A DevOps Manifesto for Speech Corpus Management

Ingmar Steiner
Saarland University & DFKI GmbH
steiner@coli.uni-saarland.de

Abstract: In this paper, we introduce certain concepts from the DevOps philosophy, and more generally from the software development lifecycle. We argue that the separation between source code and how it is built and released for distribution can be applied to speech corpora as well. We draw a distinction between the developers and maintainers of a speech corpus on one hand, and the researchers who use it on the other. We propose conventions to efficiently manage corpus metadata like source code, and speech data like static assets that can be retrieved automatically. Finally, we mention several use cases which illustrate the merits of these conventions.

1 Introduction

DevOps (from "development and operations") comes from the field of software development, and means different things to different people. Jabbari et al. [1] conducted an empirical study across the scientific literature and found a recurring set of concepts, principles, and philosophies which form the common ground of what DevOps actually is. Unfortunately, the term has also become a buzzword of sorts in the software industry, and as such has been distorted and hyped, but nevertheless, its core principles include, (a) an emphasis on communication and collaboration; (b) efficiency and quality through automated testing and continuous delivery; and (c) integration of tools and infrastructure into workflows. Some of the principles and ideas in DevOps are also associated with other movements in software development, but DevOps is permissive and practical, rather than dogmatic, and so we are free to borrow and adapt its ideas.

How is this relevant for us? Unlike enterprise programmers, we researchers tend to work in small groups or alone, on several projects at a time, but nevertheless, many of us already use one or more of the tools and ideas that DevOps promotes. Those of us who use source code management (SCM) tools (e.g., Subversion[1] or Git[2]) to track the development of software or experiment recipes might use automatic testing to validate results and guard against bugs; we might also use SCM to manage the LaTeX sources of papers we write together with our colleagues.

But how can a DevOps culture contribute to *speech corpus management*? A conventional speech corpus tends to come in the form of one or more compressed archives (e.g., zip files), downloaded or copied by hand from some webpage or network share. The archives typically contain hundreds or thousands of individual audio files, as well as equally many metadata files (phonetic annotations, etc.), and perhaps other data as well. The structure of these packages is often arbitrary, the annotations are in one or several ad-hoc formats, and in order to start using such a corpus for experiments or other research, it can take hours or even days of converting data from one format to another, rearranging directories, and – perhaps – writing scripts to automate some or all of these processes. Needless to say, these can be tedious activities, and more often

[1] `https://subversion.apache.org/`
[2] `https://git-scm.com/`

than not, mistakes are made that can lead to serious problems later, when the corpus data is finally put to use.

Incidentally, the corpus itself might not be error-free. Annotations could contain typos or misalignments, or the recordings themselves might have problems. Those in the research community who have worked with a given corpus are bound to have uncovered such mistakes or idiosyncrasies, but there is rarely a public platform to discuss them and share solutions or patches. Often enough, the creators of a given corpus have long moved on to different institutions, and there is no central entity to maintain it, fix mistakes, and publish updated versions. There is a growing awareness of these issues in the speech community; Rosenberg [2] highlights some of them, and argues for SCM for metadata as a solution.

This paper goes several steps further. We distinguish between the raw data (analogous to the "source" of the corpus) which can be collaboratively maintained using SCM, and the corpus *release*, automatically packaged for distribution in a "ready-to-use" form, so that the latest version is always available. Ideally, a ready-to-use speech corpus should be,

portable using standard container formats supported by open software programs on any operating system;

efficient compressed to save space, using a *free*, *lossless* codec for audio data, and distributed in a small number of files which support streaming and seeking;

self-documenting including all relevant metadata, synchronized with the data stream(s), and metadata structured in a format that is both human-readable and easy to process automatically.

A corpus in such a form can be used efficiently as a *dependency* of a given project or experiment. By leveraging DevOps concepts and tools designed for efficient resolution and retrieval of dependencies from software repositories, we can obtain, cache, and process speech corpora automatically. Moreover, generic software libraries and scripts can be applied and used without having to reinvent the wheel – these themselves should also be portable and openly developed.

As a case study, we will take the MOCHA-TIMIT corpus [3], which has been used by hundreds of researchers in the past 15 years, all of whom have had to familiarize themselves with its custom data format and structure. We demonstrate how to build and publish a *distribution* of the corpus, and present a portable sample project that uses this distribution for a simple articulatory analysis experiment.

2 Software analogy

In order to reflect on the lifecycle of speech corpus management, an analogy to software development can provide useful insight.

A software library or application can be created and maintained by one or more developers, who may well use SCM to track changes in the source code and collaborate; for open-source software (OSS), the source code and peripheral services (such as an issue tracker or project website) may be provided online by code hosting websites such as GitHub[3] or SourceForge[4].

The development process consists of adding new features, fixing problems, and so forth, and every so often, the developers will consider the current state of their software "stable" enough to warrant an explicit publication. This is commonly referred to as a *release* package,

[3] `https://github.com/`

[4] `https://sourceforge.net/`

and is associated with a *version* number. The purpose of the version number is to allow reference to a known, well-defined state of the software, and to indicate an explicit order in the version history. It is then possible to make statements such as, "this application depends on version 1.1 of that library", "that feature was introduced in version 1.2" or "the problem inadvertently introduced in version 0.8 was fixed in 0.8.1", with the understanding that the feature in question cannot be used in a version older than v1.2, or that people using v0.8 who are affected by the problem should use v0.8.1 (or newer) instead. Version numbers are incremental, and conventions such as *semantic versioning* place additional significance on the fields of a version number.[5]

It is considered standard practice for the developers to write "release notes" or to maintain a "change log", published with each release package, making it straightforward to understand the impact of the new release, and how it differs from, and improves over, the previous versions.[6]

Developers of the software may make use of various techniques to facilitate their work, including *continuous integration testing*, or automatically publishing "snapshot", developer versions, sometimes on a daily (or "nightly") basis; this allows consumers of their software to try out the latest development changes long before a new stable version is released (and conceivably provide feedback, which will help make the software more stable ahead of the release).

2.1 Developers vs. users

A crucial distinction is that between the *developers* one one hand, who are expected to have experience with software engineering and understand not just the source code they work on, but also the lifecycle from one release to the next, and the *users* on the other. Users are the consumers of the software who need it for some purpose or other, and who (hopefully) understand *how* to use it, but are not – and should not need to be – familiar with the source code or how it is used to prepare a release package. Crucially, users should not be forced to use SCM or work directly with the source code, but to obtain and use a released version.

Having said that, in an open-source model, nothing prevents users from inspecting the source code, and (assuming they possess the requisite competence) fixing problems or even implementing enhancements. If they provide their modifications to the developers and they include them in the "official" source code, such users become *contributors* to the software, and might even at some point join the developer team.

2.2 Source code vs. distribution

Depending on the nature of the source code, there tend to be significant differences between the source code and the form and contents of a packaged release. Programming languages such as C++ or Java are *compiled* languages, and the source code must be converted to a machine-readable form using a compiler, before it can be used. There may also be testing code used only for verifying that the main source code is free of errors, and such code is important during the development lifecycle, but not to the users. There could also be peripheral material, such as developer documentation, that forms part of the source code, but is useless to users and therefore not included in the released versions.

The development lifecycle can includes steps to retrieve software dependencies, compile the source code, process resources, run tests and report the outcome, package the relevant contents, and publish the package to one or more repositories or websites. All of this should be done in a workflow that is well documented and can be streamlined using *build automation*, to

[5] `http://semver.org/`

[6] `http://keepachangelog.com/`

make this process as efficient as possible. Many build automation tools are available – some more flexible and efficient than others – e.g., GNU Make[7], SCons[8], and Gradle[9].

A published release package can be *distributed* to users as a downloadable file on one or more websites (including the hosting website itself), or through a package "repository" that follows certain conventions to streamline the process of obtaining and installing the package on a user's device; these distribution workflows can themselves be managed by software platforms, e.g. Apt[10] (used in many Linux installations), NuGet[11] (used in Microsoft Windows), or various "app marketplaces" (such as Apple's App Store, Google Play, and so on).

3 Speech corpora

We have described key concepts in software development, and we will now argue that speech corpora can adopt similar conventions to turn from static, monolithic heaps of data into projects that can be actively and collaboratively maintained and improved, and distributed as efficiently as possible.

3.1 Annotations as source code

Speech data in a corpus is annotated using a variety of conventions, most prominently timestamped metadata such as transcriptions, phonetic segment boundaries, and labels. These annotations can take the form of text files, which allows them to be stored and tracked by SCM tools in much the same way as source code files.[12]

Researchers should of course be free to work with speech corpora, using whichever tools they prefer, from ESPS and WaveSurver[13] to Praat[14] and ELAN[15]. However, many of the corresponding file formats have design limitations, such as a lack of explicit hierarchical structure for annotations at different linguistic levels; for the sake of sustainability and flexibility, it is therefore recommended to *also* provide the metadata in self-documenting, established serialization formats (XML[16], JSON[17], or YAML[18]), which are human-readable and easy to parse into well-defined data structures using any major programming language.

3.2 Speech data as resources

In order to faithfully preserve all information stored in the acoustic signal, speech data should never be compressed in a corpus using a lossy codec [4]. While an uncompressed pulse-code modulation (PCM) sample encoding consumes a significant amount of storage space and transfer bandwidth, there are OSS implementations of lossless audio codecs, notably FLAC[19], which

[7] `https://www.gnu.org/software/make/`
[8] `http://scons.org/`
[9] `https://gradle.org/`
[10] `https://wiki.debian.org/Apt`
[11] `https://www.nuget.org/`
[12] A notable catch involves Praat's TextGrid format, which explicitly numbers each interval. This means that simply inserting a boundary can cause hundreds of lines to change, obscuring the nature of the modification. Fortunately, this issue can be avoided by using the alternative "chronological" TextGrid format.
[13] `http://www.speech.kth.se/wavesurfer/`
[14] `http://praat.org/`
[15] `https://tla.mpi.nl/tools/tla-tools/elan/`
[16] `https://www.w3.org/TR/xml/`
[17] `http://www.json.org/`
[18] `http://yaml.org/`
[19] `https://xiph.org/flac/`

is free and widely supported, as well as others, such as WavPack[20] or ALAC[21].

One critical issue must be discussed regarding the storage of speech data using SCM such as Git. Most SCM tools are designed to manage files consisting of (preferably short) lines of text. Acoustic signals on the other hand are stored in a binary format, and consume several orders of magnitude more storage space than their text-based metadata. This presents a problem for SCM, as speech data is (a) inefficient to store and transfer with SCM operations, and (b) meaningless to compare at the binary level against a reference when modified.

Some SCM tools have attempted to work around these limitations and enable the management of large binary assets such as speech data; for example, several extensions to Git (such as git-annex[22] or Git LFS[23]) allow developers to track such data by introducing a parallel remote storage system and transport layers to retrieve such the corresponding files, without "bloating" the Git object store itself.

However it is important to realize that the audio data in a speech corpus will typically not change much after the initial recording sessions that created it. This aspect makes it possible to store the speech data *separately* from the corpus source code, and to retrieve them as resources as required, from one or more remote storage locations, without tracking them explicitly.

3.3 Corpus development lifecycle

With this in mind, researchers or developers working with speech corpora can be described as "users" of these corpora. If they enhance the metadata by creating new and useful annotations, or correct any errors they find during their work, they can become contributors by providing their modifications back to the corpus developers. This way, the corpus can be improved over time, and new versions can be released.

Developers and expert users of a corpus can use build automation tools to retrieve and package a corpus for distribution, and the resulting release can be published to a remote website or repository. But normal users should not be required to use SCM to obtain the full source code or data, or to "build" the corpus in order to use it; they can simply download a given version from the internet in the most convenient and efficient format.

3.4 Media containers

Audio data is nearly always provided in some file format, such as RIFF Wave for PCM data, which acts as a *container* and describes the sample encoding, and facilitates the playback, seeking, and streaming of the audio. The more complex the data, the more important it is to choose a suitable container format. FLAC-encoded data, for instance, can use a lightweight FLAC container, but this format and its available metadata has certain limitations.

Where multimodal data, such as articulatory motion capture or video are available, a more flexible media container format must be chosen that allows synchronized access to all streams. Of course the container format itself should also be free to use and widely supported by OSS; these conditions are met, for instance, by Ogg[24] and Matroska[25]. Moreover, most modern web browsers and media players (such as VLC[26]) can stream data from such containers without requiring remote files to be downloaded first; this makes it much easier to casually "preview"

[20] `http://wavpack.com/`
[21] `https://macosforge.github.io/alac/`
[22] `https://git-annex.branchable.com/`
[23] `https://git-lfs.github.com/`
[24] `https://www.xiph.org/ogg/`
[25] `https://www.matroska.org/`
[26] `http://www.videolan.org/vlc/`

an available speech corpus and evaluate its suitability for a given scenario.

4 Case study

We use FLAC resources to efficiently store the speech data, and YAML for the corresponding metadata, in a number of speech corpora used for text to speech (TTS) synthesis voices, including the PAVOQUE corpus of expressive speech[27] and the SEMAINE TTS databases[28]; an example of the metadata in this format is shown in Figure 1.

For flexibility and efficiency in the development cycle and use of these corpora, we use this FLAC+YAML convention, and we provide a portable, lightweight implementation of this convention in the form of a Gradle plugin.[29] Using this plugin, the data is retrieved from the hosting website (which is declared as a custom repository), cached locally, and processed to package or convert the data and metadata into ESPS label files or Praat TextGrids. The corpora are also declared as data dependencies for the corresponding TTS voices[30], which integrates the speech corpus management into the continuous delivery workflow for synthesis voices in MaryTTS[31].

A more challenging use case is presented by multimodal speech corpora such as the MOCHA-TIMIT corpus [3], which contains data recorded with electromagnetic articulography (EMA), electroglottography (EGG), and electropalatography (EPG), as well as audio recordings from a number of English speakers. The audio and EGG signals can be compressed with FLAC, but with the many channels of EMA (sampled at a different rate), we have to use the more flexible Matroska container to store synchronized streams from all modalities in a single file. The range of sample values for the EMA, however presents a challenge; most audio codecs will clip values outside the $[-1,1]$ range, but we can rescale each channel and explicitly store the range to restore the original data during analysis.

Another interesting detail is the fact that one project [5] has used two of the MOCHA corpus subsets, discovered errors in the metadata, and published corrected files; we can use custom tasks to retrieve and merge these corrections as part of our automated build, producing a new version of the corpus subset. The entire workflow has been published online[32], and a simple experiment using the packaged corpus is available as well.[33].

5 Conclusion

We have discussed the DevOps approach, and some of the main parts of the software development lifecycle. We argue by analogy that speech corpora can be managed like software projects. Large binary assets that change rarely can be stored in remote repositories and retrieved automatically, while metadata, which is textual in nature and could be modified more frequently, can be managed like source code.

Moreover, we draw distinctions between developers and users of a corpus, and between the source data and published versions of distribution packages, which can be automatically deployed and retrieved. Finally, we demonstrate that these conventions can be used to streamline

[27] https://github.com/marytts/pavoque-data

[28] https://github.com/marytts/dfki-semaine-data

[29] https://github.com/m2ci-msp/gradle-flaml-plugin

[30] e.g., https://github.com/marytts/voice-dfki-pavoque-styles or https://github.com/marytts/voice-dfki-spike

[31] http://mary.dfki.de/

[32] https://github.com/m2ci-msp/mocha-msak0-data

[33] https://github.com/m2ci-msp/mocha-msak0-demo

```
- prompt: spike0008
  text: Ach ja?
  style: angry
  start: 27.0
  end: 28.92
  segments:
  - {lab: H#, end: 0.280902}
  - {lab: '?', end: 0.324898}
  - {lab: a, end: 0.408238}
  - {lab: x, end: 0.475}
  - {lab: j, end: 0.61}
  - {lab: 'a:', end: 0.963273}
  - {lab: _, end: 1.915}
```

Figure 1 – An example of metadata for one utterance from the PAVOQUE corpus in YAML format.

speech corpus management, and to model dependencies on specific versions of a corpus for efficient, reproducible research.

References

[1] JABBARI, R., N. BIN ALI, K. PETERSEN, and B. TANVEER: *What is DevOps?* In *17th International Conference on Agile Software Development*. Edinburgh, UK, 2016. doi:10.1145/2962695.2962707.

[2] ROSENBERG, A.: *Rethinking the corpus: Moving towards dynamic linguistic resources*. In *Interspeech*, pp. 1392–1395. Portland, OR, USA, 2012. URL `http://www.isca-speech.org/archive/interspeech_2012/i12_1392.html`.

[3] WRENCH, A. A.: *A multi-channel/multi-speaker articulatory database for continuous speech recognition research. PHONUS*, 5, pp. 1–13, 2000. URL `http://www.coli.uni-saarland.de/groups/WB/Phonetics/contents/phonus-pdf/phonus5/Wrench_PHONUS5.pdf`.

[4] SIEGERT, I., A. F. LOTZ, L. L. DUONG, and A. WENDEMUTH: *Measuring the impact of audio compression on the spectral quality of speech data*. In *27th Conference on Electronic Speech Signal Processing (ESSV)*, pp. 229–236. Leipzig, Germany, 2016.

[5] JACKSON, P. J. B. and V. D. SINGAMPALLI: *Statistical identification of articulation constraints in the production of speech. Speech Communication*, 51(8), pp. 695–710, 2009. doi:10.1016/j.specom.2009.03.007.