Developing without developers: choosing labor-saving tools for language documentation apps

Luke D. Gessler
Department of Linguistics
Georgetown University
lg876@georgetown.edu

Abstract

Application software has the potential to greatly reduce the amount of human labor needed in common language documentation tasks. But despite great advances in the maturity of tools available for apps, language documentation apps have not attained their full potential, and language documentation projects are forgoing apps in favor of less specialized tools like paper and spreadsheets. We argue that this is due to the scarcity of software development labor in language documentation, and that a careful choice of software development tools could make up for this labor shortage by increasing developer productivity. We demonstrate the benefits of strategic tool choice by reimplementing a subset of the popular linguistic annotation app ELAN using tools carefully selected for their potential to minimize developer labor.

1 Introduction

In many domains like medicine and finance, application software has dramatically increased productivity, allowing fewer people to get more done. This kind of labor reduction is sorely needed in language documentation, where there is not nearly enough labor to meet all the demand in the world for language documentation.

There is every reason to think that application software ("apps") could also help language documentation (LD) practitioners get their work done more quickly. But despite the availability of several LD apps, many practitioners still choose to use paper, spreadsheets, or other generic tools (Thieberger, 2016). Why aren't practitioners using apps?

1.1 Perennial problems in LD apps

The simplest explanation is that the apps are not helpful enough to justify the cost of learning and adjusting to them. While it is clear by now from progress in natural language processing that it is technically possible to make laborious tasks such as glossing and lexicon management less time-consuming by orders of magnitude, flashy features like these turn out to be only one part of the practitioner's decision to adopt an app. It seems that other factors, often more mundane but no less important, can and often do nullify or outweigh these benefits in today's apps. Three kinds of problems can be distinguished which are especially important for LD apps.

First, there is the unavoidable fact that the user will need to make unwanted changes to their workflow to work within an app. By virtue of its structure, an app will always make some tasks prerequisites for others, or at least make some sequences of tasks less easy to perform than others. Ideally, an app designer will be successful in identifying the task-sequences that are most likely to be preferred by their user and ensure that these workflows are supported and usable within the app. But no matter how clever they are, their app will always push some users away from their preferred workflow. For app adoption, this presents both a barrier to entry and a continuous cost (if a user is never able to fully adapt themselves to the app's structure).

Second, app developers often make simplifying assumptions which sacrifice generality for development speed. For instance, an app's morphological parser might not be capable of recognizing diacritics as lexically meaningful, which would limit the app's usability, e.g. for the many languages which represent lexical tone with diacritics.

Third, an app might lack a feature altogether. An app might only work on a particular operating system like Windows, or might not work without an internet connection, both of which might single-handedly make an app unusable. In another vein, it might be critical for a project to have texts

be annotated with a syntactic parse, or for all texts to be exportable into a particular format, like as an ELAN (Wittenburg et al., 2006) or FLEx (Moe, 2008) project. If a user cannot perform a crucial kind of linguistic annotation, or move their data into another app which is critically important for them, they are likely to not use the app at all, no matter how good it might be at segmenting text into tokens, morphologically parsing words, or managing a lexical inventory. Indeed, the features which might be absolutely necessary for any given practitioner are in principle at least as numerous as the number of formalisms and levels of analysis used by all linguists, and it is to be expected that an app will be inadequate for a great number of potential users.

1.2 The need for more developer labor

Whatever the exact proportions of these problems in explaining the under-adoption of apps, it seems clear that they all could be solved relatively straightforwardly with more software engineering labor. Much of the hard work of inventing the linguistic theories and algorithms that would enable an advanced, labor-saving LD app has already been done. What remains is the comparatively mundane task of laying a solid foundation of everyday software on which we might establish these powerful devices, and the problem is that in LD, software engineering labor is—as in most places in science and the academy—markedly scarce.

With a few notable exceptions[1], LD apps have largely been made by people (often graduate students) who have undertaken the project on top of the demands of their full time occupation. Considering how in industry entire teams of highly-skilled full-time software engineers with ample funding regularly fail to create successful apps, it is impressive that LD apps have attained as much success as they have. Nevertheless, most LD software will continue to be produced only in such arid conditions, and so the question becomes one of whether and how it might be possible to create and maintain a successful app even when the only developers will likely be people like graduate students, who must balance the development work with their other duties.

While pessimism here would be understandable, we believe it is possible to create an app that is good enough to be worth using for most practitioners even under these circumstances. While there is still not much that is known with certainty about productivity factors in software engineering (Wagner and Ruhe, 2018), many prominent software engineers believe that development time can be affected dramatically by the tools that are used in the creation of an app (Graham, 2004).

Apps depend on existing software libraries, and these libraries differ in dramatic ways: some databases are designed for speed, and others are designed for tolerating being offline; some user interface frameworks are designed for freedom and extensibility, and others double down on opinionated defaults that are good enough most of the time; some programming languages are always running towards the cutting edge of programming language research, and others aim to be eminently practical and stable for years to come. It is obvious that these trade-offs might have great consequences for their users: a given task might take a month with one set of tools, and only a few days with another.

We should expect, then, that the right combination of tools could allow LD app developers to be much more productive. If this is true, then if only we could choose the right set of tools to work with, we might be able to overcome the inherent lack of development labor available for LD apps.

2 Cloning ELAN

To put this hypothesis to the test, we recreated the rudiments of an app commonly used for LD, ELAN (Wittenburg et al., 2006). Choosing an existing app obviated the design process, saving time and eliminating a potential confound. ELAN was chosen in particular because of its widespread use in many areas of linguistics, including LD, and because its user interface and underlying data structures are complicated. We reasoned that if our approach were to succeed in this relatively hard case, we could be fairly certain it could succeed in easier ones, as well.

With an eye to economic problems of LD app development outlined in section 1.2, we first determined what features we thought an advanced LD app would need to prioritize in order to make development go quickly without compromising on quality. Then, we chose software libraries in accord with these requirements.

[1] FLEx, ELAN, and Miromaa all have at least one full-time software engineer involved in their development.

2.1 Requirements and tool choices

There were four requirements that seemed most important to prioritize for an LD app. These were: (1) that the app be built for web browsers, (2) that the app work just as well without an internet connection, (3) that expected "maintenance" costs be as low as possible, and (4) that graphical user interface development be minimized and avoided at all costs.

2.1.1 Choice of platform: the browser

15 years ago, the only platform that would have made sense for an LD app was the desktop. These days, there are three: the desktop, the web browser, and the mobile phone. While the mobile phone is ubiquitous and portable, it is constrained by its form factor, making it unergonomic for some kinds of transcription and annotation. The desktop platform has the advantage of not expecting an internet connection, and indeed the most widely used LD apps such as FLEx and ELAN have been on the desktop, but it is somewhat difficult to ensure that a desktop app will be usable on all operating systems, and the requirement that the user install something on their machine can be prohibitively difficult.

The web browser resolves both of these difficulties: no installation is necessary, since using an app is as easy as navigating to a URL, and the problem of operating system support is taken care of by web browser developers rather than app developers.

2.1.2 Offline support: PouchDB

The notable disadvantage of web browsers compared to these other platforms, however, is that the platform more or less assumes that you will have an internet connection. The simple problem is that language documentation practitioners will often not have a stable internet connection, and if an app is unusable without one, they will never use the app at all.

Fortunately, there are libraries that can enable a browser app to be fully functional even without an internet connection. The main reason why a typical browser app needs an internet connection is that the data that needs to be retrieved and changed is stored in a database somewhere on the other end of a network connection. But there are databases that can be installed locally, removing the need for an uninterrupted internet connection.

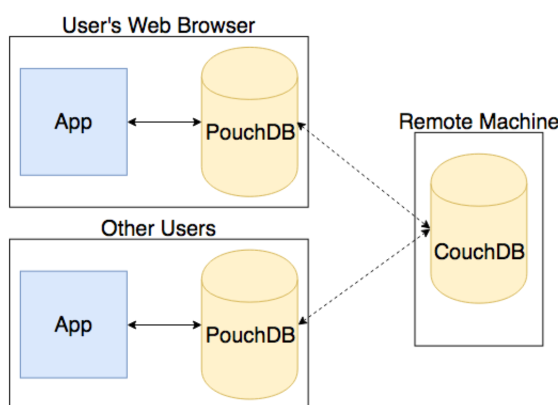

Figure 1: PouchDB imitates a traditional database, but it exists alongside the app in a user's local machine rather than on a remote server. When an internet connection becomes available, PouchDB can sync with the remote server and other clients, meaning that sharing and securely backing up data is still possible.

The most advanced database for this purpose is PouchDB. PouchDB is a browser-based implementation of the database system CouchDB. This means that PouchDB acts just like a normal database would, except it is available locally instead of over a network connection. And when an internet connection does become available, PouchDB is able to share changes with a remote instance of CouchDB, which then makes them available to other collaborators, as seen in figure 1. PouchDB and CouchDB were specially designed to make this kind of operation easy, and retrofitting the same behavior with a more traditional choice of tools would be extremely time consuming.

An added benefit of adopting this database strategy is that it dramatically reduces the need for client-server communication code. Normally, an entire layer of code is necessary to package, transmit, and unpackage data between the client and the server, which most of the time amounts to nothing more than boilerplate. With this approach, all of that is taken care of by the replication protocol which allows PouchDB instances to sync with CouchDB instances, and the server code that is required by a small set of cases (e.g. triggering a notification email) can still exist alongside this architecture.

We are not the first to realize PouchDB's potential for writing an LD app: FieldDB (aka LingSync) (Dunham et al., 2015) uses PouchDB.

2.1.3 Minimizing maintenance: ClojureScript

Software requires some degree of modification as time goes on. Almost all software relies on other software, and when breaking changes occur in a dependency, an app must also have its code modified to ensure that it can continue operating with the latest version of its dependency. For instance, much of the Python community is still migrating their code from Python 2 to Python 3, an error-prone process that has consumed many thousands of developer-hours worldwide.

This kind of maintenance is, at present, quite common for browser apps: web browsers, JavaScript, and JavaScript frameworks are evolving at a quick pace that often requires teams to constantly modify their code and compilers, and sometimes to even abandon a core library altogether in favor of a better one, which incurs a massive cost as the team excises the old library and stitches in the new one. These maintenance costs are best avoided if possible, as they contribute nothing valuable to the user: in the best case, after maintenance, no change is discernible; and if the best case is not achieved, the app breaks.

We surveyed programming languages that can be used for developing browser applications and were impressed by the programming language called ClojureScript. ClojureScript is a Lisp-family functional programming language that compiles to JavaScript. The language is remarkably stable: ClojureScript written in 2009 still looks essentially the same as ClojureScript written in 2018. The same cannot often be said for JavaScript code.

A full discussion of the pros and cons of ClojureScript is beyond the scope of this paper, but it might suffice to say that ClojureScript offers very low maintenance costs and powerful language features at the cost of a strenuous learning process. Unlike Python, Java, or JavaScript, ClojureScript is not object-oriented and does not have ALGOL-style syntax, making it syntactically and semantically unfamiliar. This is a serious penalty, as it may reduce the number of number of people who could contribute to the app's development. However, this may turn out to be a price worth paying, and as we will see in section 2.3.2, this disadvantage can be mitigated somewhat by allowing users and developers to extend the app using more familiar programming languages.

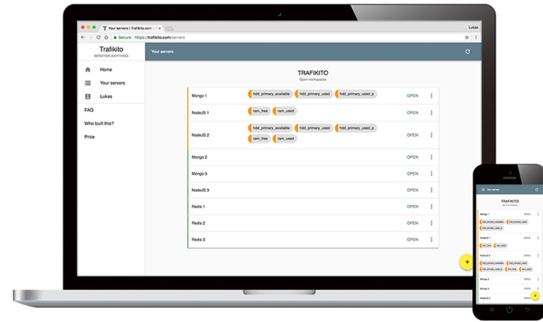

Figure 2: Trafikito, an app that uses Material-UI to support for mobile and desktop clients.

2.1.4 Minimizing GUI development: Material-UI

Creating user interface components for the browser from scratch is extremely time consuming. In the past several years, however, numerous comprehensive and high-quality open source component libraries for the browser have been released. Most parts of an LD app can be served well by these off-the-shelf components, making it an obvious choice to outsource as much UI development as possible to them.

The only concern would be that these libraries might be abandoned by their maintainers. While this is always a possibility, the communities that maintain the most prominent among these libraries are very active, and in the event that their current maintainers abandoned them, it seems likely that the community would step up to take over maintenance.

We chose to work with Material-UI, a library of components for Facebook's React UI framework that adheres to Google's Material Design guidelines. Beyond providing a set of high-quality components, with the v1.0 release of Material-UI, all components have been designed to work well on mobile clients, which has the potential to make an app developed with Material-UI usable on mobile phones at no extra development cost for the app developer, an incredible timesaving benefit. (See figure 2.)

2.2 Results

Using these tools, we were able to create a browser app that implements a subset of ELAN in about 3

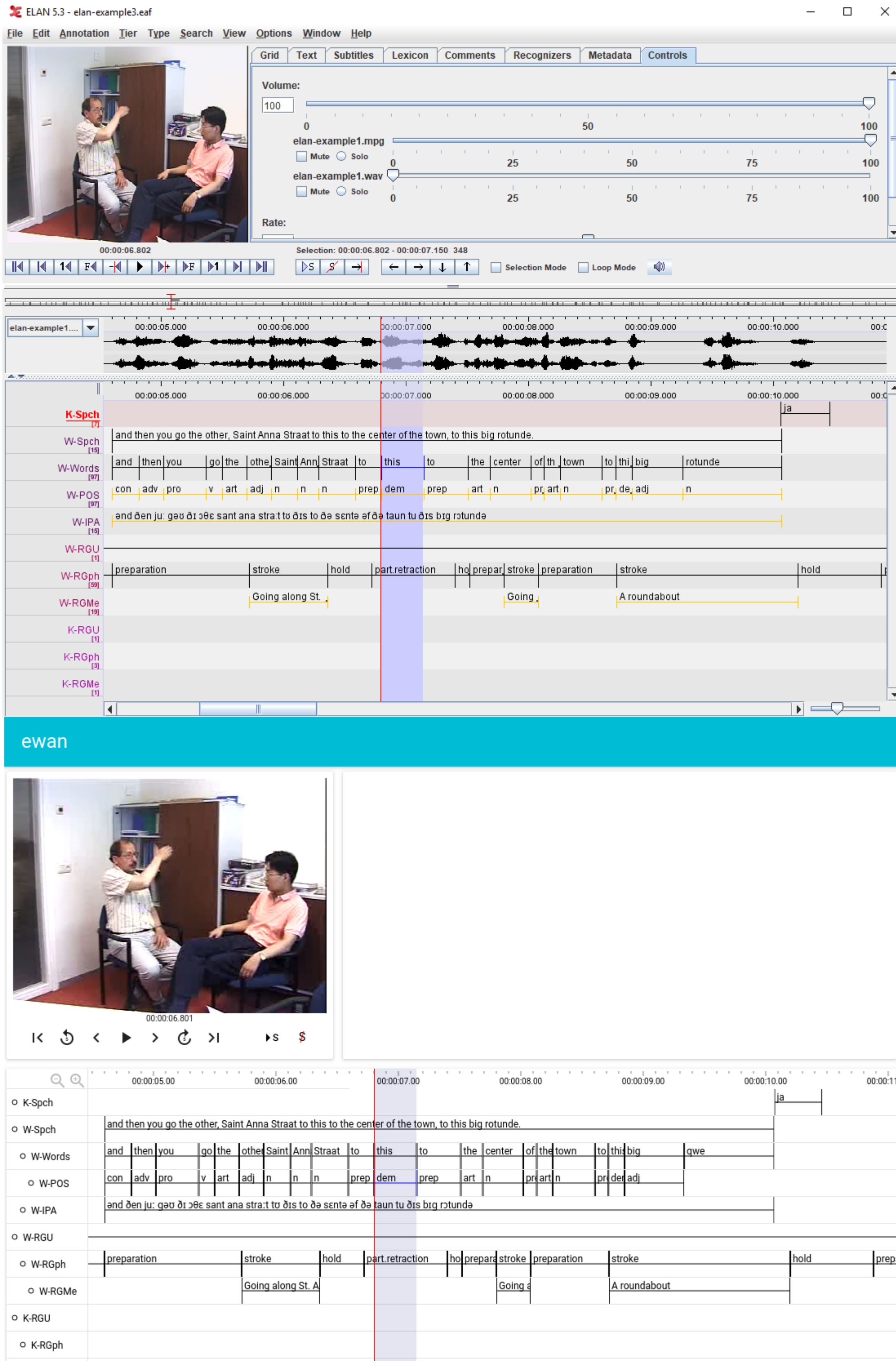

Figure 3: A screenshot of ELAN, above, and EWAN, below, for a given sample project.

weeks of development time[2]. Our implementation process basically confirmed our hypothesis: that our careful choice of tools made developers more productive, and vastly simplified the implementation of certain features that are critical to LD apps.

ELAN is an app that allows users to annotate a piece of audio or video along multiple aligned "tiers", each of which represents a single level of analysis for a single kind of linguistic or non-linguistic modality. For instance, in the example project in figure 3 (shown in both ELAN and our reimplementation, EWAN), for one of the speakers, there is a tier each for the speaker's sentence-level utterance, the speaker's utterance tokenized into words, IPA representations of those words, and the parts of speech for those words; and there is another tier for describing the speaker's gesticulations. Users are able to configure tiers so that e.g. a child tier containing parts of speech may not have an annotation that does not correspond to a word in the parent tier, and the app coordinates the UI so that the annotation area always stays in sync with the current time of the media player.

We implemented ELAN's behavior to a point where we felt we had been able to evaluate our hypothesis with reasonable confidence. In the end, this amounted to implementing the following features: (1) a basic recreation of the ELAN interface, (2) importing and exporting ELAN files, (3) read-only display and playback of projects, (4) editing of existing annotations, and (5) completely offline operation with sync capabilities. Most notably missing from this list is the ability to create new tiers and annotations, but we did not feel that implementing these features would have significantly changed our findings.

The creation of the interface with Material-UI was straightforward. The only UI components that involved significant custom development were the tier table and the code that kept the table and the video in sync: everything else, including the forms that allowed users to create new projects and import existing projects, was simply made with the expected Material-UI components.

ClojureScript's XML parsing library made importing and exporting ELAN project files a straightforward matter. Internally, EWAN uses the ELAN project file format as its data model, so there was no additional overhead associated with translating between EWAN and ELAN formats. To ensure the validity of the data under any changes that might be made in EWAN, we used ClojureScript's cljs.spec library to enforce invariants[3]. cljs.spec is one language feature among many that we felt made ClojureScript a good choice for enhancing developer productivity: cljs.spec allows developers to declaratively enforce data model invariants, whereas in most other languages more verbose imperative code would have to be written to enforce these invariants.

Projects were stored in PouchDB, and all the expected benefits were realized: an internet connection is no longer needed. Indeed, on the demo deployment of the app, there is not even a remote server to accompany it: once the app has been downloaded, users may use the app in its entirety without an internet connection. With ease, we were able to set up an instance of CouchDB on a remote server and sync our EWAN projects with it, even in poor network conditions.

2.3 Extensions

At the point in development where we stopped, there were features which we were close enough to see with clarity, even though we did not implement them. These are features we expect would be easily within reach if a similar approach were taken to creating a LD app.

2.3.1 Real-time collaboration

Google Docs-style real-time collaboration, where multiple clients collaborate in parallel on a single document, is well supported by our choice of PouchDB. Implementation is not entirely trivial, but the hard part—real-time synchronization without race conditions or data loss—is handled by CouchDB's replication protocol. It is debatable how useful this feature might be in an LD app, but if it were needed, it would be easily within reach.

2.3.2 User scripts with JavaScript

Power users delight in the opportunity to use a scripting language to tackle tedious or challenging tasks. Because this app was built in the web browser, we already have a language with excellent support at our disposal that was designed to

[2] Our app can be seen at `https://lgessler.com/ewan/` and our source code can be accessed at `https://github.com/lgessler/ewan`.

[3] An invariant in this context is some statement about the data that must always be true. For example, one might want to require that an XML element referenced by an attribute in another element actually exists.

```
// a collection of functions
const s = window.ewan.script;

// imagined feature 1: a programmatic interface for making arbitrary
// substitutions for regular expressions
s.replaceAll("tierId", /colour/g, "color");

// imagined feature 2: a more complicated interface that lets you apply a
// function to all annotation values on a tier. Suppose we made a transcription
// error in our words and forgot to lengthen [u] in front of voiced consonants
// in English. We could use this function to correct the issue:
s.updateAnns("phoneticTierId", function(ann) {
  const text = ann.value;
  const uIndex = text.indexOf("u");

  // check if we have u followed by a voiced consonant (just consider [b] and [d]
  // here for simplicity) and replace with u: if it is
  if (uIndex > -1
      && uIndex < text.length - 1
      && ['b', 'd'].indexOf(uIndex.charAt(uIndex + 1)) > -1) {
    ann.value = text.substring(0, uIndex) + "u:" + text.substring(uIndex + 1);
  }
});
```

Figure 4: Examples of what a JavaScript scripting interface for EWAN might look like. An API implemented in ClojureScript with functions intended for use in JavaScript that can modify EWAN's data structures is made available in a global namespace, and power users can use the scripting functions to perform bulk actions and customize their experience.

be easy for users to pick up and write small snippets with: JavaScript.

Since ClojureScript compiles to JavaScript and is designed for interoperation, it is very easy to make such a system available. As shown in figure 4, a special module containing user-facing functions that are meant to be used from a JavaScript console could be created. This could be used for anything from bulk actions to allowing users to hook into remote NLP services (e.g. for morphological parsing, tokenization, etc.).

The possibilities are endless with a user-facing API, and a thoughtfully constructed scripting API could do much to inspire and attract users. It's worth noting that this feature was made possible by our choice of platform: had we not been using the browser, we would have had either implement an entire scripting language or embed an existing one (like Lua) into the app.

2.3.3 Extensions

As noted in section 1.1, there will always be formalisms or levels of analysis that will not have robust support in an app. Normally, this pushes users who really need such support to either use another app or revert to a more generic medium.

To try to tackle this issue, an app structured like EWAN could be enriched with API's to allow users to customize deeper parts of the app, as shown in figure 5. Suppose that a user is a semanticist who has designed a new semantic parsing formalism, and wants to be able to annotate

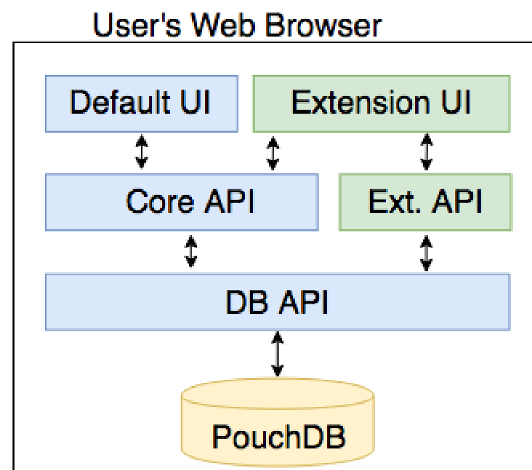

Figure 5: A high-level representation of EWAN's API layers and how UI and business logic layers could be extended if extension functions at the DB and business logic layers were made.

texts with it. This would require the creation of at least a new UI. If the user is comfortable with writing JavaScript, the user could use these extension API's to implement support for their own formalism, provided these extension API's are powerful and approachable enough for use by novice programmers. The user could then not only benefit themselves, but also share the extension with the community.

If this were done well, a major problem driving the continued fragmentation of the LD app landscape would be solved, and the LD community could perhaps begin to converge on a common foundation for common tasks in LD.

3 Conclusion

Because of the economic conditions that are endemic to software development in language documentation, app developers cannot reasonably hope to succeed by taking software development approaches unmodified from industry, where developer labor is much more abundant. We have argued these economic conditions are the major reason why LD apps have not realized their full potential, and that the way to solve this problem is to be selective in the tools that are used in making LD apps. We have shown that choice of tools does indeed affect how productive developers can be, as demonstrated by our experience recreating a portion of the app ELAN.

It is as yet unclear whether our choices were right—both our choices in which requirements to

prioritize, and in which tools to use to address those requirements. Offline support seems secure as a principal concern for LD apps, but perhaps it might actually be the case that it is better to choose a programming language that is easy to learn rather than easy to maintain. What we hope is clear, however, is that choice of tools can affect developer productivity by orders of magnitude, and that the success of language documentation software will be decided by how foresighted and creative its developers are in choosing tools that will minimize their labor.

Acknowledgments

Thanks to Mans Hulden, Sarah Moeller, Anish Tondwalkar, and Marina Bolotnikova for helpful comments on an earlier version of this paper.

References

J. Dunham, A. Bale, and J. Coon. 2015. LingSync: web-based software for language documentation. In *Proceedings of the 4th International Conference on Language Documentation Conservation*.

P. Graham. 2004. *Hackers and Painters: Essays on the Art of Programming*. O'Reilly & Associates, Inc., Sebastopol, CA, USA.

R. Moe. 2008. FieldWorks Language Explorer 1.0. *SIL Forum for Language Fieldwork*.

N. Thieberger. 2016. Language documentation tools and methods summit report http://bit.ly/LDTAMSReport.

S. Wagner and M. Ruhe. 2018. A systematic review of productivity factors in software development. *Computing Research Repository*, abs/1801.06475.

P. Wittenburg, H. Brugman, A. Russel, A. Klassmann, and H. Sloetjes. 2006. ELAN: a professional framework for multimodality research. In *Proceedings of LREC 2006, Fifth International Conference on Language Resources and Evaluation*.