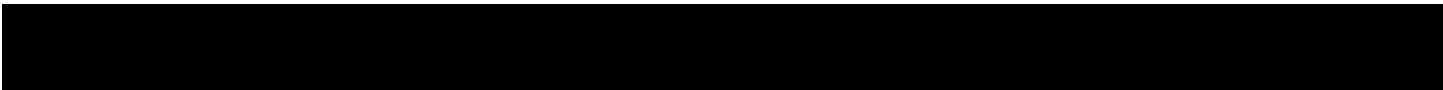


Building an AI-Powered Music Generation System for Ableton Live 11


- Building an AI-Powered Music Generation System for Ableton Live 11 1
- Chapter 1: Introduction..... 2
- Chapter 2: Background on AI Music Generation 4
 - 2.1 Symbolic vs. Audio Generation 4
 - 2.2 Key Developments and Projects in AI Music 5
- Chapter 3: System Overview and Design Requirements..... 7
 - 3.1 System Objectives and Feature Summary..... 7
 - 3.2 High-Level Architecture 8
- Chapter 4: Data Collection and Preparation 11
 - 4.1 Gathering and Organizing Your Music Data 11
 - 4.2 Data Cleaning and Preparation 13
 - 4.3 Labeling and Metadata for Conditional Generation 14
- Chapter 5: Representing Music for Machine Learning (Tokenization) 16
 - 5.1 MIDI Tokenization Strategies..... 17
 - 5.2 Tokenizing the Dataset 20
- Chapter 6: Model Architecture – Designing Your Music AI 24
 - 6.1 Choosing the Model Type..... 25
 - 6.2 Model Architecture Details 25
 - 6.3 Audio Model Integration 27
 - 6.4 Putting It Together – Model Summary..... 28
- Chapter 7: Training the Model..... 30
 - 7.1 Setting Up the Training Environment 30
 - 7.2 Data Pipeline for Training..... 31
 - 7.3 Monitoring Training 33
 - 7.5 Saving and Loading Models..... 34
- Chapter 8: Inference - Generating Music from the Trained Model 35



- 8.1 Inference Basics and Sampling Strategies..... 35
- 8.2 Prompting the Model 36
- 8.3 Generating MIDI Outputs 37
- 8.4 Generating Audio Outputs 39
- Chapter 9: Ableton Live Integration and Workflow 42
 - 9.1 Importing MIDI and Audio into Ableton..... 42
 - 9.2 Working with AI-Generated Material 43
 - 9.3 Using the AI in the Creative Process 44
 - 9.4 Example Workflow 44
 - 9.5 Limitations and Improving the Integration 45
- Chapter 10: Deployment and Scaling (Local vs Cloud)..... 46
 - 10.1 Local Environment Setup (Windows with WSL or Native) 46
 - 10.2 Cloud Training on AWS 48
 - 10.3 Collaborative or Distributed Training 49
 - 10.4 Deployment as a Tool 49
 - 10.5 Maintenance and Updating..... 50
- Chapter 11: Exercises and Next Steps 50
 - 11.1 Progressive Exercises Recap..... 50
 - 11.2 Extending the System 51
 - 11.3 Final Thoughts 52

Chapter 1: Introduction

Artificial Intelligence is reshaping the landscape of music production. Imagine a system that learns from your own musical style – your loops, samples, MIDI tracks, and full songs – and helps you create new beats and melodies on demand. In this guide, we will build exactly that: a comprehensive, AI-based music generation system that integrates seamlessly with Ableton Live 11. The system will be capable of training on your personal audio and MIDI data and generating original drum arrangements, melodic loops, or even full multitrack productions, which you can then import into Ableton as MIDI tracks or WAV audio stems.



Why AI for music? AI-driven music generation has evolved from simple algorithmic compositions to sophisticated models producing convincing, expressive music. Early approaches often generated music *symbolically* (as musical notation or MIDI), successfully creating polyphonic compositions but lacking the nuanced sounds of real audioopenai.com. Recent advances have enabled generating music in the *audio domain*, capturing timbre, texture, and even human vocalsopenai.com. Each approach has its advantages: symbolic generation is easier to train and can handle long musical structures, while raw audio generation captures the full richness of sound at the cost of higher complexityopenai.com. In this guide, we will leverage both domains – using symbolic (MIDI) techniques for composition and exploring audio techniques for rendering realistic sound – to get the best of both worlds.

What you will learn: This textbook-style guide is organized as a step-by-step journey, from fundamental concepts to a deployable system. We'll start with background on AI music generation and the specific goals of our system. Then we will delve into data preparation – how to curate and preprocess your audio/MIDI library for machine learning. From there, we'll design suitable model architectures (e.g. Transformer networks, autoencoders) that can accept various prompts like text descriptions, MIDI seeds, or audio references. We will walk through implementing the training pipeline in Python (using frameworks like PyTorch and Hugging Face Transformers), including dataset loading, tokenization, model training, and evaluation. Once the model is trained, we'll cover how to perform inference: generating new musical content from prompts with techniques like sampling and controlling output formats. Throughout, we will address practical considerations of running such a system on a local Windows machine (with Ableton) versus using cloud GPU resources (such as AWS). We'll also include examples of Jupyter notebooks, code listings (Python scripts, Dockerfiles), and automation tips to streamline the process. Finally, the guide includes progressive exercises at the end of chapters to reinforce your understanding and help you build a working AI music generation setup by the end of the book.

Who this guide is for: Our target audience is intermediate users of music production and machine learning, aiming to advance to expert level in ML-driven music creation. You should be somewhat comfortable with Ableton Live and have basic familiarity with Python. No deep learning expertise is assumed – we will introduce the necessary ML concepts along the way. By the end, you will be proficient in using modern AI techniques to produce music and equipped to push the system further with your own creativity.

Let's embark on this journey of merging your personal musical style with cutting-edge AI, turning your computer into a creative collaborator within Ableton Live!

Chapter 2: Background on AI Music Generation

Before diving into building the system, it's important to understand the background and key concepts of AI-based music generation. This chapter will cover the evolution of music AI, distinguish between symbolic and audio generation, and survey existing projects that inspire our system's design.

2.1 Symbolic vs. Audio Generation

In AI music research, there are two primary ways to represent music for generation:

- **Symbolic representation (MIDI/piano-roll):** Music is represented as a sequence of discrete events – notes with pitch, timing, velocity, and instrument information. This is analogous to a musical score or MIDI file. Models that operate on symbolic data can learn musical structure (melodies, harmonies, rhythm) abstracted away from the specific sounds. Working in this lower-dimensional space makes the task more tractable [cdn.openai.com](https://openai.com). Indeed, many early successes in AI music used symbolic approaches to generate convincing compositions in styles like Bach chorales or jazz solos openai.com. The limitation is that symbolic models don't produce the *sound* of music – they need to be rendered via instruments or synthesizers, and they may miss nuances like timbre and articulation openai.com. Symbolic generation also typically assumes a fixed set of instruments and doesn't directly include vocals or novel audio effects cdn.openai.com.
- **Audio representation (raw waveform or spectrogram):** Here, music is treated as audio signals – a sequence of samples or frequency spectra. This enables the model to generate the actual sound of music, including human voice, specific instrument timbres, and production qualities. The challenge is that audio sequences are extremely long and high-dimensional. For example, a 4-minute song at CD quality (44.1 kHz stereo) contains over 10 million samples openai.com. Modeling such long contexts with neural networks is difficult. Specialized techniques are needed to compress or chunk audio so that models can handle it. Despite the challenges, audio-generation models have made remarkable progress. They can capture nuances that symbolic models cannot – for instance, the expressiveness of a violin's tone or the reverberation of a room. We will explore strategies like autoencoders and transformers that make audio generation feasible by compressing audio into a sequence of learnable discrete codes openai.com.

Both approaches are relevant to our system. We will use **symbolic generation** to craft musical structures (drum patterns, melodies, chord progressions) in MIDI format, and then discuss options for turning those into **audio** either by using your own sample libraries or by employing an AI model to generate WAV outputs. Understanding this division will help us design an efficient workflow.

2.2 Key Developments and Projects in AI Music

AI music generation has rapidly evolved, drawing from advances in machine learning and vast amounts of music data. Here are some milestones and projects that provide context for our system:

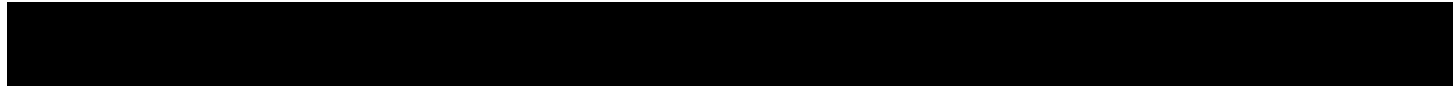
- **Google Magenta (2016→):** Google’s Magenta project has produced many open-source tools and models for music generation. Notably, **Magenta Studio** (released 2019) is a collection of plugins for Ableton Live that use trained AI models to assist in music creation magenta.tensorflow.org. The plugins include tools like *Generate* (which creates short 4-bar musical phrases using a Variational Autoencoder trained on a large MIDI dataset), *Continue* (which extends a given drum or melody clip), *Interpolate* (which blends between two musical ideas), *Groove* (which humanizes a drum beat’s timing/velocity based on learned patterns), and *Drumify* (which converts a simple rhythm guide into a full drum pattern) magenta.tensorflow.org. These tools show how AI can act as a collaborator in a DAW, generating or transforming MIDI clips that producers can then tweak. Magenta’s models (like MusicVAE, PerformanceRNN, etc.) mostly operate on symbolic data (MIDI) and require pre-training on large datasets, but they can be fine-tuned on your own data as well. We take inspiration from Magenta Studio’s capabilities – our system will aim to achieve similar tasks (generation, continuation, style transfer) but customized to your personal style and with a deeper integration into your workflow.
- **OpenAI MuseNet (2019):** MuseNet is a deep neural network that can generate multi-instrumental MIDI music up to 4 minutes long. It was based on a Transformer model (an extension of the GPT architecture) trained on a large corpus of MIDI from various genres openai.com. MuseNet could take a “prompt” of a few measures and continue it in the style of e.g. Mozart or The Beatles. It demonstrated that Transformers could handle long-term musical dependencies and produce coherent compositions with multiple instruments. While MuseNet itself is not open-sourced, it influenced many later projects. Our system will similarly use Transformer architectures to handle multitrack MIDI sequences, enabling complex arrangements.
- **OpenAI Jukebox (2020):** Jukebox took a different route – generating music in the audio domain. It introduced a *hierarchical VQ-VAE* (Vector Quantized Variational Autoencoder) to compress raw audio into a sequence of discrete codes, and then trained autoregressive transformers on those codes openai.com. By conditioning on metadata (artist, genre) and even lyrics text, Jukebox could generate full songs with singing, in the style of specific artists openai.com. For example, given the prompt “genre=rock, artist=Elvis, lyrics=...”, it outputs a new audio sample from scratch that resembles an Elvis song openai.com. The audio quality was lo-fi (due to heavy compression) and the model was enormous (multi-gigabyte and requiring powerful GPUs), but it was a breakthrough in raw audio generation. Jukebox’s approach of compressing audio to a lower rate sequence of discrete tokens is foundational – we will use a similar idea when discussing how

to generate WAV audio efficiently. However, given resource constraints, we will likely rely on more recent, lightweight methods than Jukebox’s full architecture.

- **Recent Advances (2021–2024):** The past few years have seen an explosion of improved models:
 - **Transformers for symbolic music** became state-of-the-art, with projects like *Music Transformer*, *Pop Music Transformer*, and *MT3* focusing on generating or transcribing music using attention mechanisms. Researchers developed better tokenization methods to represent music for transformers, such as REMI (RElative MI) and others (we’ll cover these in the next chapter) that allow efficient encoding of piano rolls with timing information.
 - **Diffusion models for audio:** Inspired by image generation advances, diffusion probabilistic models have been applied to music audio. Projects like *Dance Diffusion* and *Stable Audio* generate short clips by gradually refining noise into music, offering high audio quality. These models are starting to become available in open-source, though often they handle only short loops or single-instrument sounds.
 - **Meta’s MusicGen (2023):** Meta AI released MusicGen, a text-to-music generation model that is significantly more efficient than Jukebox. It uses a single-stage Transformer trained on codebooks from an audio tokenizer (EnCodec) to generate 32 kHz stereo audio conditioned on text (and optionally a melody)huggingface.co. Unlike earlier text-to-music work (e.g., Google’s MusicLM), MusicGen does not require an intermediate “semantic tokens” step; it maps text prompts directly to audio tokenshuggingface.co. Pre-trained MusicGen models are publicly available, and we will leverage them for our system’s audio generation component. For instance, we can fine-tune MusicGen on your own music to better match your style, or use it out-of-the-box to turn descriptive prompts into new sounds.
 - **Others:** Google’s *MusicLM* (2023) demonstrated high-quality text-conditioned music generation (though not released publicly), and projects like *Riffusion* showed that even image generation models can be repurposed to create music by generating spectrogram images from text and converting them to audio. While we won’t delve deeply into diffusion in this guide, it’s good to know these approaches as future avenues.

By understanding these projects, we gain insight into design choices for our system. In summary, we want to combine the **composition strength of symbolic models** with the **sound generation ability of audio models**. Our system will be unique in that it is trained on *your own data* – personalizing the generation to your style, which is something big general models don’t offer out-of-the-box. The next chapter will formalize the requirements and high-level design of our AI music generation system.

Exercise 2.1: Exploring AI Music Tools – To ground these concepts, try out a simple AI music generation tool available today. For example, install Magenta Studio (it’s free) and use the **Generate** plugin to create



a 4-bar drum beat magenta.tensorflow.org. Export the MIDI and load it in Ableton. How does the AI-generated result sound? Note down what was musically interesting and what was lacking. This experience will inform the features you want in your own system.

Chapter 3: System Overview and Design Requirements

In this chapter, we outline the system we aim to build, breaking down the requirements and sketching the overall architecture. This serves as a blueprint for the detailed implementation in later chapters.

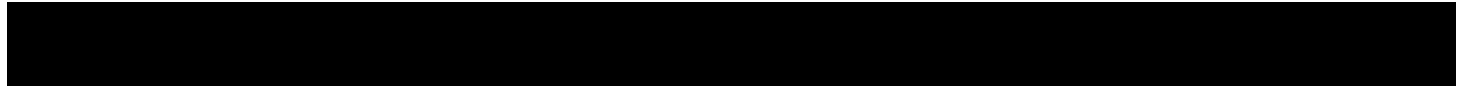
3.1 System Objectives and Feature Summary

Our AI music generation system will have the following core capabilities:

- **Training on Personalized Data:** The system can ingest your own library of musical data – including MIDI files (melodies, basslines, drum patterns), audio loops and one-shots, multitrack stem recordings, and even full tracks. By training on this data, the model will learn your style, preferred genres, and typical instrumentation. This is a custom training pipeline as opposed to using only pre-trained generic models.
- **Multimodal Prompting:** At generation time, the system accepts a variety of prompts to control what it creates. Possible prompts include:
 - *Text descriptions* – e.g. “A slow lo-fi hip-hop beat with a jazzy bassline”.
 - *MIDI seed* – e.g. a short MIDI clip (maybe a melody or drum pattern) that the model should continue or build upon.
 - *Genre/Style labels* – e.g. specifying *genre=House, style=Deep, mood=Uplifting* as tags.
 - *Audio clip* – e.g. a reference audio snippet whose vibe or groove should be emulated (this could be a drum loop, a chord progression recorded as audio, etc.).

The system will be designed to handle one or multiple of these prompts together. For example, you might give a text prompt *and* a drum loop, or you might just give a genre tag with no other input, and the model will generate something appropriate.

- **Output of Complete Musical Material:** The system will generate musically coherent outputs such as:
 - *Full drum arrangements* (kick, snare, hats, percussion on separate tracks).
 - *Melodic loops* (e.g. a 4-bar melody or chord progression).
 - *Basslines or harmonies* to accompany other material.
 - *Full multitrack beats or songs* – where multiple instrument parts (drums, bass, chords, melody) are generated in coordination.



Outputs can be in **MIDI format** (allowing you to edit notes or change instruments in Ableton) and/or **WAV audio format** (ready-to-use audio loops or stems). We aim for the output to be structured such that it can be easily imported into Ableton Live. For instance, a generated “full beat” could consist of several MIDI files (or audio stems) each corresponding to a different Ableton track (drum rack, bass instrument, synth, etc.).

- **Local and Cloud Execution:** The training and inference can run on a local machine – specifically a Windows 10/11 PC that you likely use for music production (which might have an NVIDIA GPU for CUDA acceleration) – or on cloud servers. Local training allows quick iteration with your setup (and offline use), while cloud training can provide access to more powerful GPUs (for faster training on large models or datasets). We will ensure the system is flexible to deploy in either environment:
 - On Windows, we will cover using **WSL2 (Windows Subsystem for Linux)** or Docker to set up a Linux environment for compatibility with machine learning frameworks. We’ll also note native Windows alternatives when available.
 - For cloud, we’ll discuss using AWS EC2 instances with GPU (like the p2, p3, or g4/g5 series) and how to transfer data and results between your local machine and the cloud.
- **Seamless Ableton Integration:** The ultimate goal is to make using the AI outputs in Ableton as smooth as possible. The generated MIDI or audio files should be organized and formatted for easy import. For example, if the system generates a drum beat as separate instrument tracks, it could output a folder containing Drums.mid, Bass.mid, Melody.mid etc., or corresponding WAV files. Ableton Live can directly read these: you can drag a MIDI file into Ableton and it will create a MIDI track with that clip, or drag a WAV loop in to use as audio. Our guide will also touch on best practices like tempo synchronization (ensuring the MIDI or loops align to a fixed tempo grid) so that when you drop them in Ableton, they immediately play in sync.

To make these concrete, consider a use-case: *You type a prompt: “Funky house beat at 120 BPM with a slap bass line.” The system, having been trained on your collection of house drum loops and funky bass MIDI clips, generates 16 bars of a drum pattern (kick, snare, hats) and a complementary bassline. It outputs a drums.mid and bass.mid (or audio stems) at 120 BPM. You import them into Ableton Live 11, where you have your favorite drum samples and bass synth ready. Instantly, you have the foundation of a new track which you can then modify, arrange, and mix further.* This is the kind of workflow we target.

3.2 High-Level Architecture

How will we build a system that meets these objectives? At a high level, our system will consist of the following components (illustrated in Figure 3.1):

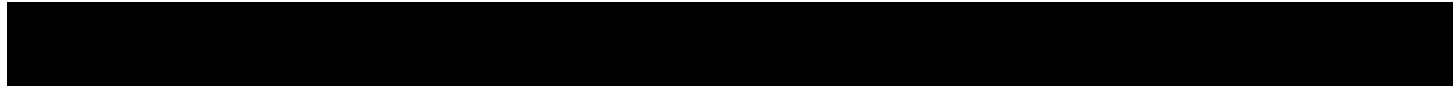


Figure 3.1: Converting a musical sequence into model-ready tokens. Here a short melody is encoded as tokens (NOTE_ON, TIME_DELTA, NOTE_OFF with specific values). This text-like encoding allows us to train language-model-style neural networks on musical data huggingface.co.

1. **Dataset Preparation Module:** A set of scripts/notebooks that take your raw data (MIDI files, audio files, metadata) and preprocess them into a structured dataset for training. This includes:
 - a. Reading MIDI files, quantizing if necessary, and converting them into a sequence of tokens (more on tokenization in Chapter 4).
 - b. Processing audio: either converting audio to a lower-dimensional representation (like discrete codec tokens or spectrograms) if we train an audio model, or extracting features if using audio as conditional input.
 - c. Organizing metadata like genre labels, track instrument labels, or any text descriptions you provide for each piece.
 - d. Splitting the data into training and validation sets.

The output of this module might be a folder or database that contains tokenized representations of music pieces (e.g. text files with tokens or a binary format) and associated labels.

2. **Training Module (Learning the Model):** This is where the machine learning model is trained on the prepared dataset. We will design a neural network architecture that can consume the tokenized music data (and conditions like genre or text). During training, the model will learn to predict the next events in a musical sequence given previous events (and given a prompt). We will likely utilize **Transformer architectures** for this, as they excel at sequence modeling and can handle the multi-track music data by appropriate tokenization huggingface.co. We might have one integrated model or multiple sub-models (for example, one model for MIDI composition and another for audio rendering). This module will also handle checkpointing (saving model weights), and possibly fine-tuning pre-trained models (like MusicGen) on your data. We'll implement training in PyTorch, using either plain PyTorch training loops or higher-level frameworks (PyTorch Lightning or Hugging Face Transformers Trainer) for convenience.
3. **Inference Module (Generation Pipeline):** After training, we need a procedure to use the model for generating new music. The inference pipeline will:
 - a. Accept user prompts (text, MIDI, etc.) and encode them in the same way training data was encoded (e.g. text prompts might be converted to tokens, MIDI seeds to tokens).
 - b. Feed the prompt into the trained model and autoregressively sample new tokens step by step to create a sequence of musical tokens. We will incorporate techniques like

temperature scaling and *top-k/nucleus sampling* to control randomness and creativity in generation [upwork.com](https://www.upwork.com).

- c. Decode the generated token sequence back into a usable format: either constructing a MIDI file from it, or decoding audio tokens to a WAV file (using the audio codec or vocoder).
- d. Post-process the outputs if needed (e.g., merging tracks, ensuring the length is a multiple of bars, etc.) and save the result files.

4. **Integration & Utility Module:** Around the core ML components, we will have utility scripts for things like:

- a. Converting generated MIDI to audio using synthesizers or sample libraries (if we choose to render MIDI to WAV automatically, or we can leave this step to the user in Ableton).
- b. A simple user interface (could be command-line or a minimal web or GUI interface) to input prompts and trigger generation.
- c. Packaging the model and inference code, potentially with a **Docker container** so that you can run it easily on different machines (for example, spin up a Docker container on AWS that has the model ready to generate via an API).
- d. Managing experiments and versions – e.g. keeping multiple trained models (you might train different models for different styles or purposes).

Behind these modules, there will be **storage** for datasets and models. If working locally, this might just be folders on your disk. In the cloud, you might use cloud storage (AWS S3 buckets or a data versioning tool) to keep the large files. We will ensure the guide covers how to handle these in both scenarios.

Design Considerations: A few important design choices we will face:

- *Unified model vs. multiple models:* We could train one Transformer that handles everything (multi-track MIDI and even audio tokens in one sequence). This unified approach is elegant but can be complex. Alternatively, we might train separate specialized models (e.g., one for MIDI composition, and if audio output is needed, a second stage model or existing tool to render audio). We will assess these options given the data available and resource constraints. A likely approach is to focus on a strong **symbolic model for composition** first, and use either existing audio generation models or simple rendering for the audio part.
- *Sequence length and structure:* Music sequences can be long. We need to decide how many bars or seconds of music the model will handle in one generation. We might train the model on relatively short sequences (say 8 or 16 bars of music) as examples, which can then be extended during inference by iterative generation or looping. This is related to tokenization and memory of the model.
- *Real-time vs. offline generation:* Our current scope is offline generation (you input a prompt and wait for the model to output a result). Real-time generation (like an interactive jam assistant) is

outside our scope for now, but the outputs should be fast enough to get results within perhaps a few seconds to a minute, depending on length. Using a GPU for inference will help.

By the end of this design phase, you should have a clear mental model of what we're building. In the next chapters, we will start executing on this plan, beginning with preparing your data for the AI to learn from.

Exercise 3.1: Map Your Use-Cases – List three specific scenarios where you want to use this AI system in your music production. For each, write down (a) what prompt you would give, (b) what output you expect (MIDI or audio, which instruments), and (c) how you would use that output in Ableton. For example: “Generate a lo-fi drum loop with vinyl crackle – Prompt: genre=lofi, mood=chill – Output: 8-bar drum loop audio – Use: import to Ableton, add as background percussion.” This will clarify your priorities (e.g., if all your use-cases involve drum loops, you'll know to focus the model on drums, etc.).

Chapter 4: Data Collection and Preparation

Garbage in, garbage out – the quality of our AI music system is directly tied to the quality of data we train it on. In this chapter, we'll focus on building a high-quality dataset from your own music files. We will discuss how to organize the data, annotate it with useful metadata (like genre or instrument tags), and preprocess it into a form suitable for our model (including tokenization of MIDI and possibly encoding of audio). This step is perhaps the most important in the whole process huggingface.co, so we will go into detail on best practices.

4.1 Gathering and Organizing Your Music Data

Start by aggregating all the music material that you want the AI to learn from. Typical sources might include:

- **MIDI files** – If you have projects in Ableton or other DAWs, export individual MIDI tracks or entire arrangements as MIDI files. You might have downloaded MIDI files or transcriptions of songs you like – those can be included if they represent the style you want (keeping in mind any right considerations if using commercial music). MIDI is extremely valuable since it explicitly shows the notes and rhythms for each instrument.
- **Loops and Samples (WAV/AIFF)** – These could be drum loops, instrument riffs, one-shot samples, etc. Consider categories: drum loops (by genre, tempo), bassline loops, melody loops, vocal samples, etc. Since raw audio is harder to model than MIDI, think about whether you want the AI to generate audio that mimics these loops, or if it will learn patterns to replicate in MIDI. For example, a library of drum loops could teach a MIDI-based model about common rhythmic patterns, even if you ultimately output new patterns as MIDI.

- **Stem recordings** – If you have multitrack stem exports of your songs (each instrument isolated), those are great for analysis. For instance, separate drum, bass, chords, and melody stems from complete songs allow the AI to learn how different parts work together. Even stereo mixes of full tracks can be useful to some extent (perhaps for an audio model to learn overall sound), but they're less structured than stems.
- **Miscellaneous** – You might also include things like chord progressions or lead sheets (in text or JSON form), or project files. However, since our training will revolve around MIDI and audio, these may need conversion (e.g., chord charts could be converted to a MIDI sequence of block chords).

Once collected, **organize the data** in a directory structure that will be convenient for processing. One approach is to separate by data type:

mathematica

Dataset/

```

├── MIDI/
│   ├── drums/
│   ├── bass/
│   ├── chords/
│   └── melody/
├── Audio/
│   ├── drum_loops/
│   ├── instrument_loops/
│   └── full_tracks/
└── Metadata/
    └── track_metadata.csv

```

You might also organize by song or session:

rust

Dataset/

```

└── Song1/
    ├── stems/ (audio stems)
    ├── midi/ (MIDI files for each stem)
    └── metadata.yaml (BPM, key, genre tags for Song1)

```

Choose an organization that makes sense to you. The key is to be able to iterate through the data systematically in a script. A **metadata file** (like a spreadsheet or CSV) can be very helpful. For example, you can have a `tracks.csv` with columns: `filename`, `type[MIDI/audio]`, `genre`, `bpm`, `instruments`. That allows you to programmatically use the genre or bpm as labels.

Tip: If your data is scattered across drives or projects, spend time upfront to consolidate and label it. This might involve exporting MIDI from Ableton projects (Ableton has an “Export MIDI Clip” function, or you can drag a MIDI clip to your desktop to get a `.mid` file). It might also involve normalizing names (e.g., ensure files have descriptive names like `FunkyBeat120bpm.mid` which can hint at tempo or style).

4.2 Data Cleaning and Preparation

Raw musical data often needs cleaning:

- **Trim and Align Loops:** Make sure loops are cut to exact bar lengths. For example, a drum loop that’s meant to be 4 bars at 120 BPM should be exactly 8 seconds long (if 4 bars of 4/4 at 120bpm). Trimming off any silence or extra tail ensures the AI doesn’t learn incorrect lengths. Similarly, align MIDI clips to start at beat 1 and end at a bar boundary if possible. Consistent loop lengths help in training (especially if we decide to chunk training data into fixed-length segments).
- **Tempo Handling:** Note the tempo (BPM) of each piece. In MIDI, tempos can be embedded. In audio loops, you might know the BPM from the name or metadata. We will need tempo info if we mix data of different tempi or if we want the model to condition on tempo. One strategy is to normalize all training data to a fixed tempo (say 120 BPM) by time-stretching audio and adjusting MIDI timing. However, this can alter the feel. Another strategy is to include tempo as part of the input features (e.g., adding a token like `TEMPO=120` at the start of a sequence huggingface.co). We’ll revisit how to handle tempo in tokenization. For now, ensure you at least record the original tempo of each piece in the metadata.
- **Key and Transposition:** If your music spans different keys, consider transposing MIDI files to a common key (or at least a smaller set of keys). For example, many datasets transpose everything to C major/A minor (the “white keys”) to reduce complexity, since the model can then learn patterns independent of absolute key and you can always transpose the output later. This isn’t mandatory, but if your dataset is small, normalizing keys can help the model focus on patterns of intervals and harmony. You might do this for melodic content but not for drums (drums have no pitch). Keep track if you transpose, so you know how to invert the transposition on output if needed.
- **Quantization vs Human Feel:** MIDI data might have imprecise timing if it was performed live (not strictly on grid). You have a choice: quantize it (lock notes to a grid, e.g., 16th notes) or keep the human imperfections. Strict quantization makes the data simpler and might be easier for the

model to learn patterns (especially drum patterns). However, it loses the “groove”. An alternative: keep them as is, and even introduce “swing” or timing tokens in the representation (some tokenizations support micro-timing). As a compromise, you could quantize most material but perhaps keep a slight swing on genres that require it (like a slight shuffle in jazz). For our initial model, we’ll assume mostly quantized data, and we’ll incorporate groove/timing as needed in a simpler way (e.g., the model could output slightly offset MIDI if we include time-shift tokens).

- **Splitting by Sections:** Full songs might be too long to feed entirely to the model. It’s common to split songs into segments (e.g., 8-bar or 16-bar segments) and treat those as independent training examples huggingface.co. For instance, a 3-minute track could be split into, say, 6 segments of 30 seconds each. When splitting, ensure segments make musical sense (split on bar boundaries). If a song has distinct sections (verse, chorus, etc.), you might want to label segments accordingly or at least ensure variety. Later we can allow the model to generate longer pieces by stringing together segments or via iterative generation.

After cleaning, you should have a well-structured set of MIDI and audio that’s consistent in format:

- All MIDI files perhaps using the same resolution (ticks per beat) and type (we might convert Type 1 MIDI files, which have multiple tracks, into multiple single-track files or a merged multi-track representation).
- All audio files in a consistent format: e.g., WAV format, 16-bit or 24-bit, 44.1 kHz (or 32kHz if we plan to use MusicGen’s default, which is 32kHz). Consistent stereo/mono (mono might be fine for many things like drum loops; stereo if spatial info is important).
- No extraneous files or noise.

Exercise 4.1: Curate Your Dataset – Create a spreadsheet listing at least 20 pieces of data you will include. For each, note: file name, length (bars or time), tempo, key (if applicable), genre/style, and instruments present. Review this list critically – is each piece representative of what you want the model to learn? Remove anything that you wouldn’t want the model to imitate (for example, if there’s a track with a lot of mistakes or noise). If your list is short, consider augmenting it with external data in the public domain (e.g., MIDI files from GNU MIDI collection, or drum loops from royalty-free packs) that align with your style.

4.3 Labeling and Metadata for Conditional Generation

To enable the model to generate music based on prompts like “genre” or “mood,” you should label your training data with those attributes. This way, we can teach the model via examples – e.g., all reggae files have the label “GENRE=reggae” so later if we prompt with that label, it will tend to produce reggae-like output.

Useful metadata to consider:

- **Genre:** e.g., hip-hop, techno, rock, classical, etc. This can be a broad category or even multiple labels if your music blends genres. You might label each piece manually. If you have the artist or album info, you could infer genre (as one tutorial did by using the Spotify API to get genre tags for artists huggingface.co). In our case, since it's your own music, use your knowledge of your tracks. Keep genre labels somewhat general unless you have enough data to support many fine-grained ones.
- **Style/Modifiers:** Sometimes within a genre, you have variations like “acoustic”, “synthwave”, “lo-fi”, “uplifting”, “dark”, etc. These could be considered as additional tags (you might call them “style” or “mood”). For example, Genre: House, Style: Deep. Or Genre: Hip-hop, Mood: Chill. Use any descriptors you find relevant. The more consistent you are, the better (don't have 50 different adjectives each used only once – focus on a set of key styles that repeat across songs).
- **Instrument roles:** For multi-instrument data, it's useful to label which instrument a track is (drums, bass, lead, chords, pad, etc.). If your MIDI files are single-instrument, the filename or folder might indicate this. For audio stems, similarly, know which stem is which. We will likely encode instrument information in the token representation (like INST=37 meaning instrument #37, or a token “<Drums>” to indicate we're in a drum track) huggingface.co.
- **Tempo and Key:** As discussed, these can be important. We might include them as special tokens (e.g., TEMPO=120 at start). Key could be included (like KEY=C_minor), though models can possibly infer key from the notes, an explicit token might help if you want to constrain generation to a key.
- **Lyrics or other data:** If any of your songs have lyrics and you want to incorporate lyric-conditioned generation (like Jukebox did with lyrics), that's a whole additional modality. It's probably beyond our scope unless you have a lot of paired lyrics. We will likely skip this, focusing on instrumental music. But theoretically you could supply lyric text as another input channel.

Create a **metadata file** (CSV or JSON) that contains a record for each training item with all these labels. For example:

CSS

track_name, type, lengthBars, tempo, key, genre, style, instruments

Beat001, midi, 8, 120, N/A, hip_hop, boom_bap, [drums]

Song002, stems, 16, 140, A_minor, rock, aggressive, [drums, bass, guitar, vocal]

Loop003, audio, 4, 90, D_minor, reggae, mellow, [drums, guitar_skank]

This will guide how we create input tokens. For instance, a MIDI track labeled as hip_hop and aggressive might be converted into a token sequence starting with `PIECE_START GENRE=hip_hop MOOD=aggressive TEMPO=90 KEY=D_minor` etc., followed by the notes huggingface.co.

4.4 Tools for Data Processing

To effectively prepare data:

- Use Python libraries like **mido** or **pretty_midi** to read and write MIDI files. They allow you to iterate over notes, get tempos, etc. We might use these in a preprocessing script to convert MIDI to our token format.
- **librosa** or **torchaudio** for audio processing – to load audio files, resample them, do time-stretch if needed, etc.
- **pandas** or simply Python CSV for handling metadata spreadsheets.
- If you have a lot of data and need to do batch processing (like transpose all MIDI to C, or time-stretch all loops to 120 BPM), consider writing small scripts to automate this offline before training.
- Check out open-source libraries like **MidiTok** huggingface.co, which implements many music tokenization methods and might save you time in converting MIDI to sequences. We will explain the concepts enough that you can either use such a library or implement a custom tokenizer.

Now that we have our data collected, cleaned, and labeled, we're ready for one of the most crucial steps: turning this musical data into a form our model can understand (i.e., tokenization), which we will tackle in the next chapter.

Exercise 4.2: Preprocessing in Action – Write a short Python script (or use a Jupyter notebook) to perform a simple preprocessing task on one of your files. For example, use `pretty_midi` to load a MIDI file, print out the instruments and notes, and then transpose all notes down by 2 semitones and save the result as a new MIDI file. Or, load an audio loop with `librosa`, trim silence at the ends, and save it. This will both test that your files are accessible and get you familiar with the libraries we'll use.

Chapter 5: Representing Music for Machine Learning (Tokenization)

Now that our dataset is organized, we need to translate the musical data (especially MIDI) into a sequence of tokens that our model can learn. This process is analogous to converting text into words or subword tokens for NLP models. The design of this token representation is critical: it needs to capture all the musical information (notes, timing, instruments, etc.) in a linear sequence of tokens. A good

tokenization will make the learning task easier for the model, while a poor one could make it impossible for the model to understand the structure.

5.1 MIDI Tokenization Strategies

Representing a single-track MIDI performance as tokens can be done in various ways. Researchers have developed several tokenization methodshuggingface.co:

- **MIDI-like (Event-Based):** This straightforward approach treats MIDI events as a stream. For example, a NOTE_ON event, a NOTE_OFF event, and time delays between events are all separate tokens. We might have tokens like NOTE_ON=60 (meaning note middle C on), NOTE_OFF=60 (note off), and something to represent waiting or advancing time. One way to handle time is using a special token like TIME_DELTA= n where n might represent some ticks or a 16th-note step. This is illustrated in **Figure 5.1** (which showed a sequence of notes with specific time steps) – e.g., NOTE_ON=64 TIME_DELTA=4 NOTE_OFF=64 to indicate a note was held for 4 ticks (or some unit)huggingface.co. MIDI-like tokenization can also include tokens for other MIDI controls (tempo changes, control changes) if needed. It's simple but can lead to long sequences, especially if every tick or small time unit is a token.
- **REMI (RElative Music International):** REMI is a tokenization introduced to better handle meter and tempohuggingface.co. It uses tokens for *Bar Start* and *Beat* positions, allowing the model to understand musical measures. It also explicitly includes chords and tempo tokens. The idea is to have a high-level structure (bars) and then within each bar, events relative to the beat. REMI tokens might look like: <BAR> <Position 0> Note=60 Duration=4 Velocity=... <Position 48> Note=62 ... etc. We may not need to implement REMI fully, but we can borrow the idea of bar and position markers to give structure.
- **CP Word (Compound Word) and Others:** Some methods group multiple attributes into one token. For example, CP Word might combine (Note, Velocity, Duration) into one token to reduce sequence lengthhuggingface.co. This requires a fixed vocabulary of combinations and is a bit like treating a chord or multi-note event as one token.
- **MMM (Multi-track Music Machine):** Since we have multi-instrument music, MMM is relevant. MMM extends tokenization to multiple tracks by introducing tokens that mark track boundaries and instrument assignmentshuggingface.co. As shown earlier, it wraps note sequences within <TRACK_START> and <TRACK_END> tokens, and uses tokens like INST=30 to denote what instrument (30 might be Overdriven Guitar in the MIDI program standard) the track ishuggingface.co. It also has <BAR_START> and <BAR_END> to wrap bars, and can allow multiple tracks concurrently by how the sequence is orderedhuggingface.co. Essentially, an entire song with multiple tracks becomes one long token sequence that interleaves tracks, but structured with these delimiters.

Choosing a tokenization is a trade-off between completeness and conciseness. For our project:

- We want to include drums, bass, etc., in one sequence. We can use an approach similar to MMM. We might not need the full complexity of MMM (which also had some advanced features like bar “fill” for incomplete bars), but at least include tokens for track separation and instrument labels.
- We want the model to understand bar boundaries, so including a bar token every fixed number of beats is wise.
- We should include tempo as a token if it varies or if we have multiple tempos. If all training data is at similar tempo or we decided to fix tempo, we might skip it. But to be safe, let’s plan to include a TEMPO=X token near the start of sequences huggingface.co.
- We should include any meta tags (genre, etc.) as tokens at the start of the sequence (after a PIECE_START token) huggingface.co. For example: PIECE_START GENRE=hip_hop MOOD=chill TEMPO=90 KEY=D_minor.

Let’s outline a custom token format for our needs:

- Start and end tokens: PIECE_START and PIECE_END to denote a complete piece (or segment).
- Metadata tokens immediately after start: e.g. GENRE=hip_hop, STYLE=lofi, TEMPO=90, KEY=C_minor. (If any of those are not applicable to all, we can include or omit them as needed. The model can learn to optionally condition on them.)
- Then, for each track/instrument:
 - TRACK_START INST=drums (we can use instrument names or a consistent label like “drums”, “bass”, since it’s your own data you can define these; or use General MIDI instrument numbers if convenient).
 - Within a track:
 - BAR_START at the start of each bar.
 - Musical events inside the bar in chronological order:
 - We can use time step tokens to move the time forward. One approach: quantize everything to 16th notes (or whatever smallest subdivision you need) and use a TIME_STEP token repeatedly. For example, a half note rest in 4/4 could be represented by 8 consecutive TIME_STEP tokens if one TIME_STEP = a 16th note. This can lead to many tokens if there are long gaps. Alternatively, use a TIME_DELTA=n token that directly jumps n ticks. Many implementations use a maximum value (say 16 or 32) for time delta to keep vocabulary limited, and if longer gaps needed, just multiple tokens or bars.
 - Notes: We will have NOTE_ON=pitch and NOTE_OFF=pitch tokens as needed. Optionally, we might not use explicit NOTE_OFF if we use durations. Another method: Use NOTE_ON=pitch and a separate DURATION=d token

right after to specify how long it lasts. That's another style (some tokenizers do that). If using NOTE_OFF, the advantage is explicit, but it doubles the events. Using DURATION can be more compact: e.g., NOTE=60 D=4 to mean note 60 with duration 4 (in some unit).

- Velocity: We might skip explicit velocity tokens to keep it simple, or incorporate it as part of NOTE token (some do like NOTE_ON=60:vel127). For drums, velocity can matter (ghost notes vs accented), and for melody expression too. But including velocity increases vocab and sequence length. Given an intermediate-level audience, we might decide to include velocity in a limited way or not at first iteration.
 - BAR_END at end of bar (or the next BAR_START implies end of previous).
 - TRACK_END after finishing a track's token sequence.
 - Then possibly another TRACK_START for next instrument, etc., until all tracks are done.
 - Finally PIECE_END.

This is similar to MMM's described format huggingface.co. In the MMM example snippet given in Chapter 2, they had actual pitch numbers and instrument numbers in tokens and grouped in that way.

We will implement something along these lines in code. One important point: *How to order the tracks in the sequence?* Typically, one could put tracks one after the other (serializing the multi-track). This means the model generates an entire track's events, then moves to next track. However, that loses synchronization between tracks (the model might not know that track 2 is supposed to align rhythmically with track 1). MMM addressed this by an interleaved approach called "Bar Fill" where they alternate tracks by bars: e.g., all tracks' bar1 events, then all tracks' bar2 events 【24†】. But implementing that is more complicated.

A simpler compromise: If we always give a drum track first, the model could implicitly time other tracks to it (drums give a reference grid). But that's not guaranteed. Given complexity, we might choose to focus on generating one track at a time with context of others, or generate everything sequentially and then align offline.

A workable strategy: train the model on combined sequences but maybe bias it with a positional encoding that resets each bar, etc. For now, let's not overcomplicate – we'll serialize by track (drums then bass then melody, etc.). The model can still learn correlations because earlier track events are in its attention history when generating later tracks, but it might not perfectly align them without explicit bar tokens. Our inclusion of bar tokens and possibly time resets at bar boundaries will help maintain an alignment.

5.2 Tokenizing the Dataset

With the rules set, we now actually convert our dataset:

- For each piece (MIDI or stems):
 - Parse the MIDI file (or if you have stems' MIDI, use those; if only audio stems without MIDI, we might skip purely audio pieces for now or only use them in an audio model separately).
 - Determine the metadata (genre, etc.) and start forming the token list: [PIECE_START, GENRE=X, ... TEMPO=Y, KEY=Z].
 - For each track in the piece:
 - Identify instrument. For drums, we might label as “DRUMS” instrument. For melodic, if the MIDI uses a program number or track name, map it to a token. Perhaps define a small mapping: e.g., if track name contains “Bass”, use INST=BASS token; if “Piano” use INST=PIANO, etc. Uniformity helps (maybe only a handful of instrument tokens that cover your common instrumentation).
 - Append TRACK_START and INST=instrument tokens.
 - Split the MIDI events by bar. You can use the time signature (assume 4/4 for simplicity unless your music has others, then you'd need to include time signature tokens too). If 4/4, one bar = ticks_per_beat * 4 ticks in MIDI. Many MIDI files have 480 ticks/beat, so a bar is 1920 ticks. You could iterate events and whenever the time exceeds the bar, insert a BAR_END and then BAR_START.
 - Inside a bar, add events:
 - Sort events by time. For each event:
 - If it's a Note On, add NOTE_ON=pitch. Immediately after, if representing duration explicitly, add DURATION=d (d in, say, 16th note units). If not using duration tokens, you'll later also process the Note Off events; in that case, also include NOTE_OFF=pitch when encountered.
 - If using time step tokens: you need to add something to move time forward between events. One approach: keep a current_time counter. For the next event, compute delta = (next_event_time - current_time). If delta > 0, convert that into one or multiple time tokens. E.g., if delta = 120 ticks (and we decide 480 ticks = 16 time units of 30 ticks each, just as an example), you could output some combination to sum up (like TIME_DELTA=30 repeated 4 times). MidiTok's [table huggingface.co](https://github.com/tylertomlinson/miditok) shows each method's capability for rest, etc. Simpler: if we quantize to 16th note (120 ticks if 480 per beat), then

delta=120 ticks = one token, say TIME_DELTA=1 meaning one 16th. If delta=240 (an eighth note), that would be two TIME_DELTA=1 in a row, or we allow TIME_DELTA=2 meaning two 16ths. We can define that as we like.

- We also might want a BAR_START at the beginning of each bar inside the track. Possibly also at the very start of track output (before first bar's events).
- After all events in all bars are processed, put a TRACK_END.
 - After all tracks, append PIECE_END.

This results in one sequence. If your piece is long and you're splitting by segments, you might do the above for each segment (with perhaps segment-level metadata).

The output of tokenization can be stored as a list of strings (each token is a string like "NOTE_ON=60"). You can then map these to integers by building a vocabulary (a dictionary mapping token -> ID). Common tokens like NOTE_ON=60 (pitch 60) mean you need tokens for each pitch you use, each instrument label, etc. We'll generate the vocab from the data.

Vocabulary size considerations: MIDI pitches 0-127 -> 128 possible NOTE_ON tokens (and similarly NOTE_OFF if separate). If velocity considered, multiply that in. Instruments: maybe 10-20 types. Time steps: depends on resolution – if we allow TIME_DELTA=n for n=1 to 16 (for 16 16th-notes max rest, etc.), that's 16 tokens. So overall, likely a few hundred tokens, which is fine. If we incorporate every possible combination event as unique token (like CP Word does), could be bigger, but we'll keep it moderate.

Remember to include special tokens in vocab like PIECE_START, etc.

We may also tokenize **audio data** if we plan to train an audio model. Audio tokenization is different: typically one uses an audio *codec* to convert waveform to tokens (e.g., EnCodec, as used by MusicGen huggingface.co, or a VQ-VAE). Covering how to implement a codec from scratch is heavy, but if we plan to fine-tune MusicGen, we can use their pre-trained EnCodec which outputs 50 tokens per second per codebook (MusicGen uses 4 codebooks sampled in parallel) huggingface.co. Fine-tuning MusicGen would involve feeding in those tokens and training the transformer. Instead of doing all that manually, one approach is to use the audiocraft library to handle audio tokenization and just provide it with audio and prompts.

Given the scope, we'll likely focus our code on symbolic tokenization and use MusicGen for audio, so we won't dive deeply into manually tokenizing audio in this guide. However, a brief mention:

- If you wanted to include audio loops in training without a pre-trained model, you could use a technique like training a VQ-VAE on your audio first. That compresses audio into discrete codes

(like Jukebox did with 3 levels openai.com, or simpler one-level VQ-VAE with a certain codebook size). Each code (embedding index) becomes a token. You would then train a model on sequences of those tokens. But doing this from scratch requires a lot of data and tuning. Instead, we leverage existing ones (like EnCodec from Facebook).

After tokenization, you should convert tokens to numbers and possibly save them in a file or database that the training code can load efficiently. Some choose to save each sequence as a line in a text file (since our tokens are texty, that's easy). Others use numpy arrays or Torch tensors.

For example, you might create data/tokens.txt where each line is a space-separated token sequence for a piece. Or since we might have thousands of tokens per piece, storing as compressed numpy might be better.

5.3 Example of Tokenization

Let's walk through a simple example to illustrate:

Suppose we have a 1-bar drum loop (4/4) with a kick on beats 1 and 3, snare on 2 and 4, and hihats every 8th note. In a grid:

- Kick: at beats 1 and 3 (positions 0 and 2 in 0,1,2,3 for quarter-note indices).
- Snare: at beats 2 and 4 (positions 1 and 3).
- Hi-hat: 8th notes means on 1&,2&,3&,4& (positions 0.5, 1.5, 2.5, 3.5 in quarter units, or in 16th note indexing: positions 0,2,4,6 out of 8 sixteenth per bar).

We label this as genre=hip_hop, instrument track=drums.

Token sequence could be:

makefile

```
PIECE_START GENRE=hip_hop TRACK_START INST=DRUMS BAR_START
NOTE_ON=KICK TIME_DELTA=2 NOTE_ON=SNARE TIME_DELTA=2 NOTE_ON=KICK TIME_DELTA=2
NOTE_ON=SNARE TIME_DELTA=?
NOTE_ON=HIHAT ... etc ... BAR_END TRACK_END PIECE_END
```

This is pseudo because we need to correctly intermix them timewise. Perhaps better, break by 16th:

- 16th 0: Kick + Hat
- 16th 1: (hat already covered as it rings? If using Note_on/off, hat might have short duration)
- 16th 2: Snare + Hat

- 16th 4: Kick + Hat
- 16th 6: Snare + Hat

This gets detailed; representing polyphonic events at exactly same time might require either sorting by event type or issuing them sequentially with zero time delta in between. Usually, one would output all events at a given time before moving time forward. So multiple NOTE_ONs in a row then a time shift.

So at 0: output NOTE_ON=KICK NOTE_ON=HIHAT (the order can be fixed e.g. kicks before hats or by pitch order) then TIME_DELTA=1 (to move one 16th).

At 1: no new events? (if hat closed or continuous? Let's assume hats play every 8th so nothing at the off-16th except the hat from previous still ringing, which we ignore here as new events).

TIME_DELTA=1 move to 2.

At 2: output NOTE_ON=SNARE NOTE_ON=HIHAT, then TIME_DELTA=1.

At 3: move to 4 (TIME_DELTA=1).

At 4: NOTE_ON=KICK NOTE_ON=HIHAT, then TIME_DELTA=1.

At 5: move to 6 (TIME_DELTA=1).

At 6: NOTE_ON=SNARE NOTE_ON=HIHAT, then TIME_DELTA=1 (to reach end of bar position 7).

At 7: TIME_DELTA=1 (to reach end of bar at 8? Actually if we had 8 sixteenths, index 7 is last event position, then bar_end).

Then BAR_END.

This shows how granular it can get. It might yield many TIME_DELTA=1 tokens; an alternative is combine them: e.g., TIME_DELTA=2 to jump two 16ths at once, etc. We can decide that our TIME_DELTA tokens represent multiples of a 16th to shorten sequences. For example, define that TIME_DELTA=4 means move forward 4 sixteenths (i.e., one beat). Then Kick at beat1, next Snare at beat2: we could encode Kick then TIME_DELTA (for one beat) then Snare. Actually, one can incorporate a Beat token or Bar token which resets time count every bar as well.

Given the complexity, using an existing tokenizer is tempting. **MidiTok** library supports many of these out-of-the-box huggingface.co. For instance, we could use MidiTok's MMM or REMI implementation to convert all MIDI to token sequences, saving us from reinventing the wheel.

However, doing at least one manually helps understand.

5.4 Handling Audio Data (Brief)

If you have audio loops that you want the model to learn to generate *in audio form*, you will need to decide whether to:

- Convert them to some time-frequency representation like a Mel-spectrogram and train a model on that (like treating it as an image generation problem, which some have done with Spectrogram Transformers).
- Or use a pretrained audio tokenizer. We will likely use the latter (EnCodec tokenizer from MusicGen). The details of that will be handled by the MusicGen model rather than us coding it. Essentially, EnCodec will convert an audio waveform into a sequence of discrete codec codes (at 50 Hz). If using MusicGen for fine-tuning, you provide pairs of (audio token sequence, conditioning prompt) to train it. We won't manually tokenize audio to text here.

If you want to incorporate audio as *input prompt* (e.g., give a melody audio clip and generate accompaniment), an approach is to extract some features from the audio prompt (like a latent embedding from a model or a mel-spectrogram as input to the transformer). Designing that is advanced; MusicGen handles melody conditioning by actually feeding the melody audio through EnCodec and then as a sequence to condition the generation of the new piece huggingface.co. We might skip deep integration of audio prompts and instead allow something simpler (like providing a beat and expecting the model to detect its tempo or something – that might be done outside of the model for now).

At this point, we have defined how to represent and tokenize our data. In the next chapter, we will set up the model architecture that will be trained on these token sequences to learn to generate new music.

Exercise 5.1: Tokenize a MIDI file – Using either your own script or a library like MidiTok, convert a simple MIDI file (like a one-track melody) into a token list. Print out the token sequence. Does it make sense to you? (For example, do you see note numbers and durations in a logical order?) This will give you a concrete feel for what the model will actually ingest. If using MidiTok, you can try their REMI or MMM tokenization and inspect the output.

Chapter 6: Model Architecture – Designing Your Music AI

With data in tokenized form, we turn to designing the machine learning model that will learn from this data. We need an architecture that can handle long sequences, understand the structure in our token sequences, and produce coherent outputs conditioned on prompts. Given the state of the field and our requirements, a Transformer-based architecture is a natural choice cdn.openai.com. In this chapter, we'll outline the model's components, discuss how we incorporate conditional inputs (like genre tags or prompt tokens), and consider any modifications needed for music data (versus plain text).

6.1 Choosing the Model Type

Why Transformer? Transformers have become the dominant architecture for sequence generation (in NLP, vision, and also music) because of their ability to capture long-range dependencies with self-attention. Music, especially when represented in a linear token sequence (as we have), is essentially a sequence problem like language. There are repeating motifs, structures (verse/chorus or bar patterns), and long-term dependencies (e.g., a motif that reappears after 16 bars). Recurrent Neural Networks (RNNs) like LSTMs were used in earlier music models, but they struggle with very long sequences and can be harder to train. Transformers can look at the entire sequence context when generating each token, which is great for consistency in music.

We will implement a **Transformer Decoder** model (similar to GPT-style) which is trained to predict the next token given all previous tokens in the sequence. Essentially, it treats our music token sequence as a language to model. This model will be an **autoregressive language model** for music. When we generate, we will feed in a prompt (some starting tokens) and have it continue token by token.

Model size considerations: Depending on your dataset size and compute, the transformer can be relatively small (say, 6 layers with 8 attention heads and an embedding size of 512) or larger (e.g., 12 layers, 12 heads, embedding 768 or more). Larger models can capture more complexity but require more data and time to train. Since we are working with possibly a limited personal dataset, we should not go too large to avoid overfitting. We might start with ~GPT-2 small sized model (around 100 million parameters or less) or even smaller if data is very small. One thing in our favor is that music sequences can be shorter (if we limit to, say, 16 bars at a time, that might be a few hundred or couple thousand tokens) compared to text which can be longer; so we might not need an extremely large context length or model.

6.2 Model Architecture Details

The core components of our Transformer model:

- **Embedding Layer:** We will have an embedding matrix that maps each token ID in our vocabulary to a vector (embedding). This could be size 256 or 512 for example. We will also likely have a **positional encoding** mechanism, since transformers are order-aware via position encodings. We can use the standard learned positional embeddings or sinusoidal. However, music has some periodic structure (bars, beats). Some research uses relative position encodings or specific music-aware encodings. To keep it simple, we might just use a basic positional encoding for the absolute token index in the sequence. Given we have tokens like BAR_START, the model can also

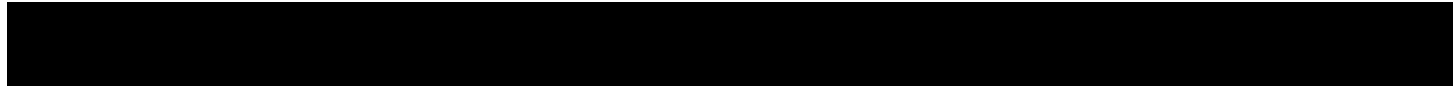
learn positions from content. But absolute position helps differentiate first bar vs later bars if needed.

- **Self-Attention Layers:** Each transformer layer has a multi-head self-attention sublayer. This allows the model to attend to all previous tokens when predicting the next (thanks to causal masking). So when the model is generating, at a note in bar 4 it can attend back to notes in bar 1 if needed, which is great for repeating themes or ensuring consistency with earlier chords, etc.
- **Feed-Forward Layers:** After attention, each layer has a position-wise feed-forward network (a couple of linear layers with ReLU or GELU) to process the attended information.
- **Decoder Stack:** We'll have N such layers stacked.
- **Output Projection:** Finally, a linear layer projects the transformer output to logits over the vocabulary, and softmax gives a probability distribution for the next token.

This is a standard language model architecture. If using a framework like Hugging Face Transformers, we could simply use their GPT2LMHeadModel or TransformerDecoder classes to define this, specifying the vocab size and number of layers.

Conditional inputs handling: How do we feed in prompts like text or genre? In our tokenization, we actually inserted genre and style as special tokens at the start of the sequence. That means the model can learn to associate those with what follows (just as GPT can learn a prompt at the start). For example, many sequences might start with GENRE=jazz and then contain swing rhythms; the model will pick up that pattern. So at generation time, we can just include that token to steer it. This is a simple yet effective conditioning method huggingface.co. We don't necessarily need a complex multi-modal architecture unless we wanted to feed raw audio or an actual separate text description (beyond just a categorical label). If we had arbitrary text prompts (like "A happy song with piano"), we could tokenize that text with a language model and somehow fuse it. But that's complex. Instead, perhaps we limit "text prompt" to a fixed set of tags (genre/mood/instrument), which we already handle as tokens.

If we wanted to condition on an audio or MIDI seed: The way to do that is simply to prepend those notes as part of the sequence. For example, if you want the model to continue a given drum pattern, you would start the generation with PIECE_START GENRE=hip_hop ... TRACK_START INST=DRUMS BAR_START ... (some sequence of note tokens for say 2 bars) ... and then let the model continue from there with new tokens. Training the model to handle this means in training we should include cases where sequences have a beginning and the model continues. But our training data usually are full sequences. One trick is to train the model on next-token prediction for full sequences, which inherently means it learns to continue from any intermediate point. We can further train it on *prefixes* (taking the first part of sequence as input and still predicting next tokens beyond what was in input) if we wanted to emphasize continuation, but it's generally not needed.



So, we might not need to architecturally distinguish between prompt tokens and generated tokens – as long as we feed them in properly during generation.

Multi-track coherence: Our model sees the multi-track sequence as one long stream. If we ordered tracks sequentially, it might not perfectly align them since, for example, by the time it generates the bass track tokens (which come after drum track tokens in sequence), it already has the drum track in context. That's good – it can condition the bass on the drums it saw. But when generating, we likely will generate track by track in that fixed order. This could actually be powerful: generate drum track first, then bass, etc., in one pass, with the model internally using previously generated tracks as context for later ones. That means if we want a different ordering (like maybe allow bass first), we'd have to either retrain or generate differently. It might be wise to always follow a consistent order of tracks in the training data (say: drums, then bass, then chords, then melody) so that the model expects that order. We should document and stick to that convention.

Another approach is to generate all tracks interleaved by time (like bar-by-bar). That is more complex and we didn't fully implement that in tokens. So we'll stick to ordered tracks for now.

Handling long sequences / memory: If our sequences (e.g. 16 bars of multi-track music) result in, say, 500 tokens, a small transformer can handle that easily. If we tried to do entire songs of 100+ bars (thousands of tokens), we might need to consider memory or using approaches like longformer or compressing structure. But presumably we'll train on shorter segments.

Special considerations:

- We should probably use *masking* during training if needed to ensure the model doesn't look at future tokens. In a causal LM like GPT, that's standard.
- We might tie input and output embeddings (weight tying) to reduce parameters since vocab might be large-ish (though it's often done in language models).
- We might consider *relative positional encoding* (where the model learns relative distances rather than absolute positions) since a piece could be placed anywhere in timeline and we want things like "this note is 4 beats after the previous bar" to matter more than absolute index. But for simplicity, absolute works okay and the bar tokens themselves give a periodic signal.

6.3 Audio Model Integration

If we incorporate audio generation (e.g., using MusicGen), what's the architecture? MusicGen's architecture is also a Transformer, but one that outputs audio token indices. We likely won't rewrite MusicGen from scratch; instead, we might fine-tune it. In that scenario, "architecture design" means figuring out how to integrate it with our system:

- Option 1: Two-stage approach. First use our custom transformer to generate MIDI (multi-track). Then separately, either:
 - Use a synthesizer (non-ML) to convert MIDI to audio (e.g., load it in Ableton and assign instruments). This is a perfectly valid approach if your goal is just to hear something – not everything has to be ML-generated. In fact, using your high-quality sample library might sound better than an AI generating raw audio from scratch. We’ll discuss this in integration.
 - Or use an AI like MusicGen directly: you could give MusicGen a text like “lofi hip-hop drum loop” and it generates an audio. But to align with a specific MIDI you generated, MusicGen does allow a melody audio as input to influence output. But that might not tightly sync either.
- Option 2: Unified approach. The model itself outputs both MIDI and audio. For example, one could design a model that outputs multiple modalities – but that’s research-level complicated (like having some tokens represent audio frames and some represent notes – likely not feasible together). More straightforward is training a separate model that generates audio conditioned on symbolic. That is like training a neural synthesizer (e.g., WaveNet conditioned on MIDI). Google’s Project Magenta had a model for that called PerformanceNet for drums or the NSynth which generates instrument sounds from a pitch input.

Given our scope, we won’t design a new audio model but use an existing one (MusicGen) in a two-stage pipeline. So our main architecture focus remains the symbolic model.

6.4 Putting It Together – Model Summary

Our Plan: We will implement a Transformer decoder model for multi-track MIDI sequences with the vocabulary we defined. We’ll condition on tags by including them as tokens. For audio, we will fine-tune or use pre-trained models separately as needed.

Pseudocode for model (PyTorch-like):

python

```
class MusicTransformer(nn.Module):
    def __init__(self, vocab_size, n_layers=6, n_heads=8, embed_dim=512, ff_dim=1024,
max_seq_len=1024):
    super().__init__()
    self.token_emb = nn.Embedding(vocab_size, embed_dim)
    self.pos_emb = nn.Embedding(max_seq_len, embed_dim)
```

```

self.layers = nn.ModuleList([
    TransformerDecoderLayer(embed_dim, n_heads, ff_dim) for _ in range(n_layers)
])
self.fc_out = nn.Linear(embed_dim, vocab_size)

def forward(self, input_ids):
    # input_ids: (batch, seq_length)
    seq_length = input_ids.size(1)
    positions = torch.arange(0, seq_length, device=input_ids.device).unsqueeze(0)
    x = self.token_emb(input_ids) + self.pos_emb(positions)
    # causal mask for self-attention (ensure model can't see future tokens)
    mask = generate_causal_mask(seq_length, device=input_ids.device)
    for layer in self.layers:
        x = layer(x, mask) # each layer does self-attn with mask + FFN
    logits = self.fc_out(x) # (batch, seq_length, vocab_size)
    return logits

```

In practice, we might utilize `nn.TransformerDecoder` from PyTorch which can simplify this.

Parameter initialization and training – we’ll rely on PyTorch defaults or Xavier init. We will train this model with a cross-entropy loss to predict next token.

We should also consider how to feed the model for training: typically we take each training sequence, and the model sees it from token 0 to N-1 as input and predicts token 1 to N as output. We’ll implement that logic in the training loop or use an existing Trainer.

Multi-GPU or not – Our model might be moderate enough to train on one GPU. If using multiple GPUs, PyTorch DDP or DeepSpeed can help, but given an intermediate audience, we might assume single-GPU training (or at most we mention you can scale if needed). In cloud, a single A100 could handle a lot.

Integration with Hugging Face – Hugging Face’s Transformers library can handle a lot of this. For example, one could create a Tokenizer (though here we rolled our own specialized one), and then use `GPT2TokenizerFast` with a custom vocab if desired. Then use `AutoModelForCausalLM` with a config (GPT-2 like config with our vocab size) and train. This could speed up development.

But implementing it manually also teaches how it works. We might lean on their Trainer for simplicity of training loop.

Evaluation – We’ll want to monitor training loss, maybe sample some outputs periodically to hear/see if it’s learning (maybe generate a short clip from a fixed prompt every epoch).

Finally, keep in mind the **progressive training** idea: we might start training on just one track (like just drums) to see if the model can learn basic rhythms, then add complexity. Or train shorter sequences then longer. Curriculum learning can help, but not strictly necessary.

We will cover training details in next chapter. At this point, the architecture is decided.

Exercise 6.1: Examine a Transformer Example – If you’re not very familiar with Transformer code, it would help to look at an example implementation. You can visit Hugging Face’s documentation and find the GPT2 model implementation (or use `model = GPT2LMHeadModel.from_pretrained('gpt2')` in a notebook and examine `model.config` and `model.transformer` structure). Identify the number of layers and heads in GPT-2 small. This provides a reference for what a transformer’s components are. Think about how you might adjust those for your music model (e.g., maybe fewer layers since our dataset is smaller than GPT-2’s text data).

Chapter 7: Training the Model

With data prepared and model architecture in place, it’s time to train the model. In this chapter, we will set up the training pipeline: loading data in batches, defining the loss function, and running epochs of training so that the model learns to generate music like your data. We will also discuss tips for effective training, avoiding overfitting, and leveraging hardware (GPUs) efficiently. By the end of this chapter, you should be able to kick off training of your music generation model either on your local machine or in the cloud.

7.1 Setting Up the Training Environment

Before writing training code, ensure your environment is ready:

- **Install Dependencies:** By now, you should have Python (preferably 3.9+), PyTorch (with CUDA support if using GPU), and possibly Hugging Face Transformers installed. Also ensure you have libraries like `pretty_midi`, `torchaudio`, etc., as needed. If using a Docker container, these would be in your Dockerfile (we will provide a sample Dockerfile in Chapter 10). If on local, a `pip install torch transformers pretty_midi librosa` (with correct CUDA version for torch) should suffice.
- **Hardware Check:** Verify that your GPU is available. In Python, do:

```
python
```

```
import torch
print(torch.cuda.is_available())
print(torch.cuda.get_device_name(0) if torch.cuda.is_available() else "CPU only")
```

This should show your GPU's name (e.g., "NVIDIA RTX 3080"). Training on GPU will massively speed things up. If you have to use CPU, consider training a very small model or just following the steps conceptually, as CPU training will be slow.

- **Reproducibility:** Set a random seed for experiments to reproduce results (optional but good practice). E.g., `torch.manual_seed(42)` and similarly for `numpy` if used.

7.2 Data Pipeline for Training

We need to feed the tokenized data to the model in batches. Here's how:

- Load your tokenized dataset. If you prepared a `tokens.txt` or similar, you can read it and parse the tokens to their IDs. Alternatively, if you have them in memory (list of token ID lists).
- Create a mapping for tokens to IDs (vocab dictionary) and IDs to tokens. This should be saved so you can use the same mapping at inference. If using a library's tokenizer, you can save that as well.
- Decide on a sequence length. If all your training sequences are roughly the same length (say we use 16-bar segments, maybe 500 tokens each), you could pad them to a fixed length for batching (or use dynamic batching if some are shorter). It might be easier to pick a max length and pad shorter ones with a special `<PAD>` token (which our model can ignore in loss via masking).
- Create a PyTorch Dataset class that yields sequences (input IDs and target IDs). In a causal LM, the target is typically the next token for each position. One approach: for each sequence, return `input_ids = sequence[:-1]`, `labels = sequence[1:]`. The model will be trained to predict labels from `input_ids`.
- Then wrap it in a `DataLoader` with batching. Use `shuffle=True` for training loader. A batch will be a tensor of shape `(batch_size, seq_len)`. Ensure to pad sequences in a batch to the same length (PyTorch's `DataLoader` with a custom `collate_fn` can do this, or use `transformers.DataCollatorForLanguageModeling` if using their Trainer).
- We also create a validation set (some fraction of the data set aside) to monitor performance on data the model hasn't seen in training, to detect overfitting.

Batch size: This depends on your GPU memory. Each token in sequence turns into an embedding and flows through the model. If we have sequence length ~ 500 and model embedding 512, that's $500 \times 512 \sim$

256k values per sequence per layer, etc. A batch of 8 or 16 might be reasonable on a 12GB GPU, but you can adjust. Start small and increase if you see GPU memory headroom. If using gradient accumulation (to simulate larger batch), that's possible but may not be needed.

Loss function: We will use CrossEntropyLoss (which internally does softmax and computes negative log-likelihood). In PyTorch, nn.CrossEntropyLoss() can compare output logits of shape (batch, seq_len, vocab_size) with target of shape (batch, seq_len). We might need to flatten these or use ignore_index for padded tokens. Hugging Face's Trainer handles this if you provide labels with -100 for padded positions as ignore index.

Optimization: Use an optimizer like Adam or AdamW (AdamW is standard for transformers). We'll choose a learning rate – often something like 1e-4 to 5e-4 for training from scratch, but it varies. We may also schedule the learning rate: e.g., a warm-up for a few hundred steps then decay. The Hugging Face Trainer can set up a scheduler easily (like linear decay with warmup).

Training loop: If writing custom:

```
python
```

```
model.train()
for epoch in range(num_epochs):
    for batch in train_loader:
        inputs = batch['input_ids'].to(device)
        labels = batch['labels'].to(device)
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs.view(-1, vocab_size), labels.view(-1))
        loss.backward()
        optimizer.step()
    # Validation after each epoch:
    model.eval(); compute val loss ...
    model.train()
```

If using Trainer, define a TrainingArguments and call trainer.train().

Epochs and iterations: How long to train? Music models can be tricky – too little and they output nonsense, too much and they overfit (memorize training songs). There's no fixed rule; monitor validation loss. Perhaps start with, say, 50 epochs or until it seems to converge. You might find it plateaus earlier.

Also since our dataset could be small, even a few epochs might overfit. Pay attention to when training loss << validation loss (overfit sign).

To mitigate overfitting:

- Use dropout in transformer layers (typical values 0.1).
- Possibly use regularization like weight decay (AdamW uses weight decay).
- Data augmentation: For MIDI, you can transpose some tracks randomly during training to augment (especially for melodic data – e.g., randomly shift melodies up/down a step, except drums). This increases variety. You can implement this in the data loader (randomly shift pitches on the fly for some samples).
- Early stopping if needed.

7.3 Monitoring Training

Keep an eye on:

- **Loss curves:** Plot or print the training and validation loss each epoch. They should decrease. If validation loss starts increasing while training loss keeps dropping, that's overfitting. You might then reduce learning rate or stop training.
- **Sample outputs:** A fun and useful way to monitor is to periodically generate a sample from the model and listen or examine it. For example, every epoch, take a fixed prompt (like maybe just a genre token and Track start) and generate a few bars, then convert to MIDI and see if it's getting more realistic. In text models, people generate sample text to see if it's learning grammar; similarly, generating a short music clip can qualitatively show learning progress (initially it might be random notes, later it might start to hold a beat or scale).
- **Gradient norms:** If using a high learning rate, watch out for divergence (loss becoming NaN). Gradient clipping (e.g., clip norm to some value like 1.0) can help stability.

Compute considerations: Each epoch iterates over the dataset. If dataset (post-tokenization) has, say, 100 sequences of length 500, that's 50k tokens per epoch. With batch size 8, that's $\sim 100/8 = 12.5$ iterations per epoch – very few, so you might do many epochs quickly. If dataset is larger, adjust accordingly. The concept of “steps” (batches seen) might be more relevant. Sometimes schedules are given in steps, e.g. 1000 steps warmup.

If training in the cloud, you could use tools like Weights & Biases (wandb) to log metrics and even sample outputs.

Let's say after some training, validation loss is low and not improving much, and sample outputs sound good – then training is complete. Save the model weights.

7.4 Fine-tuning Pre-trained Models (Optional)

Another training scenario is fine-tuning an existing model. For instance, fine-tuning Meta's MusicGen on your audio data. This requires their code and possibly a good GPU. Fine-tuning can be faster as the model starts from a knowledgeable state. If you go this route:

- You'd load the pre-trained model, freeze or not freeze some layers (maybe freeze the audio codec, only train the transformer part further).
- Provide your audio clips with text labels to the training loop. E.g., if you have 50 ambient loops, label them "ambient". The model can adapt to generate more ambient style after fine-tuning.
- Keep learning rate low (like $1e-5$) to not ruin pre-trained weights too quickly.

We won't detail this fully due to complexity, but references like the blog on fine-tuning MusicGen [active-loop.ai](https://active-loop.ai/active-loop.ai) highlight that modeling music audio is challenging due to long sequences and high frequencies, so fine-tuning can help the model focus on a narrower style.

For our system, you might fine-tune MusicGen on your stems to better generate similar audio timbres. This could be done after training the symbolic model, so you generate a MIDI, and then you want an audio model to produce that in your style – not trivial but possible: one hack is to feed your generated MIDI into your DAW to get audio, or fine-tune an audio model to directly go from genre prompt to a full mix resembling your productions.

We'll proceed assuming our main training effort is on the symbolic model.

7.5 Saving and Loading Models

During training, especially long runs, save checkpoints. This could be every N epochs or just the final model. Using HuggingFace Trainer, it handles saving. If manual, use `torch.save(model.state_dict(), 'model_epoch10.pt')`. Also save the vocab/tokenizer mapping (maybe as a JSON or pkl). Also note any special needed info (like how to detokenize, instrument mapping) for later use.

After training, you'll have a trained model weight file. We can then use it for generation.

Exercise 7.1: Dry Run a Tiny Training – It's often useful to test your training loop on a tiny example to verify it works. Create a trivial dataset (e.g., two sequences: [PIECE_START, NOTE_ON=60, NOTE_OFF=60, PIECE_END] and [PIECE_START, NOTE_ON=62, NOTE_OFF=62, PIECE_END]) and train the model for a few iterations. This is not about meaningful music, just about catching bugs. Does the

loss decrease to near 0 (since the model can memorize two sequences)? If so, your training loop is functioning. Then you can replace with real data.

Chapter 8: Inference - Generating Music from the Trained Model

Training is complete – congratulations! Now comes the exciting part: using the model to generate new music. In this chapter, we'll set up the inference pipeline: how to feed prompts into the model and sample outputs to create MIDI or audio files. We will cover various prompt scenarios (text, MIDI, etc.) and how to post-process the model's output tokens into usable formats. Additionally, we will discuss strategies to steer generation (controlling randomness or forcing certain tokens) and troubleshoot common issues (like outputs that are too short or repetitive).

8.1 Inference Basics and Sampling Strategies

When generating (also called decoding), the fundamental process is:

1. **Provide a prompt** (a sequence of tokens to start with, can be empty or just the start token or include genre tags, or even some notes).
2. **Iteratively predict the next token** using the model, then append that token to the sequence, and repeat.
3. Stop when some stopping condition is met (like an End-Of-Piece token generated, or a length limit).

At each prediction step, the model gives a probability distribution over the vocabulary for the next token. Simply taking the most likely token each time (greedy decoding) often results in dull or repetitive sequences. Instead, we introduce randomness in a controlled way:

- **Temperature:** This is a parameter that smooths or sharpens the probability distribution. A higher temperature (>1) makes the output more random by flattening the probabilities (encouraging less likely tokens to be chosen occasionally), while a lower temperature (<1) makes it more deterministic (peaks more pronounced). As an example, at temperature 1.0, we sample according to the model's softmax probabilities. At temperature 0.8, we are closer to greedy, and at 1.2, we might get wild variations [upwork.com](https://www.upwork.com). In music, a temperature around 1.0 is usually a good starting point; you can adjust if output is too boring (raise it) or too chaotic (lower it).
- **Top-k / Top-p (nucleus) sampling:** These are techniques to cut off the long tail of unlikely tokens. **Top-k** means at each step, consider only the top k most probable tokens and sample from them (set rest to 0 prob). For example, top-k=5 would only allow the 5 most likely next tokens [upwork.com](https://www.upwork.com). **Top-p (nucleus)** means consider the smallest set of tokens whose

cumulative probability exceeds p (e.g., 0.9) and sample from those. These methods ensure we don't sample extremely unlikely tokens that could derail the composition. We can tune these: for instance, in text generation, top- $p \sim 0.9$ is common.


For music, using temperature with a fixed top- k (like 10 or 20) might be wise to keep things coherent (especially for note values where you don't want a completely off-scale note with low probability randomly sneaking in).

We will implement a sampler that given the model and a prompt, loops and generates tokens one by one using these strategies.

8.2 Prompting the Model

We have several prompt types to handle:

- **Genre/Style Prompt (Text Labels):** If you want a certain genre, you should include those tokens at the start. For instance, `PIECE_START GENRE=hip_hop STYLE=lofi TRACK_START INST=DRUMS BAR_START` as a beginning. If you do this, the model knows from training that typically after “lofi hip_hop” you might have certain instruments and rhythms. You might include just genre/style and then let it generate everything else from scratch.
- **Instrument Prompt or Partial sequence:** If you have a specific drum pattern or melody as a seed, you would encode that entire sequence of tokens (with the appropriate track tokens and notes) as the prompt. The model will then continue from the end of that sequence. For example, you have a drum loop (maybe 2 bars) that you like, and you want the model to generate a bassline for it and continue drums. You could feed the 2-bar drum tokens (closing the bar and track if needed) and then include `TRACK_START INST=BASS BAR_START` and let the model generate bass notes.
- **Audio Prompt (style transfer):** If you have an audio clip that you want to use as a style reference, our symbolic model doesn't directly ingest audio. One way is to extract features from that audio and convert to a token. But we haven't trained it that way. This is where using the audio model might come in: e.g., MusicGen can take an audio clip (like a reference melody) as input in addition to text huggingface.co. If we fine-tuned MusicGen, you might do something like `musicgen.generate(prompt="techno 130bpm", audio_reference="clip.wav")` and it will output an audio sample that follows the style of clip.wav. But with our symbolic model, a simpler approach: maybe treat the audio prompt externally, e.g., detect its BPM or key and just feed those as tokens (if the audio is of a drum loop, you could transcribe it to a simple drum token sequence as seed using an onset detection). This is advanced and likely outside our current pipeline. For now, we'll assume prompts are given in symbolic form or category tags.



To prompt effectively:

- Always start with `PIECE_START` so the model knows it's a new piece. Then any global tokens (genre, etc.).
- If you know the first track should be drums (as we trained sequences in that order), you might include `TRACK_START INST=DRUMS BAR_START` but not give any notes, and let the model generate the drum track from scratch. Alternatively, if you want to start with a melody, you might prompt with an empty drum track then start a melody track.
- It's possible to prompt the model to only generate certain tracks by providing others. For instance, to get it to generate accompaniment to your melody: include full tokens of your melody track (with `TRACK_END`), and then add `TRACK_START INST=PIANO BAR_START` (assuming you want a piano comp) and let it continue. The model will see the melody in the prompt context and (hopefully) create something fitting after.

Keep in mind the model always generates one token at a time. It doesn't "know" not to touch the prompt tokens; we simply won't ask it to generate those, they are given. So you have to craft the prompt in such a way that the next token it should generate is exactly where you want new material. If the prompt ends in the middle of a track, the model will continue that track. If the prompt ends right after a `TRACK_START` of a new instrument, it will start generating that instrument's notes.

8.3 Generating MIDI Outputs

Once we have a sequence of tokens generated by the model, we need to turn it into a MIDI file (or multiple MIDI files for each track). Steps:

- Parse the token sequence:
 - It will contain `TRACK_START` and `TRACK_END` tokens for each track (if our model properly outputs them). We should split the sequence by tracks. For each track, identify the instrument (from `INST=x` token) and gather all note events (Note on, off, durations, etc.) with timing.
 - Reconstruct timing: when we tokenized, we decided how to encode time (e.g., a `TIME_DELTA=4` meaning move 4 16th notes). So we will accumulate time as we scan tokens. For example, set a `current_time = 0`. As we read tokens:
 - If token is `TIME_DELTA=n`, increase `current_time += n * (time_unit)` (time_unit might be 1/16th note or specific tick count).
 - If token is `NOTE_ON=pitch`, record a note start at `current_time` for that pitch. We might need to store it until we find when it ends:

- If we had DURATION=d token following, then note end time = current_time + d*(time_unit).
- If we instead rely on a NOTE_OFF token, then note off will appear later in sequence when current_time has advanced.
- Either way, collect notes with start and end times.
 - If token is BAR_END or track switches, you might just note that as structure (maybe reset current_time per track if we didn't interleave tracks).
- Since we likely did tracks sequentially, each track's notes are relative to track's own timeline starting at 0. But if they were meant to align, they should have the same number of bars. If one track ended early, perhaps we add a rest until others finish. This is a bit complicated: ideally the model output each track with equal bars. If not, we may need to align by padding shorter tracks with silence.
- Use a MIDI library to create a MIDI file: create instruments, set tempo (if we have a TEMPO token or use a default). For each note (pitch, start, end, velocity), add to MIDI track.
- Save to .mid file.

During generation, we might also enforce some constraints:

- If we want exactly 8 bars, we could stop generation after the model outputs 8 BAR_ENDs for each track or something. Alternatively, include a special end token. Perhaps our PIECE_END can be used by model to terminate. If the model doesn't output it, we might forcibly stop when desired length reached.
- We can programmatically stop generation after N tokens to avoid infinite loop. Many models when well-trained will output a PIECE_END at the end of a piece because training sequences ended with it. If not, implement a stop: e.g., if token is PIECE_END or if length > max.

Quality and coherence: The generated MIDI might need some clean-up:

- Perhaps quantize it if slight timing irregularities appear (if using time tokens it should be quantized by design).
- If the model left a note hanging (no NOTE_OFF and we used that scheme), you may need to terminate it at end of bar.
- Remove any overlapping notes in monophonic lines if that matters (in polyphonic tracks, overlaps are fine if chords, but double triggers might be weird).
- If some tracks are empty (model created a TRACK_START with almost no notes), you can decide to drop those tracks from final MIDI.

After this, you have a MIDI file. You can load that into Ableton. Typically, Ableton will interpret each MIDI track with a default piano sound if it's a .mid with multiple tracks, or if you import separate .mid files, you assign instruments.

8.4 Generating Audio Outputs

If we want audio directly:

- **Via symbolic route:** We could take the generated MIDI and render it. Options:
 - Open Ableton and use your instruments (manual step).
 - Or use a script: If you have a synth or sampler accessible, e.g., use fluidsynth with a SoundFont for quick instrument sound. For drums, you could pre-map certain MIDI notes to drum samples (e.g., if using General MIDI mapping: 36 = kick.wav, 38 = snare.wav, etc.) and programmatically mix those samples according to the MIDI. That might be a small separate project (writing a simple sampler).
 - There are also Python libraries (like magenta's synthesizer, or MIDIUtil combined with a software synth).
 - Given this is a textbook for internal publishing, we can describe how to do it conceptually rather than provide full code for audio rendering.
- **Using a generative audio model (MusicGen):** If we fine-tuned or want to use MusicGen to produce an audio:
 - E.g., call MusicGen with a text prompt, or give it a melody audio to follow. Notably, MusicGen can accept a melody audio to influence the output huggingface.co, but our output from the symbolic model is MIDI, not audio. We could turn the MIDI of, say, melody track into audio (maybe using a simple synth), then feed that to MusicGen as the melody prompt, with a text prompt describing overall intent.
 - For instance: we have a generated drum and bass MIDI. We might not have an easy way to feed both to MusicGen, as it takes one optional audio. Maybe just take the melody or chords as audio prompt and in text say "with groovy drums".
 - This approach is a bit roundabout and might not perfectly reconstruct specific patterns.

If the user really wants AI to output final audio, fine-tuning MusicGen on their data to use text or tags might allow generating similar music from text alone, bypassing the symbolic model. However, that means losing precise control (the model directly produces complete audio).

Possibly a combination: use symbolic model for structure, then another model for style. There is research on pipeline like: generate MIDI -> render with style transfer network. But for simplicity, one might just treat the symbolic output as the final product (i.e., user plays it with VSTs).

Given the likely workflow in Ableton, the simplest path: **Output MIDI for composition, and let the musician use their own instruments to get the final sound.** This is typically what tools like Magenta Studio do – they output MIDI clips that you then assign sounds to in Ableton magenta.tensorflow.org. We will emphasize this, and treat audio generation as an optional extension.

8.5 Example Generation Session

Let's simulate a generation:

- User wants a *lofi hip hop beat with 8 bars*:
 - Prompt tokens: `PIECE_START GENRE=hip_hop STYLE=lofi TEMPO=80 TRACK_START INST=DRUMS BAR_START`.
 - We feed that, then run sampling. The model generates drum pattern tokens for a number of bars, then presumably ends drum track: `TRACK_END`.
 - It might then automatically start another track if in training it learned to often do drums then maybe a melody. If not, we can force start a bass track by adding `TRACK_START INST=BASS BAR_START` to the prompt after drum track is done, and continue generating.
 - We continue until we have, say, drums, bass, maybe chords.
 - When to stop? If we want exactly 8 bars for each, we might stop after the model has placed 8 `BAR_END` in a track and maybe a `TRACK_END`. We might have to trim extra if it goes beyond.
 - Append `PIECE_END` if not present and stop.
 - Now we decode that token list to MIDI: create separate MIDI tracks for drums, bass, chords.
 - Save and import into Ableton. The drums track will be just MIDI notes (likely general MIDI drum notes like 36 for kick, etc., since our model was trained on your drums data, it will use the notes from that data – ensure you use a matching drum kit).
 - The bass track MIDI you route to a bass instrument in Ableton. And so forth.
 - Press play in Ableton and enjoy the AI-created loop. Maybe it's a bit raw, but you can now tweak velocities, swap samples, arrange it into a song structure.

Controlling length and structure: If the output is too short or the model stops early (maybe it output a `TRACK_END` too soon), you can try to coerce it by continuing generation even after a track ends (maybe start a new track or repeat structure). If it's too long or rambling, you cut it or instruct model via prompt like including an explicit `PIECE_END` token after a certain bar (though if you put that in prompt, generation stops since end reached).

Avoiding repetition: Models sometimes get stuck in loops, e.g., repeating the same bar. Top-k/p helps, but if it's an issue, you can implement heuristic filters (like if the last 4 bars tokens exactly repeated,

maybe break). But that complicates the decoding logic and usually a well-trained model with correct temperature should be okay.

8.6 Using the Cloud vs Local for Inference

If your model is large and you trained on the cloud, you might also want to generate on the cloud (perhaps to leverage the GPU for faster generation, especially audio). However, often generation can be done on CPU for small sequences (though slower). You can also move the model to your local machine (the saved weights) and load it there if you have enough memory. For example, a 100M param model might be ~400MB on disk, which might be fine to load on a decent PC.


If you built a Docker for training, reuse it for inference so that all dependencies match. You could also wrap the generation in a simple script or even set up an API if you want (like a Flask server that returns generated MIDI for given prompts). That's beyond our scope but possible for integration.

8.7 Troubleshooting Generation Issues

- *Model outputs invalid token order (like TRACK_END without TRACK_START):* This could happen if training didn't sufficiently teach those rules. If minor, you can post-process to remove anomalies. If major, you might need to enforce constraints during generation (e.g., you could tell the model that after a TRACK_END, if it outputs something other than another TRACK_START or PIECE_END, you ignore it or adjust).
- *The music is unsatisfactory (too simple or odd):* This could be due to under-training or overfitting to limited data. Possibly fine-tune more, or provide a stronger prompt. Sometimes giving the model a bit of a start (one bar of something) leads to better results than asking it to generate from scratch.
- *Timing issues:* If tracks don't align well (e.g., bass notes off beat relative to drums) and you expected alignment, possibly tokenization ordering caused this. You might need to refine how tracks are generated (maybe interleaved by bar in prompt).
- *Performance:* If generating long pieces or many variations, ensure you use batch generation if needed (some frameworks allow generating multiple sequences in parallel). Or offload heavy parts to GPU. Usually generation one token at a time is not too heavy compared to training.

Now you have a pipeline to generate music. The next chapter will discuss how to integrate these results into Ableton Live and how to set up the system for ease of use on your workstation.

Exercise 8.1: Generate and Listen – Use your trained model to generate a short piece (even 4 bars). Convert the output to MIDI and import it into Ableton or a MIDI player. Listen to it critically. Does it reflect your style? Identify one thing you like (e.g., “the hi-hat pattern was nicely varied”) and one thing to



improve (“the melody was too random”). This will guide you on perhaps adjusting model parameters or editing the output musically. Remember, the human is still in the loop – you can always edit the AI output to polish it!

Chapter 9: Ableton Live Integration and Workflow

At this point, we can train and generate music with our AI system. The final step is integrating this into your music production workflow with Ableton Live 11. In this chapter, we’ll cover how to best import the AI-generated material into Ableton, how to set up a workflow for iterating between the AI and Ableton, and some tips for using the AI as a creative assistant (for example, generating variations or ideas that you then build on). We’ll also discuss any technical considerations specific to Ableton Live, such as tempo syncing and file management.

9.1 Importing MIDI and Audio into Ableton

MIDI Import: Ableton Live 11 makes it easy to import MIDI files:

- Drag-and-drop the MIDI file from your file explorer directly onto an open Ableton set. If the MIDI contains multiple tracks (type 1 MIDI), Ableton will create multiple new MIDI tracks, each with a default instrument (usually a piano sound) unless the file has track names that match an existing instrument rack name. You can then replace those instruments with ones you want (e.g., put a Drum Rack on the drum MIDI track).
- Alternatively, you can go to the File menu and use “Import MIDI File...”. This does the same thing.
- If you have separate MIDI files per instrument (e.g., you saved drums.mid, bass.mid separately), you can drag each into Ableton onto specific tracks. This is useful if you have a project open with your favorite drum kit on one track – you can drag the drums MIDI onto that track to have it played with your samples.

Tempo and Time Signature: Make sure to set the Ableton project tempo to what the AI used (we included a TEMPO token, say 120 BPM). Ableton might not automatically adjust the tempo from a MIDI file (MIDI files can have tempo info that Ableton will read, but often it uses the current project tempo). To be safe, manually set it. If the AI output was at 90 BPM and your Ableton is 120 BPM, the timing of MIDI notes will shift in musical meaning (though MIDI is tempo-relative so they will just play faster or slower; audio loops would require warping).

If your piece has a different time signature (like 3/4), set that in Ableton for the relevant bars, though our guide assumed mostly fixed 4/4.

Audio Import: If you generated audio loops or stems with the AI:

- Drag the WAV file into Ableton. Ableton will place it on an audio track. Check the tempo: if the loop file has BPM in its filename or you know it, adjust Ableton's tempo or use Ableton's warp function to match it. Ableton might auto-warp short loops to the project tempo; if it guesses wrong, you might disable warp and just set project tempo to loop's tempo for seamless playback.
- If you have multiple stems (drums.wav, bass.wav, etc.), drag each onto separate audio tracks and line them up at bar 1. They should all be exactly the same length (if our generation ensured equal bars), so they stay in sync. If not, you may need to trim or loop them accordingly.

File Management: It's a good idea to keep your AI outputs organized:

- Perhaps have an "AI_Generated" folder in your project, with subfolders for each session.
- Save the MIDI files with descriptive names and BPM, like ChillHopIdea1_80bpm.mid.
- When you import into Ableton, those files are referenced; if you save the Ableton project, it doesn't embed the MIDI file, but the MIDI notes are saved in the .als project. For audio, Ableton may the WAV into its project folder (depends on your file collection settings).
- Consider saving the prompt and model settings that produced a given output, in case you want to reproduce or vary it. For example, keep a text note: "Idea1: Prompt = genre=hip_hop, style=lofi, generated 8 bars drums+bass at temp1.0 topk=10."

9.2 Working with AI-Generated Material

Treat the AI output as a draft or inspiration. Some tips:

- **Quantize / Groove:** If the timing is slightly off or too mechanical, use Ableton's quantize or Groove Pool. For instance, if the model generated slightly uneven timing on a drum beat, you can quantize it to the grid. Or if it's too rigid, apply a groove template (maybe MPC swing).
- **Transpose and Modify:** If the melody is great but in the wrong key, transpose the MIDI. If the chord progression is okay but one chord seems off, change that chord in the piano roll.
- **Split and Rearrange:** The model might generate through-composed 8 bars. You can slice it up: maybe the first 4 bars are good for a verse, and the next 4 for a chorus. Or you can loop certain parts. Don't feel obligated to use it exactly as is.
- **Layering:** You can layer the AI part with your own played parts. Maybe the AI gave you a bassline idea – you could double it with a synth or modify the rhythm.
- **Editing Drum Patterns:** Often AI might produce a drum pattern that's a bit busy or sparse. You can easily mute or add hits in Ableton's drum MIDI editor. For example, if there are too many kicks, delete a few. If you want a crash at the start, add it.

- **Velocity and Dynamics:** AI might not perfectly vary velocities unless we incorporated that in training. You might want to humanize velocities (Ableton has a randomize velocity feature or you can manually adjust). This adds expressiveness.

The key is that the AI system can give you a starting point – something to react to. You as the producer can then apply your judgment to polish it.

9.3 Using the AI in the Creative Process

There are a few ways you can incorporate this AI system in your routine:

- **Idea Generation:** Stuck with writer's block? Generate a few 8-bar snippets in different styles and audition them. Even if one is 50% good, that's material to work with. Sometimes a surprising riff or chord change from the AI can inspire a new direction.
- **Variation and Fills:** You have a main drum groove but need a fill every 8 bars? You could prompt the model with your main groove and ask it to continue for 1 bar; maybe it comes up with a cool fill. Or generate 10 variations of a bassline and pick the best parts.
- **Continuing a Track:** Suppose you have 4 bars and you wonder what could come next. Provide those 4 bars as prompt, and let the model continue a few more bars. Maybe it predicts a nice continuation.
- **Multi-track Arrangement:** Perhaps generate multiple tracks together for an arrangement idea – then you might mute some and only use one. For instance, the model might generate drums, bass, and piano, but you only liked the bass – that's fine, you got a bassline out of it.
- **Sound Design with AI audio:** If you did incorporate audio generation, you could generate weird ambient textures or percussion sounds, then import those as audio and treat them like samples in Ableton (chop, effect, etc.).

Staying Organized: It's easy to generate a lot of content. Keep track of what model settings or prompts yielded outputs you like, and keep those seeds for later. Conversely, if something consistently doesn't work (e.g., model always makes chaotic output for a certain genre), note that and maybe retrain or avoid that.

9.4 Example Workflow

Let's illustrate a workflow for clarity:

- **Step 1:** You set Ableton to 130 BPM, want a synthwave drum beat. You prompt the AI: `GENRE=electronic STYLE=80s TEMPO=130 TRACK_START INST=DRUMS` It generates a drum

pattern with big gated snares. You import the MIDI to Ableton, assign a LinnDrum kit – sounds good! Minor adjustments on hi-hats and it's solid.

- **Step 2:** You then ask AI to generate a bassline for that. You prompt it with the drum track from step 1 plus TRACK_START INST=BASS. It produces a bass riff. You bring it in, put a bass synth. It's a bit busy, so you remove some passing notes. Now it locks nicely with drums.
- **Step 3:** You manually play a pad chord progression that you feel fits. Now, you use the AI one more time: give it the drums and bass and chord track as context, and ask it to generate a lead melody (TRACK_START INST=LEAD). It gives something. You use parts of it, tweaking a melody line to better match the chords.
- **Step 4:** Now you have an 8-bar section. You duplicate it, maybe change some notes in the second iteration to create a variation or use AI to generate a small fill. Then you arrange, etc. The AI parts are now fully in Ableton, and you treat them like any clip – cut, effect, automate.

The result is a hybrid of your input and AI suggestions, hopefully achieving a result you might not have gotten alone as quickly.

9.5 Limitations and Improving the Integration

While the system can be powerful, be aware of limitations:

- **Musical Coherence:** AI doesn't truly "understand" music theory or emotion; it's imitating patterns from training. Sometimes it might produce technically correct but soulless phrases. Always add your musical intuition.
- **Style limitations:** If your dataset was small, the model might not generalize widely. For example, if you only trained on slow ambient music, asking it for a fast metal riff might not yield good results. It's bounded by what it saw. To broaden it, you'd need to train on more diverse data or explicitly augment data.
- **Integration Overhead:** Right now, using the system involves running a Python script outside Ableton and then importing results. This is somewhat offline. A future improvement could be to integrate it more into Ableton (for example, using Max for Live to call a Python backend that generates MIDI directly in Ableton). That's advanced, but possible (there are examples of using Python in Max for Live devices medium.com).
- **Real-time jamming:** Our system isn't real-time. If you play some chords and want the AI to instantly respond with a bassline, that's not doable without significant modifications. We generate offline then import. So it's more for composition than live performance at this stage.

Ensuring Ableton Compatibility: Always double-check the output format:

- For drums, ensure the MIDI notes correspond to the drum rack's mapping (you might need to adjust either the model's output or your drum rack).
- For keys, if the model output a program change or something odd (it shouldn't if we didn't include those tokens), you might see an empty MIDI track in Ableton with a program number – just delete those events.
- Ableton has a limit on MIDI clip length (hours long), not an issue for 8 bars but just a note.

This integration chapter solidifies how to effectively use the AI system in practice. With this, we have covered the full journey from data to trained model to making music in Ableton.

Exercise 9.1: Jam with the AI – Pick a musical element you struggle with (say, drum fills or chord progressions). Use the AI to generate a few alternatives for that element. Import them into Ableton and try each in context of a song you're working on. Does any AI-generated element improve your track or spark a new idea? This practical trial will show the value of having such a tool at your disposal.

Chapter 10: Deployment and Scaling (Local vs Cloud)

Our focus so far has been on development and usage. In this chapter, we address how to set up the system for long-term use and how to scale or deploy it if needed. Whether you run everything on your personal machine or leverage cloud resources for heavy training jobs, you should manage environments and dependencies for reliability. We will discuss creating a reproducible setup (using Docker or similar), using cloud services (like AWS EC2 or SageMaker) for training/inference, and the considerations of each approach.

10.1 Local Environment Setup (Windows with WSL or Native)

Windows Native vs WSL2: Since Ableton Live runs on Windows, you likely use Windows for music production. Training deep learning models is often easier in a Linux environment due to better compatibility with certain libraries and drivers. WSL2 (Windows Subsystem for Linux) allows you to run Linux within Windows. Microsoft, in recent Windows 10/11 updates, supports GPU compute in WSL2, meaning you can use your NVIDIA GPU inside an Ubuntu environment on Windows learn.microsoft.com/youtube.com. This is a great option to run training without dual-booting.

To set up WSL2 for CUDA:

- Enable WSL and install a Linux distro (e.g., Ubuntu) from Microsoft Store.
- Install the NVIDIA drivers that support WSL (the standard driver now includes support; ensure you have the latest driver).

- In the Ubuntu shell, install CUDA toolkit or simply install PyTorch with CUDA (Conda or pip).
- Microsoft's documentation learn.microsoft.com provides step-by-step. In short, ensure Windows is updated, WSL is updated, then `sudo apt-get update && sudo apt install nvidia-cuda-toolkit` inside WSL, etc.
- Test with `nvidia-smi` inside WSL2.

Alternatively, you can install Python and PyTorch directly on Windows (there are Windows wheels). That can work too, especially now PyTorch supports Windows fairly well. But some libraries (like `pretty_midi`) might need Visual Studio Build Tools to compile on Windows, which can be a hassle. Using WSL avoids those issues because you're in Linux land.

Python environment: Use a virtual environment or Conda environment to keep things tidy. For example, in WSL Ubuntu:

```
bash
```

```
sudo apt-get install python3-venv
python3 -m venv ai-music-env
source ai-music-env/bin/activate
pip install torch torchvision torchaudio transformers pretty_midi librosa
```

Also install Jupyter if you plan to run notebooks (`pip install jupyterlab`).

Code management: Keep your project code (the scripts for training, generation, etc.) in a folder that's accessible. WSL can access your Windows files (under `/mnt/c/...`). Perhaps put the project in your Documents.

Docker container (Optional): For ultimate reproducibility, you can create a Docker image that has all dependencies. Docker can run on Windows (with WSL2 backend). For example, a Dockerfile might use `nvidia/cuda:11.8-cudnn8-runtime-ubuntu20.04` as base, then `RUN pip install torch transformers` You can then run this container with `--gpus all` to allow GPU use. The benefit is you can easily deploy the same container on a server. However, using Docker for GPU in WSL requires some setup but it is supported (Docker Desktop with WSL2 integration and the NVIDIA Container Toolkit).

We won't go deep into Docker specifics, but here's a simple **Dockerfile** snippet as a reference:

Dockerfile

```
FROM nvidia/cuda:11.7.1-cudnn8-runtime-ubuntu20.04
RUN apt-get update && apt-get install -y python3.8 python3-pip
RUN python3 -m pip install --upgrade pip
RUN pip install torch==1.12.1+cu117 torchvision torchaudio --extra-index-url
https://download.pytorch.org/whl/cu117
RUN pip install transformers pretty_midi librosa
# project files
WORKDIR /app
. /app
```

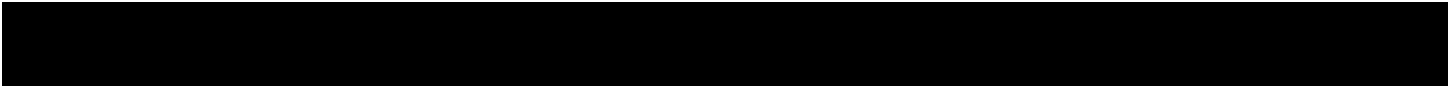
Then you'd build and run this. This ensures if you or colleagues want to replicate the environment, they can run the container.

10.2 Cloud Training on AWS

If your local GPU isn't powerful enough or you prefer not to tie up your PC for hours of training, using AWS or another cloud provider is an option.

AWS EC2:

- AWS offers EC2 instances with GPUs (like g4dn.xlarge with a Tesla T4, or p2.xlarge older K80, or more powerful p3 or newer g5 instances with NVIDIA A10Gs).
- These instances can be spun up on demand. You'd typically use an Amazon Machine Image (AMI) that has GPU drivers preinstalled. Amazon has Deep Learning AMIs (DLAMI) which come with Conda environments for PyTorch, etc.
- Steps: Launch an EC2 instance (Ubuntu or DLAMI) with a GPU, attach enough storage (to hold your dataset and to save models), and configure security (for SSH).
- SSH into the instance, and either git clone your project or files. If using Docker, you could build/run the Docker there (install nvidia-docker toolkit).
- Make sure to transfer your dataset to the instance. You can use SCP or AWS S3. For large datasets, S3 is handy: you upload data to S3, then from the EC2 download it (fast if same region).
- Run training as you would locally (maybe in a screen or tmux session so it continues if your SSH disconnects). Monitor logs.
- When done, download the trained model files to your local machine (via SCP or pushing to S3).
- Important: GPUs on AWS are not cheap by the hour. Always shut down (or stop) the instance when not in use to avoid charges. You can also use spot instances for lower cost, but they can terminate unexpectedly (so make sure to save progress often if so).



AWS SageMaker: This is a more managed service where you can run training jobs with code and data in S3. It can automatically handle spinning up instances and logging. For our scope, it's maybe overkill, but if you were professionalizing this, SageMaker training jobs or Google Colab/Vertex AI, etc., are options to consider. They have their own way of specifying the environment (like a Docker image or conda spec).

Colab/Google Drive: If you don't want to pay directly, Google Colab offers free GPU runtime for limited time. You could try training a smaller model there by uploading data to Google Drive. But free Colab is limited (K80 GPU for 12 hours etc.). Colab Pro gives better GPUs. It's good for prototyping though.

10.3 Collaborative or Distributed Training

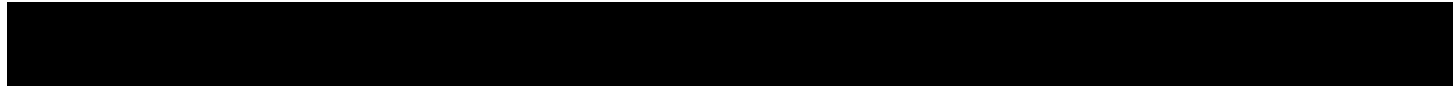
If you imagine extending this system with more data, you might need to scale out:

- Distributed data parallel across multiple GPUs or even machines: libraries like PyTorch Lightning or Accelerate can make multi-GPU training easier. But unless you have access to multi-GPU setups, you likely won't do this personally.
- If you had two GPUs on a PC, you could use `torch.nn.DataParallel` or ideally `DistributedDataParallel` to split batches. The complexity is beyond this guide's scope, but it's possible.
- Another scale method is mixed precision (FP16 training) which can speed up and reduce memory. PyTorch has `torch.cuda.amp` for that. If using HuggingFace Trainer, just set `fp16=True` in `TrainingArguments`.

10.4 Deployment as a Tool

Once satisfied, you might want to package this AI system as a more user-friendly tool:

- A simple CLI script could wrap prompt input and call the model to generate, saving to a MIDI file. For example, `python generate.py --genre hip_hop --bars 8 --output out.mid`.
- A lightweight web app (maybe a Flask or FastAPI server) could present a form to input prompt parameters and then trigger generation, allowing you to download the output. If you do this, ensure to handle concurrent calls if needed and load the model once at startup to reuse.
- **Max for Live device:** For deep integration, one could create a Max for Live (M4L) device that communicates with a Python backend. Max can send/receive UDP or OSC messages, so you could run a Python server that listens for a "generate" message from Max (with parameters), then outputs MIDI notes back to Max for placement in a clip. This is advanced but would allow using the AI directly in Ableton via a custom device. Some have done similar things for simpler AI tasks.



Remember to consider system performance; generating with a large model can take several seconds for a few bars. That's fine for offline use, but for interactive use you might need a smaller model or compromise on length.

10.5 Maintenance and Updating

As you use the system:

- You might retrain or fine-tune the model as you create more music (incorporating new data so the AI evolves with your style). Keep your dataset updated and perhaps do periodic training runs adding the new material.
- If new techniques come out (say, a better tokenization or a new model like MusicLM being released) you could integrate those to improve quality.
- Monitor the sizes of things: model file, dataset. If model size grows, ensure your inference environment can handle it.

Ethical/right Consideration: Since you're training on your own data, you should be fine regarding rights. If you added external data, be mindful of right (the outputs might inadvertently parts of training data). However, working with your own material mitigates that risk and ensures uniqueness.

Finally, ensure to back up your trained models and any important generated MIDI that you liked – treat them as part of your creative archive.

Exercise 10.1: *Environment Reproduction* – To test if your deployment setup works, try running your generation script on a different machine (or a fresh cloud instance). Use your documented steps or Docker to set it up. If it runs smoothly and produces the same output for a given random seed, you've successfully made your project portable. This is valuable if you ever upgrade your PC or share the tool with a collaborator.

Chapter 11: Exercises and Next Steps

Congratulations on building an AI-based music generation system tailored to your style! As a final chapter, we provide some structured exercises to reinforce what you've learned and suggest next steps to continue advancing your system and skills.

11.1 Progressive Exercises Recap

Let's recap the exercises from each chapter and ensure you have their solutions or outcomes:

- *Data Curation (Exercise 4.1 & 4.2):* You should have a spreadsheet of your dataset and perhaps a small script that demonstrates reading a MIDI file and transposing or otherwise manipulating it. Make sure you're comfortable writing code to iterate over your dataset – that skill is important in customizing this system further.
- *Tokenization (Exercise 5.1):* You tried tokenizing a simple MIDI. For instance, a C major scale might turn into tokens like NOTE_ON=60, TIME_DELTA=some, NOTE_OFF=60, If you used MidiTok, compare its output with our described format. Understanding that will help if you want to adjust the representation.
- *Transformer understanding (Exercise 6.1):* By examining an existing model like GPT-2, you should identify components like number of layers, etc. Perhaps you even wrote a snippet to instantiate a small Transformer and print its summary. This exercise ensures you can later tweak the architecture (e.g., “what if I add two more layers?”).
- *Training loop test (Exercise 7.1):* By running a tiny training on dummy data, you likely saw the loss go down to near 0 (the model memorized the dummy sequences). If that didn't happen, there may be a bug in how you set up inputs or loss. This test is crucial because it isolates the training code from the complexity of real data.
- *Generation and listening (Exercise 8.1):* You hopefully generated a short output from the trained model. Maybe it sounded off, maybe surprisingly good. The key is you managed to go through the full generate->MIDI->listen pipeline. If something broke (like the MIDI file was empty or corrupt), now is the time to debug the generation decoding.
- *Jam with AI (Exercise 9.1):* This was more open-ended: you integrate the AI output into a real music context. The outcome is subjective, but ideally you found at least a small element from AI that was useful. If not, consider why – maybe the model needs more training or maybe you need to adjust how you prompt it. The feedback loop between you and the AI can guide further training (for example, “the AI always makes too many notes in chord progressions” could lead you to train it on sparser examples or implement a rule to drop some notes after generation).
- *Environment reproduction (Exercise 10.1):* You tested deploying on another environment or using Docker. If you succeeded, you have confidence that you can set this up again (important when moving between studio computer and laptop, etc.). If not, identify where the gap was – update documentation or scripts to include missing steps.

11.2 Extending the System

Now that the basic system works, here are some ideas for extending or improving it:

- **More Data or Different Data:** If you only trained on your own music and you have, say, 20 songs, the model might be limited. You could consider incorporating external datasets to augment it. For

example, the open MIDI dataset Lakh (as mentioned earlier) or Drum MIDI libraries. You could then fine-tune the model on your style so it blends general music knowledge with your specific vibe.

- **Better Conditioning:** We used simple tokens for genre and mood. You could introduce more nuanced controls. For example, add a token for “intensity=high/medium/low” to indicate energy level, and label your data accordingly (maybe a loud part of a song as high intensity). Or specific instrument requests like “ADD_STRINGS” token that if present, the model should include a strings track. This requires training the model with such scenarios (you might need to generate some synthetic training data or cleverly label).
- **Structure Generation:** Currently, the model generates a fixed length. Real songs have structure (intro, verse, chorus, etc.). One extension could be generating longer sequences with sections or using multiple passes (generate a chord progression outline, then generate a melody on top, etc.). There is research on hierarchical music generation that might inspire you.
- **User Feedback Loop:** You as a user can guide the model by selecting outputs you like. One could imagine using your preferences to further train or adjust the model (reinforcement learning or at least iterative refinement). That’s advanced (Reinforcement Learning from Human Feedback, RLHF, is how ChatGPT was improved for example).
- **Diffusion or Other Models for Audio:** If you become ambitious to generate full audio, you might look into open-source diffusion models for music. There are projects like Diffwave for drums or other community projects. These could potentially be conditioned on your MIDI output (i.e., generate an audio drum loop conditioned on a symbolic drum pattern). That’s like building your own Drumify but for audio.
- **Real-time plugins:** Perhaps build a VST or Max for Live that wraps the generation. This would be a big project in itself but would make the tool more accessible in a production environment. Tools like JUCE (C++ framework) could theoretically host a PyTorch model, or one could use Outsourcing the heavy compute to a backend script.

11.3 Final Thoughts

This journey combined technical depth (machine learning, coding) with music knowledge. By working through it, you’ve not only built a unique tool for yourself, but you’ve also gained insight into how modern AI “thinks” about music. As you use the system, keep evaluating it critically: sometimes it will surprise you with creative ideas, other times it might fail hilariously. Each output, good or bad, is an opportunity to understand both your music and the model better.

Remember, the goal is not to have AI replace creativity, but to enhance it – like a tireless collaborator who can churn out ideas at your command. You hold the artistic vision; the AI provides sparks and sometimes labor (like quickly generating variations).



Next Steps Learning: To further boost your skills:

- Dive deeper into music AI research papers (the references we cited like MusicVAE, Music Transformer, etc., are great reads to understand the motivations and math).
- Learn more about deep learning by perhaps experimenting with simpler examples (try building an RNN music generator for comparison, etc.).
- If you haven't already, explore Magenta's open source code or others; you might find useful utilities or pre-trained models to incorporate.
- Engage with communities (there's a Magenta Google group, and many on Reddit [r/musictheoryandAI](#) or [r/AudioProgramming](#)) where people share progress on similar projects.

Most importantly, have fun with it. You now have a custom AI music tool – something relatively few producers have. Use it to create something truly original, and keep pushing the boundary of what's possible with technology and art.