Peer-reviewed article

How OER webapps came to be the way they are (and how to fix them!): a developer's guide to the SUSTAIN model

Robert Reynolds

Brigham Young University

robert_reynolds@byu.edu

Abstract

Many Open Educational Resource (OER) web applications have suffered from limited longevity and impact, even when teachers and students are receptive. A major cause is hiding in the subtle decisions about which technologies are used to build the apps. This "publish, then perish" trend is a direct result of the fact that OER developers are not always professional web developers, and are susceptible to following industry trends that do not align with educational and scholarly goals. I discuss factors that contribute to this tendency, and propose an alternative deployment model to extend the impact of educational applications: the SUSTAIN model. Using the example of the now-defunct VIEW browser extension, I highlight struggles that have complicated deployment and maintenance. I then present a consolidation and re-write of two webapps (VIEW and RuMOR) into a single new browser extension, the Russian Language ToolKit (RLTK). The new build prioritizes long-term viability by compiling underlying tools to WebAssembly and porting all language processing functionality from the previous implementations' back-end servers to the browser's javascript runtime. By isolating all technology to a static webapp, most potential sources of deployment failure are completely avoided. Finally, I discuss possibilities and limitations of this approach that can help researchers determine whether it is appropriate for their work.

Slovo 66:1 (2026), 160–176. DOI https://doi.org/10.33063/slovo.v66i1.1240
CC BY 4.0

1. Introduction

Despite decades of promising research, the long-term impact of many Open Educational Resource (OER) web applications has been limited. Friesen (2009) observes that OER projects often face a "sustainability crunch," where initiatives falter once initial funding is exhausted. Sancho-Gil et al. (2020) characterize this as the "predictable failure" of Ed-Tech initiatives, and Mishra (2017) notes that barriers to OER adoption often come from within the movement itself. A key technical reason for this limited impact is that the technology stacks (i.e., the combination of programming languages, frameworks, and services used to build an application) chosen for these projects are often brittle—prone to breaking when any single component changes. As a result, an OER prototype described in a conference paper is often no longer accessible only a few years later. Ahlf et al. (2024) analyze this ephemerality and propose the TRS (Time, Resources, Sustainability) model, which emphasizes forward-thinking planning to prevent obsolescence.

In this article, I identify specific factors that make an OER technology stack brittle. I then propose the SUSTAIN model, a set of design principles aimed at improving the effective longevity of OER webapps. Finally, I illustrate these principles in action by describing how two webapps—1) *VIEW*, a Russian reading assistant (Reynolds et al. 2014), and 2) *RuMOR*, a Russian morphology/orthography tutor (Reynolds et al. 2022)—were consolidated and redesigned into a single new browser extension, the Russian Language ToolKit (RLTK). My goal is to provide concrete guidance for the OER community to build tools that not only showcase pedagogical innovations, but also remain usable and impactful for many years.

2. Related work

The problem of longevity and project maintenance has been the topic of ongoing discussion. Volodina et al. (2012) observed that very few educational web applications were actively maintained or reused, despite the availability of accurate Natural Language Processing (NLP) tools that could benefit language learning. They proposed a "Service Oriented Architecture" (SOA), but such approaches drastically complicate deployment and reliability, since the failure of any one service can break the entire app. Similarly, Amaral et al. (2011) discuss how even technically successful educational prototypes often fail to integrate into real-world teaching, partly because the effort to keep them running is too great given their brittle underpinnings. While the TRS model of Ahlf et al. (2024) provides a holistic framework encompassing both technical and organizational dimensions, the present work focuses specifically on technical design principles for web application longevity.

3. Factors Limiting OER Webapp Longevity

3.1 Complex, Multi-Component Architectures

OER web applications often combine multiple advanced technologies to analyze and respond to learner input. Using language-learning apps as an example, a typical architecture might include a web frontend, a backend server with one or more NLP libraries (for tokenization, parsing, error detection, etc.), a database for student models or logging, and possibly additional servers or native programs for specialized tasks. While each component brings useful capabilities, the overall system becomes complex and fragile. The use of different programming languages and frameworks can complicate maintenance. This challenge is not unique to OER; Lercher et al. (2023) highlight similar issues in microservice architectures, where the evolution of Application Programming Interfaces (APIs) and communication between loosely coupled services creates significant integration and maintenance overhead.

3.2 Dependence on Native Binaries and External Servers

Many OER projects leverage existing NLP tools developed outside the language-learning domain. For example, many applications rely on popular NLP frameworks, such as CoreNLP (Manning et al. 2014), stanza (Qi et al. 2020), spacy (Honnibal et al. 2020), nltk (Bird et al. 2009), huggingface (Wolf et al. 2020), and others. Likewise, many rule-based systems rely on finite-state morphological analyzers (e.g., HFST[1] or Foma[2]) and constraint grammars (VISL CG-3[3]) to provide detailed linguistic analysis for learner input in systems for morphologically rich languages (e.g., Antonsen 2013; Wiechetek et al. 2022). These tools are often written in low-level languages (C/C++) and distributed as native binaries or source code. As such, the most natural implementation for them is on a back-end server.

Reliance on a remote server or service is inherently fragile. The end-of-life of Adobe Flash in 2020, which rendered numerous educational applications unusable overnight, starkly illustrates the risk of depending on proprietary runtimes. Using remote servers to host NLP components pushes the maintenance burden onto the project team: running servers incurs ongoing costs and requires security updates. Research projects often cannot keep servers running indefinitely after funding ends or developers graduate. When the

[1] Helsinki Finite-State Tools; https://hfst.github.io/
[2] (Hulden 2009); https://fomafst.github.io/
[3] (Bick and Didriksen 2015); https://edu.visl.dk/constraint_grammar.html

server goes offline, the OER tool becomes inaccessible. Remote servers also introduce latency and dependency on internet connectivity, which can hinder classroom adoption.

4. The SUSTAIN model

To address the above challenges, I propose a new model for developing OER web applications that is optimized for long-term viability. This model encourages applications that are Simple, User-less, Static, Transferable, Accessible, Independent, and web-Native (SUSTAIN). As an optimized model, the SUSTAIN model may not fit all applications, but it can still serve as frame of reference to evaluate whether a given OER webapp's features justify the reduced sustainability those features entail. Adhering to these principles can greatly increase a tool's lifespan and ease of maintenance, thus amplifying its impact.

4.1 Keep it simple

Keep the app's features and user interface as simple as possible. Each additional feature adds complexity, which increases the chances of bugs and maintenance burdens. Determine the core functionality that delivers the most educational value, and focus on that.

Keep the app's architecture as simple as possible by limiting the number of technologies used. Each additional technology, framework, or service in your stack is another thing that can break or become obsolete. By limiting the number of distinct components, you reduce integration bugs and the effort required to update the system. In practical terms, this might mean using a single general-purpose programming environment to handle most of the application's logic. For instance, a modern web application can be written entirely in JavaScript, leveraging libraries within that ecosystem, instead of split across a Java backend, a JavaScript frontend, a Python server, etc.

If an app can be implemented with just "vanilla" web technologies (HTML/CSS/JS/WASM) in the client browser, that eliminates the need for any specialized server or installation. A simpler stack also eases the onboarding of new developers or contributors. Open-source OER projects (as discussed in 4.7) benefit when others can readily understand how to run and modify the system. A simpler architecture lowers the so-called "bus factor" (i.e., the risk that only one person knows how it works). Given that research software often changes hands or is inherited by new students, simplicity can make the difference between a project that dies with the original developer and one that others can pick up and carry forward.

Finally, simplicity aids portability (cf. 4.4). The fewer external requirements, the more easily a system can be deployed in new contexts (different institutions, different devices,

etc.). This increases the chances of surviving beyond the initial use case and being reused or repurposed, thereby extending its life.

4.2 Store user data on client

The storage of user data on a remote server has profound implications for maintenance and compliance. Vázquez-Cano et al. (2023) emphasize "safety and ethics in data protection" as a key factor in digital sustainability for learning apps. If your OER system collects and stores personal user information, you inherit responsibilities for data privacy, security, and regulatory compliance (such as GDPR in Europe, or COPPA for minors in the US). Indeed, Otto (2019) found that legal concerns are a significant barrier to OER adoption in educational institutions. These obligations can become a burden that outlives the initial research project. For example, an OER platform that requires student accounts and stores progress data on a server must ensure that data is kept secure and private. Over time, managing user accounts can require updates for security (e.g., password hashing algorithms) and policy changes (privacy laws or platform policies). If those are neglected, the system may have to be shut down to avoid risks of data breaches or policy violations.

By keeping all processing local and not storing user data centrally, one can avoid such policy pitfalls entirely. Minimizing stored data also makes an OER tool easier to hand off or shut down cleanly. If no personal data is stored on a server, there is no liability or ethical issue if the server is discontinued; you can simply open-source the code and let others run it without worrying about migrating user databases. Many app components that traditionally use a central database–such as learner models for adaptive content–do not actually require a central database. This user data can be stored on the client, e.g. using HTML5's `localStorage` or `sessionStorage`. Besides the benefits for privacy and compliance, this often simplifies the app's architecture (c.f. 4.1).

If tracking user progress is part of a broader research objective, it is still advantageous to use local storage first, and then sync that data with a remote server when possible. This allows for capturing data while the user is not connected to the internet, and makes it trivial to remove data collection functions from the app without affecting general functionality.

4.3 Static webapp deployment

A static web application is one that consists solely of static files (HTML, CSS, JavaScript, WebAssembly) that can be served by any standard web server without the need for server-side processing or databases. On the other hand, dynamic web applications rely on server-side code (e.g., PHP, Node.js, Python) to customize the content of each page. Note that

static webapps can still include complex client-side logic, including interactivity and adapative behavior. The important distinction is that all such processing happens in the user's browser, rather than on a remote server.

Static webapps are inherently simpler to deploy and maintain, since they do not require a running backend server or database. Many hosting providers offer free or low-cost static web hosting, making it possible to provide access to the app with literally no ongoing costs.

4.4 Reusability and Transferability

Design the app to be easily reusable and extensible by others. This includes clear documentation, modular code structure, an open-source license, and publication in a public repository.

One of the best ways to ensure that the underlying logic of an OER webapp is reusable is to build two apps at the same time, sharing underlying classes and other components. For example, you might build a static webapp, a desktop app (using Electron or a similar framework), and/or a browser extension that all share the same core codebase. This forces you to write modular, well-documented code that can be adapted to different contexts.

Publishing the source code of OER web applications in an open repository (e.g. GitHub or GitLab) is crucial for longevity and impact (Duan and Lee 2022). Open-sourcing allows other developers and researchers to inspect how the system works, reuse components, and update parts as needed to keep it running. It also guards against the tool disappearing entirely when the original authors move on. Even if a live demo goes down, the code remains available for someone else to deploy or extend.

If your project has dependencies or deployment considerations, clearly document them; better yet, create a `Dockerfile` or `compose.yaml` to effortlessly deploy the app in a container. Consider including a copy of the container's base image in your codebase in case the image is removed from container repositories in the future.

Making OER code open-source and placing it in a well-known public repository (with documentation and an appropriate license) significantly increases the chances that the software remains usable and useful in the long term. It invites collaborative maintenance and signals to the community that building on it is welcome, which can lead to the software being adopted in unexpected new contexts rather than fading away.

4.5 Accessibility

Ensure that the app is accessible to users with diverse needs and abilities. Vázquez-Cano et al. (2023) identify accessibility as a core pillar of digital sustainability, ensuring that tools

remain usable by all students. Many governments and institutions have legal requirements for digital accessibility (e.g., the Americans with Disabilities Act in the US, or the Web Content Accessibility Guidelines internationally), and a failure to comply can result in legal liability or exclusion of users.

Designing for accessibility from the start ensures that the app remains legally compliant as standards evolve. Use semantic HTML elements, provide alt text for images, ensure keyboard navigability, and test with screen readers.[4] Using standard web technologies, as discussed in 4.7 below, helps ensure compatibility with assistive technologies. Avoid Javascript frameworks, such as React, that dynamically manipulate the Document Object Model (DOM) in ways that break core accessibility features.

4.6 Independence

Although many OER researchers have followed industry trends and advocated for a service-oriented approach to building educational applications (e.g., Volodina et al. 2012), experience has shown that such architectures dependent on multiple APIs are brittle and difficult to maintain over time. As noted by Lercher et al. (2023), relying on external APIs introduces risks related to breaking changes and the need for constant communication between service providers and consumers—a luxury often unavailable in loosely funded OER projects. Similarly, Assunção et al. (2023) found that technical changes drive the evolution of microservices more than the actual business logic, and even when focused on business logic, "isolated microservices are an illusion.". For this reason, the SUSTAIN model discourages the use of external APIs altogether, whenever possible.

Similarly, each external library dependency is a potential point of future failure. If dependencies are necessary, vendor them (i.e., include a copy of the source code directly in your project) to avoid relying on package managers or content delivery networks (CDNs) that may disappear. The longevity of any software is tied to the longevity of its dependencies. When possible, eliminate dependencies and frameworks altogether. When necessary, lean towards well-established, widely adopted open-source dependencies over niche or cutting-edge solutions.

4.6.1 Artificial Intelligence APIs

Especially in the age of large language models (LLMs) and generative AI, it may be tempting to rely on third-party AI APIs (e.g., OpenAI, Cohere, HuggingFace) to handle some of the heavy lifting for NLP tasks. While these services can provide an easy proof-

[4] https://developer.mozilla.org/en-US/docs/Learn_web_development/Core/Accessibility

of-concept, they introduce significant long-term risks. API providers will change pricing models, rate limits, or even discontinue services, which can render your OER tool unusable.

When possible, avoid relying on external APIs. Instead, consider using open-source models that can be run locally (see 4.7). If an external API is necessary, design the app with fallback functions for when the API is unavailable or changes.

4.7 Web-native technologies

Choose technologies and frameworks that have a strong commitment to backward compatibility and long-term support. Web standards (e.g. HTML, CSS, JavaScript, and WebAssembly) are the consummate example of this. Not only are these technologies mature, stable and powerful; they are governed by organizations[5] that are dedicated to openness, accessibility, and backward compatability. These groups are committed to the mantra "don't break the web!", and by using only those technologies, you are virtually guaranteed future-proof, maintenance-free operation. A simple JavaScript web page from 1998 will likely still work today. In contrast, a browser plugin using non-standard or proprietary technology, such as a Java applet, a Flash app, or a Silverlight app from that time would not, since those plugins have been discontinued.

4.7.1 WebAssembly and Natural Language Processing (NLP)

Many OERs rely on compute-heavy models, such as Natural Language Processing (NLP) and machine learning. A static webapp may seem like a strange environment to deploy such applications, since most major machine-learning frameworks are written in C/C++/Java/Python. However, aside from Java, the rise of WebAssembly (WASM) has made it possible to run many tools and libraries in the browser. The compiled WebAssembly binaries operate on the client's machine at speeds comparable to native execution.

Python is currently one of the most popular programming languages for NLP research and the Pyodide project provides a WebAssembly build of the Python interpreter along with many packages commonly used for NLP, such as `nltk` and `scikit-learn` (team 2021). In addition, Pyodide makes all pure-python libraries simple to use in the browser.[6]

[5] https://www.w3.org/; https://whatwg.org/; https://ecma-international.org/technical-committees/tc39/

[6] Unfortunately, two popular Python NLP libraries – `spacy` and `stanza` – are not yet officially published for WebAssembly, but work is underway to change that.

Although Python's popular deep learning libraries are not yet supported in WebAssembly, there are JavaScript alternatives for running trained models directly in the browser, provided the model is not too large for users' local devices.

Many popular NLP tools are written in C or C++. Emscripten[7] makes it possible to compile such tools to WebAssembly, sometimes with little to no change to the source code. The resulting WebAssembly build can then be called directly from Javascript in the browser, without the need for a back-end server.

Although the SUSTAIN model is not usually the most natural way to write an educational app or resource, the inconvenience is far outweighed by the simplicity of deployment and the near future-proof result. There is good reason to believe that as long as an HTTP server can deliver the files of your app to a client, it will work.

5 Case Studies

5.1 Case Study 1: Redesigning VIEW's Russian module into RLTK

As a proof of concept, I present the redesign of the Russian module of VIEW, an OER web application for Russian. VIEW (Visual Input Enhancement for the Web) is a browser extension that enables learners to get on-the-fly language exercises from any webpage they visit (Amaral et al. 2006; Metcalf and Meurers 2006; Dimitrov 2008, 2018; Meurers et al. 2010; Reynolds et al. 2014). It is a multilingual successor to the WERTi system, extending the concept of automatic text enhancement to languages beyond English. The webapp has seen three incarnations (WERTi -> VIEW -> VIEW 2.0) and unfortunately all three suffered from a brittle tech stack that eventually rendered it an inoperable maintenance liability. This prompted a complete overhaul guided by the principles above.

5.1.1 The demise of VIEW 2.0

The report on VIEW 2.0 (Dimitrov 2018) clearly describes the technical shortcomings of the two preceding versions, so I will not mention those here. The latest version of VIEW also followed the following client-server model: the webapp would send the text of the current webpage to a remote server, which ran a pipeline of NLP tools. This pipeline included a morphological analyzer (for tokenization and lemmatization, using HFST-based transducers for Russian), a constraint grammar module (VISL CG-3 rules to disambiguate morphosyntactic tags and detect certain grammatical forms), and a set of exercise

[7] https://emscripten.org/

templates. The server would return the annotated text and exercise data to the extension, which would then overlay highlights or interactive exercises onto the webpage for the user.

As shown in Figure 1 (left), the backend server was implemented in a Java tomcat server along with a number of C++ executables (HFST and VISL-CG3 binaries), and required a running PostgreSQL database for user authentication and interaction logging. The extension itself was written in JavaScript. For a time, the system worked and was even deployed in a small user study. However, two developments quickly undermined this architecture.

The first issue was maintenance: the remote server and its NLP components required occasional upkeep that was displaced by more pressing teaching and research goals. Javascript dependencies were sometimes deprecated, or had breaking changes because of security vulnerabilities. Occasionally, the backend server was left in a state of dysfunction, breaking the extension for all users.

The second issue arose when the Chrome Web Store introduced stricter privacy policies. Because the extension was transmitting the content of any webpage a user visited to our server, it was pulled from the store. Chrome began requiring extensions to limit data collection or provide justifications, and in some cases disallowed the kind of broad access VIEW required. Complying would require a significant refactor to remove data tracking or undergoing complex audits and adding privacy disclosures that might deter users.



Figure 1: Architecture comparison showing the evolution from VIEW's Russian module to RLTK. Parallel components share colors across both diagrams (e.g., green for NLP tools). Left: the original client-server design required a Java application server, the UIMA framework, C++ NLP binaries, and a PostgreSQL database. Right: the RLTK architecture compiles all NLP tools to WebAssembly and runs entirely in the browser, satisfying all SUSTAIN principles

5.1.2 Static JavaScript + WebAssembly Redesign

Learning from that experience, I undertook a redesign of VIEW as part of a new consolidated browser extension, the Russian Language ToolKit (RLTK),[8] with the goal of eliminating the backend entirely and simplifying the technology stack as described above. The new approach was to use a purely client-side application, delivered either as a browser extension or a static web page that users can load locally. This meant reimplementing all necessary NLP processing in the browser environment.

I successfully compiled HFST and its dependencies – `foma`, `OpenFST`, and `ICU`[9] – as well as `VISL CG-3` to WebAssembly modules. Because I needed javascript to interact directly with HFST, it was necessary to write emscripten bindings for two HFST classes, but all the

[8] https://github.com/reynoldsnlp/rltk

[9] ICU is a library for handling unicode. It comes with a fairly large data library that I filtered for only those datasets used by HFST.

other dependencies compiled without significant changes. This means that the full pipeline of analyzing text (tokenizing, tagging, disambiguating) can execute within the user's browser, securely and efficiently, without a server connection.

With these components in WASM, the new stack is drastically simpler. RLTK is essentially a JavaScript application that loads the WASM modules and applies them to page content. All code runs in one environment (the browser) and is written in one primary language (JavaScript, with WebAssembly as a complement). There is no need for a separate database, as we avoid storing data altogether. Thus, it inherently complies with Chrome's privacy requirements and does not even need to ask for extensive permissions (beyond reading the page content locally).

RLTK exemplifies the principles of the SUSTAIN model:

- **Simple:** I removed entire layers of complexity (no separate web server, database, or external NLP service). Everything is packaged together in a single client-side application.
- **User-less:** The new webapp does not maintain any user accounts or logs. All user interaction (like which words were clicked or which answers were given) stays in the user's session. This dramatically reduces legal and maintenance burdens.
- **Static:** The application consists entirely of static files. Deployment is as easy as opening an HTML file or installing the extension; no backend configuration is required.
- **Transferable:** The entire RLTK codebase, including NLP components compiled to WASM, is hosted in a public repository under an open-source license. Because it is static, anyone can clone the repository and get the system running by simply opening the index page.
- **Accessible:** By relying on standard web technologies rather than proprietary plugins, the application is more compatible with assistive tools.
- **Independent:** The system no longer relies on external APIs or services that might be discontinued. It is self-contained.
- **web-Native:** By using standard WebAssembly and JavaScript, I rely on web platform stability. Browsers are very likely to support this code for the foreseeable future without breaking it.

An added benefit is performance and scalability. Since each learner's computer performs its own analysis, the system scales with the number of users without any server cost—and even works offline.

This client-side paradigm is not limited to rule-based NLP. Recent developments have shown that even large neural network models can be run in-browser via WebAssembly or WebGPU. Libraries like `transformers.js`[10] allow state-of-the-art models (for machine

[10] https://huggingface.co/docs/transformers.js/en/index

translation, grammar correction, etc.) to be executed entirely in the browser with no server. This means that future OER web applications could incorporate advanced NLP in a way that is still consistent with the SUSTAIN model.

5.2 Case Study 2: Consolidating RuMOR into RLTK

RuMOR (Reynolds et al. 2022) is an OER web application that helps learners of Russian improve their writing. When a learner submits text, RuMOR identifies morphological and orthographic errors—such as incorrect case endings or misspelled words—and provides corrective feedback with grammatical explanations.

The original version ran on a Python server that invoked the same NLP tools used by VIEW: HFST tokenized and analyzed each word form, and VISL CG-3 applied hand-written rules to detect errors and select the appropriate correction. Because these tools were compiled C++ programs, they could only run on the server, making the application dependent on a dedicated machine.

Because RuMOR relies on the same HFST and VISL CG-3 tools as VIEW, its functionality was consolidated into RLTK. The same WebAssembly builds of these tools power both the reading-enhancement and error-detection features, allowing the entire pipeline to execute directly in the user's browser. The consolidated app requires no backend server: it is a fully static webapp that follows the SUSTAIN principles and can be hosted on any standard web server at no ongoing cost.

5.3 Results and Outlook

RLTK demonstrates the viability of the SUSTAIN approach. Since the consolidated extension is functionally equivalent to the original apps, no user testing or similar evaluation is relevant to the purpose of the redesign. RLTK's footprint is moderate: the WASM files for the Russian HFST analyzer and CG are a few megabytes, which load quickly and can be cached by the browser. RLTK is publicly available online.[11]

RLTK provides a template for how other OER web applications can be modernized. By embracing the SUSTAIN model, OER developers can eliminate many traditional points of fragility. The threshold of what can be done client-side has shifted dramatically in recent years, and browsers increasingly serve as universal runtimes across all devices.

[11] https://chromewebstore.google.com/detail/hofbpcgdhdaihhlcjegbfdnmaplnjnco

6. Conclusion

Many OER web applications have yielded impressive educational innovations, yet too often their impact quickly fades as the software becomes inaccessible or unmaintained. To break the cycle of "publish, then perish" in OER software, we must prioritize sustainability in our technology choices. By simplifying architectures, sticking to backward-compatible platforms, minimizing user data entanglements, and open-sourcing our code, we can create OER tools that remain operational and useful for far longer, potentially benefiting many more learners and researchers. Importantly, these principles do not usually require sacrificing functionality or sophistication.

The availability of WebAssembly and in-browser machine learning libraries today provides opportunities to implement even complex OER systems in a more durable form. This shift toward WebAssembly has already been evident in other published research (Pine et al. 2023). By collectively valuing sustainable design, we can ensure that OER research builds a cumulative legacy rather than a series of dead-end prototypes. In sum, a less brittle tech stack is not just a technical convenience; it is central to increasing the real-world impact of OER. When our tools last longer and are easier to share, more educators can adopt them, more learners can benefit from them, and more researchers can build upon them.

The technical recommendations of the SUSTAIN model complement the organizational and planning guidelines advanced by Ahlf et al. (2024), who advocate for integrating sustainability considerations throughout the OER project lifecycle. Together, such approaches can help ensure that OER initiatives remain accessible, non-stagnant, and effective over time.

6.1 Limitations

This article demonstrated how a handful of OER web applications were successfully implemented as SUSTAIN webapps. However, these webapps are based on a rule-based NLP pipeline including finite-state transducers for morphological analysis (using HFST) and constraint grammar (using VISL-CG3), which require less compute power compared to some other NLP methods, particularly Large Language Models (LLMs) and other deep learning architectures. When applications require significantly more storage or compute resources, you must consider whether users' devices will be capable of running the required models. Even if their devices can technically run the models, the impact on system memory and/or battery life may justify deviations from the SUSTAIN model.

References

Ahlf, Michael, Sara McNeil & Phuong Nguyen. 2024. Open Educational Resources: Time, Resources and Sustainability in an Ephemeral Digital World. *In Educational Research and the Question (s) of Time*. Cham : Springer.

Amaral, Luiz, Vanessa Metcalf & Detmar Meurers. 2006. Language Awareness Through Re-Use of NLP Technology. *Pre-Conference Workshop on NLP in CALL–Computational and Linguistic Challenges*. CALICO.

Amaral, Luiz, Detmar Meurers & Ramon Ziai. 2011. Analyzing Learner Language: Towards a Flexible Natural Language Processing Architecture for Intelligent Language Tutors. *Computer Assisted Language Learning* 24(1). 1–16.

Antonsen, Lene. 2013. Constraints in Free-Input Question-Answering Drills. *Proceedings of the Second Workshop on NLP for Computer-Assisted Language Learning at NODALIDA*, 11–26.

Assunção, Wesley KG, Jacob Krüger, Sébastien Mosser & Sofiane Selaoui. 2023. How Do Microservices Evolve? An Empirical Analysis of Changes in Open-Source Microservice Repositories. *Journal of Systems and Software* 204. 111788.

Bick, Eckhard & Tino Didriksen. 2015. Cg-3—Beyond Classical Constraint Grammar. *Proceedings of the 20th Nordic Conference of Computational Linguistics (NODALIDA 2015)*, 31–39.

Bird, Steven, Ewan Klein & Edward Loper. 2009. *Natural Language Processing with Python: Analyzing Text with the Natural Language Toolkit*. Sebastopol, CA: O'Reilly Media.

Dimitrov, Aleksandar. 2008. Rebuilding WERTi: Providing a Platform for Second Language Acquisition Assistance. *Seminar für Sprachwissenschaft*, Universität Tübingen.

Dimitrov, Aleksandar. 2018. *VIEW 2.0: An Extensible Platform for Web Based Interactive Language Learning and Research*. M.A. thesis, Eberhard Karls Universität Tübingen.

Duan, Chenggui & Tracy K Lee. 2022. Educational Use of Free and Open Source Software (FOSS): International Development and Its Implications for Higher Education. *Interactive Technology and Smart Education* 19(1). 39–57.

Friesen, Norm. 2009. Open Educational Resources: New Possibilities for Change and Sustainability. *International Review of Research in Open and Distributed Learning* 10(5).

Honnibal, Matthew, Ines Montani, Sofie Van Landeghem & Adriane Boyd. 2020. *spaCy: Industrial-strength Natural Language Processing in Python*. https://doi.org/10.5281/zenodo.1212303.

Hulden, Mans. 2009. Foma: A Finite-State Compiler and Library. *Proceedings of the 12th Conference of the European Chapter of the Association for Computational Linguistics*, 29–32.

Lercher, Alexander, Johann Glock, Christian Macho, and Martin Pinzger. 2023. Microservice API Evolution in Practice: A Study on Strategies and Challenges. *arXiv Preprint arXiv:2311.08175*.

Manning, Christopher, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven Bethard & David McClosky. 2014. The Stanford CoreNLP Natural Language Processing Toolkit. In Kalina Bontcheva & Jingbo Zhu (eds.), *Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations*. Association for Computational Linguistics. https://doi.org/10.3115/v1/P14-5010.

Metcalf, Vanessa & Detmar Meurers. 2006. Generating Web-Based English Preposition Exercises from Real-World Texts. In *Eurocall 2006 Granada: Integrating CALL into Study Programmes*, 174.

Meurers, Detmar, Ramon Ziai, Luiz Amaral, et al. 2010. Enhancing Authentic Web Pages for Language Learners. In *Proceedings of the NAACL HLT 2010 Fifth Workshop on Innovative Use of NLP for Building Educational Applications*, 10–18.

Mishra, Sanjaya. 2017. Open Educational Resources: Removing Barriers from Within. *Distance Education* 38(3). 369–80.

Otto, Daniel. 2019. Adoption and Diffusion of Open Educational Resources (OER) in Education: A Meta-Analysis of 25 OER-Projects. *International Review of Research in Open and Distributed Learning* 20(5). 122–40.

Pine, Aidan, David Huggins-Daines, Eric Joanis, et al. 2023. ReadAlong Studio Web Interface for Digital Interactive Storytelling. In Ekaterina Kochmar, Jill Burstein, Andrea Horbach, et al. (eds.), *Proceedings of the 18th Workshop on Innovative Use of NLP for Building Educational Applications (BEA 2023)*. Association for Computational Linguistics. https://doi.org/10.18653/v1/2023.bea-1.14.

Qi, Peng, Yuhao Zhang, Yuhui Zhang, Jason Bolton & Christopher D. Manning. 2020. Stanza: A Python Natural Language Processing Toolkit for Many Human Languages. In Asli Celikyilmaz and Tsung-Hsien Wen (eds.), *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics: System Demonstrations*. Association for Computational Linguistics. https://doi.org/10.18653/v1/2020.acl-demos.14.

Reynolds, Robert, Laura Janda & Tore Nesset. 2022. A Cognitive Linguistic Approach to Analysis and Correction of Orthographic Errors. *Russian Journal of Linguistics* 26(2). 391–408.

Reynolds, Robert, Eduard Schaf & Detmar Meurers. 2014. A VIEW of Russian: Visual Input Enhancement and Adaptive Feedback. In *Proceedings of the Third Workshop on NLP for Computer-Assisted Language Learning*, 98–112.

Sancho-Gil, Juana M, Pablo Rivera-Vargas & Raquel Miño-Puigcercós. 2020. Moving Beyond the Predictable Failure of Ed-Tech Initiatives. *Learning, Media and Technology* 45(1). 61–75.

The Pyodide development team. 2021. *Pyodide/Pyodide*. V. 0.27.5. Zenodo, released August. https://doi.org/10.5281/zenodo.5156931.

Vázquez-Cano, Esteban, Mercedes Quero-Gervilla, Paz Díez-Arcón & Carolina Pascual-Moscoso. 2023. Analysis of Digital Sustainability Factors in the Adoption of

Learning Apps in Primary and Secondary Education. *Edutec, Revista Electrónica De Tecnología Educativa* 83. 24–40.

Volodina, Elena, Lars Borin, Hrafn Loftsson, Birna Arnbjörnsdóttir & Gumundur Örn Leifsson. 2012. Waste Not, Want Not: Towards a System Architecture for ICALL Based on NLP Component Re-Use. In *Proceedings of the SLTC 2012 Workshop on NLP for CALL*, 47–58.

Wiechetek, Linda, Katri Hiovain-Asikainen, Inga Lill Sigga Mikkelsen, et al. 2022. Unmasking the Myth of Effortless Big Data - Making an Open Source Multi-Lingual Infrastructure and Building Language Resources from Scratch. In Nicoletta Calzolari, Frédéric Béchet, Philippe Blache, et al. (eds.), *Proceedings of the Thirteenth Language Resources and Evaluation Conference*, European Language Resources Association. https://aclanthology.org/2022.lrec-1.125/.

Wolf, Thomas, Lysandre Debut, Victor Sanh, et al. 2020. Transformers: State-of-the-Art Natural Language Processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, 38–45. https://www.aclweb.org/anthology/2020.emnlp-demos.6.