

6-13-2019

# The Original Beat: An Electronic Music Production System and Its Design

Eli Yale

Christian Quintero

Matt Kordonsky

Follow this and additional works at: [https://scholarcommons.scu.edu/cseng\\_senior](https://scholarcommons.scu.edu/cseng_senior)



Part of the [Computer Engineering Commons](#)

---

## Recommended Citation

Yale, Eli; Quintero, Christian; and Kordonsky, Matt, "The Original Beat: An Electronic Music Production System and Its Design" (2019). *Computer Engineering Senior Theses*. 152.  
[https://scholarcommons.scu.edu/cseng\\_senior/152](https://scholarcommons.scu.edu/cseng_senior/152)

This Thesis is brought to you for free and open access by the Engineering Senior Theses at Scholar Commons. It has been accepted for inclusion in Computer Engineering Senior Theses by an authorized administrator of Scholar Commons. For more information, please contact [rsroggin@scu.edu](mailto:rsroggin@scu.edu).

**SANTA CLARA UNIVERSITY**  
**DEPARTMENT OF COMPUTER ENGINEERING**

Date: June 13, 2019

I HEREBY RECOMMEND THAT THE THESIS PREPARED UNDER MY SUPERVISION BY

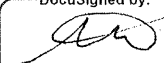
**Eli Yale**  
**Christian Quintero**  
**Matt Kordonsky**

ENTITLED

**The Original Beat: An Electronic Music Production System and Its Design**

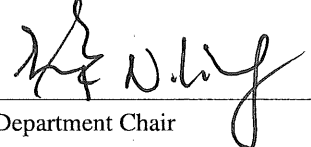
BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

BACHELOR OF COMPUTER SCIENCE AND ENGINEERING

DocuSigned by:  
  
D8129749E98B404...

---

Thesis Advisor

  
Department Chair

# **The Original Beat: An Electronic Music Production System and Its Design**

by

Eli Yale  
Christian Quintero  
Matt Kordonsky

Submitted in partial fulfillment of the requirements  
for the degree of  
Bachelor of Computer Science and Engineering  
School of Engineering  
Santa Clara University

Santa Clara, California  
June 13, 2019

# **The Original Beat: An Electronic Music Production System and Its Design**

Eli Yale  
Christian Quintero  
Matt Kordonsky

Department of Computer Engineering  
Santa Clara University  
June 13, 2019

## **ABSTRACT**

The barrier to entry in electronic music production is high. It requires expensive, complicated software, extensive knowledge of music theory and experience with sound generation. Digital Audio Workstations (DAWs) are the main tools used to piece together digital sounds and produce a complete song. While these DAWs are great for music professionals, they have a steep learning curve for beginners and they must run native on a user's computer. For a novice to begin creating music takes much more time, effort, and money than it should. We believe anyone who is interested in creating electronic music deserves a simple way to digitize their ideas and hear results. With this idea in mind, we created a web based, co-creative system to allow beginners and pros alike to easily create electronic digital music. We outline the requirements for such a system and detail the design and architecture. We go through the specifics of the system we implemented covering the front-end, back-end, server, and generation algorithms. Finally, we will review our development time-line, examine the challenges and risks that arose when building our system, and present future improvements.

# Table of Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Motivation . . . . .	1
1.2	Solution . . . . .	2
<b>2</b>	<b>Requirements</b>	<b>3</b>
2.1	Functional Requirements . . . . .	3
2.2	Non-functional Requirements . . . . .	4
2.3	Design Constraints . . . . .	4
<b>3</b>	<b>Use Cases</b>	<b>5</b>
<b>4</b>	<b>Activity Diagram</b>	<b>6</b>
<b>5</b>	<b>Conceptual Model</b>	<b>8</b>
5.1	Homepage . . . . .	8
5.2	Digital Input Piano Roll . . . . .	9
5.3	Download Page . . . . .	10
<b>6</b>	<b>Technologies Used</b>	<b>11</b>
6.1	Languages . . . . .	11
6.2	Packages . . . . .	11
6.3	Frameworks . . . . .	11
6.4	Databases . . . . .	11
6.5	Explanation . . . . .	12
<b>7</b>	<b>System Architecture</b>	<b>13</b>
7.1	Architectural Diagram . . . . .	13
7.2	Production Environment . . . . .	14
7.3	Security . . . . .	15
7.4	Backend Architecture . . . . .	16
<b>8</b>	<b>Music Generation Algorithms</b>	<b>19</b>
8.0.1	KeyChord . . . . .	19
8.0.2	KeyChord2 . . . . .	19
8.0.3	BayesNet . . . . .	20
8.1	Drum Layer Generation . . . . .	21
<b>9</b>	<b>Design Rationale</b>	<b>22</b>
9.1	Requirements Rationalization . . . . .	23
9.2	Package and Front-end Rationalization . . . . .	23
9.3	Generation Engine Rationalization . . . . .	24

<b>10 Test Plan</b>	<b>25</b>
10.1 Testing Done . . . . .	25
10.1.1 Unit Testing . . . . .	25
10.1.2 White Box Testing . . . . .	25
10.2 Testing Needed . . . . .	26
10.2.1 Black Box Testing . . . . .	26
10.2.2 Stress Testing . . . . .	26
<b>11 Development Timeline</b>	<b>27</b>
<b>12 Social Issues</b>	<b>29</b>
12.1 Ethical Analysis . . . . .	29
<b>13 Conclusions</b>	<b>30</b>
13.1 Results . . . . .	30
13.2 Future Work . . . . .	30
13.3 Obstacles Encountered . . . . .	31
13.3.1 Lack of Percussive Pitches in Music21 . . . . .	31
13.3.2 Debugging Bayes Net . . . . .	31
13.3.3 Deployment Challenges . . . . .	31
13.3.4 Integration of IFrames . . . . .	32
<b>14 References</b>	<b>33</b>
14.1 Open Source Products . . . . .	33
14.2 Research Paper . . . . .	33
<b>A User Manual</b>	<b>34</b>
A.1 Access . . . . .	34
A.2 Accounts . . . . .	34
A.3 Production . . . . .	34
A.3.1 Create Melody . . . . .	35
A.3.2 Upload Melody . . . . .	35
A.3.3 Generated Layers . . . . .	35

# List of Figures

- 3.1 Use Cases . . . . . 5
- 4.1 Activity Diagram . . . . . 6
- 5.1 Homepage . . . . . 8
- 5.2 Melody Creation Page . . . . . 9
- 5.3 Download Page . . . . . 10
- 7.1 Architecture Diagram . . . . . 17
- 7.2 UML Class Diagram . . . . . 18
- 8.1 BayesNet Design . . . . . 21
- 11.1 Development Timeline . . . . . 27
- 11.2 Task Breakdown . . . . . 28

# Chapter 1

## Introduction

### 1.1 Motivation

Electronic music, including Progressive House, is a genre of electronic music where songs consist of layers of digital sounds generated by a computer. It is up to the producer to arrange the drum sounds into a rhythm, then select synthesizer notes to create a melody. Many producers add additional digital sounds to complement the melody and rhythm and add to the complexity of the track. The layered nature of electronic music means there is lots of room for auto-generation and co-creation with computers.

There exists AI music tools like Jukedeck, Mubert, and Amper which provide full composition of music intended for backing tracks. The problem is they take little to no input from the user and produce a pseudo random track. Other tools like Humtap are app based and produce tracks that are not easily integrated into a DAW for the professional producer. They also provide little inspiration and minimal ability to change sounds within the track.



## 1.2 Solution

This is where our app aims to fill the gaps. Rather than requiring a user to generate every layer in their song, we enable them to start with one layer, and our system will generate the other layers and sounds. For example if a user has a simple melody in their head, they may upload it in MIDI format to our system, and our system will work to generate an accompanying harmony. Since end-to-end song generation is very difficult, we focused initially on harmony generation. After we built several crude algorithms, we implemented the web application portion of the system. From there we refined our generation algorithms and built a machine learning model. The final system met all but one of our initial functional system requirements and we met one out of four recommended functional requirements. The remaining features are discussed in the future work section of this report. Furthermore, we met all of our non-functional requirements, and stayed within our design constraints. We are confident that with our final system, even experienced professionals will be able to create quick sketches of their ideas and use the generated layers as inspiration. Since our system is co-creative and web-based, it is unlike any other electronic music tool on the market. With a simple UI that presents the song as a layer of tracks, beginners will have no problem navigating the app and producing their first song. Instead of complicated music software being a barrier to entry, our system will be a path for new and experienced producers to explore the world of music.

# Chapter 2

## Requirements

It was our priority to create a music generation engine and enhance it through a website. The music generation engine generates the musical elements to be presented by the website. It was important for the website we created to have a user-friendly experience by being both easy to use and contain all of the necessary information. For our engine and website to complete these tasks we outlined the following requirements.

### 2.1 Functional Requirements

- The system will have a secure log in system that will ensure the safety of the clients information
- The system will allow the upload of a MIDI file
- The system will allow users to enter a short melody on a digital keyboard
- The system will use the algorithm chosen by the user to generate a MIDI file that harmonizes with the initial MIDI upload
- The system will allow for the download of the newly generated MIDI file
- The system will allow users to see generated tracks in a piano roll
- The system will allow the user to change the sounds associated with the generated track
- The system will add a drum beat to accompany the melody input

## **2.2 Non-functional Requirements**

- The system should be user friendly and easy to use to ensure musicians can enjoy their creation process
- The system should have a quick response time
- The system should be a reliable system with few system bugs
- The system should have a minimalist design to ensure an elegant design style
- The system will scale to allow for more than 50 users in a moment

## **2.3 Design Constraints**

- The system will be deployed on DigitalOcean
- The front-end will be compatible with all browsers while the external MIDI device features will only be compatible with Google Chrome
- The system will be relatively OS and server stack independent allowing deployment on servers other than a DigitalOcean instance

## Chapter 3

# Use Cases

The use cases diagram shown in Figure 3.1 demonstrates the possible uses for our web application. There is only one type of user and that is a general visitor that wants to produce harmonies for their chord progression.

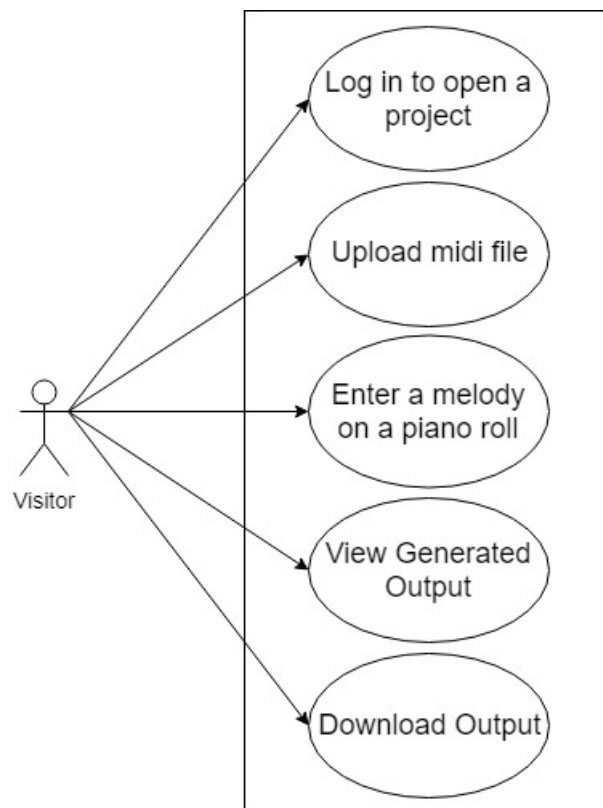


Figure 3.1: Use Cases

## Chapter 4

# Activity Diagram

The Activity diagram outlines the path a user takes when navigating throughout our web application and its functionalities.

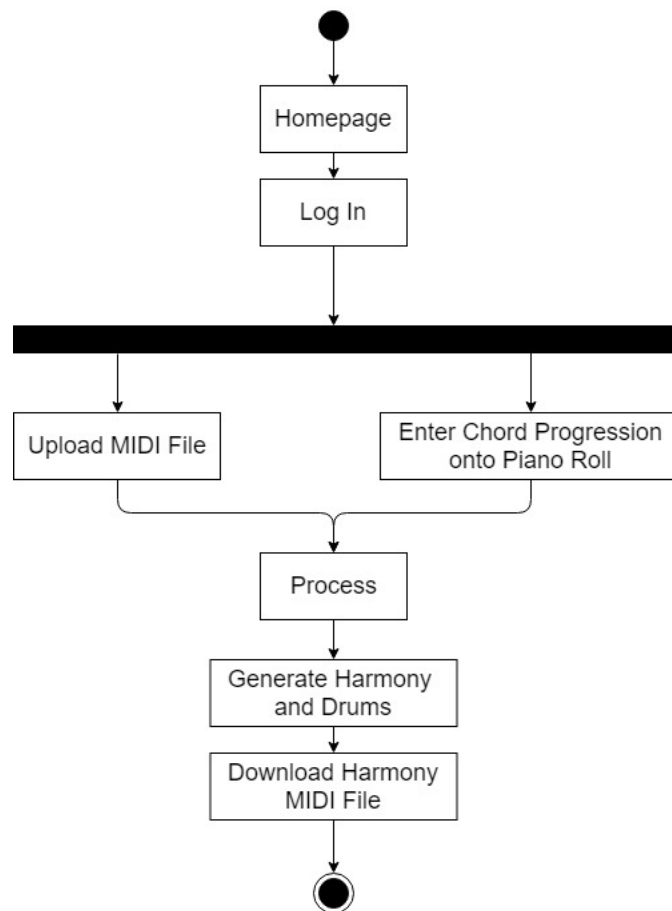


Figure 4.1: Activity Diagram

When a user visits our website, they will be presented with a login page. Once the client logs in, the user will have two options. The first option is the ability to upload a midi file to process and the latter is the ability to use an in browser piano roll to create a musical melody. The website will proceed to process the midi input by the client. After the website has processed the midi file, the user will be presented with one piano roll for the generated harmony and one piano roll for a drum beat to the melody input. The client will ultimately be able to choose to download any of the tracks separated from each other or all in a single midi file.

## Chapter 5

# Conceptual Model

The conceptual model is a visual representation of what the web application looks like.

### 5.1 Homepage

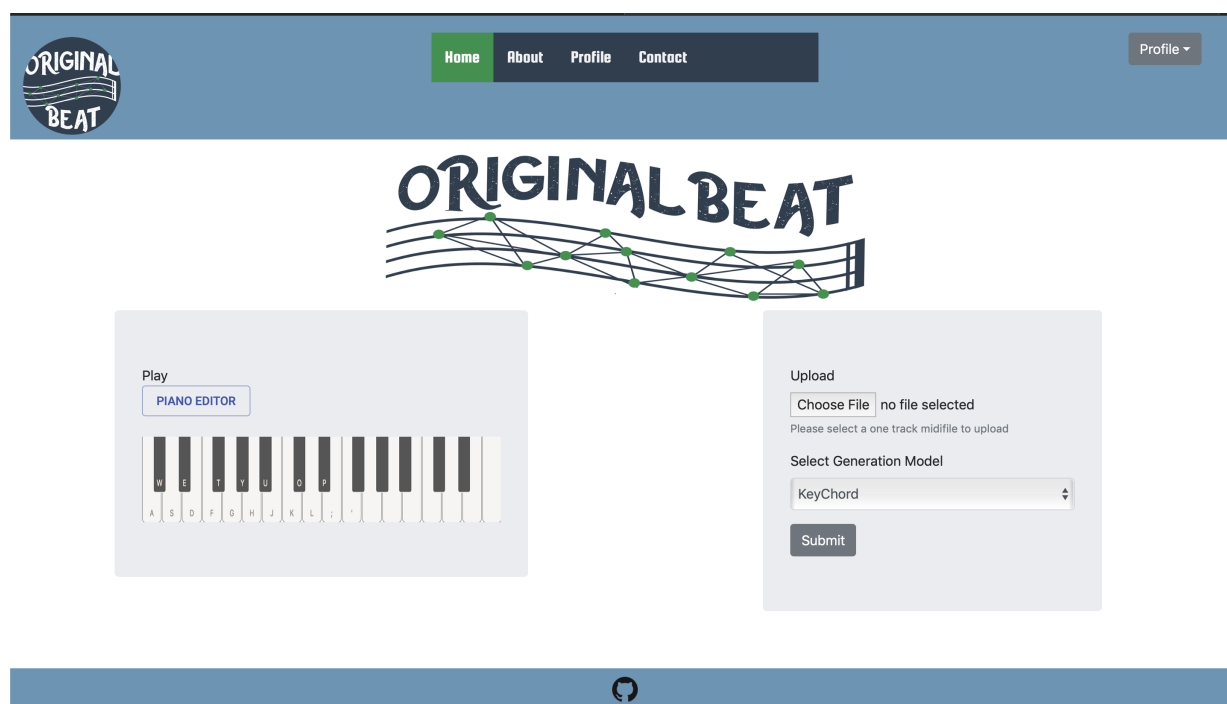


Figure 5.1: Homepage

Figure 5.1 presents the web applications homepage. For our homepage, we wanted a simple user interface, immediately allowing the user to choose whether to upload a midi file or create one in browser.

## 5.2 Digital Input Piano Roll

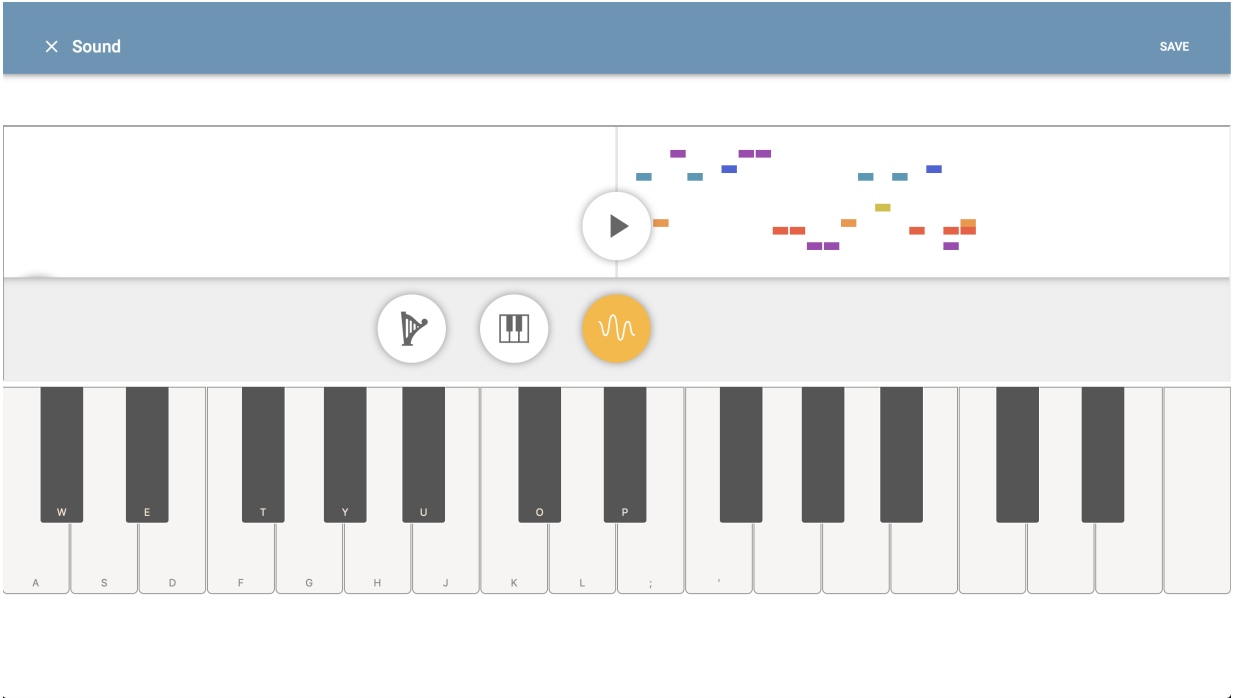


Figure 5.2: Melody Creation Page

Figure 5.2 models the layout for our in browser piano roll. Users will be presented with a simplified piano roll if they choose to create their melody in the browser using their keyboard or mouse, rather than upload a midi file.



### 5.3 Download Page

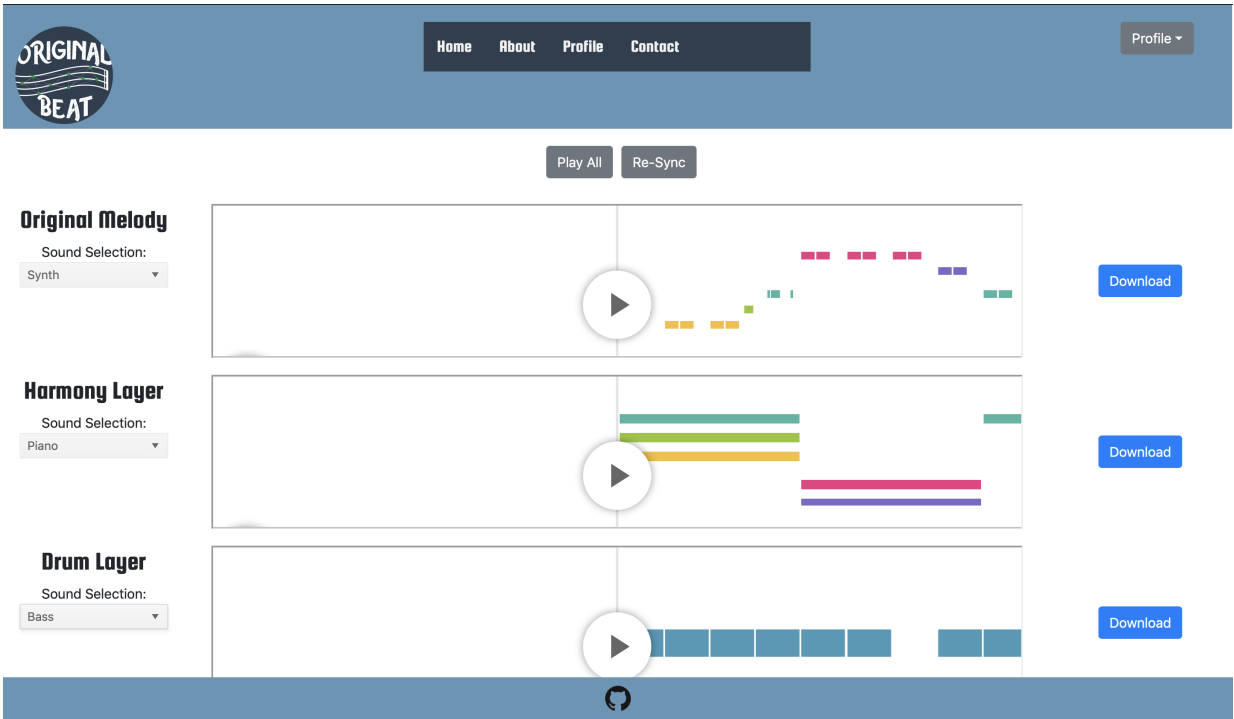


Figure 5.3: Download Page

Figure 5.3 models the download page, which will present users with three piano rolls: one for the original melody, one for the harmony created from the original processed melody, and one for the generated drums. The user can then change the sound of each piano and listen to them individually or synchronously. The user will be able to ultimately download all three layers as a midi file using the download button.

## Chapter 6

# Technologies Used

### 6.1 Languages

- Python
- JavaScript

### 6.2 Packages

- mido
- music21
- Tone.js
- Pomegranate
- MIDI.js

### 6.3 Frameworks

- React
- Django

### 6.4 Databases

- SQLite3 DB

## 6.5 Explanation

We can divide the technologies we use into two groups, the core application technologies upon which the system runs, and the supplementary libraries and packages that perform our midi manipulation and front-end. Because our system is web based, the foundational technologies are the standard web development languages. In order to improve security, robustness, and to make interfacing with our generation engine straightforward, we use the Django server framework with a SQLite3 database. Django uses python which is the language we use in our music generation engine. SQLite3 is a modern database technology that provides ease of use for storing our user's data including, credentials, session info, and projects and integrates well with Django. As part of our implementation we experimented with generation models and determined which algorithms resulted in the best layering. We experimented with Markov chains and used the python package Markovify combined with manual scraping for a corpus of midi files in our specified format. In addition to Markov Chains and basic Expert System algorithms, we tried the more complex Bayesian Network model. For this we used the statistics package Pomegrante, which provides a convenient Bayesian Network class.

On top of the above core architectural technologies, we built our app to adhere to the midi standard by using additional python and client side packages. Our generation engine requires extensive midi manipulation and for this we used a combination of mido, and music21; mido for its ease of use and our familiarity with it, and music21 for its powerful feature set. On the javascript side, MIDI.js was used as a midi sequencer to capture the sequence from the onscreen piano roll, while Tone.js helped us with audio samples. Finally we used the React front-end library with JSX syntax to render our complete front end including an onscreen keyboard and display of the midi in the standard piano roll.

## Chapter 7

# System Architecture

### 7.1 Architectural Diagram

With these technologies in mind, we now lay out the architecture for our system as seen in Figure 7.1. To begin, we chose a web based approach for several reasons. We want our app to be beginner friendly as well as powerful and a web app will satisfy both these goals. New users dont need to bother downloading our software and we are free to modify it and improve the generation engine at anytime. Modern browsers also provide powerful languages and processing which is sufficient to render our UI and process the input events.

We selected client server architecture since we have high compute jobs to be done on both the client and server side. While data is a significant component of our system and we are storing several types in our database, we feel that architectures such as data centric or data flow would be ill suited to the heavy client side compute. The largest type of data we will need to manage will be midi data. Standard two track midi songs are typically between 10-20KB. If we use a training set of 100,000 songs, we can still be under 1GB. Hence, storing both training data, and user generated tracks on our server should not be an issue.

## 7.2 Production Environment

In our original design, we planned to use the Heroku cloud server platform to host our final application. With our initial research, the Heroku Dynos seemed to fulfill our needs at a fair price but as we attempted the initial deployment, we realized a major shortcoming. The Heroku Dyno VPS does not allow for local storage which meant we would have no way to store the users uploaded midi files on our server. We would instead need to purchase bulk storage on another server and transfer midi files between them. This clearly seemed like unnecessary overhead so we explored other options. A simple Amazon Web Services E3 instance would be more expensive than a server of similar specifications on Digital Ocean so we decided to use DigitalOcean as our cloud provider. The DigitalOcean Droplet turned out to be very intuitive to instantiate and provided a convenient management console.

Our droplet instance runs Ubuntu 16.04.6 LTS and is physically located in Digital Oceans San Francisco server farm for the fastest delivery to the Bay Area. With our Droplet instance ready to go, we began customizing our cloud server to our needs. In our initial design, we did not choose a web server software but with a little research, it was clear that Nginx would be the simplest and fastest server to run with Django. We used Django as our python web server framework as per our design constraints but we realized that Django recommends using a WSGI compliant middleware to communicate with Nginx for added security in a production environment. For this, we installed Gunicorn (a WSGI compliant HTTP server) and set it up to run as a system service. Next, we installed the same versions of Django and SQLite as our development environment for seamless deployments. Git is used as version control to download the production build of the site. Finally, so that unicorn and Django restart automatically in case of a server reboot, we run supervisor, a common server management tool. When all is said and done, deployment can be done in just a few commands. Please see the docs section of our repository and the `server_notes.md` file for more info on our production server and the steps taken for deployment.

## 7.3 Security

It was our goal and a part of our system requirements that our production web application be secure and encrypted. In order to do this, we started with server security. The server follows traditional linux security protocols with only one user being given root permissions. Nginx is started as root but not run with root permissions. All configuration files can only be edited by a user with root permissions, and all log files can only be written by processes with user permissions identical to the log files. Log files are rotated automatically by Nginx. The Linux utility Uncomplicated Firewall (UFW) was used to set iptables to only allow traffic on port 443 for HTTPS and port 22 for ssh connections. With these measures in place, we believe our physical server meets basic security standards.

It was also important for us to encrypt all traffic between our web server and clients so for this we used the free Certificate Authority Lets Encrypt along with their certbot to verify our server and obtain a TLS certificate. With a few changes to the Nginx configuration, we were able to serve all traffic over https with end to end encryption.

## 7.4 Backend Architecture

The initial design for our software architecture was fairly general with the main goals being modularity and simplicity. We wanted our backend architecture to be intuitive, yet powerful so we imaged a hot-swappable architecture where we could swap out different generation algorithms with a common interface for passing midi data into them. This would allow us to rapidly deploy with simple algorithms while experimenting with machine learning algorithms. So, in the initial stages of our build, we refined this concept and defined the classes and components of such an architecture. The resultant architecture we call the Beat Engine and can be seen in the UML diagram in figure 7.2.

This architecture begins with the Django views.py file which presents a view when the user visits a specific URL on our site. When a user submits a midi file, a POST request is sent to the server and one of our upload views is hit. This view instantiates a Beat Engine object which is where generation begins. The Beat Engine is responsible for pre-processing the midi file, creating a generation algorithm, and writing the output back to a file. After the Beat engine is finished, control returns to Django. Each user to our site has their own beat engine instantiated when the upload a melody.

When the Beat Engine is initialized, it creates an instance of the Beat class. The Beat class is a static container for the song data and has no methods. It is used simply as a standard for passing data to our generation classes. The Beat instance has instance variables such as key, tempo, and notes found in the uploaded melody. The most important attribute is the Music21 stream representation of the inputted melody. This is the data format that our algorithms use, and they will then generate a separate stream for the output.

Next, the Beat engine instantiates a generation class based on the method that the user selects. For the purpose of this Senior Design. We have built three generation mechanisms: KeyChord, KeyChord2, and BayesNet. KeyChord and KeyChord2 are Expert System algorithms and will be briefly explained in the following Chapter. BayesNet is the single Machine Learning model we experimented and will similarly be explained in detail in the following Chapter. The beat engine passes the Beat instance to the generation model that has been selected and the Generation Model performs the generation. The output from the Generation model is written to the harmony stream instance variable of the beat class and the generation engine returns control to the Beat Engine. Finally, the Beat Engine writes the harmony and melody combined stream to an output file and returns control to Django. Django now serves the outputted file back to the user and the user can view their generated track. For more info on the Beat Engine and generation process, please the documentation on the classes inside the engine directory of our repository.

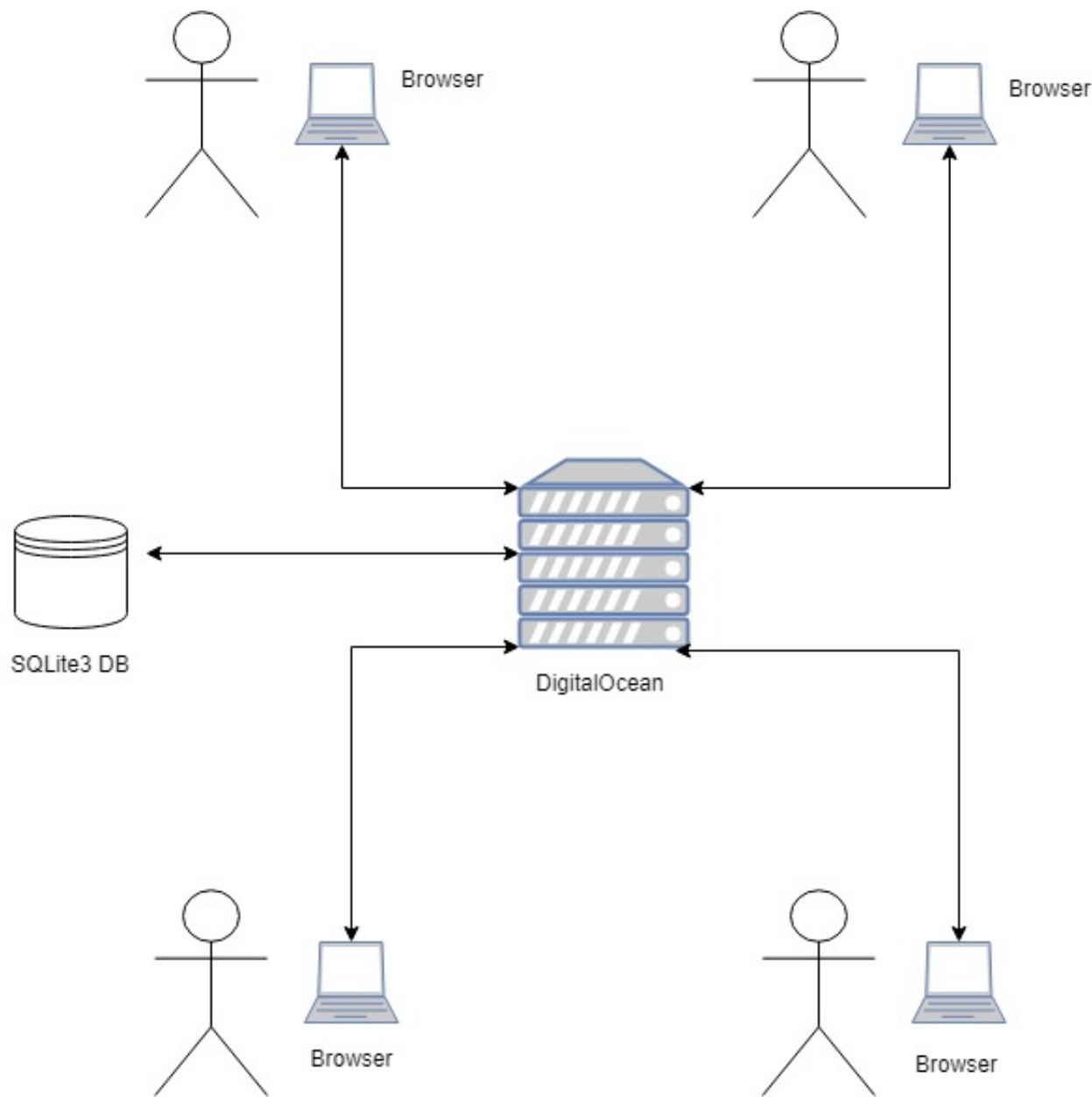


Figure 7.1: Architecture Diagram



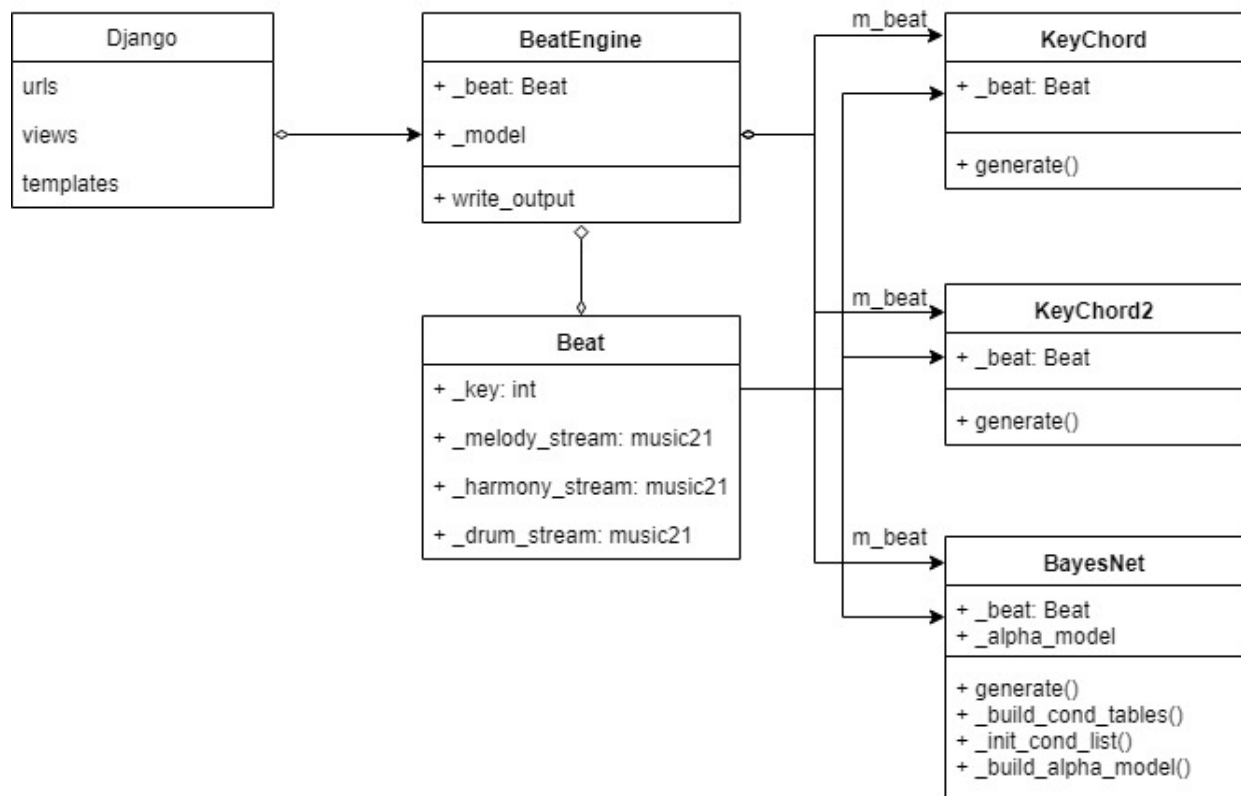


Figure 7.2: UML Class Diagram

## Chapter 8

# Music Generation Algorithms

Our web application uses 3 different harmonizing generation algorithms. This includes a combination of simple expert systems and a complete machine learning system. Individuals who use our web application will be able to select which of the 3 generation engines they would like to use to process their melodies and generate unique harmonizing layers. Our goal was to allow users the flexibility to decide which engine to use in order to generate diverse layers. We created 2 expert system engines which we called KeyChord and KeyChord2, and called our machine learning engine BayesNet.

### 8.0.1 KeyChord

KeyChord is the initial expert system engine that we built for our application. This initial expert system uses a very crude algorithm to generate a simple harmony layer using the user-provided melody. This algorithm parses the midi file of the melody and takes every unique note played in a specific measure and creates a harmonizing chord from those notes for that specific measure. This algorithm continues to do the same thing for every measure in the midi file until a simple, yet effective harmony chord is produced for every measure.

### 8.0.2 KeyChord2

KeyChord2 was our improved expert system. To begin our design of KeyChord2, we took our implementation of KeyChord and added music theory to add diversity between chords used. According to music theory every musical key should have a total of 7 harmonizing chords. To find these chords one would permute through the scale of the key and begin the scale at the current scale partial. From there the algorithm takes the first, third, and fifth note and performs a reverse chord look-up to find all of the chords that can harmonize with the key. Our algorithm then chooses a random chord and adds one of the possible chords to each measure.

### 8.0.3 BayesNet

The BayesNet class is our attempt at a machine learning approach to the problem. The Bayesian Network is a well known, generative, probabilistic model based on Bayesian statistics. The general idea is that we can estimate the probabilities of the next chord or harmony note in a sequence given the previous melody note and harmony note. We were initially unfamiliar with this method but discovered its potential for music generation while exploring our initial plan of using a Markov Chain. We came across the paper Music Generation Using Bayesian Networks [1] which presented a general procedure for the use of Bayesian Networks. We began by interpolating their schema and simplifying the Bayesian network structure to suit our needs for electronic music. We assume that the basic music structure of an electronic song can be deconstructed into a sequence of melody notes, voicing nodes, and chords. The power of a Bayesian network comes from describing the conditional dependence between these components. As seen in figure 8.1, we determine the successive melody note is dependent on the preceding note and similarly the next chord is dependent on the the previous chord. Each melody note is dependent on both the current voicing note and current chord. Finally the voicing note is conditional on the current chord. The dependencies are expressed with arrows in our network structure. The white nodes are the notes to be given to the model, and the grey nodes are the nodes to be statistically imputed.

In order to perform the imputation we must train the model on a data set of midi songs and generate the conditional probability tables. The data set consists of 60, two track, midis songs that are derived from popular electronic music tracks. Music21 is used to parse the midi files and extract the song data. Notes are represented by their pitch class and chords are represented with their Roman Numeral notation in the given key. With music data extracted for each song, the probability tables can be filled in. These tables describe the probabilities of notes and their successors for all possible notes and chords in the corpus. The Bayes\_Net class has several methods to form these tables. After the conditional probability tables are found we construct the network using the pomegranate python library where nodes are initialized with their respective conditional probability tables and the edges are defined. Finally the trained model is baked and written to disk in JSON format. When the model is ready to be used, the generate method reads the model from disk and seeds it with an initial chord and voicing note. Then, we can iterate through the user uploaded melody and generate chord and harmony output with our model.

We have yet to perform a formal quantitative analysis of the Bayes Net model, but initial qualitative tests reveals the main shortcomings of this model is the repetition that appeared in the generated output. This stems from the fact that our data set is rather small and a majority of the songs have a significant amount of repetition between notes and chords. This leads to high probabilities for repeated notes in our model and a high likelihood of repeated output. There are a couple ways to solve this and the first would be to expand our data set to include more varied song. Of course this method would also increase the training time. The second method would be to manually configure the probabilities

according to prevailing music theory. This approach is interesting because it would be a hybrid of a machine learning model and an expert system and would certainly be something we would like to pursue in the future.

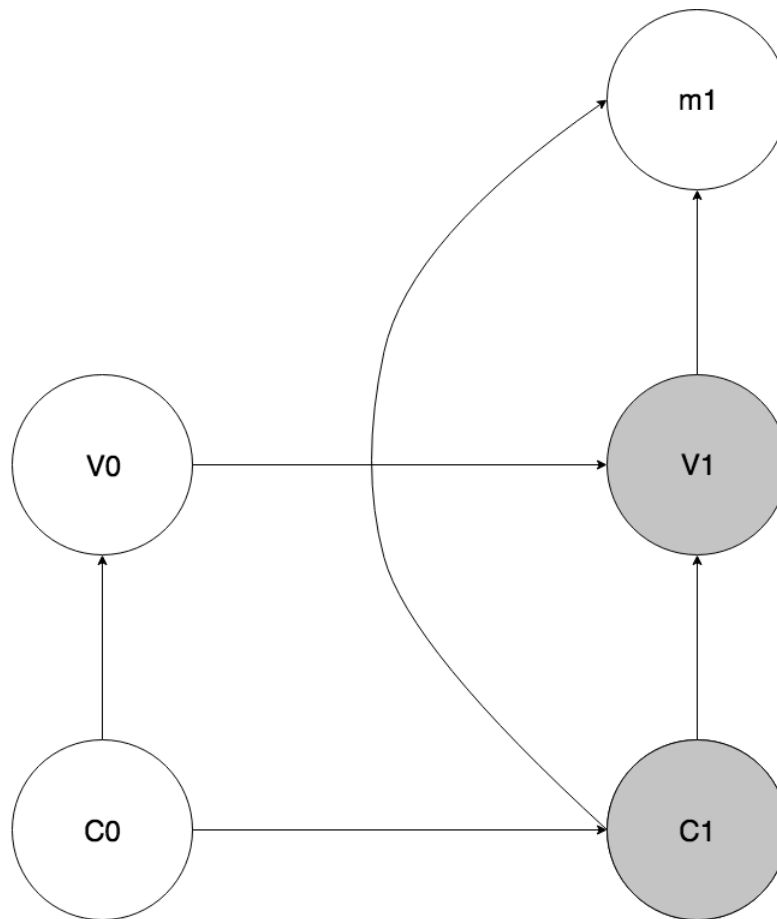


Figure 8.1: BayesNet Design

## 8.1 Drum Layer Generation

For our Drum Layer, we used a crude algorithm that adds a quarter note beat of bass notes. In order to add variation we decided to add a 30 percent chance of skipping a specific beat. We chose this algorithm because it follows the classic quarter note bass beat that progressive house is known for.

## **Chapter 9**

# **Design Rationale**

In the following section we rationalize our design choices starting from a high level abstraction and diving into the details of the system.

## 9.1 Requirements Rationalization

In order to fulfill our goal of creating a web based tool for beginners and professionals to create electronic music we specifically chose our aforementioned requirements. Our core music file format is the midi standard. This provides many benefits including widespread documentation and support as well as language specific tools for working with song data. Since midi is the widely accepted standard, it was our requirement that the system can take midi input from a variety of sources. We decided to allow midi upload so that experienced producers can improve melodies they may have already created. We also aimed to allow external midi device input when used in Google Chrome so musicians who already own a midi instrument can use our system as a co-creative DAW. However, this is a feature we have not implemented at this point. Finally, we provide an on screen keyboard so anyone anywhere can use our system. We present the downloaded track in a piano roll since this is a visualization most musicians are familiar with.

We included a section on recommended requirements since we had several stretch goals that would be "nice to have" features if we have extra time in our development period. These features are basically add-ons to the generated track that will provide the user with more ideation and a longer, more complete song. A bass and drum line combined with looping would help the user workout the structure of their song while sound and instrument manipulation would give the user more control over the final project.

The last requirements section involves the constraints on our system. These stem mostly from time and cost restrictions. We used the free level of Digital Ocean for deployment and we can scale our instances as users join the system. Since we chose the Django server framework for our backend, our app can technically run on any server regardless of the underlying server software, be it Apache, Nginx etc. Our front-end is optimized for Google Chrome but is compatible with most modern browsers including Safari, Chrome, and Edge.

## 9.2 Package and Front-end Rationalization

Since we built an on screen keyboard and piano roll, we wanted to have a component based dashboard which lends itself nicely to the React framework. Although the complex UI was a stretch goal of ours, we built simple React components for these two elements and React assists with the state management and data passing. Finally, we used several midi manipulation packages in python and javascript throughout our system which makes converting and displaying the midi much easier. Mido and music21 allow us to create an object for each track and note in python and manipulate them with functions. This is necessary when feeding a user track into our generation engine as well as manipulating songs when training the engine.

## 9.3 Generation Engine Rationalization

Initially, a large "unknown" in our system and development process was the generation engine. While we have done extensive research on the subject of algorithmic music composition, specifically EDM production, we were still aware that significant challenges would present themselves during development. To that end, we outlined a general methodology in which we approached the implementation of our engine which gave us an incrementally better algorithm. We decided to make the generation engine completely modular with respect to the mechanism of generation. It takes as input a midifile with a single track and single loop of melody. With this rigorous definition, we can effectively "hot swap" the generation module. The first generation module we chose to build was a Markov chain since this is a model we were familiar with, but ended up not being as useful since markov chains did not help us create harmony layers that harmonized with our original melodies, but rather assisted in composition production. We then decided to start off by building a crude expert system in order to start creating simple harmonization layers for our processed melodies. After building a minimum viable product, we decided to improve this expert system model to create more complex, better-sounding harmonies, however we noticed that this approach would lead us into a rabbit hole of implementing endless music theory logic. Therefore, we decided to do more research into a machine learning model and ultimately build another generation engine using the Bayesian network model. We found this graphical probabilistic model to possess more potential for more complex and pleasing harmony generations, provided that we gather a large selection of quality midi samples to train our model. We ultimately decided to allow users to select between the three generation models we created in order to allow more variety in the harmonies that users can produce.

## **Chapter 10**

# **Test Plan**

### **10.1 Testing Done**

#### **10.1.1 Unit Testing**

Throughout the production of our web application, we testing individual units as we made progress to ensure that we were diagnosing potential problems as early as possible before connecting these individual components together. We tested individual components with expected midi file types, as well as unintended files in order to test how our units handled expected data and edge cases. We ensured that each component behaved as intended and was stable before linking with other units.

#### **10.1.2 White Box Testing**

As we finished our components and integrated them together, we then performed function tests, which ensured that the components had been correctly connected and satisfied our requirements. This is white box testing since we performed functional tests with knowledge of the codebase. An example these tests was passing a midifile from the front-end to the back-end and ensuring it was parsed and fed into the generation engine. These functional tests are non-automatic and the inputs must be driven manually. Finally we deployed our app to our server as soon as possible to help us catch platform compatibility issues between our development and production servers early on.



## **10.2 Testing Needed**

### **10.2.1 Black Box Testing**

Unfortunately due to delays in debugging and system implementation, we were unable to have other users outside of the development team test out our web application and utilize its functions. Ideally, we want to advertise our web application to the general public so that we can get user feedback on what they like about our web application, what they think needs changing, and what bugs they discover that we did not resolve in our white box testing. Ultimately, this is a product open to musical creators and we want to ensure that we can provide the helpful product that we envisioned.

### **10.2.2 Stress Testing**

Along with opening our web application to the general public, we would want to also stress test the stability of our platform and server with many active users to guarantee that our framework is reliable. This will be an integral step in the deployment of our website because we hope to allow as many musicians as possible to use our website as a tool to encourage creativity. With stress testing we will ensure that the website will not crash and loose all of the progress a musician has made in the development of their song.

# Chapter 11

## Development Timeline

The Development Timeline as seen in Figure 11.1 demonstrates the schedule we chose to follow for the development of our project. Development of the project took a total of 10 weeks and after winter quarter we planned to begin the delivery phase of our project. The Task Breakdown as seen in Figure 11.2 details which group member was responsible for each task and how much of their task they had completed.

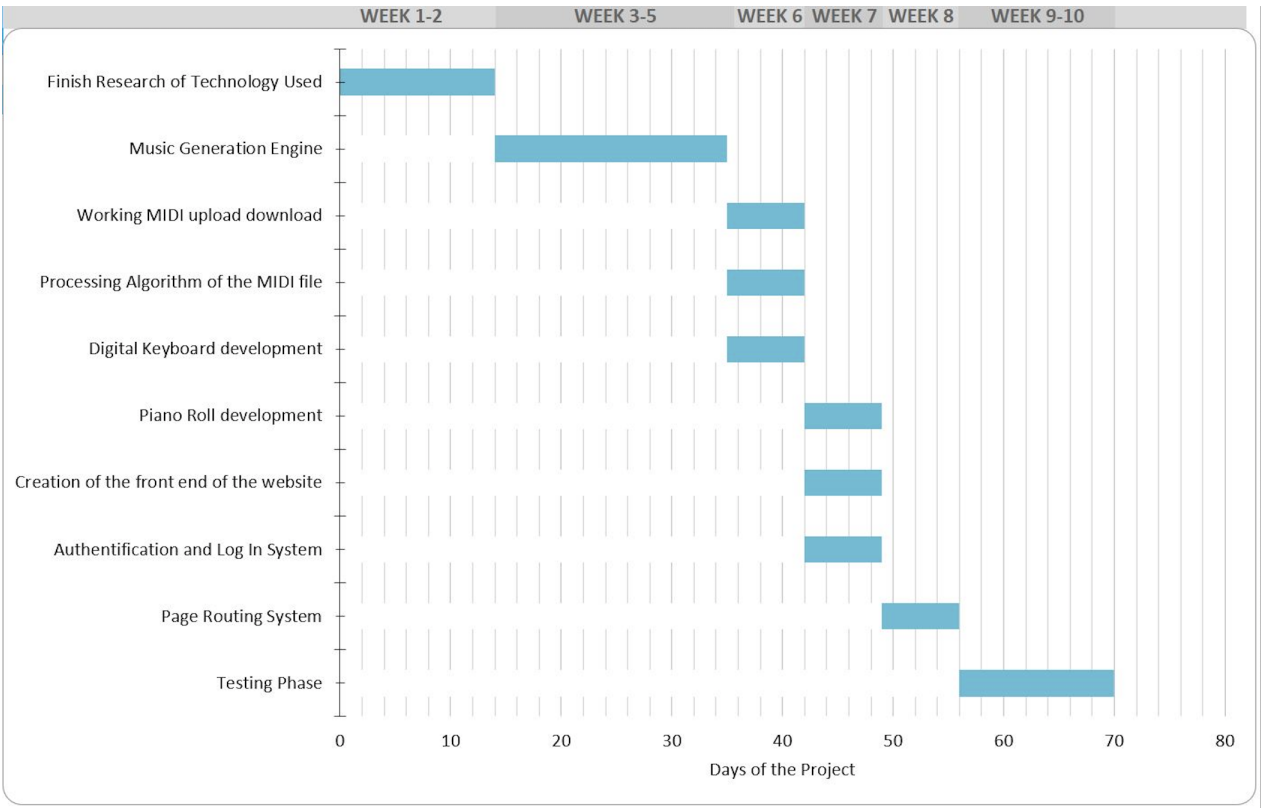


Figure 11.1: Development Timeline

TASK NAME	START DATE	END DATE	SART ON DAY	DURATION	TEAM MEMBER
Finish Research of Technology Used	1/7	1/21		0	14 Everyone
Beat Engine	1/21	2/11		14	21 Eli
KeyChord	1/21	2/11		14	21 Christian & Eli
KeyChord2	1/21	2/11		14	21 Matt
Bayes Net	1/21	2/11		14	21 Eli
DrumBeat	1/21	2/11		14	21 Matt
Working MIDI upload download	2/11	2/18		35	7 Eli
Processing Algorithm of the MIDI file	2/11	2/18		35	7 Eli
Digital Keyboard development	2/11	2/18		35	7 Eli
Piano Roll development	2/18	2/25		42	7 Christian & Eli
Creation of the Front End of the Website	2/18	2/25		42	7 Christian & Eli
Authentification and Login System	2/18	2/25		42	7 Eli
Page Routing System	2/25	3/4		49	7 Eli
Testing Phase	3/4	3/25		56	14 Everyone

Figure 11.2: Task Breakdown

## **Chapter 12**

# **Social Issues**

### **12.1 Ethical Analysis**

One of the main ethical issue that was raised during the presentation of our web application was the potential for users to use copyrighted music with our application to ultimately alter and claim as their own. In the music industry, copyright infringement is a serious issue and can lead to several lawsuits. We understand the severity of this issue and are fully against such behavior. Although we created our web application with the desire to inspire and assist in music creation, it is indeed possible for individuals to use our application with copyrighted material. However, our application serves as only an musical design platform, not an actual publishing platform. The issues with copyright infringement occurs when users decide to steal work that isn't theirs then publish and claim as their own. Therefore, this potential issue lies outside of our control and instead into the hands of publishing platforms. It is common for musical engineers to manipulate and create remixes from copyrighted music, but again becomes a legal issue only once they decide to publish or share this copyrighted material without permission from the original owner.

## Chapter 13

# Conclusions

### 13.1 Results

After completing the senior design process, our group was able to reach all of the goals we intended. We ended with a visually pleasing website with a very user friendly design. Our final product included a total of three hot-swappable music generalization algorithms that learn from the users input to help produce the best output possible. We hope that our website will be useful to all music producers to inspire them as they work on their future music production.

### 13.2 Future Work

While we were brainstorming the functional requirements of our system, we included many ambitious requirements for us to develop. Unfortunately due to lack of time these features had to be delayed until future implementations of The Original Beat. These features would all improve the usability of the system and help make The Original Beat more user friendly for all musicians alike. The features we hope to implement in the future include:

- The system can allow input from an external MIDI Device as a method for inputting a melody
- The system can include Percussive SoundFonts
- The system will allow the user to save and load multiple projects previously worked on
- The system will include a larger quantity of training data for Bayes Net Algorithm
- The system will contain an improved implementation of the digital keyboard which will work on deployment

## 13.3 Obstacles Encountered

### 13.3.1 Lack of Percussive Pitches in Music21

One of the major problems we ran into when we were trying to add the percussive pitches was the fact that music 21 doesn't allow the programmer to adjust the midi channel it is adding notes to. On a midi file, drums are stored on channel 10 but unfortunately Music21 only allows one to write to channel 1. To get around this problem, we decided to dedicate specific notes in channel 1 to drums and adjust the sound in our website.

### 13.3.2 Debugging Bayes Net

The Bayes Net package proved slightly tricky to use when we began experimentation. We were stumped on the best way to represent notes and chords so as to avoid having too many entries in our probability tables. When encoding the notes as their note name and pitch class, (i.e. C4) and encoding chords as their common name, (Cmaj) we quickly ran into very large probability tables that were inefficient for imputation. We soon learned that notes should be encoded simply by their pitch class, (i.e. c=0) and chords as their roman numeral representation, (i.e. cmaj=I). Finally, we found if all possible combinations of notes and chords weren't defined in the probability tables, the library would run into a segmentation fault. These faults were particularly nasty to debug since the underlying code in Pomegranate was written in C. There is no easy way to debug python code in python.

### 13.3.3 Deployment Challenges

This deployment was our first experience with end to end deployment of a production website so naturally, we ran into some challenges. The major challenges were with files, permissions, and paths. To begin, the production build of our piano roll used javascript source map files which were confusing at first. These files are used for debugging minified javascript in the web console, but we were unaware of this. To fix this we simply built the map file and placed it in the directory where static files are served from. Next, our application was working fine on our local environment, but the file upload wasn't working in the production environment and the user was facing a server 500 error instead of being redirected after uploading their midi file. To complicate matters, Nginx and Django were not logging the server error in the log files we configured. After lots of digging, it turned out that our logging errors were being buffered and not written to the files. We needed to set the environment variable:

```
export PYTHONUNBUFFERED=TRUE
```

After fixing this we were able to see the server error occurred when attempting to use the python Path library for opening and writing files. The Path library was not supported on Ubuntu like it was on our macOS local environments. We simply changed the file paths in python to use strings and the os package functions and everything worked fine.

### **13.3.4 Integration of IFrames**

One of the challenges we faced on the front-end was the integration of IFrames to appropriately display our in-browser piano roll, as well as manipulate individual components to alter the piano roll to serve all of our needs. In order to do this, a fair amount of time was spent analyzing the original code to determine how appropriately add more instruments, reformat the piano roll to fit in our page, and how to provide easy user interactions with the piano roll. A lot of trial and error was required in order to determine the most effective ways to do so, but in the end we were able to manage these IFrames effectively.

## Chapter 14

# References

### 14.1 Open Source Products

- Google Creative Lab Piano Roll
- react-piano
- SoundFontProvider

### 14.2 Research Paper

Kitahara, Tetsuro. 2017. "Music Generation Using Bayesian Networks." *Machine Learning and Knowledge Discovery in Databases Lecture Notes in Computer Science*, 368372. doi:10.1007/978-3-319-71273-4\_33.



# **Appendix A**

## **User Manual**

### **A.1 Access**

To access the web application:

1. Open a Chrome web browser (Note: The Original Beat is currently optimized only for Chrome web browsers).
2. Enter <https://www.theoriginalbeat.com> in the browser's search bar to access The Original Beat.

### **A.2 Accounts**

Upon accessing the site, users will be prompted to sign in to begin using the service. If user has not created an account with The Original Beat, they will be required to create an account first. Afterwards, they will be logged in and able to use the service.

### **A.3 Production**

Users will now be able to view the homepage after successfully logging in, where they will be presented with the option to either create their own melody using an in-browser piano roll, or upload their own melody using a midi upload feature.

### **A.3.1 Create Melody**

If user chooses to create their own melody:

1. User will click the "Piano Editor" button, which will present them with an on-screen piano.
2. Here they can either select notes with their mouse or enter in notes with their keyboard, which will be labeled on the piano itself. Notes will be recorded on the piano roll automatically as the user enters notes.
3. After the user is done creating their melody, they can hear a playback of their melody using the play button on the piano roll or select the save button in the upper right of the screen to save their melody.
4. After selecting the save button, the user will be presented with a pop-up card presenting them with their saved melody and a drop-down list with the harmonizing generation engines, allowing them to select which generation engine to use to process their melody when clicking the "Upload" button.

### **A.3.2 Upload Melody**

If user chooses to upload a midi file that they have saved on their computer:

1. User will click the "Choose File" button, which will allow them to select a midi file saved on their computer (Note: only files of type .midi will be accepted).
2. After selecting the midi file, user can select which engine to use with the "Generation Model" drop-down list, which is set to KeyChord by default.
3. Lastly, user will click the "Submit" button to start processing their melody with the selected generation engine.

### **A.3.3 Generated Layers**

Once the midi file is fully processed, user will be presented with a 3 piano rolls, one for the original melody, one for the generated harmony, and one for the generated drums.

The layers will be labeled accordingly and the user can play each layer individually or play all layers at the same time using the "Play All" button on the top of the page. If the piano rolls are not in sync, user can select the "Re-Sync" button to re-sync the piano rolls before selecting the "Play All" button.

Users can also use the drop-down list to the left of each piano roll to select the instrument sound for that layer. Users can ultimately download all of these layers as midi files using the "Download" button next to the piano rolls.