ARTICLE

Balancing Community and Local Needs

Releasing, Maintaining, and Rearchitecting the Institutional Repository

Daniel Coughlin

ABSTRACT

This paper examines the decision points over the course of ten years of development of an institutional repository. Specifically, the focus is on the impact and influence from the open-source community, the needs of the local institution, the role that team dynamics plays, and the chosen platform. Frequently, the discussion revolves around the technology stack and its limitations and capabilities. Inherently, any technology will have several features and limitations, and these are important in determining a solution that will work for your institution. However, the people running the system and developing the software, and their enthusiasm to continue work within the existing software environment in order to provide features for your campus and the larger open-source community, will play a bigger role than the technical platform. These lenses are analyzed through three points in time: the initial roll out of our institutional repository, our long-term running and maintenance, and eventual new development and why we made the decisions we made at each of those points in time.

THE INSTITUTIONAL REPOSITORY (IR)

A university institutional repository (IR) provides long-term access to the scholarship and research outputs of an institution.[1] The outputs can be in the form of scholarly publications, data sets to support publications or other research, electronic theses and dissertations, and other digital assets that have value to the university to preserve and to the research community and beyond to disseminate. There is additional value in keeping these otherwise scattered resources collected in a single repository to showcase the scholarly accomplishments of an institution.[2] There is value to the university to collect and disseminate the scholarly outputs of the university to understand the strengths of the university and promote that research to outside audiences, attract new faculty, and provide opportunities for new faculty where fields may be emergent or void of an institutional presence. Furthermore, there is value to the research community to be able to find peer research without having to pay publisher access fees.

Reducing the burden on faculty to meet various policy demands from a federal, publisher, and institutional perspective provides another motivation for IRs. Federal policies can require making anonymized research data and scholarship publicly available because it is publicly funded through tax dollars; publishers can make authors provide access to the data that supports the research that is being published.[3] In the United States, a growing number of academic institutions, from 2005 to 2021, have adopted an open-access policy that requires researchers to provide a copy of any published scholarly article in a publicly accessible repository. The institutional repository is a way for a university to meet this increasing demand from research organizations and funding institutions for their researchers.[4]

As the size of a campus grows in disciplines, it inherently grows in complexity and a diversity of digital needs and use cases from its researchers. For example, high-resolution images or

Daniel Coughlin (dmc186@psu.edu) is Head Libraries Strategic Technologies, Penn State University.

https://doi.org/10.6017/ital.v41i1.14073

atmospheric data are likely to create a higher demand in storage needs than a discipline that relies largely on text. Performance-based research may require multimedia resources and streaming capabilities while other large files can be shared in a more asynchronous manner. The diversity of needs contributes to the complexity of finding a solution for an institutional repository that meets all, or many, of the needs on a campus from a file storage, discovery, and access perspective.

This paper broadly addresses Penn State University's development of its IR at three distinct points in time: (1) choosing a platform for our IR and its initial release; (2) maintaining an IR; and finally, (3) our current solution nearly 10 years later. At each point in time, we analyze our decision process through four lenses. These lenses provided a thorough examination for us to decide on how to proceed; they are community needs and potential tension that exists with local needs, our team dynamics, and finally the platform we built our software on and the infrastructure required to maintain it. We discuss why we made the decisions we made through these four lenses, the benefits and drawbacks, and what we have learned along the way. Penn State is the state of Pennsylvania's land grant university in the United States. The University has 24 campuses physically located in the Commonwealth of Pennsylvania, the World Campus which is online, two law schools, and a medical school. In the fall of 2021, Penn State had 73,476 enrolled undergraduate students and 13,873 graduate students, with research expenditures totaling over $850 million for the last four years.[5] Penn State is a large, public research institution with a diverse set of needs. This is significant because when the university is considering developing a large system such as an institutional repository, we need to meet the needs of a broad set of disciplines and domains. We are fortunate enough to have software developer and system administration resources that smaller institutions may not have. This provides a bit of context into our considerations for an institutional repository.

SELECTING A REPOSITORY

In January 2012, Penn State University Libraries and Penn State's central Information Technology department collaborated on developing an institutional repository for the University's growing data management needs. The University Libraries was interested in becoming more involved in open-source software community development efforts. At that point, many universities that we had spoken with had an existing IR solution in place, and we had a lot of freedom to choose a platform without the burden of data migration. We considered investigating (1) off-the-shelf, turnkey solutions such as DSpace, (2) a prototype we had just built called Curation Architecture Prototype Services (CAPS) using a microservices approach, or (3) building on top of an existing platform. Ultimately, we decided to build on top of an existing platform, Samvera (named Hydra at the time).[6] We did not want a turnkey solution, because we felt that we had distinct needs that would require a level of customization that these solutions would not be able to offer. Based on discussions with others, we decided to develop something of our own. We wanted to leverage the experience of others in the repository development domain. The microservices approach at the time was more of a conceptual approach towards development than an existing software solution. The ability to build on an existing platform was a happy middle ground for us and we evaluated this decision through several lenses that led us to our selection at that time.

Community Involvement

We did not want to develop a solution in a vacuum and thought a group with a (relatively) common set of problems would be helpful to problem solve. The Samvera community was a small but growing community working towards repository solutions like what we were trying to

achieve. Members of the community were both managerial and technical. This was valuable to us for understanding the strategic direction for the community and the ability to collaborate and problem solve on technical implementations. Some of the key partners for our early work were University of Hull (UK), Stanford University, University of Virginia, and Notre Dame. There was communication throughout the year over community email, chat platforms, and phone calls; however, the quarterly partner meetings were the most valuable time for collaboration. These quarterly meetings were a couple days in length, typically at a partner institution's campus (physically) attended by managers and software/systems developers. This provided the ability to work together on specific problems, showcase our work, and get to know each other more closely at lunch and after-hours meetups. Working within the community would also get our team increased exposure and help with recruiting future colleagues. Working in the open-source software community has been seen to benefit both candidates and employers in future job recruitment.[7] We were excited by the promise of working with and contributing toward a larger community. Our team had apprehension about building this alone, and we were happy to be working with the support of a community and within their set of processes.

Local Needs

Early on in our requirements for the repository we created a MOSCOW chart that provided our "must have," "should have," "could have," and "won't have" features.[8] The platform we were choosing was going to provide us with a significant set of these features for our repository with very little work on our end. These features were built in and included search, discovery, and basic file edit functionality. Essentially, we were going to quickly meet the needs of our stakeholders by using this software. This was important for a couple of reasons. First, providing features to our stakeholders quickly gave them ample time to provide feedback so that we could make necessary customizations for their specific needs. A less quantitative benefit was gaining the trust of colleagues at the start of a new project and new initiative. Rather than continually suggesting "that feature will be done next week," we were able to deliver results quickly and get feedback. For example, our repository integrated with our campus authentication system, restricting access. We were able to deliver these features and get feedback on both the functionality as well as terminology to improve the usability. In particular, the way our developers described permissions was initially too confusing for our users and we were able to make necessary adjustments prior to a production release of the IR.

Team Dynamics

We believed it was a significant professional development opportunity for many people on the team to work with a larger community and learn from and with those in the open-source community. The team working on the IR consisted of three full-time, or near full-time, developers (one joining after we started the project), and a systems administrator. This project was our first large project that included a project manager invested in Agile project management methodologies and with a systems administrator in place at the beginning of the project.

Platform and Infrastructure Stability

There was a desire to get to a common solution to easily set up other repositories for various needs within the Libraries and we hoped there would be an ability to plug and play various components or features. The three common components of this system were Fedora Commons to store both metadata and our digital assets; Solr as an index for fast search; and Blacklight as a web interface that sits on top of Solr. One of the primary components, active-fedora, would sync content between the Fedora and Solr persistence layers. Our hope was that with this model, we

would be able to write code that could be used in other repositories, and we could use the code that other institutions had written for our repository needs to build other applications more quickly.

The Samvera community was initially called Hydra because of the relationship with the mythical creature that has several heads (see figure 1). We were considering the potential of running a core storage infrastructure and discovery infrastructure, while developing several heads for our various applications. We knew this was a lofty expectation, but also thought that it was a good design principle for us to advance. Additionally, the pilot that we developed on microservices (CAPS) seemed to have a relatively large storage service and we could not determine how to get away from that. Although this was a bit of a shift in our philosophy, it was less of a shift based on our practical experience.

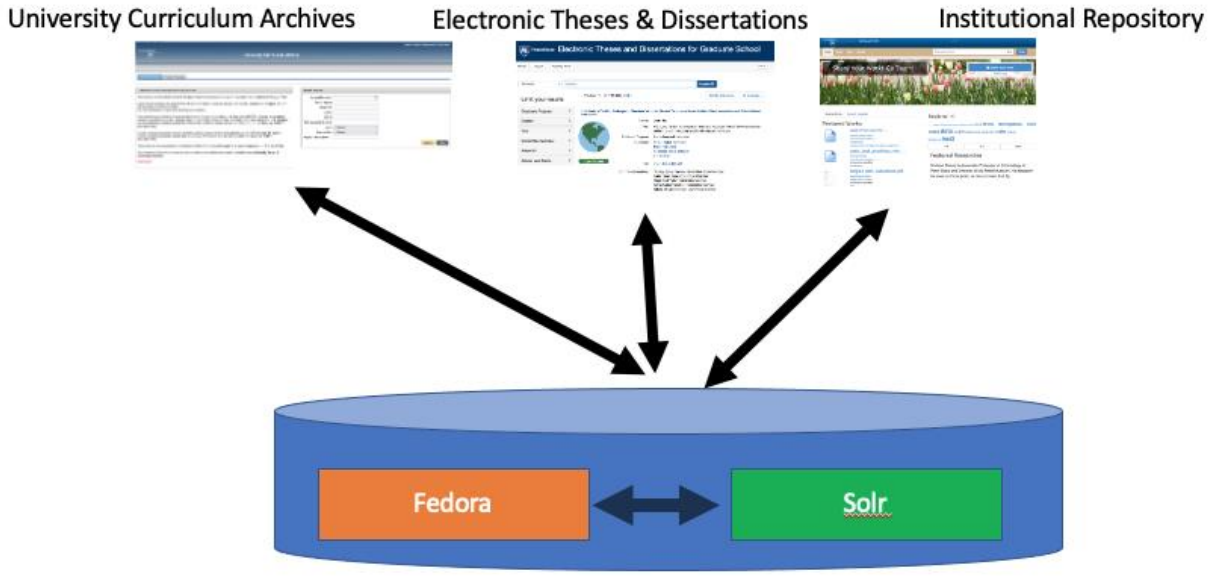

Figure 1. Aspirational intentions of running many applications on one access and discovery system.

Initial Release

The initial release of our IR, ScholarSphere, was for research data, scholarly articles, and presentations. We considered the repository file agnostic and left the definitions of scholarly materials up to the depositor. The self-deposit process made very few assumptions to limit the barriers to deposit—there were few mandated fields for deposit in ScholarSphere.

The initial rollout of ScholarSphere had met the "must have" and many of the "should have" needs that we had defined initially in our development requirements. The list of "must haves" included upload files via the web, create and assign metadata to the uploaded files, set three access levels to the files, search for files, display files, etc. The list of "should haves" were faceted browse, faceted search results, share files with a group, etc. The benefit of working on a community-developed platform provided some of these features for us (search, faceted browse), and gave us the flexibility to customize where necessary. For example, we had our own data model of metadata to assign to files based on our users' needs. We were able to update the existing metadata that was

provided out of the box, to accommodate that. This was a tremendous win for us to leverage community-provided solutions and local needs. Additionally, the platform provided a search index with Solr. This enabled our infrastructure to have a common solution with community support on configuration questions. Using the Blacklight UI on top of Solr created another opportunity for us to customize where desired and ease of development efforts.

Community: Following the initial release, we worked with other members of the community to pull out some of the core functionality and place it in a separate Ruby library. This library (Sufia) could then be leveraged as a default set of repository features for other developers. The release of a new IR, and this library, provided us with a lot of positive exposure at various community events.

Local Needs: Locally, we used this library to develop a repository for our digital archives. It previously took two to three developers nearly nine months to develop ScholarSphere; however, we used the Sufia module to roll out a separate repository in six months with a single developer. This was another successful production rollout and a successful use of a product created by and for the community.

Team Dynamics: We had a successful release and were getting support for new developers to hire. We continued to move more of our projects toward an Agile approach and permanently embed systems administrators into our development projects.

Infrastructure: We had not released a new system for archives on the exact same system that ScholarSphere was developed on, but we were happy that our projects were relatively homogeneous technology stacks and provided a familiarity to run.

MAINTAINING THE IR

Over the next several years we released three major updates to ScholarSphere:

1. Migrating the data object store to a major version
2. Overhauling the user interface
3. Migrating our data model to the Portland Common Data Model (PCDM)

Simultaneously, the Sufia library that we developed had also grown in usage by other institutions and contributions from other developers. We were excited to have additional contributors, and with that came an understandable sense of competing priorities within our community's development roadmap. We were building ScholarSphere features and functionality to meet the needs of our local institution and managing the tension between community direction and local needs. Again, we look at these lenses as evaluating the period during maintenance, upgrades, and feature adds.

Community Involvement

Two of the major releases mentioned above were largely community driven. In one case—migrating the data object store—we were one of the initial repositories within our open-source community to migrate our data storage system. We anticipated that doing this work early would prevent us from having to rewrite any code that relied on the data storage layer. Ultimately, this may have been a bit early for us, because we never were able to create the momentum for others in the community to make this same migration. This created a bit of a divergence, but at this layer in our technology it did not prevent us from continuing to work closely with the community.

We were able to add locally developed features for managing files and uploads, community components that allowed for controlled vocabularies, and Cloud provider uploads.[9] In all, from 2012 to 2019 we were an active member of the community: we provided technical contributions, we were being asked to present at community events, and our developers were frequently asked to help at several workshops. The community provided many opportunities for professional development and code from the community provided new features to our users. We felt this work was successful.

We had three major releases. One was something that our local users were able to experience directly. Two of our upgrades were largely on the back end and, while there is no argument on their importance, it can be a challenge to illustrate the significance of largely opaque technology upgrades to users.

Concurrently, we were coming up against other challenges that were proving difficult to solve in a sustainable and scalable way. Large file size (larger than 1 GB) for uploads and downloads remained an issue that researchers seemed to be encountering more frequently. Our mechanisms for getting around some of these obstacles led us to looking at an API for administrators and other applications to integrate. For example, if the web browser upload was not working, perhaps we could physically get the file from a user and upload it to the system ourselves. If we could do that, maybe we could use an API to do this upload, but we did not have an API.

When developing new features, we would question if it should be code to contribute back to the community or only for our (Penn State) needs. Frequently, the devil is in the details and, while several institutions were interested in a feature based on a conversation, implementation could be much more detailed and it was difficult to find common ground. This complexity could lead to longer timelines and more difficult planning for local development features.

Team Dynamics

Over this time period we advanced our team by adding several highly skilled developers (some of whom have now moved on to other positions and remain highly respected within the community), and enriched the collective skill set of the group. The team was enriched by this experience overall.

The balance between community involvement and local needs became a frequent conversation point for our team. We spent a lot of effort on initiatives that had not solved some of the bigger problems our users were experiencing locally; our community disengagement was likely a combination of common reasons, for example, our lack of time to make meaningful contributions.[10] In the spring of 2019, the development team that worked on ScholarSphere shrunk from three developers down to one. We had a strong number of developers within the Samvera community to collaborate with; however, we had difficulty bringing on new members at the time because the complexity within the ScholarSphere system created a high learning curve that was not necessarily transferable to other technology stacks.

At the end of the summer in 2019 we were given 25 GB of video files to upload in ScholarSphere and make accessible. The parameters of the request were outside of what we could support from our web interface, and we had no API to allow a product owner to develop against and work with the researcher to meet this request. After approximately one month of working with the data and our system, we successfully ingested the files into ScholarSphere. At the end of this month, we

decided that we needed to more urgently evaluate our path forward because we could not have our lone developer spending this amount of time on single-user requests.

Platform and Infrastructure Stability

Each of the major versions released between 2012 and 2019 had several patches and feature releases to enhance the system, the interface, and/or our processes for change management within the software system. For example, we went from a typed script containing a series of commands to Chef (a language used to automate software deployment) for deployment management; we upgraded infrastructure core components (Fedora, Solr, Travis, RedHat, etc.); and we added infrastructure to keep up with the system demands. In terms of adding infrastructure, we both enhanced the virtual capabilities (CPU and RAM) of our systems and had tasks offloaded to other systems. We did not want the systems our users interfaced with to be responsible for all the heavy lifting. These tasks included characterization, indexing metadata for search, creating thumbnails, etc. (see figure 2).

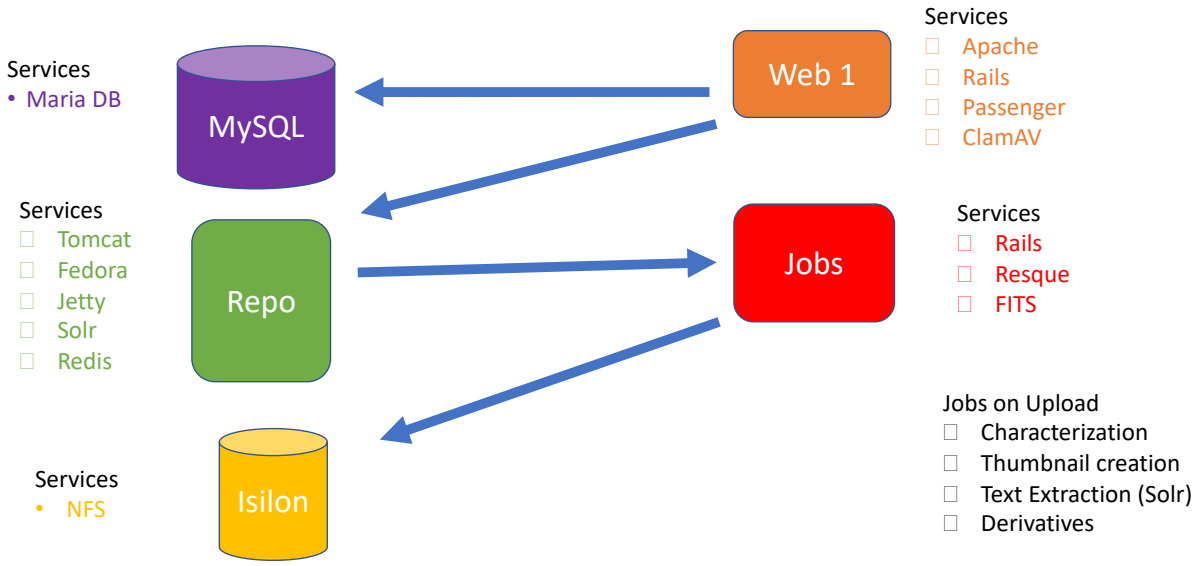

Figure 2. Systems and services with basic workflow process for uploading a file to ScholarSphere, including the background jobs that ran on file upload.

Adding additional components improved the user experience but made our infrastructure difficult to manage. We were continually trying to push our systems to reflect the best practices of the twelve-factor app.[11] However, over time, we had certain "infrastructure smells." The infrastructure smells were essentially anti-patterns of these best practices or symptoms of a bigger problem.[12] These anti-patterns included:

- Storage coupled closely to application
- Lack of flexibility to scale storage to integrate
- Inability to spin up a ScholarSphere instance

- Taking days to set up a dev environment
- Lack of flexibility to decouple small tasks that may require increased resources (create derivatives)

Evaluating Next Steps

Although we were coming up against some struggles and continued maintenance with ScholarSphere it was a successful software project that had several things we liked (and likely took for granted). It was important for us to recognize what features and characteristics of ScholarSphere were a part of this list. ScholarSphere's data model was flexible enough to support several current use cases and future needs and was developed with a significant amount of community input. There were other development teams within our organization that were also developing new applications in Ruby, so the language continued to be relevant within our larger group as was Ruby on Rails, Blacklight, and Solr. Some of the libraries developed with these frameworks were providing us with struggles and we knew that tools and infrastructure could be barriers to newcomers onboarding and orientation.[13] However, the languages themselves were still flexible enough for us to continue our work. We had three permission levels to access the full text of an uploaded file: (1) public, (2) Penn State only, and (3) private, and we didn't want to develop anything more complex than that around access permissions. Fedora provided us with versioning capability of our objects and we thought that this was something not only to continue but potentially enhance. We also had strong support from the Samvera community for ScholarSphere. Many people had worked on the code that helped provide functionality and we could collaborate within that community when problems arose. At that point we largely decided to continue to develop needed features for ScholarSphere while the community pushed forward. In part we were hoping that our divergent paths would converge within a year (give or take).

The month following the relatively manual process of ingesting the 25 GB of video files into ScholarSphere was spent making important updates to the system and fixing any low-hanging fruit. In October 2019 we decided to start from scratch and spend about two months developing a new solution and to evaluate our path forward after that.

CURRENT SOLUTION

We turned to the same four established lenses when evaluating our needs in 2019. However, it is worth noting that organizationally we were in a much different position than when we started in 2012. The software development and infrastructure team that managed the service was organizationally moved from Central IT to the Libraries where the service and product owner resides. Being in the same building and having the same priorities improved communications. Also, people within the teams had changed, and our leadership had changed, which changed how we approached some of our decisions. We had more experience in technical skills, specifically in repository development; we were more refined in our implementation of Agile methodologies; and having run a service for years, we had a better sense of our users' needs.

Community Involvement

The community saw a tremendously successful period of growth during this time in adoption of software, exposure for funded grants, and number of partners. There was renewed excitement about multiple solutions including turnkey repository solutions, hosted solutions, the merging of two highly regarded software libraries for performance, and improvement in developer friendliness. The latter improvement stripped some of the design patterns that developers struggled with to something more familiar and made it easier to onboard new developers.

Local Needs

The pressure to meet our local needs and competing priorities for the community-based software became a sticking point for us. We needed to have a more scalable backend and we were not sure when our needs and the priorities of the community would merge. We had also been behind on several dependencies and the lift to get back up to date, before being able to add anything new, was considerable. This situation led us to create a prototype for evaluation. Our initial goal was to see how difficult it would be to build a system to meet the needs of uploading the video files that ScholarSphere currently could not handle. We had confidence we could develop features, but this area was a consistent challenge and we considered it a primary hurdle for us to jump.

Team Dynamics

Our development team consisted of a single developer. However, we had an infrastructure developer who was able to help with systems configuration, automation, and containerization. Our developer thoroughly understood ScholarSphere and the underlying codebase and architecture and had the resources to hire a consultant to help with our efforts.

We had considerable work performed by a local software development company on other repositories (Electronic Theses & Dissertation system, a digital cultural heritage repository, and our Researcher Metadata Database). We valued this partnership and wanted to continue to utilize them as our staff numbers were down. We needed to be able to more quickly onboard others than we previously had in the past. If we were able to have three relatively new members of the team contributing to this progress, then we would also potentially have chosen a technology stack that was comfortable for others outside our development team to make a more immediate impact.

Platform and Infrastructure Stability

As with many systems that are actively developed for years, our current system had several dependencies that had organically grown over time to become burdensome to put together in order to set up a development environment. Additionally, a local development environment was not an exact replica of the production environment because networked storage was implemented on production and our development systems had a local storage. We also took this opportunity to test out Amazon S3 storage options as our production storage system. We chose this alternative to see if we had increased reliability in our storage and to see how well we could manage data in S3 and get a production service using this to provide an example of the annual operating cost by using the Cloud vendor. We were able to simplify our rollout a bit, and modernize the technologies used to run our systems (i.e., Docker containers, Kubernetes cluster) (see figure 3).

Development

We had three general goals: (1) to improve stability/scalability for local needs; (2) to improve our ability to get an environment up for developers more simply; and (3) to be able to onboard new developers more quickly. Shortly after our prototype test proved we could meet local needs in scalability, we were able to test out our second goal, getting a ScholarSphere environment set up easily. The process of setting up a development environment went from days to hours. We had reached two of our three goals with these tests and believed our development team (that was two to three new developers) contributing to our first two goals was proof that we could onboard new developers quickly (our third goal).

After several months of development in early 2020 we had accomplished moving several of the obstacles that had been in our way in recent years but were nowhere near feature parody with ScholarSphere.

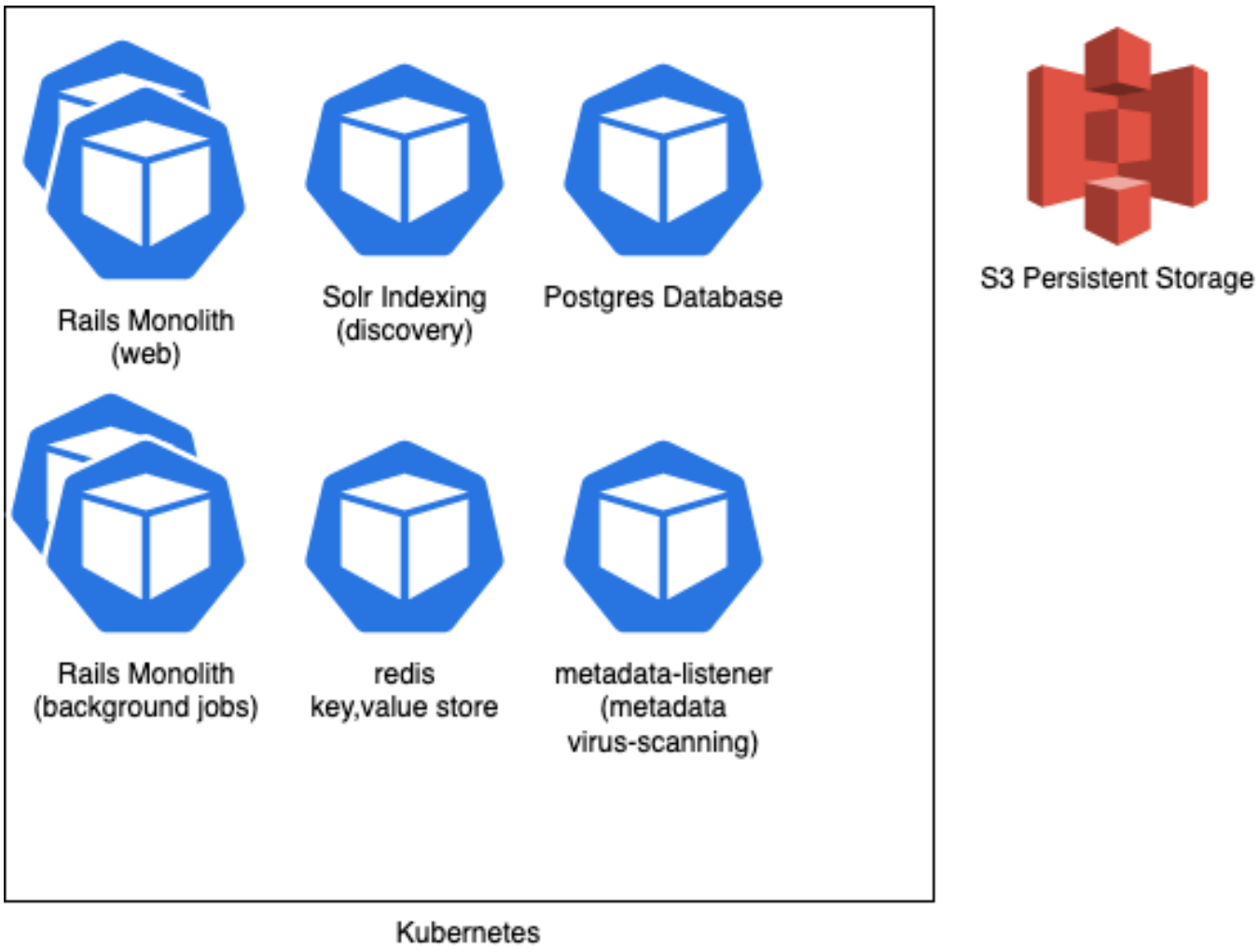

Figure 3. Current infrastructure for ScholarSphere, released in November of 2021.

We had a rich feature set to transfer from the existing ScholarSphere and did not want to simultaneously run two systems until we achieved some level of feature parody. We wanted to get to a minimal viable product (MVP) for our new prototype, migrate data, release our new version, and retire our existing system. Our product owner had been working directly with ScholarSphere users and was able to help us determine priorities for the features we needed in order to have an MVP. The following were some of those features:

- an API, at the very least an internal one for
    - our migration script
    - other home-developed applications
    - internal Library employees
- versioning and the ability to view versions
- updated status (pending published)
- updated user interface
- URLs that were harvestable
- maintaining our data model for continued support of concepts such as collections
- enhanced support for DOIs

We also identified some features that had been developed over the years to either simplify or eliminate. The profiles within ScholarSphere were not heavily used and over the years the University had more mature systems for this type of purpose. Similarly, finding a featured researcher for the home page seemed to create more work than it was worth, and our social media integrations were not going to be a priority. We also thought a user's dashboard—the default page after logging in—could be greatly simplified based on the most prominent actions our researchers wanted to perform.

CONCLUSION

After a little over a year of development, in November 2020, we released our new version of ScholarSphere. We used our own internal API, as planned, for data migration from our existing Fedora Commons storage system into the new one in Amazon S3. Over the past seven months we have done nine feature releases, including collections, and an enhanced API to support Penn State's Open Access initiative. We learned some lessons along the way within all of these lenses. We have also more than doubled the physical storage size of our repository since releasing in November 2020. Over the summer, we were able to meet a faculty member's request to upload 30 to 40 videos of 300 to 400 GB, a request we never would have been able to meet in our prior solution.

Community & Local Needs

Working with the Samvera community has provided countless opportunities for our entire team. We were able to sharpen our technical skills, were given opportunities to lead workshops, organized community development sprints, and collaborated on a plan for a community roadmap (to name a few). Our entire team benefitted in several ways by the involvement in the community: our software knowledge is higher, our problem-solving skills are more creative, and our outside professional opportunities expanded. Ultimately, our paths diverged in a way that made it difficult to justify the time and resources required for merging back.

There are several benefits to community-based software: more eyes looking at potential security issues in code, more voices to let you know when a dependency of your code has become vulnerable, shared software ideas for developing issues, and shared solutions for common problems. The cost of all these benefits comes with increased complexity in organizing a solution (you need to take multiple institutions into account), workflows for development (your local workflow may not be the same as the community approved workflow), competing priorities within the community, and competing priorities with the community and local roadmap. Open-source communities are largely online, these groups typically have a more shared, informal leadership structure and that lack of formal leadership can make it difficult to find solutions to these complexities.[14]

Team Dynamics, and Platform and Infrastructure Stability

Rewriting a system can be a daunting task, and several prominent developers would argue against it.[15] Reasons we believe we were successful are that (1) we did not change our data model, (2) although we changed our architecture, we did not change our coding conventions or our agile development process, and (3) the benefits of our changes were multidimensional. We were meeting users' needs with our development work and our infrastructure was enhancing our capabilities and making the work of our developers easier and less frustrating. Our deployment process has improved to the point that we can perform a release easily and without downtime.

Our technology is no longer based on Samvera, and is now, largely, a more generic Ruby on Rails application. We migrated from using Fedora as both a metadata and object store (retrieving objects on our central Isilon system through Fedora) to using Postgres as a metadata store and Amazon's S3 storage service for our files. We migrated our background jobs processing services from Rescue to Sidekiq. We continue to use Blacklight discovery and search interface, with Solr as our search platform.

Many of these technical decisions were made because of the change in dynamics of our team, and perhaps the single biggest change was around experience and the confidence that comes with that. Selecting a platform and infrastructure to support that platform is daunting. It is particularly difficult when you have so many questions in front of you about how the system will be used, the demand it may be under, the need to scale, how to deploy new features and update dependencies, etc. Our decisions in 2019 were made with much more experience and understanding of what was required of our system as well as what desired by our users. This gave us the confidence to branch off slightly from the joined technical path and recognize all the value (beyond technical solutions) to remain members of the community albeit in a modified capacity.

ACKNOWLEDGEMENTS

Many people put in tremendous time, effort, skill, thought, and enthusiasm into ScholarSphere over the years. We want to acknowledge all those that have contributed to the development and advancement of the system and appreciation for their work: Carolyn Cole, Hector Correa, Michael Tribone, Michael J. Giarlo, Adam Wead, Ryan Schenk, Jeff Minnelli, Dann Bohn, Justin Patterson, Joni Barnoff, Seth Erickson, Kieran Etienne, Calvin Morooney, Jim Campbell, Paul Crum, Chet Swalina, Matt Zumwalt, Justin Coyne, Elizabeth Sadler, Valerie Maher, Jamie Little, Brian Maddy, Kevin Clair, Patricia Hswe, and Beth Hayes.

ENDNOTES

[1] Helen Hockx-Yu, "Digital Preservation in the Context of Institutional Repositories," *Program* 40, no. 3 (2006): 232–43, https://doi.org/10.1108/00330330610681312.

[2] Raymond Okon, Ebele Leticia Eleberi, and Kanayo Kizito Uka, "A Web Based Digital Repository for Scholarly Publication," *Journal of Software Engineering and Applications* 13, no. 4 (2020), https://doi.org/10.4236/jsea.2020.134005.

[3] Research Data Access and Preservation, "Browse Data Sharing Requirements by Federal Agency," SPARC, September 29, 2020, http://researchsharing.sparcopen.org/compare?ids=18&compare=data; "Publisher Data Availability Policies Index," CHORUS, October 8, 2021, https://www.chorusaccess.org/resources/chorus-for-publishers/publisher-data-availability-policies-index/.

[4] "Registry of Open Access Repository Mandates and Policies," ROARMAP, http://roarmap.eprints.org/view/country/840.html.

[5] "Student Enrollment – Fall 2021," *The Pennsylvania State University Data Digest 2021*, https://datadigest.psu.edu/student-enrollment/.

[6] Stephen Abrams, John Kunze, and David Loy, "An Emergent Micro-Services Approach to Digital Curation Infrastructure," *The International Journal of Digital Curation* 5, no. 1 (2010): 172–86, https://doi.org/10.2218/ijdc.v5i1.151.

[7] Jennifer Marlow and Laura Dabbish, "Activity Traces and Signals in Software Developer Recruitment and Hiring," in *CSCW '13: Proceedings* (ACM, 2013): 145–56, https://doi.org/10.1145/2441776.2441794.

[8] Dai Clegg and Richard Barker, *CASE Method Fast-Track: A RAD Approach* (Reading: Addison-Wesley, 1994).

[9] "Questioning Authority," GitHub, accessed September 2021, https://github.com/samvera/questioning_authority; "Browse-Everything," GitHub, accessed 09/05/2021, https://github.com/samvera/browse-everything.

[10] Sophie Huilian Qiu et al., "Going Farther Together: The Impact of Social Capital on Sustained Participation in Open Source," *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)* (2019): 688–99, https://doi.org/10.1109/ICSE.2019.00078.

[11] Adam Wiggins, "The Twelve-Factor App," accessed September 2021, http://12factor.net.

[12] Akond Rahman, Chris Parnin, and Laurie Williams, "The Seven Sins: Security Smells in Infrastructure as Code Scripts," *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)* (2019): 164–75, https://doi.org/10.1109/ICSE.2019.00033.

[13] Christopher Mendez et al., "Open Source Barriers to Entry, Revisited: A Sociotechnical Perspective," in *Proceedings of the 40th International Conference on Software Engineering* (May 2018): 1004–15, https://doi.org/10.1145/3180155.3180241.

[14] Lindsay Larson and Leslie A. DeChurch, "Leading Teams in the Digital Age: Four Perspectives on Technology and What They Mean for Leading Teams," *Leadership Quarterly* 31, no. 1 (2020), https://doi.org/10.1016/j.leaqua.2019.101377.

[15] Fredrick P. Brooks Jr., *The Mythical Man-Month: Essays on Software Engineering* (Reading, Mass.: Addison-Wesley Pub. Co., 1982) https://search.library.wisc.edu/catalog/999550146602121; Joel Spolsky, "Things You Should Never Do Part I," *Joel On Software*, April 6, 2000, https://www.joelonsoftware.com/2000/04/06/things-you-should-never-do-part-i/.