Automatic Music Transcription at scale

Alex Hicks

Abstract

Traditional transcription methods for music involve extensive listening, trial and error, and frankly incorrect output. Most musicians cannot identify sound pitches without a reference, and so automatic music transcription (AMT) is essential to producing accurate, timely transcriptions of audio. However, existing reliable AMT systems are difficult for non-technical users to take advantage of, and so this paper puts forth a service to solve this need. The described service deploys resources to Google Cloud which allow a user to upload audio, and quickly receive a transcription generated by a machine learning (ML) model. The components of this are the model, Omnizart; the REST (REpresentational State Transfer) API providing public access to Omnizart; the web interface providing basic user access to the REST API; and of course, the Google Cloud services which the entire system lives on. In short, this service utilizes the flexibility and long ramp-up times afforded by cloud providers (like Google) to democratize music recognition, while maintaining low cost of operation to permit the widest possible usage.

Keywords

Automatic music transcription; AMT; Google Cloud; GCP; Google Cloud Run; Google Cloud Storage; Omnizart; Docker; Cloud finances.

Table of Contents

Introduction

Most musicians can't name a note they hear without a reference sound – if they can, they're said to have "absolute pitch." This rare ability lets a musician not only recognize but reproduce music they hear, without aid.[1] Unfortunately, even though it might be a partially learnable skill, the vast majority of musicians are unable to achieve this artistic flexibility and ease, and so we turn to tools like software. Traditional algorithmic software has been unable to solve this problem, referred to as Automatic Music Transcription (AMT) any better than the average human – recognizing distinct notes from noisy audio is a significant challenge, especially for non-traditional music.[3] Even worse, most cutting-edge AMT software depends on a strong technical understanding which most musicians lack. One such tool is Omnizart, a library which provides an interface to an AMT-focused machine learning (ML) model, so that command-line users can transform audio files into transcribed musical notation.[4] We describe a system which solves the myriad problems associated with imperfect pitch, by deploying the Omnizart project to Google Cloud alongside a web application providing end-user access to the ML model.

In Section 1, we describe the overall architecture of the service and its guiding principles. In Section 2, we analyzed possibilities for cost efficiency, user activity patterns, and long-term maintenance, to make the service as financially viable as possible – knowing that without users, a service is hardly worth the silicon it sits on. Section 3 discusses the specifics of the Omnizart deployment, and how it could be changed to focus on different features. The REST API's integration with Omnizart, and indeed the entire web back-end, are detailed in Section 4. Finally, in Section 5, we analyzed some potential bottlenecks if the service were to be run at scale, and put forth ideas to ease them.

Description

This paper describes a service which provides global access to accurate automatic music transcription, at relatively low cost and without significant ongoing maintenance requirements.

1 – Service architecture

We designed the transcription service with a few goals in mind – the main one, of course, is cost. In the ecosystem of the modern Internet, there is very little room for financially-inefficient services to gather users, and so the costs of operation, maintenance, and subsequent development must be an essential factor when designing the service. Specifically, Omnizart is a large component of the overall complexity, and so we devote the most attention to it in Section 1.1. Google Cloud is the backing provider which hosts the service and stores its data, so covering it in some detail (in Section 1.2) is useful. Finally, the REST API and front-end are the simplest pieces of the architecture, but we do describe their design in Section 1.3.

1.1 – Omnizart

Omnizart is the engine powering the entire service; without it, this paper would be focused in a very different, lower-level direction! Its Python interface gives us the flexibility to wrap it in our own script, transforming its command-line interface into adherence to a cloud-focused specification of sorts, detailed in Section 4.2. This storage-based interface allows minimal communication between Omnizart and any consumers (including our REST API), providing flexibility for those consumers and a lean implementation for our Python wrapper.

Omnizart's datasets are quite large, and therefore much of the service's bottleneck is related to downloading those "checkpoints," or updates for Omnizart's ML models.[4] Of course, executing those models isn't exactly quick either – but in practice, audio files are small, and musical audio smaller, so the relative impact of processing them is low. Our main concern when optimizing Omnizart for scale will be reducing the download/startup time of the container, and thereby lowering the overall costs.

Omnizart accepts a wide variety of file formats, but the web-app limits uploads to basic MP3 files for simplicity's sake.[4] Transcriptions are output as files in MIDI format, which describes the notes played and their durations. These files can be easily converted into full, playable scores, either by the musician or automatically by the service.

1.2 – Google Cloud

Google Cloud provides two essential components, but it's important to note immediately that they are not irreplaceable components. The service uses Google Cloud Run to deploy and invoke instances of Omnizart, and Google Cloud Storage to store audio files, output files, and transcription state. As you may notice, these are generic use cases, and many providers could support them – we chose Google Cloud for ease of use and their free tier's permissiveness.

Google Cloud Run permits us to spin up instances of Omnizart on demand, providing high performance on short notice using the new Jobs feature. It's not a perfect fit, but it allows just enough parameterization so that we can start a new instance for every transcription request, ensuring cost-efficient flexibility at lower request scales.

Both Google Cloud Storage and our usage of it are perfectly generic – we store binary blobs (files) for the user's audio uploads, the webapp's transcription state tracking, and Omnizart's MIDI output. This is in a small way imperfect, as state tracking is certainly not atomic in this setting, but at the low scale anticipated for this research, it's quite appropriate.

1.3 – REST API and front-end

These pieces make the Omnizart model actually accessible to end-users, and they are as important as they are underdeveloped. In order to focus this exploration on the deployment and scaling aspects of AMT, most features which would cause a user to actually use this service were cut. The user interface is basic, and the API has no authentication, but the core flow is

implemented via a Node.JS REST API, interacting with Omnizart based on the web interface's requests. The user uploads an audio file via a simple form, receives status updates via polling, and eventually is able to download their transcription.

1.4 – Service architecture summary

The service architecture is mainly designed to lower cost, and that is reflected in the technologies we selected, but several secondary considerations led to the system's specific layout. Primary among these was simplicity – simpler systems are easier (cheaper) to design, maintain, and improve, so this is also fulfilling the cost goal. Simplicity ties into the other secondary goal of reliability, but much of the reliability requirement is able to be offloaded to Google Cloud. As designed, this service should provide great functionality to end-user musicians without sacrificing funds or long-term viability.

2 – Costs and financial viability

2 – Introduction

The measure of any non-governmental service's success in a capitalistic economy is inherently tied to its financial viability; this may seem obvious, but is too often overlooked when starting to design a service. Our primary goal when considering finances is to ensure long-term viability. Profit is important in a general motivational sense, but the absolute requirement of not losing money is our focus in this analysis – if we can break even, we're happy.

2.1 – Costs of Google Cloud

The service is deployed on Google Cloud, and uses its resources for data storage & state management. We use Cloud Run, specifically a new beta feature called Jobs, but its pricing is standardized alongside the rest of Cloud Run into region-based tiers.[5] Google Cloud Storage uses a similar region/usage-based pricing scheme, so calculating overall costs per transcription is relatively straightforward.[6]

Cloud Run costs are based on usage and region, with higher CPU and memory costs for some regions. Charges are applied based on monthly CPU usage, memory, and requests. Cloud Run supports both request-only CPU allocation, and always-on allocation; but this service only uses the cheaper request-only allocation, and so we disregard the higher always-on costs here. Conversely, there is also a discount for usage commitments, but due to low expected usage, we ignore them too. Finally, we assume consistent usage over a long period of time, and so we discount the time-limited free tier's impact on the service's financial viability.

Table 1 – Cloud Run costs by region tier.[5]

Tier	$ / vCPU-second	$ / RAM GiB-second	$ / million requests
1	$0.0000240	$0.0000025	$0.40

2	$0.0000336	$0.0000035	$0.40
Average	$0.0000288	$0.0000030	$0.40

As laid out in Table 1, the costs of tier 2 regions are significantly higher in the main category we care about; if CPU costs are raised by 50%, the CPU-heavy ML model provided by Omnizart will incur a much higher portion of overall costs. Tier 2 regions should therefore be avoided for production deployment if possible, but since most transcription jobs are quick, the increased costs would still be manageable. South America, Australia, and most of Asia are uncovered by Google's tier 1 regions, so our viability assessment includes tier 2 pricing despite the higher costs.[5] The expanded user-base is anticipated to compensate for the higher processing costs, especially since those costs will only be incurred by actual users – a low user population in tier 2 regions would not negatively impact overall viability.

Cloud Storage bills monthly for data stored, as well as for operations on that data. Several different object lifetimes are available, but this service's data is all ephemeral and can be safely deleted after a transcription's completion – we therefore rely on Standard storage, to avoid the more expensive operations of longer-term storage classes. The per-region pricing varies considerably more than Cloud Run, so we'll use an global average for the cost of storing one gigabyte for a month; $0.02268.[6] Alongside that, Google Cloud charges operation costs: $0.005 per 1,000 non-read operations, and $0.0004 per 1,000 reads.[6] Finally, a cost of $0.01/GB is applied to transfer from a Cloud Storage bucket to a destination within Google Cloud, such as Cloud Run.[6]

In all, the costs of Google Cloud are predictable, and scale smoothly with user count. Since these are the only financial costs of deploying the service, we can move on to discuss how these costs will be assessed in practice; i.e., how our service uses Google Cloud and incurs bills.

2.2 – Expected usage patterns

Musicians don't generally play enormous repertoires, simply because it takes a lot of time and energy to learn new pieces.[8] They therefore don't need to transcribe huge chunks of data at once, which is incredibly convenient for our service's usage patterns (and therefore costs). Instead, musicians tend to focus on a few pieces at once, and so we can plan for individual users to repeat their usage infrequently. In practice, this means that the service's usage costs would scale relatively linearly with the number of users, allowing it to accrue a large user-base without ballooning costs.

When a user transcribes an audio file, they trigger a series of Google Cloud actions via our REST API. That workflow, and those actions, are detailed in Section 4.2; in summary, there are 5 writes and 5 reads per transcription. These will incur $0.027 per 1,000 transcriptions.

The Cloud Run costs are less straightforward to calculate! Since we only use one Cloud Run request per transcription (to start Omnizart), the request fee is only $0.0004 / 1,000. When we assume an average 20 seconds per transcription and an even spread across tier 1 and 2 regions, we arrive at $0.6204 per 1,000 transcriptions. This is over 20x the Storage costs, and obviously worth reducing if possible; so if we exclude tier 2 regions from the deployment, the cost drops by 14.5%, to $0.53. This might be worthwhile if tier 1 regions would sufficiently cover the desired user-base, but that seems unlikely to us – there are plenty of musicians in "tier 2" regions! Further improvements to the wrapper script and other performance options are discussed in Section 3.3.

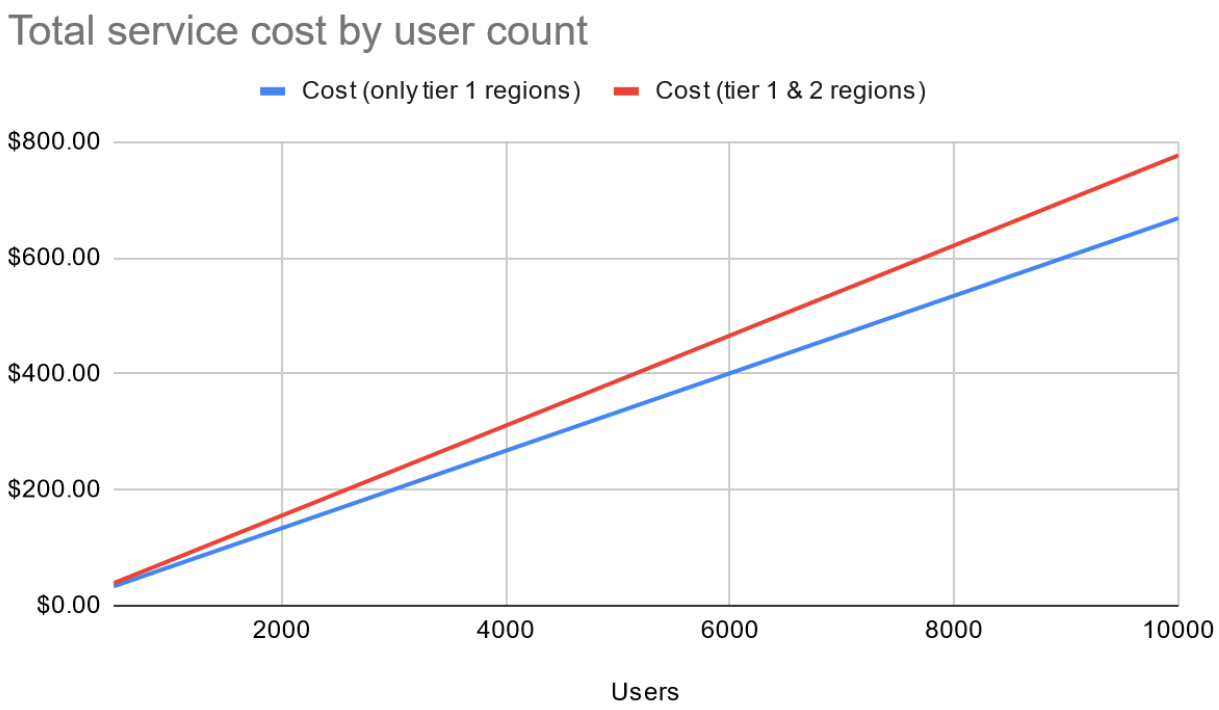

Figure 1 – Total service cost by user count and region tier.

The final cost, accounting for both Cloud Run and Storage usage, is between $0.557 and $0.6474 per 1,000 transcriptions. Since this is constant, and the total cost therefore scales linearly with the number of users / operations, we can refer to Figure 1 to determine the overall costs at a given user count. For instance, the difference between exclusively using tier 1 regions and using both tiers 1 and 2 is very, very small at low user counts. As the user count scales, the cost difference does increase, but the vast user-base made available by that increase in (total, not even per-user) cost makes the extra deployments worthwhile.

2.3 – Projections and viability assessment

The costs of running the service are very low, and operating without any database or other always-on infrastructure allows the costs to be tied directly to usage. This means that the

service can ramp up its user-base for an extended period of time while remaining financially viable. For example, 1,000 regular users each transcribing 10 audios per month would only incur $6, and the entire population of Earth doing the same would result in a mere $540,000 per year! This is eminently affordable (for an organization with a global user-base), and proves that the service is quite viable from a financial perspective. There are a variety of possibilities for funding such a low amount, including ads, subscriptions, or donations; the relatively small sums involved allow great flexibility in determining the best sources of funding.

2.4 – Costs and financial viability summary

Running transcriptions via the service incurs several costs from Google Cloud, and these costs will vary based on region, usage, and Omnizart's performance. Nevertheless, we've assembled a reasonable estimate of the long-term costs of deploying the service on Google Cloud. The expected range of $0.557 to $0.6474 per 1,000 transcriptions appears very reasonable – such a small, linearly scaling cost should pose little difficulty when obtaining funding.

3 – Omnizart on Google Cloud

Omnizart is an open-source Python library, providing wrappers and utilities for a machine-learning model which is purpose-built for automatic music transcription.[4] The authors of Omnizart have provided a Dockerfile to wrap the library in a container, and we extend and deploy that container on Google Cloud Run. When a user starts a transcription, a new container is started with parameters for that specific transcription. Each Omnizart instance is therefore ephemeral, and no data is stored beyond the transcribed output.

3.1 – Jobs and alternatives

Google Cloud Run has several options for executing arbitrary Docker containers in production. A Service in Cloud Run allows a container to be run as a web server, always remaining available to handle requests. This functionality (and the accompanying auto-scaling) is great – and totally unnecessary for our service. Luckily, a recent new offering called Cloud Run Jobs is available in beta, allowing us to execute containers as simple one-off tasks.[9] The costs are identical to Services, and the main downside is a maximum runtime of 10 minutes.[10] During testing, we never observed any transcription job's duration approach that limit, and so we expect Jobs to serve our purpose admirably despite its limitations.

Jobs have another quirk; you cannot pass parameters into them at execution-time.[9] You can only set parameters when creating the Job, not when invoking it. It's designed to handle huge parallel batch jobs, but that's basically our use case – but at a smaller scale! We work around this minor limitation by creating a new Job for every transcription, invoking it, and subsequently deleting it. Since there are no financial costs associated with creating new Jobs, this method affords us the best of both worlds; minimal cost alongside near-instantaneous container creation and fast run times.

3.2 – Docker

If you're not familiar with Docker, it's fantastic - you can run code without installing dependencies on the server, and consequently you can quickly spin up new instances (containers) as needed.[7] Both of these traits are excellent for our use case; Omnizart has a large dependency tree, including Tensorflow, as well as even larger datasets for its machine learning models.[4] Using Docker allows interoperability with the majority of cloud providers which support it, and flexibility for local development – developers don't need to install Omnizart, which would lead to (slightly) increased participation from an open-source community.

3.3 – Improving Omnizart's performance

The principal cause of Omnizart's occasional sluggishness is the data – it loads large checkpoints to back its machine learning, and these add significant delay. We effectively download them freshly every time a transcription is run, since Cloud Run Jobs don't cache Docker images for us. The obvious solution is to keep some servers always available, with Omnizart pre-downloaded and ready to transcribe – but this incurs considerable cost, with no gain from a small user-base. The existing system design of instantiating a new Omnizart container for each transcription is, financially, the least scalable part of the system. From a pure software perspective, it'd scale wonderfully (as fast as Google allows it!), but the same number of transcriptions could be served by a smaller number of always-on worker nodes.

Determining the value (and therefore priority) of this change is beyond the scope of this paper, as it depends very heavily on actual usage data. To be clear, the existing system is ideal for our particular case of very little funding and an unfulfilled desire for technical scalability. But if the system were to scale suddenly beyond expectations, the unnecessarily-increasing costs would likely be the largest burden, along with the impact to Omnizart's run time. Further improvements to the backing machine learning layer would require considerable expert analysis, as AMT is a complex and cavernous topic.

S3 Summary

Google Cloud is an ideal host for Omnizart, particularly given the new Jobs feature and our transient use of them. Some limitations exist, but none affect our transcription service in a meaningful way. Utilizing Docker as a wrapper for Omnizart allows it to remain portable, and smooths the developer experience. This does introduce some performance concerns, particularly around image size and re-downloading unnecessary data, but we deem these tradeoffs worthwhile at small scale, and particularly with low funding.

4 – Public access to Omnizart

S4 Introduction

The goal of this service is, of course, to provide the greatest access possible to Omnizart; i.e., democratizing automatic music transcription. This goal is not achievable if we

were to only use Google Cloud's services out-of-the-box. Therefore, we describe an implementation of a REST API which allows public users to perform transcriptions with Omnizart, and a corresponding basic web interface for laypeople to use the service with minimal effort. These services are also hosted on Google Cloud, but our primary interest in discussing them is their direct interactions with Omnizart and relevant Google Cloud services; their actual hosting is fairly standard.

4.1 – REST API flow

The REST API provides the frontend with a consistent, HTTP-friendly interface to Omnizart. It is implemented in TypeScript, and runs on Node.JS. Operating without authentication or frameworks, it's certainly a bare implementation, but it does its job of facilitating access to the transcriptions. To begin, the user (via a browser) sends a GET request to `/upload-url`, which replies with a URL and a transcription ID. The browser can then PUT an audio file to this URL, and POST the transcription ID to `/transcription`. This starts a transcription job (using the uploaded audio file), and replies with a status.

Once the transcription starts, the browser uses the ID to poll for a completed status at `/transcription/:id`. Once the transcription is completed, that endpoint will return a URL to the transcription, in the form of a MIDI file – which the user can download and view, or convert to another format.

4.2 – Interactions with Omnizart

The URLs referenced in Section 4.1 aren't magical, or black boxes – they're signed Cloud Storage URLs.[11] These allow the REST API to pre-authenticate URLs to Cloud Storage objects ("blobs"), whether to allow the browser to upload directly or download directly. When the user requests an upload URL, a random ID is generated. Using that ID, a Cloud Storage path is generated, and a signed URL to PUT to that path is returned to the user. Then, when the user starts a transcription with `POST /transcription`, the API can recreate the upload URL with the provided transcription ID.

Omnizart doesn't have any native concept of Google Cloud, or even HTTP – its sole interfaces are command-line, and Python. Our Dockerized wrapper around Omnizart (written in Python) uses Google Cloud's Python SDK to pull user-uploaded audio files to the local filesystem. It then runs a transcription using Omnizart, which also outputs (a MIDI file) to the local filesystem. Finally, we upload the MIDI file to Google Cloud Storage, and mark the job complete.

4.3 – Improving the interface

The job's status (and other metadata) is currently tracked by ID-named JSON files in Google Cloud Storage. This is, as previously noted, not atomic. However, for our purposes, this shouldn't be a problem – only one worker node can/should operate on a given audio file at once. The only edge case would be when a polling read fails to read the correct status during a

simultaneous write; but failure in this case would be an inaccurate “in progress” status, and a repoll several seconds later with the correct result. State and data management could be moved to a relational database or other, more structured store – but as our pattern goes, an always-on database would incur constant costs unrelated to the user count, which is inappropriate for our present use case.

The other point of probable improvement is the interface between API and Omnizart – they currently communicate via an architecturally brittle combination of Google Cloud Storage writes and Google Cloud Run Job parameters. A more ideal implementation would communicate via an event-based system, or a client-server model like HTTP, or even a combination such as RPC. The main downside of the current implementation is developer experience and understanding, and so it’s an acceptable flaw in a system with few developers and work. However, it would likely be a priority refactor for any team aiming to mainstream the service.

4.3 – Summary

The layers required to provide the public web access to Omnizart are complex, but they boil down to a simple flow from user to REST API to Omnizart, and back again. Owing to time and budgetary constraints, there are major improvements to the system’s financial scalability and ease of development on the horizon but not quite implemented. We described these in some detail in order to aid future development, since the goal of this paper is to not remain a paper!

Summary

The service we’ve described will support non-technical musicians in their endeavors to play music without access to the sheet music (“source code”). Omnizart allows flexible transcription of monophonic audio into MIDI files – our web interface & REST API allow a casual user to perform transcription with minimal effort. The costs of the system scale linearly with usage, so the operator will not pay excessive hosting bills without commiserate user activity. This provides a long ramp for the business to monetize its operations efficiently, without frantic periods of overspending without any returns. The system’s performance scales well given the constraints of budget and time, but this would be the first point of improvement for future development.

Appendix

AMT: Automatic music transcription
API: Application programming interface
HTTP: Hypertext transfer protocol (PUT, POST, and GET are HTTP verbs)
JSON: Javascript Object Notation
MIDI: Musical instrument digital interface

ML: Machine learning
REST: Representational state transfer

References

AUTHORS. TITLE. PUBLISHER. DATE. PAGE COUNT. URL.

[1] Moulton, Calum. “Perfect pitch reconsidered.” Royal College of Physicians. Oct 2014. 3 pages. https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4951961/

[2] Van Hedger, Stephen C., et al. “Auditory working memory predicts individual differences in absolute pitch learning.” Cognition. Jul 2015. 15 pages. https://pubmed.ncbi.nlm.nih.gov/25909580/

[3] Holzapfel, Andre; Benetos, Emmanouil. “Automatic music transcription and ethnomusicology: a user study.” International Society for Music Information Retrieval. 2019. 7 pages. https://www.diva-portal.org/smash/get/diva2:1474663/FULLTEXT01.pdf

[4] Wu, Y.-T., et al. “Omnizart: A general toolbox for automatic music transcription.” Journal of Open Source Software. Dec 12 2021. 5 pages. https://joss.theoj.org/papers/10.21105/joss.03391

[5] “Cloud Run Pricing.” Google Cloud Run. https://cloud.google.com/run/pricing

[6] “Cloud Storage Pricing.” Google Cloud Storage. https://cloud.google.com/storage/pricing

[7] “Docker Overview.” https://docs.docker.com/get-started/

[8] Zhukov, Katie. “Effective practising: A research perspective.” Australian Journal of Music Education. 2009. 10 pages. https://files.eric.ed.gov/fulltext/EJ912405.pdf

[9] “Create Jobs.” Google Cloud Run. https://cloud.google.com/run/docs/create-jobs

[10] “Cloud Run Quotas and Limits.” Google Cloud Run. https://cloud.google.com/run/quotas

[11] “Signed URLs.” Google Cloud Storage. https://cloud.google.com/storage/docs/access-control/signed-urls

[12] “google-cloud.” Python Package Index. July 2018. https://pypi.org/project/google-cloud/