('mongoose')
const Schema = mongoose.Schema

const TodoSchema = new Schema({
    title: String,
    subtitle: String,
    deadline: Date,
    priority: {
        type: String,
        enum: ['none', 'low', 'medium', 'high'],
        default: 'none'
    },
    blocks: [{
        type: Schema.Types.ObjectId,
        ref: 'block'
    }],
    scheduled: Boolean,
    author: {
        type: Schema.Types.ObjectId,
        ref: 'user'
    }
})

module.exports = Todo = mongoose.model('todo', TodoSchema)
```

PDF document made with CodePrint using Prism

```
def getFreeBlocks(self, date):
    day = self.getDay(date)
    # sorted_events = day.sort_values(by=['time_start'], ascending=True)
    free_blocks = []
    last_event = None
    for event in day.itertuples():
        curr_event = event
        if curr_event and last_event:
            last_end = getattr(last_event, "time_end")
            curr_start = getattr(curr_event, "time_start")
            if last_end < curr_start:
                # print('open', last_end, curr_start)
                free_blocks.append((last_end, curr_start))
            else:
                # print('closed', last_end, curr_start)
                pass
        last_event = curr_event

    return free_blocks


def schedule_event_on_days(self, dates, duration_seconds, name, location, category):
    datetime_blocks = []
    for elem in dates:
        block = self.schedule_event_on_day(elem, duration_seconds, name, location, category)
        if block:
            datetime_blocks.append((datetime.combine(elem, block[0]), datetime.combine(elem, blc

    # print(datetime_blocks)

    # First free block week heuristic
    first_free_block = None
    first_free_block_dur = -1
    for datetime_block in datetime_blocks:
        first_time = datetime_block[0]
        second_time = datetime_block[1]
        diff = second_time - first_time
        if diff >= duration_seconds:
            first_free_block = datetime_block
            first_free_block_dur = diff

    # Maximum sufficiently large free block in the day
    max_free_block = first_free_block
    max_free_block_dur = first_free_block_dur
    for datetime_block in datetime_blocks:
        first_time = datetime_block[0]
        second_time = datetime_block[1]
        diff = second_time - first_time
        if diff >= duration_seconds and diff > max_free_block_dur:
            max_free_block = datetime_block
            max_free_block_dur = diff

    # Minimum sufficiently large free block of day
    min_free_block = first_free_block
    min_free_block_dur = first_free_block_dur
    for datetime_block in datetime_blocks:
        first_time = datetime_block[0]
        second_time = datetime_block[1]
        diff = second_time - first_time
        if diff >= duration_seconds and diff < min_free_block_dur:
            min_free_block = datetime_block
            min_free_block_dur = diff

    # chosen_block = first_free_block
    # chosen_block = max_free_block
    chosen_block = min_free_block

    print(chosen_block)
    self.addEvent(chosen_block[0].date(), chosen_block[0].time(), chosen_block[1].time(), name,

    return chosen_block
```

PDF document made with CodePrint using Prism