# Package Picker:
## A Recommendation Engine for Software Packages

Daniel Sauble, Gerald Arocena, Dennis Cherchenko,

Muwei Gu, Edward Kang, and Volodymyr Popil

Harvard Extension
CSCI E-599 Software Engineering Capstone
Spring 2020

# Index

# Journal paper

**Abstract**

Package Picker is a recommendation engine for software packages. It uses an item-to-item collaborative filtering approach to recommend packages that are frequently used together with packages from a user's GitHub repository. The results are weighted using four different metrics: cosine similarity, number of sibling packages, absolute growth in terms of download count, and percentage growth in terms of download count.

This approach is superior to keyword search or other generalized-search-criteria approaches in the following ways: First, it reduces the amount of knowledge a software developer must have about their application to get good recommendations. Second, it allows developers to benefit from choices made by existing projects in the wild. Third, it surfaces useful packages that would ordinarily be buried by popularity-based metrics, making it easier for new package maintainers to gain adoption.

Our initial analysis focuses on the npm package ecosystem for JavaScript. We decided to focus on npm because it contains more packages than the next six most popular ecosystems combined, and there are twice as many JavaScript repositories on GitHub as the next most popular programming language. Having lots of data helped decrease the sparsity of our cosine similarity matrix. We conclude by showing that our technique can be extended to support the Python Package Index (PyPI), in preparation to support additional programming languages in the future.

**List of keywords**

Recommendation engine
Collaborative filtering
Cosine similarity
npm
JavaScript
GitHub

**Introduction**

Developers are only as good as their tools, which include the software packages they import. When a developer has a functional need, they must decide whether to use an open-source project or build it themselves. Building new functionality is expensive, so developers are incentivized to reuse software that others have written in the form of packages. This is especially true for npm, which takes the idea of small, reusable modules much further than other package ecosystems.

The traditional ways in which developers decide which packages to use for their project are not the most efficient. Developers looking for new open source packages are forced to rely on basic search tools, word-of-mouth, and popularity metrics to make their decision. Because this adds complexity to the search, developers invariably gravitate to packages that they, their co-workers, or random strangers on the internet have recommended, whether or not these are actually the best packages to use. While these methods generally get the job done, they are neither the best ways nor do they promote the discovery of new and superior packages. With millions of packages to choose from, how can a developer gain the full picture of possible options?

Package Picker solves this problem and helps developers find JavaScript or Python packages that are commonly used with the packages in their project Package Picker has ingested a mapping between repositories and packages from GitHub, and using this mapping, has encoded the data and computed the cosine similarity of all columns. The resulting similarity matrix allows us to determine the degree to which any two packages in our corpus are likely to appear in the same package.json file together. This serves as an effective item-to-item collaborative filtering approach, in that we can pass in a list of packages and get back a list of suggested packages. This targeted list of suggestions is then passed through a secondary filter or search interface to provide highly contextual recommendations. For example, users can sort, filter, or search by a variety of metadata: recommendation score, package name, category, and download count.

This sort of tool is useful for software architects who are tasked with choosing new packages for their application, or engineering managers who need control over how open-source software is being used by their teams. Package Picker fills a gap between recommendation engines that suggest projects for OSS maintainers and search engines that provide suggestions for package consumers based on keyword alone. Package Picker provides a one-click path to relevant package recommendations without the tedium of multiple keyword searches or going through GitHub repositories by hand. Through automation, Package Picker leads developers to the right packages for their project.

**Current Recommendation Engines and Metrics**

Sophisticated recommendation engines have existed for years, but their focus is different from what Package Picker was designed to do. Existing solutions have focused primarily on five problems: 1) the source code level rather than the package level (Mens, 2014), 2) finding open source projects in need of developer maintenance (Matek, 2016), 3) open source collaboration (Allaho, 2014), 4) onboarding new project maintainers (Malheiros, 2012), and 5) identification of domain experts (Christidis, 2012). These source-code-based recommendation engines meet a variety of needs throughout the software development lifecycle by helping developers choose an API method, spot missing code, reuse existing code, and get started with handy aids like auto-suggest. Many of these even integrate with common IDEs.

There are several source code recommendation engines such as Rascal, FrUiT, Strathcona, Hipikat, and CodeBroker (Robillard, 2014). There are many different methods used across these engines: 1) collaborative filtering and association rule mining to spot common usage, 2) space patterns to identify particular frameworks or APIs, 3) structural content of methods to build a similarity database for recommendation, 4) heuristics to retrieve metadata

related to project onboarding, and 5) latent semantic analysis (LSA). However, most of these methods assume that the developer already knows what packages they want to work with. If the developer wishes to discover new packages, a different approach must be used. This is the main difference between source code recommendation engines and package recommendation engines such as Package Picker.

There are also existing systems such as "npm Discover" and "npm packages PageRank" that work by allowing users to enter keywords to find packages of interest. The npm PageRank uses Google's PageRank algorithm to calculate package importance by the number of other packages that reference it as a dependency. Beyond npm PageRank, all of the other package discovery systems in common use are search index based, including npm Discover (deprecated since the 2016 study). The most commonly used search engines for npm packages are npmjs.com and yarnpkg.com.

Package Picker is different because it makes recommendations based on which packages tend to appear in the same applications together. By seeding it with a list of dependencies used by an existing application, we quickly reduce the search space from the global set of packages in the ecosystem to a much smaller list of packages that are all relevant to the application in question. Recommendations from our approach are also more relevant than a PageRank-based approach, which is based on relationships between parent and child dependencies. When developers add packages to an application, those packages become peer dependencies, so a peer-based approach is needed. Package Picker maps applications to their top-level dependencies, then computes the cosine similarity of the resulting matrix, which gives us the odds that any two peer dependencies appear in the same application together. A recommendation from Package Picker indicates that other people have successfully used the recommended package in other applications with a similar set of dependencies to yours.

**Our approach**

Package Picker is designed to provide recommendations for npm (JavaScript) and PyPI (Python) packages. We only consider runtime dependencies, not development or optional dependencies, mainly for simplicity but also because some ecosystems (e.g. PyPI) don't include a concept of non-runtime dependencies in their manifest files. We built the system for npm initially because it is the largest package ecosystem and thus has the most data for training. We added support for PyPI because our project is built in Python and we wanted to test our project on itself.

The machine learning pipeline takes a three phase approach: 1) querying GitHub for all repositories containing a package.json (npm) or requirements.txt (PyPI) manifest file with dependencies, 2) querying npmjs.com and pypi.org for package metadata related to those dependencies, and finally 3) building the cosine similarity matrix.

First, the pipeline builds its training set by querying all repositories in GitHub for those containing a npm or PyPI manifest file with dependencies (packages). Because the GitHub GraphQL API only allows up to 1,000 repositories to be fetched in a single search query, we fetch repositories based on their created date, one day at a time. To be fetched, a repository must have more than 1 star and be classified as a JavaScript or Python repository by GitHub. We include the star criteria to filter out repositories that aren't popular and are probably of low quality. Once fetched, a repository must have a package.json or requirements.txt file in its root folder to be included in our training dataset.

The repository and package name (including version) are persisted to the applications and packages tables in the database. We differentiate between major versions of a package, but not minor or patch versions, as we found this to be the best tradeoff between precision and having too many dimensions in our training data. We generate an integer id for each application and package, and use these ids in the training process and map back to package names when recommendations are requested by the user.

Second, the machine learning pipeline goes through all packages stored in the database, and retrieves the metadata from npmjs.com and pypi.org for each. This metadata includes the categories, the date each package was last modified, and download counts for last month and the same month a year prior. Categories in particular are arrays of strings that are optionally associated with each package, and provide the package author with a means of tagging the purpose of a package. Download counts are used to calculate the absolute trend and percentage trend component scores, while the categories and package modified date are used solely to provide the user with additional filtering, sorting, and search options.

Third, the machine learning pipeline determines the similarity of all packages in the corpus. It retrieves all records from the dependencies table, and creates a vectorized matrix of the application to package mapping using sparse vectors. Next, it computes the package cosine similarity of all columns in the matrix, which results in a package similarity matrix where each cell is a value between 0 and 1 inclusive, representing the degree to which two packages tend to appear in the same package.json or requirements.txt files together. We iterate through all cells in the matrix, ignore cells with a zero score to save space, and write the rest to a similarity table which is used by the web server to generate recommendations. The similarity table has three columns, the `package_a` and `package_b` columns are the packages with a non-zero similarity score, and the similarity column contains the score.

To retrieve recommendations, a user can either input dependencies manually, or log in to GitHub and retrieve a list of their repositories from GitHub with a package.json or requirements.txt file, respectively. When they select one of these repositories, we fetch the 10,000 packages that have the highest similarity scores relative to the packages in the user's repository. We refrain from returning packages that already exist in the user's repository. For each recommendation, we calculate its PkgPkr Score, which is the weighted sum of 1) its similarity score, 2) popularity in terms of how many other distinct packages appear together with it, 3) absolute growth in terms of year-over-year downloads, and 4) percentage growth in terms of year-over-year downloads. We assign weights of 0.5, 0.3, 0.1, and 0.1 to these scores, respectively. We log transform the popularity and download counts prior to computation of the scores because the data distribution is highly skewed. The resulting score is an integer between 1 and 10, inclusive, where 10 indicates a very strong recommendation. The web server has filtering and sorting capabilities which allow the user to browse these recommendations and their associated metadata, including the computed PkgPkr Score. By default, the list of recommendations is sorted by their PkgPkr Score.

**Results**

We assess the performance of our recommendation engine from three angles. 1) Is the system capable of recommending packages that are deliberately removed from a project? 2) Is the system capable of recommending a package based on the package's own dependencies? 3) Is the system able to make recommendations that are at least on par with a more traditional search engine?

First, we take a popular project and remove one package at a time from its package.json. If the recommendation engine recommends each missing package back to us, it's a good sign that our system recognizes which packages are commonly used together. We use express 4.17.1, a web framework for Node.js that has 30 dependencies. As shown in Figure 1, our system is able to recommend each missing package except qs@6, safe-buffer@5, and serve-static@1, with scores between 4 and 6.

| **Package** | **Score** | | **Package (cont…)** | **Score** | | **Package (cont…)** | **Score** |
|---|---|---|---|---|---|---|---|

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| accepts@1 | 6 | | escape-html@1 | 5 | | qs@6 | - |
| array-flatten@1 | 6 | | etag@1 | 6 | | range-parser@1 | 5 |
| body-parser@1 | 6 | | finalhandler@1 | 4 | | safe-buffer@5 | - |
| content-disposition@0 | 6 | | fresh@0 | 6 | | send@0 | 5 |
| content-type@1 | 5 | | merge-descriptors@1 | 6 | | serve-static@1 | - |
| cookie@0 | 5 | | methods@1 | 5 | | setprototypeof@1 | 5 |
| cookie-signature@1 | 5 | | on-finished@2 | 6 | | statuses@1 | 5 |
| debug@2 | 6 | | parseurl@1 | 5 | | type-of@1 | 6 |
| depd@1 | 5 | | path-to-regexp@0 | 6 | | utils-merge@1 | 5 |
| encodeurl@1 | 5 | | proxy-addr@2 | 5 | | vary@1 | 6 |

Figure 1: Ability of recommendation engine to recommend packages that are purposely removed from express.js

To verify this result against a baseline, we choose a completely unrelated project with a completely non-overlapping set of dependencies, retrieve scores for all 30 of express's dependencies, and see how the scores compare. We use cucumber.js 6.0.5, an automated test framework for Node.js. As shown in Figure 2, scores are 1-2 points lower for all packages except qs@6, safe-buffer@5, and serve-static@1. This is what we would expect if the context of a package has an impact on its PkgPkr Score.

| **Package** | **Score** | | **Package (cont…)** | **Score** | | **Package (cont…)** | **Score** |
|---|---|---|---|---|---|---|---|
| accepts@1 | 4 (-2) | | escape-html@1 | - | | qs@6 | 4 |
| array-flatten@1 | 4 (-2) | | etag@1 | 4 (-2) | | range-parser@1 | 4 (-1) |
| body-parser@1 | 5 (-1) | | finalhandler@1 | 4 (-0) | | safe-buffer@5 | 4 |
| content-disposition@0 | 4 (-2) | | fresh@0 | 4 (-2) | | send@0 | 4 (-1) |
| content-type@1 | 4 (-1) | | merge-descriptors@1 | 4 (-2) | | serve-static@1 | 4 |
| cookie@0 | 4 (-1) | | methods@1 | 4 (-1) | | setprototypeof@1 | 4 (-1) |
| cookie-signature@1 | 4 (-1) | | on-finished@2 | 4 (-2) | | statuses@1 | 4 (-1) |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| debug@2 | 4 (-2) | | parseurl@1 | 4 (-1) | | type-of@1 | 4 (-2) |
| depd@1 | 4 (-1) | | path-to-regexp@0 | 4 (-2) | | utils-merge@1 | 4 (-1) |
| encodeurl@1 | 4 (-1) | | proxy-addr@2 | 4 (-1) | | vary@1 | 4 (-2) |

Figure 2: Comparison of PkgPkr Scores using cucumber.js as a baseline

Second, we see if our recommendation engine is capable of recommending a package based on that package's dependencies only. Because Package Picker takes a peer-dependency approach, we wouldn't necessarily expect this to work unless it is common for a project's dependencies to also be peers with it in a parent application that relies on the project. We pick five unrelated projects and see if they recommend themselves and each other, and if so, what the scores are. As shown in Figure 3, our system was only capable of self-predicting express@4, cucumber@6, and mathjs@6. However, in each case the self-prediction score (in bold) was either the sole or highest score, which is what we would expect if context matters.

| | express@4 | cucumber@6 | serverless@1 | johnny-five@1 | mathjs@6 |
|---|---|---|---|---|---|
| express@4 | **8** | - | - | - | - |
| cucumber@6 | 5 | **4** | - | - | - |
| serverless@1 | 5 | - | - | - | - |
| johnny-five@1 | 4 | - | - | - | - |
| mathjs@6 | 4 | - | - | - | **3** |

Figure 3: Ability of recommendation engine to self-predict different packages

Third, we compare our recommendation engine to the authoritative search for npm and PyPI packages, npmjs.com and pypi.org, respectively. We use "express" for npm and "django" for PyPI, as both are popular packages that are also web frameworks, so the results should be similar in terms of domain. As shown in Figure 4, the quality of search results varies greatly. The top three results from pypi.org are django itself and some test packages, while the rest seem like specific django extensions. In contrast, the results from npmjs.com seem to do a better job of picking out packages that are relevant to web development in general while happening to also be compatible with express. In both cases, the results from Package Picker seem similarly focused on the domain of web development for django and express, respectively.

| **npmjs.com** | **Package Picker** | | **pypi.org** | **Package Picker** |
|---|---|---|---|---|
| express | body-parser@1 | | Django 3.0.5 | asgiref@3 |

| | | | | |
|---|---|---|---|---|
| path-to-regexp | morgan@1 | | Django-504 2.2.9 | sqlparse@0 |
| cors | cookie-parser@1 | | django-503 0.1 | whitenoise@5 |
| http-proxy-middleware | express-session@1 | | django-scribbler-django2.0 0.9.3 | gunicorn@20 |
| morgan | cors@2 | | django-filebrowser-django13 3.0 | pytz@2019 |
| is-regex | passport@0 | | django-jchart-django3-uvm 0.4.2 | requests@2 |
| express-handlebars | serve-favicon@2 | | Django-tracking-analyzer-django2 0.3 | six@1 |

Figure 4: Comparison of top search results and recommendations

As shown above, the results from Package Picker indicate that it is capable of making relevant recommendations for a particular package context. It can fill holes in an existing application's dependencies, recommend a parent package based solely on its own dependencies in many cases, and also deliver results on par with a good search engine. The relative ease with which we were able to add Python support indicates that our approach is extensible to other ecosystems, opening the possibility of a universal recommendation engine that is not language specific.

**Next steps**

Adding new languages should not be difficult based on our experiences with PyPI, though we would want to create more of a framework to guide and simplify this work for future developers. The larger issue is that scraping data is quite time consuming. It takes about 50 hours to scrape 10 years of data for npm and PyPI alone. This time could be reduced with parallelization, but GitHub has anti-abuse measures in place which prevent too much optimization here.

We did notice some bad packages in our recommendations, as some projects seem to use malformed dependency strings (e.g. _better-assert@1.0.2@better-assert@1.0.2). These could be filtered out at the time we fetch repositories from GitHub, by applying a regex to the name of each package and only accepting those which are valid names. A larger issue is adversarial attacks that could take advantage of our filtering criteria to promote their own packages. For example, a person could create a number of repositories with a meaningless collection of dependencies, then create a bunch of dummy accounts which they could use to star the repositories and get them above our inclusion threshold (currently >1 star). To mitigate this, we could increase the follower count requirement on repositories we fetch from GitHub, but it's not a robust solution and reduces the amount of data we can capture. We should consider more robust spam detection measures in addition to changing our filter criteria.

Another area of additional research would be to investigate the right weights to use for the PkgPkr Score. If we started crowdsourcing feedback on recommendation quality, we could conceivably use a neural network to learn the right weights from this feedback. This would also make it easier to add new metrics and retune the recommendation engine accordingly.

Given that the same recommendation can appear multiple times for different dependencies, we could also explore adding an additional weighted component to the score which represents the number of times a recommendation

appears. For example, a recommendation that appears 10 times for 10 different associated packages could be scored more highly than another recommendation which only appears once.

Finally, we make no attempt to include non-runtime dependencies, nor do we deal with the cold-start problem when someone has no packages but wants a recommendation to get started with their project. We also don't accommodate repositories with a non-traditional file structure. If a package.json or requirements.txt file isn't in the root of a repository, our system won't pick it up. These are all improvements that would improve the utility of Package Picker.

**Related work**

Programmers spend up to 57% of their daily work researching (Sadowski et al., 2015), as well as learning new APIs, finding code samples, and helping with implementation (Bajracharya et al., 2012). *Code search* (searching by extracts of code) is a common way developers find information about their code but is quite different from typical search engine queries. Code search engines usually focus on searching source code and returning a list of links to code that matches the query, but this classical approach to measuring performance of ranking algorithms may not measure the quality of results correctly (Martie, 2017). Martie also suggested adding synonyms, related words to the keywords in the query to improve these recommendation algorithms. Adding descriptive terms like words from documentation, tags from the community, or natural language on web pages may also help (Chatterjee et al., 2009; Zagalsky et al., 2012), but these only had marginal improvements over regular code search. While there are many search engines available for software packages, they are typically keyword-based and require developers to have a good idea of what packages are available before they start. Furthermore, search engines by their very nature provide very little information about the developer's application, so they have only a very limited ability to suggest packages that would fit an application. These limitations result in a tendency to choose packages primarily by popularity, which is not always a good proxy for functional requirements.

Niu et al. (2017) implemented a machine learning algorithm application called Codota Learning-to-Rank which outperformed a commercial online code example search engine by 44%. Codota used several criteria including textual similarity between a query and code, popularity that represents a higher acceptance rate, code metrics like line length, number of identifiers, or call sequence length, and context similarity. Codota produced a normalized discounted cumulative gain and a 48.42% improvement in expected reciprocal rank using the RankBoost learning-to-rank algorithm (Freund et al., 2003). These findings were helpful when considering how to build Package Picker.

**References**

Allaho, M. (2014): Recommendation Services for Open Source Software Community. Retrieved February 16, 2020 from http://search.proquest.com.ezp-prod1.hul.harvard.edu/docview/1656184887?accoun tid=11311

Bajracharya, Sushil, and Krishna Lopes. "Analyzing and Mining a Code Search Engine Usage Log." Empirical Software Engineering, vol. 17, no. 4-5, 2012, pp. 424–466.

Bogart C, Kästner C, Herbsleb J, Thung F (2016) How to Break an API: Cost Negotiation and Community Values in Three Software Ecosystems. In: Proceedings of the 24th International Symposium on the Foundations of Software Engineering (FSE), pp 109–120

Buse, Raymond P L, and Westley R Weimer. "Learning a Metric for Code Readability." IEEE Transactions on Software Engineering, vol. 36, no. 4, 2010, pp. 546–558., doi:10.1109/tse.2009.70.

Chatterjee, Shaunak, et al. "SNIFF: A Search Engine for Java Using Free-Form Queries." Fundamental Approaches to Software Engineering Lecture Notes in Computer Science, 2009, pp. 385–400., doi:10.1007/978-3-642-00593-0_26.

Christidis, K. (2012): Combining Activity Metrics and Contribution Topics for Software Recommendations. Retrieved February 16, 2020 from https://ieeexplore-ieee-org.ezp-prod1.hul.harvard.edu/document/6233408

Freund, Yoav, et al. "An Efficient Boosting Algorithm for Combining Preferences." Journal of Machine Learning Research, vol. 4, no. 6, 2004, pp. 933–969.

Kim, Jinhan, Lee, Sanghoon, Hwang, Seung-won, AND Kim, Sunghun. "Towards an Intelligent Code Search Engine" AAAI Conference on Artificial Intelligence (2010): n. pag. Web. 18 Feb. 2020

Malheiros, Y. et. al. (2012): A Source Code Recommender System to Support Newcomers. Retrieved February 16, 2020 from https://ieeexplore-ieee-org.ezp-prod1.hul.harvard.edu/document/6340250

Martie, Lee Thomas. "Understanding the Impact of Support for Iteration on Code Search." 2017, p. 323.

Matek, T. and Zebec, S. (2016): GitHub Open Source Project Recommendation System. Retrieved February 16, 2020 from https://arxiv.org/abs/1602.02594

Mens, K. and Lozano, A. (2014): Source Code-Based Recommendation Systems. Retrieved February 16, 2020 from https://www.researchgate.net/publication/258630290_Source_Code-Based_Recomm endation_Systems

Niu, Haoran, et al. "Learning to Rank Code Examples for Code Search Engines." Empirical Software Engineering, vol. 22, no. 1, 2017, pp. 259–291.

Robillard, M. P., Maalej, W., Walker, R. J., & Zimmermann, T. (2014). Recommendation systems in software engineering. Berlin: Springer.

Sadowski, Caitlin, et al. “How Developers Search for Code: a Case Study.” Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, 2015, pp. 191–201.

Thummalapenta, Suresh, and Tao Xie. “Parseweb.” Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering - ASE 07, 2007, doi:10.1145/1321631.1321663.

Zagalsky, Alexey, et al. “Example Overflow: Using Social Media for Code Recommendation.” 2012 Third International Workshop on Recommendation Systems for Software Engineering (RSSE), 2012, doi:10.1109/rsse.2012.6233407.

Zhong, Hao, et al. “MAPO: Mining and Recommending API Usage Patterns.” Lecture Notes in Computer Science ECOOP 2009 – Object-Oriented Programming, 2009, pp. 318–343., doi:10.1007/978-3-642-03013-0_15.

# Requirements

The purpose of this project was to build a package recommendation engine that helps software developers choose better packages, faster. We had twelve customer-facing requirements.

1. Fetch a list of repositories from GitHub that have at least 100 followers and contain a package.json in the root directory.

   ***Note:*** *The package.json file contains npm relevant metadata, including a project's dependencies. We also expanded the scope of the project to include PyPI, whose equivalent file is named 'requirements.txt' by convention.*

2. Filter out any repositories whose package.json does not include a start script.

   ***Note:*** *Because our tool is designed to recommend packages for applications, we were asked to filter out libraries from our training data, which typically do not have a start script (often used, among other things, to start an application).*

3. For each of these repositories, retrieve the list of first-order dependencies from the package.json.

   ***Note:*** *A package.json file consists of a top-level JSON object, one key of which is named 'dependencies' with a list of library names and their versions as a corresponding value. Additionally for PyPI projects, the requirements.txt file contains a list of project dependencies, separated by newlines, and an (optional) specification (e.g. >=, <, ==, etc.) followed by the desired version.*

4. Filter out any packages with fewer than 1,000 downloads over the last month.

   ***Note:*** *This threshold was based on an internal npm support policy. Package maintainers are allowed to automatically unpublish their packages unless they exceed 3,000 downloads. This is to balance self-service with a desire to avoid breaking downstream dependencies (the left-pad incident). We chose 1,000 downloads to be slightly more permissive, while still filtering out*

*those that are not popular and thus might contaminate our recommendations.*

5. Accept a set of dependencies as input to the recommendation engine.

   ***Note:*** *We wanted to choose an ecosystem agnostic naming scheme to make it easier to support additional ecosystems beyond npm. We ended up using the Package URL (PURL) format. The full PURL spec can be found at https://github.com/package-url/purl-spec.*

6. Return a list of *good* package recommendations that are based on other applications with dependencies similar to yours.

   ***Note:*** *While this requirement was intentionally vague to allow us room for experiment, we ultimately ended up defining "good" as a combination of package similarity, popularity, and download trends.*

7. Allow users to authenticate using GitHub.

   ***Note:*** *Because most developers use GitHub, it ended up being far more convenient to use a Github OAuth App to authenticate our users than to build authentication ourselves. Additionally, we store the token in a user session to allow access to user metadata and repositories.*

8. Retrieve a list of their repositories.

   ***Note:*** *After authenticating with the GitHub OAuth App and getting a token, it is straightforward to retrieve their public and private repositories (including the package.json or requirements.txt files in those repositories).*

9. Display any repositories that have a package.json [or requirements.txt] with at least one dependency.

   ***Note:*** *Because Package Picker was not required to support all ecosystems, we filter out any repositories that aren't supported. This simplifies the list and makes it clearer for what repositories a user can request recommendations. To further convenience the user, we paginate the list of repositories and show recommendations for the master branch by default.*

10. Let people select one of their repositories and retrieve a list of package recommendations for that repository.

    ***Note:*** *The ultimate goal of Package Picker is to let users retrieve package recommendations for a particular repository. Because this list ended up containing 10,000 recommendations, we expanded the scope of this requirement to include pagination, search, and sorting (including multi-column sort). We also ended up showing the name, categories, monthly download count, and score for each recommendation. As a further convenience, we also allow users to retrieve recommendations for a non-master branch if desired.*

11. Group package results by category.

    ***Note:*** *Users are typically looking for a particular type of package. To accommodate this, we show the categories associated with each package and let the user click a category label to filter the results to only packages with that category. Categories are the aspects to which dependencies relate to (e.g. web, api, logger, cookie)*

12. Allow the user to filter their package recommendations by name or category.

    ***Note:*** *In addition to category filtering, we also allow the user to identify the most popular packages from the list of recommendations by their raw download count. Even though download count isn't necessarily a good measure of whether a package is a good choice for the user's application, it is a common criteria for package selection and so this requirement was created as a concession to that particular desire.*

There are four main components to our solution: a ML pipeline, a database, a web service, and the GitHub GraphQL API. These requirements break down into the following four components.

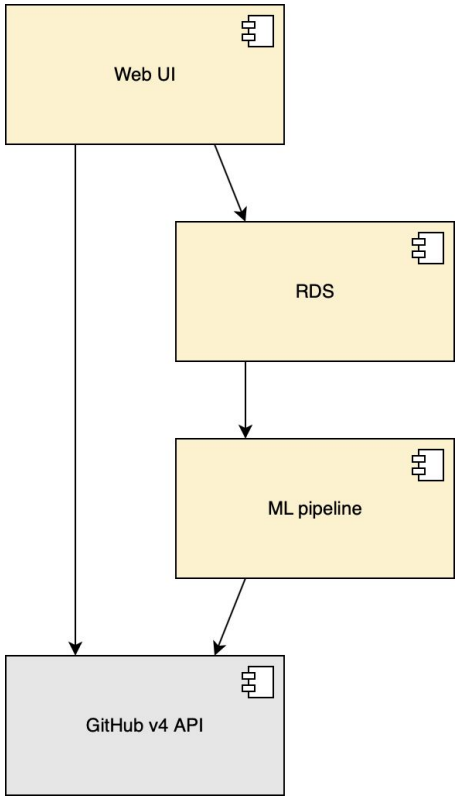


Figure 5: The four main components of Package Picker

# Design diagrams

The system is built around two services, a ML pipeline service (Python and Apache Spark) and a web service (Django). These services live in Docker containers hosted on AWS Fargate, and use PostgreSQL hosted on AWS RDS to store and fetch recommendation data. The ML pipeline service fetches data from GitHub, npmjs.com, and pypi.org. The web service uses GitHub for authentication and for fetching a user's repositories.

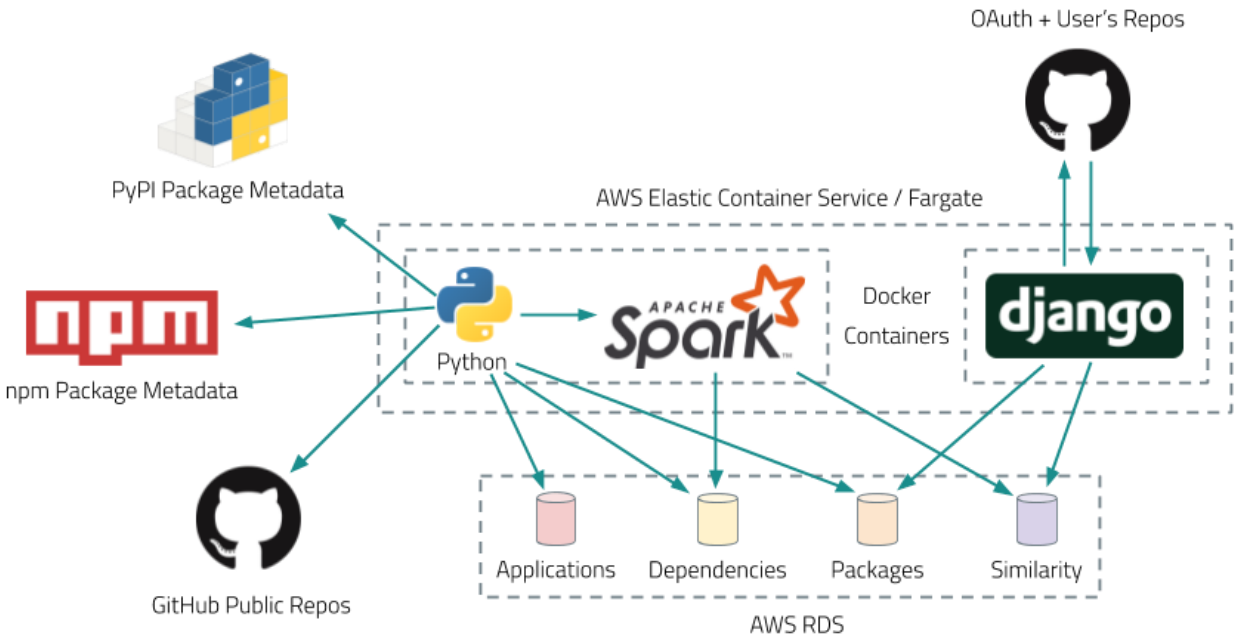


Figure 6: System Architecture

Our CI/CD pipeline is built around GitHub Actions, AWS ECS, AWS CloudWatch Rules, and AWS Step Functions. When code is committed to GitHub, unit tests are automatically triggered. When PRs are merged into the dev branch, the Docker images are built and deployed to ECR and the corresponding task definitions are updated, which causes AWS ECS to update any running web server instances. Finally, we have a CloudWatch rule which executes a State Machine once per day, this State Machine manages a single run of our ML pipeline. Because the full ML pipeline currently takes more than 50 hours to complete, we have disabled the CloudWatch rule and run the ML pipeline manually instead.

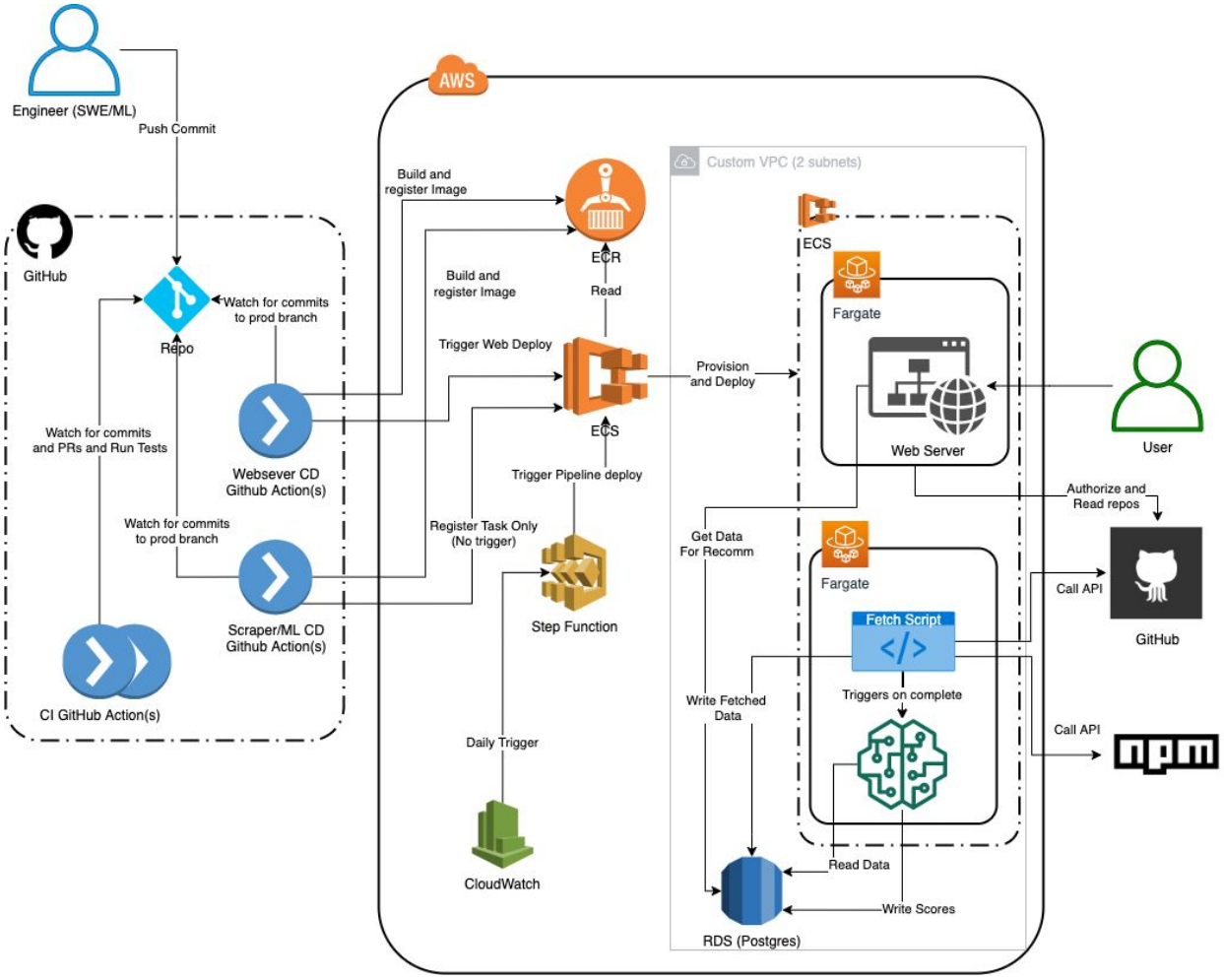


Figure 7: System Architecture and CI/CD process

The ML pipeline is driven by run_scraper.py. It imports modules to work with GitHub, npm, and PyPI data. The npm module uses a month_calculation.py utility module to compute download statistics. Our test suite also exercises these modules.

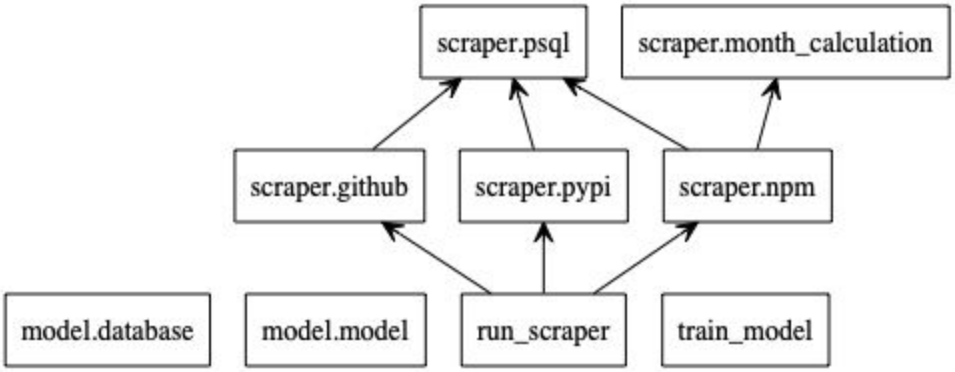


Figure 8: ML Pipeline File Diagram

Finally, the web server has a fairly traditional structure. It renders views, while calling the RecommenderService for recommendations and github_utils for various utility functions, such as getting a list of a user's repositories from GitHub or fetching dependencies for a particular repository.

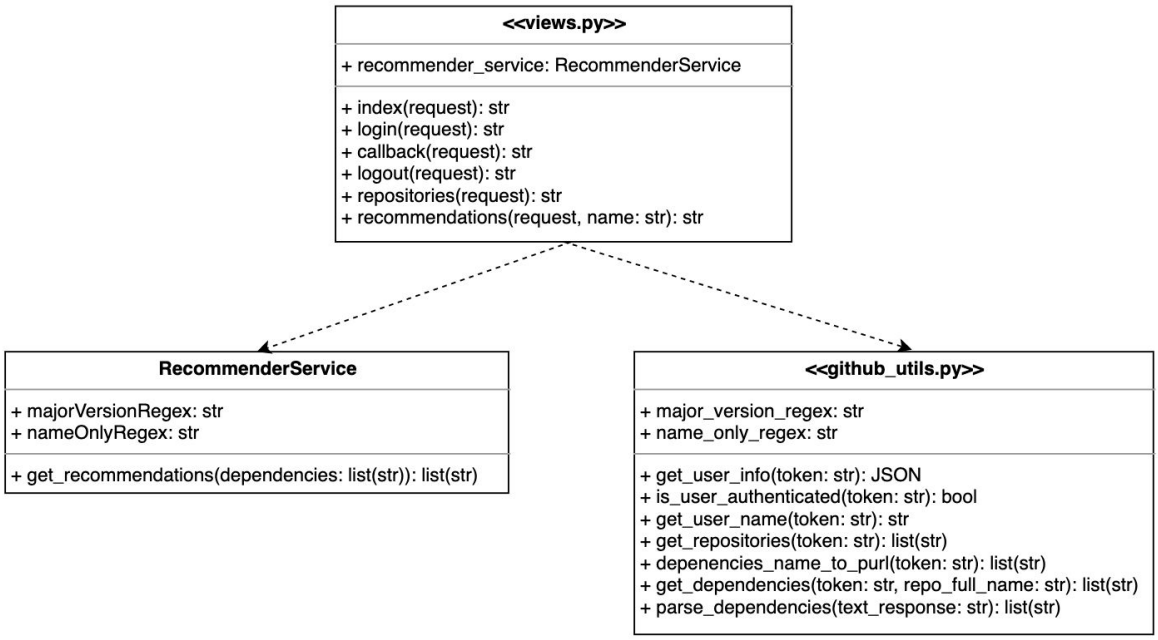


Figure 9: Web server diagram

# Design specifications

Our system is designed to provide recommendations for npm (JavaScript) and PyPI (Python) packages. We only consider runtime dependencies, not development or optional dependencies, mainly for simplicity but also because some ecosystems (e.g. PyPI) don't include a concept of non-runtime dependencies in their manifest files. We built the system for npm initially because it is the largest package ecosystem and thus has the most data for training. We added support for PyPI because our project is built in Python and we wanted to test our project on itself. Adding new languages should not be difficult based on our experiences with PyPI, though we would want to create more of a framework to guide and simplify this work for future developers.

The machine learning pipeline takes a three phase approach: 1) querying GitHub for all repositories containing a package.json (npm) or requirements.txt (PyPI) manifest file with dependencies, 2) querying npmjs.com and pypi.org for package metadata related to those dependencies, and finally 3) building the cosine similarity matrix.

First, the pipeline builds its training set by querying all repositories in GitHub for those containing a npm or PyPI manifest file with dependencies (packages). Because the GitHub GraphQL API only allows up to 1,000 repositories to be fetched in a single search query, we fetch repositories based on their created date, one day at a time. To be fetched, a repository must have more than 1 star and be classified as a JavaScript or Python repository by GitHub. We include the star criteria to filter out repositories that aren't popular and are probably of low quality. Once fetched, a repository must have a package.json or requirements.txt file in its root folder to be included in our training dataset.

The repository and package name (including version) are persisted to the applications and packages tables in the database. We differentiate between major versions of a package, but not minor or patch versions. To make it easier to process the data, we generate an integer id for each application and package, respectively. These application and package ids are unique to their respective tables. We then store the mapping from application to package, referencing these ids as foreign keys in the dependencies table. We avoid duplicates by using PostgreSQL INSERT ON CONFLICT statements to perform upserts. If the data doesn't already exist, it is inserted, otherwise it is updated.

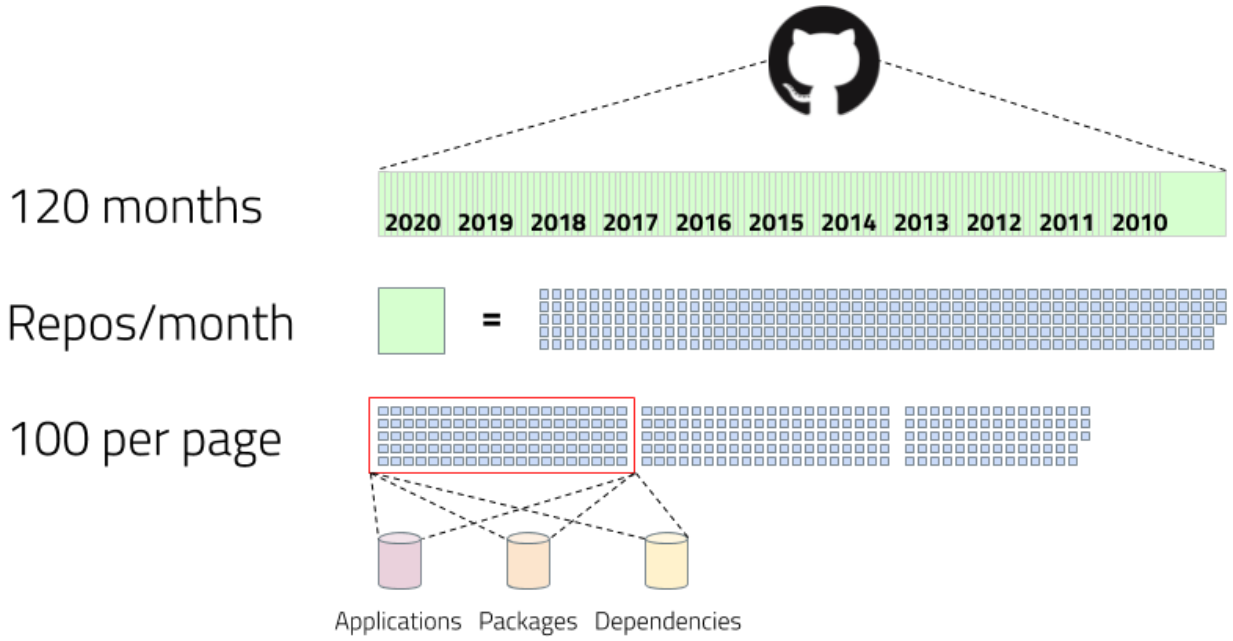


Figure 10: GitHub query process

Second, the machine learning pipeline goes through all packages stored in the database, and retrieves the metadata from npmjs.com and pypi.org for each. This metadata includes the categories, the date each package was last modified, and download counts for last month and the same month a year prior. Categories in particular are arrays of strings that are optionally associated with each package, and provide the package author with a means of tagging the purpose of a package. Download counts are used to calculate the absolute trend and relative trend component scores, while the categories and package modified date are used solely to provide the user with additional filtering, sorting, and search options.

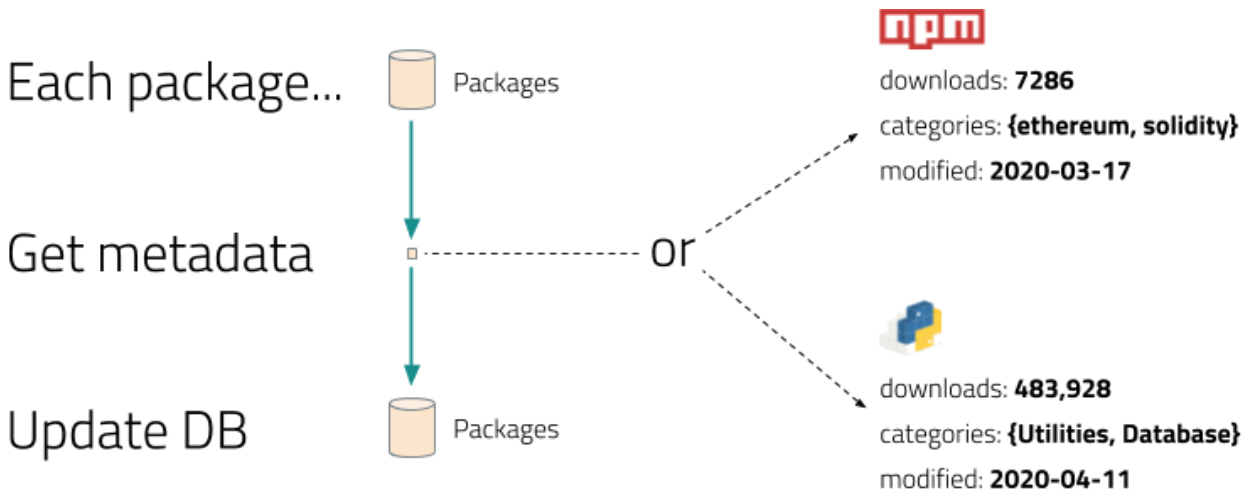


Figure 11: Package metadata query process

Third, the machine learning pipeline determines the similarity of all packages in the corpus. It retrieves all records from the dependencies table, and creates a vectorized matrix of the application to package mapping using sparse vectors. Next, it computes the package cosine similarity of all columns in the matrix, which results in a package similarity matrix where each cell is a value between 0 and 1 inclusive, representing the degree to which two packages tend to appear in the same package.json or requirements.txt files together. We iterate through all cells in the matrix, ignore cells with a zero score to save space, and write the rest to our similarity table, which is then used by the web server to generate recommendations. The similarity table has three columns, the package_a and package_b columns are the packages with a non-zero similarity score, and the similarity column contains the score.

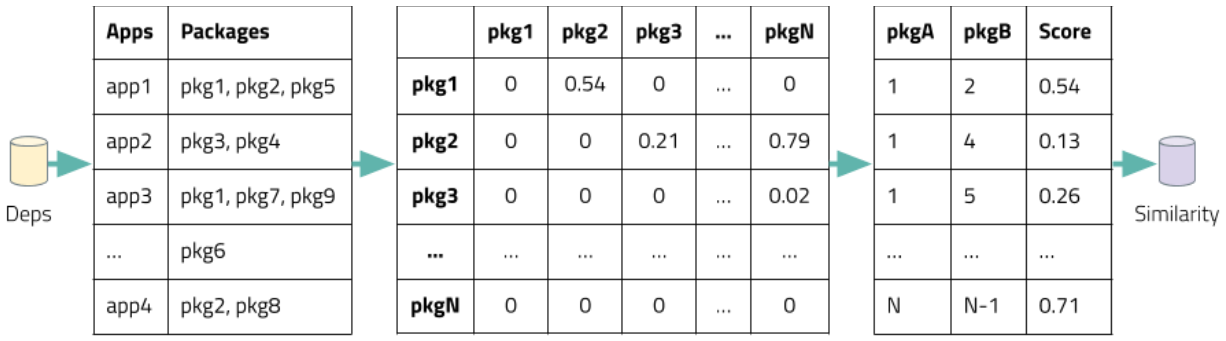


Figure 12: Process of generating similarity scores

To retrieve recommendations, a user can either input dependencies manually, or log in to GitHub and retrieve a list of their repositories from GitHub with a package.json or requirements.txt file, respectively. When they select one of these repositories, we fetch up to 10,000 packages that have the highest similarity scores relative to the packages in the user's repository. We refrain from returning packages that already exist in the user's repository. It is important to note that the recommendation engine does *not* suggest package upgrades at present. For example, if you are using React 15, the system will suggest packages that are frequently used with React 15, but it is unlikely to suggest upgrading to React 16.

For each recommendation, we calculate its PkgPkr Score, which is the weighted sum of 1) its similarity score, 2) popularity in terms of how many other distinct packages appear together with it, 3) absolute growth in terms of year-over-year downloads, and 4) percentage growth in terms of year-over-year downloads. We log transform the popularity and download counts prior to computation of the scores because the data distribution is highly skewed. The resulting score is an integer between 1 and 10, inclusive, where 10 indicates a very strong recommendation. The web server has filtering and sorting capabilities which allow the user to browse these recommendations and their associated metadata, including the computed PkgPkr score.

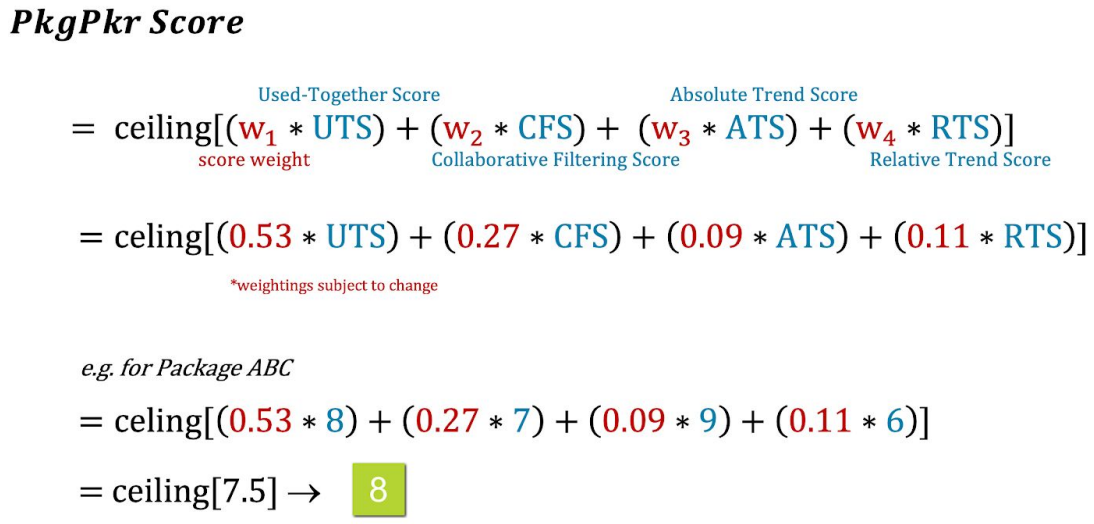


Figure 13: Components of a PkgPkr Score

One of the nice benefits of hosting our web service on AWS ECS is that it is trivial to scale it out horizontally. When we create the cluster, we define how many instances we want to be running,

and AWS ECS spins up new AWS Fargate instances from our task definition file until it meets the desired number. This also allows for zero-downtime when deploying new changes, as new instances with the changes are spun up and the load balancer switched over before the old instances are taken down.

**Include diagrams showing the location of various packages with respect to the client/server, database schema.**

Our database schema consists of four tables: applications, packages, dependencies, and similarity. These tables store information about GitHub repositories, npm and PyPI packages, mapping between applications and packages, and similarity scores for each pair of packages, respectively.

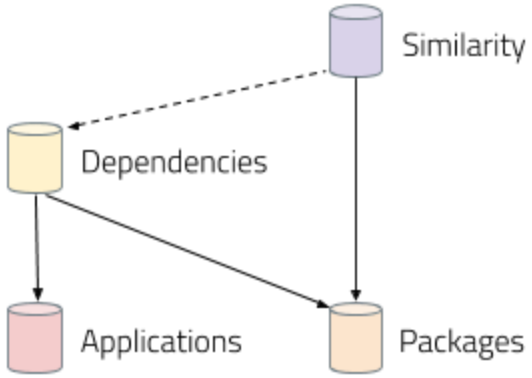


Figure 14: Relationship of tables in AWS RDS

The applications table contains an integer id, used when computing the similarity matrix, and several other fields that store metadata related to each repository.

```
CREATE TABLE applications (
  id SERIAL PRIMARY KEY,
  url TEXT NOT NULL,
  name TEXT NOT NULL,
  followers INTEGER,
  hash TEXT NOT NULL,
  retrieved TIMESTAMPTZ NOT NULL,
```

```
  CONSTRAINT unique_url UNIQUE (url)
);
```

The packages table also contains an integer id, for computation of the similarity matrix. It too contains metadata related to each package, some of which is used for computation of the PkgPkr Score and the rest used for display, filtering, and sorting in the UI.

```
CREATE TABLE packages (
  id SERIAL PRIMARY KEY,
  name TEXT UNIQUE NOT NULL,
  short_name TEXT,
  url TEXT,
  monthly_downloads_last_month INTEGER,
  monthly_downloads_a_year_ago INTEGER,
  absolute_trend INTEGER,
  relative_trend INTEGER,
  categories TEXT[],
  popularity INTEGER,
  bounded_popularity INTEGER,
  modified TIMESTAMPTZ,
  display_date TEXT,
  retrieved TIMESTAMPTZ NOT NULL
);
```

The dependencies table maps each application to the packages it depends on. It references the integer ids we created for each application and package. This table is consumed by the ML pipeline when generating the similarity matrix.

```
CREATE TABLE dependencies (
  application_id INTEGER REFERENCES applications (id),
  package_id INTEGER REFERENCES packages (id),
  CONSTRAINT unique_app_to_pkg UNIQUE (application_id, package_id)
);
```

Finally, the similarities table contains the results of generating the similarity matrix. Each row represents a pair of packages and their non-zero similarity score. We don't include packages with a zero similarity score in this table because they aren't used by the recommendation engine and would consume significantly more space, since the similarity matrix is quite sparse.

```
CREATE TABLE similarity (
  package_a INTEGER REFERENCES packages (id),
  package_b INTEGER REFERENCES packages (id),
  similarity FLOAT(4) NOT NULL,
  bounded_similarity INTEGER,
  CONSTRAINT unique_pkg_to_pkg UNIQUE (package_a, package_b)
);
```

Finally, we define a few indices on the similarity table for performance. The web service requests recommendations from the similarity table directly, so it needs to be performant.

```
CREATE INDEX ON similarity (package_a);
CREATE INDEX ON similarity (package_b);
CREATE INDEX ON similarity (similarity);
```

**Explain interesting design choices, or particularly complex issues that we managed to work through**

Because we use cosine similarity to determine whether two packages are related, the dimensionality of our data is a major concern. Our initial requirements filtered out any repositories with fewer than 100 followers, which resulted in a very small data set of 19,000 packages and a similarity matrix that was 99.88% sparse. Part of this sparseness was due to the fact that we initially considered each package version separately (e.g. 1.0.1, 1.2.0-rc2, etc), which added an order of magnitude more columns to our similarity matrix than using package name alone. We addressed these issues in two ways:

First, we lowered the followers requirement in our ML pipeline to increase the number of repositories fetched. The reason we originally filtered out repositories with fewer than 100 followers is because we didn't want low quality repositories to skew our results. While low quality results are still a concern, this extra data helped us start generating recommendations where previously there were few or none. We also discovered that we could fetch 10x more repositories by changing our GitHub query from a topic search to a language search.

Second, we stripped out everything but the major version number for each package to reduce the number of columns in our dataset. We realized that most packages adhere to semver, which means that upgrading to the latest minor or patch version of your current major version is not only desirable from a security standpoint, but is also unlikely to break the functionality of your application. Thus, we realized we could get away with only recommending major versions, and assume people will just install the latest minor and patch version available for that major version. Even though we stopped differentiating between minor and major versions, we still had almost 200,000 distinct package/version combinations thanks to the increased data ingest. This, plus the fact that we were generating a similarity matrix across two package ecosystems (npm and PyPI) with little overlap, resulted in the sparsity of our similarity matrix staying about the same, at 99.93% sparse.

As a result of these adjustments, we increased the average number of recommendations (i.e. package pairs with a non-zero similarity score) by over 300x. However, this also made the performance of our web server too slow, because we were fetching package metadata from npmjs.com on demand right before loading the recommendations page. To compensate for the much larger number of recommendations we were now retrieving, we began fetching metadata from npmjs.com and pypi.org at the same time that we fetched repositories from GitHub, storing the metadata in our database for retrieval at recommendation time. We also capped the maximum number of recommendations to 10,000.

**Explain our choices of language/ platform. Discuss how the choices worked out.**

We decided to use Python instead of Java for both the ML pipeline and web server, due to Python having better machine learning libraries in general, and team familiarity with Django in particular. Apache Spark was the one Java-based project we continued to use in our service, but because it has

an excellent Python front-end wrapper (PySpark), this posed no issues to our project beyond mildly confusing our linter (pylint) which was unable to account for wrapper libraries. In all other cases, our use of Python has posed no significant risk to the project, and is particularly well-suited to the scripting of our ML pipeline. It also allowed us to quickly experiment with different machine learning approaches in Jupyter Notebook, including an alternating least squares (ALS) approach. We ultimately decided against using ALS because it's better suited for user-to-item collaborative filtering, which would have prevented us from fetching recommendations for repositories our recommendation engine had not yet been trained on.

From a platform perspective, we were initially learning toward Heroku due to its ease of use, but eventually standardized on AWS because of its overall maturity and support for containers. We had some difficulty getting a CI/CD flow working on AWS Elastic Container Service, but once we passed the learning hurdle, we've had no issues working with it since. In addition, we've added the ability to automatically provision the 40 required AWS resources with Terraform, which has given us more confidence that others will be able to replicate our project.

**What are some unique aspects of your project in terms of features?**

We aren't aware of any tools that use a project's existing dependencies to recommend packages that they should consider using in their project. Our approach applies cosine similarity, a standard approach to item-item collaborative filtering, to package recommendation, which seems like a new application of the technique. We also augment the results of cosine similarity with more traditional metrics, like popularity and download trends, which gives our recommendation engine more flexibility. Because we load 10,000 of the best recommendations into the front-end, our project allows for near-zero lag as users search, filter, and sort through the recommendations.

Our recommendation engine is essentially a database. We populate a similarity table with pairs of packages and their similarity score, so getting recommendations simply requires a database query, which is very fast and we have successfully scaled our similarity table up to 22 million rows with very minimal degradation in performance. In fact, our production database is running on an AWS T2.small RDS instance, so we have lots of room to scale up with our existing architecture.

While not entirely unique to our project, our use of GitHub to authenticate users and fetch their repositories so they can get recommendations is a very streamlined user experience. We also provide a manual input mode so people can paste in their package.json or requirements.txt and get recommendations without having to grant access to their GitHub account. This also allows people who don't use GitHub to use our tool. We've even added an open API that other developers can use in their own projects.

## Wireframes

Figures 15-20 below show the original wireframes for the UI before implementation.


Figure 15: Homepage wireframe

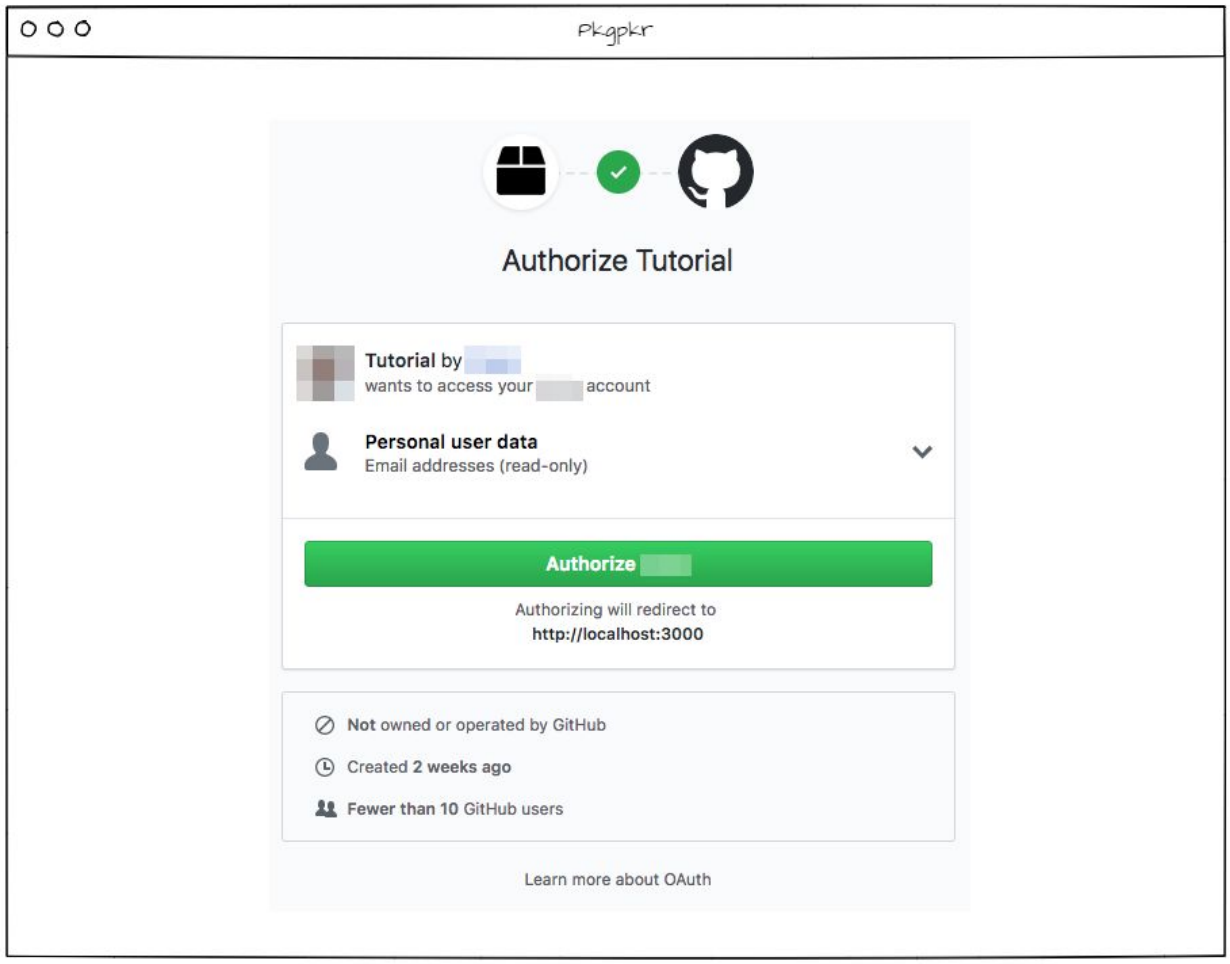


Figure 16: GitHub login screen

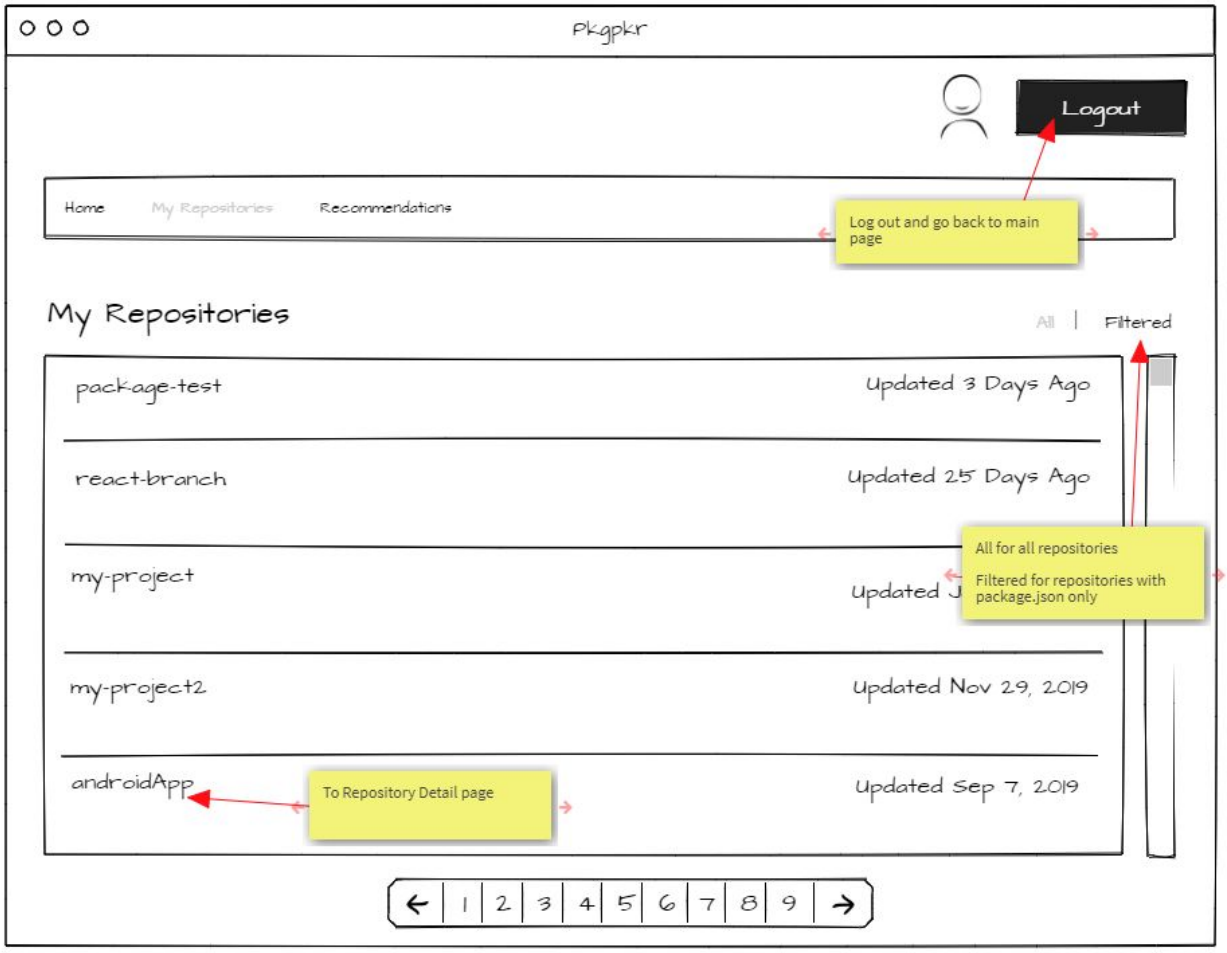


Figure 17: List of the user's repositories

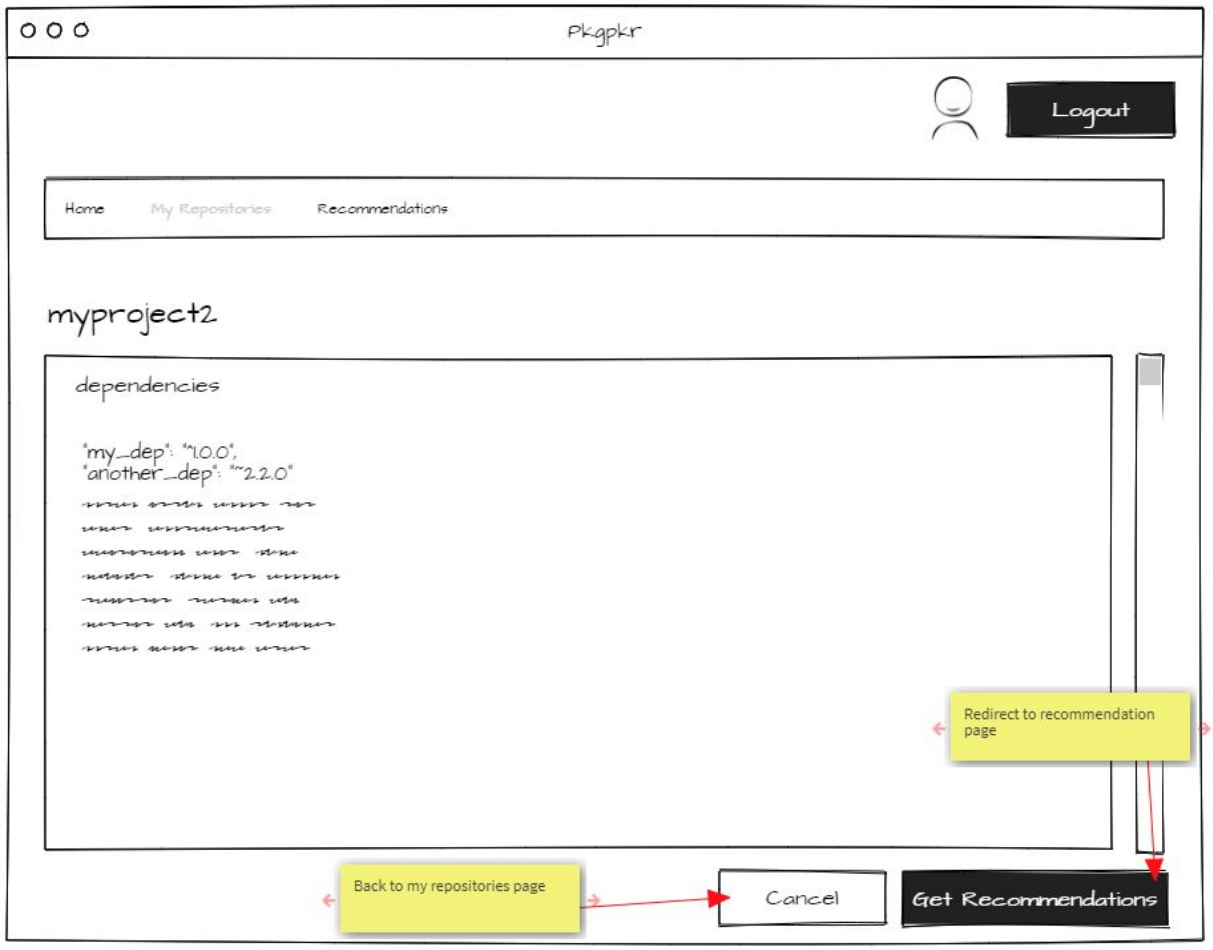


Figure 18: List of a repository's dependencies

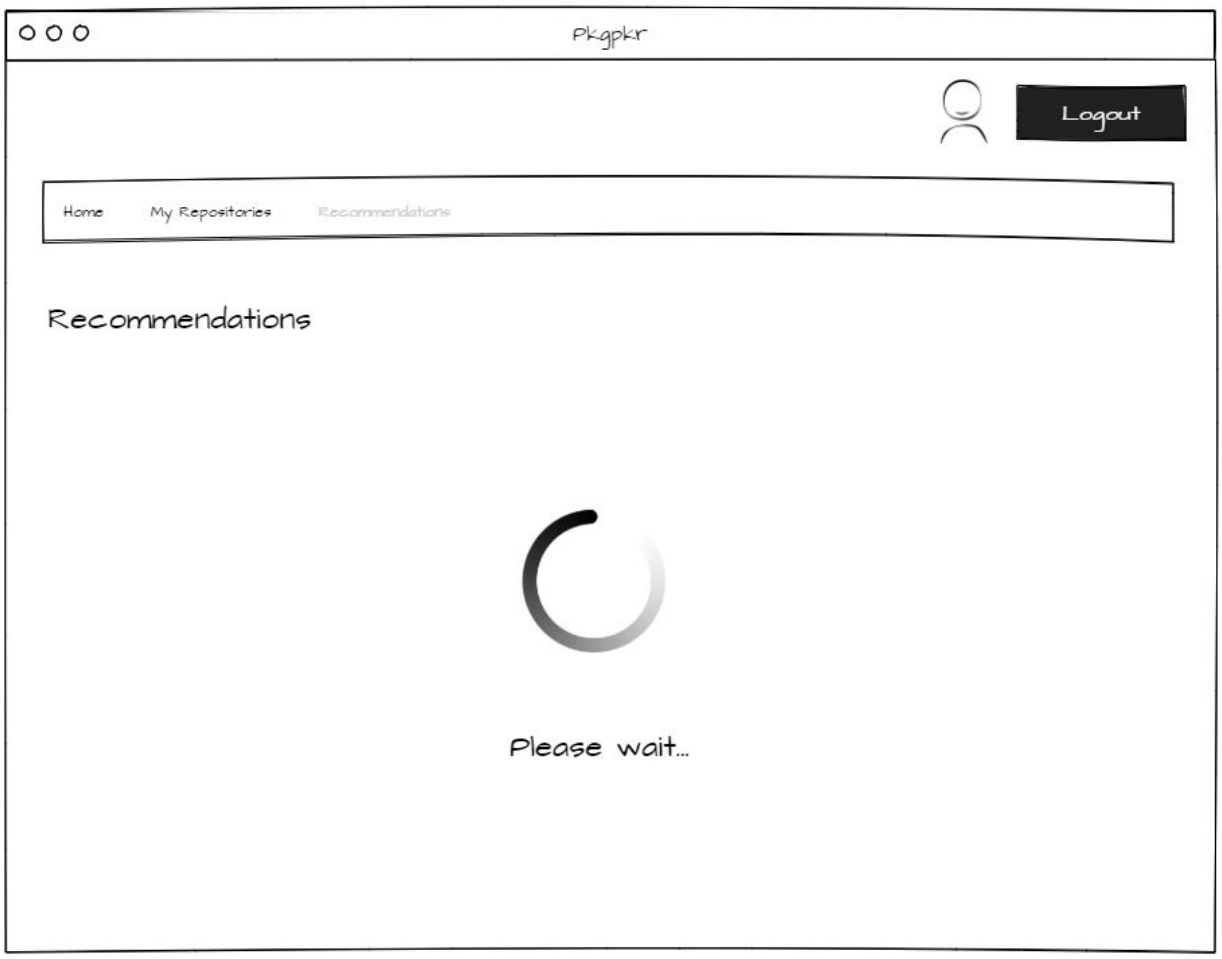


Figure 19: Loading screen while fetching recommendations

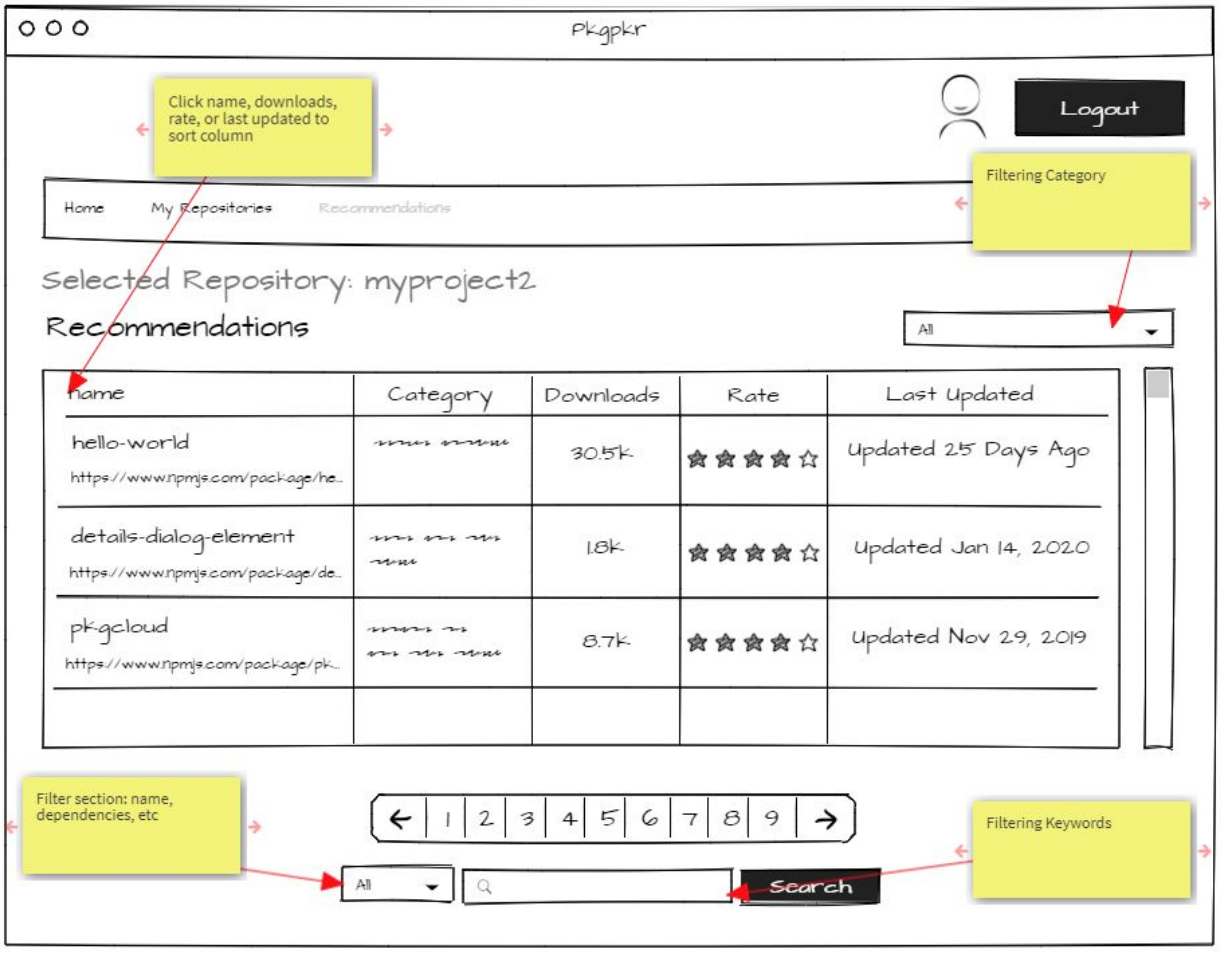


Figure 20: List of recommendations

# Changes to the original design

- We're using PostgreSQL to store the recommendation engine's similarity matrix instead of using a Parquet file stored in S3.
- We decided against React, since JQuery had everything we needed and our UI is currently very simple.
- We decided to combine the ML pipeline and recommendation engine into a single service. Building the cosine similarity matrix is cheap relative to fetching all the data, so we rebuild it every time we fetch new data.

- We decided to use Docker containers deployed to AWS ECS and AWS Fargate via GitHub Actions instead of deploying code to EC2 instances manually. This improved our CI/CD automation significantly.
- We added some extra sorting and filtering capabilities to our web UI since we found a great table control which provides that functionality for free.
- We've moved from building the cosine similarity matrix and storing it in S3 for use by the web service, to creating usable mapping output that is stored in the DB and consumed by the application directly.
- Changed standard for repositories (from 100+ followers to 2+ star counts), and haven't started filtering out packages with <1,000 monthly downloads. Originally, the idea behind these filters was to keep out low-quality repositories and packages. We lowered the filters in Milestone 2 to increase the amount of data available for training. Now that we're collecting more than enough data, we should re-evaluate reinstating these original requirements or at least adjusting the thresholds as far upward as makes sense.
- Added a new branch feature that allows a user to request recommendations for a particular branch of their GitHub repository.
- Many small UI design changes to improve usability after running two user tests with external users.

# Test results

We have ~35 tests in total, with 80% test coverage for the ML pipeline and 99% test coverage for the web service. All tests run automatically when code is committed to GitHub. We've written unit tests with Python's unittest framework, and UI feature tests with Selenium. The latter goes through the process of logging in and fetching recommendations through both repository and manual input paths, for both npm and PyPI. The current state of the tests can be seen from the GitHub Action badges in our repository's README.md.


Figure 21: Different CI/CD integrated as visible on the Package Picker repository

We've had to resort to manual testing of the PySpark and post-AWS deployment pieces. Generation of cosine similarity scores via PySpark is difficult to unit test because they have very specific environment requirements (e.g. Java 8), and we haven't gotten around to mocking the tests yet. We've also resorted to manual testing of the ML pipeline because it takes 50+ hours to run, so running automated tests would be too time consuming. Automated testing of the web server post-deployment would be easier, we just haven't added health checks yet. In lieu of those, we manually trigger the Step Function which runs the ML pipeline on AWS and observe that it finishes successfully. We also open pkgpkr.com in a browser after deployments to make sure that the site is still running.

Our manual testing has revealed some weak points in our design. First, our ML pipeline initially had a tendency to error out after running for several hours, which made it difficult to fix issues and validate the fix in a timely fashion. Second, hosting our live instance on AWS with a proper domain name (pkgpkr.com) has surfaced some integration issues between Django and GitHub OAuth. For example, if the GitHub App hasn't been set up properly or the `CLIENT_ID` or `CLIENT_SECRET` haven't been passed correctly to the web service, it will show an error page when users attempt to login. These aren't things that are easy to test locally, so we've compromised by architecting the system so we can pass it a GitHub Token instead of using GitHub OAuth for testing.

We set up scripts to run Jmeter load testing to see how our application handles increased usage. The Jmeter scripts were written in the Jmeter application and used the Selenium Chromium webdriver jp@gc to simulate user actions as part of the process. The Jmeter testing supported running in non-headless mode for one user to see how everything is clicking successfully, and then an arbitrary number multiple-user (e.g. 10+) headless mode in order to see how the performance will be. The potential downsides of this type of testing is that while we are testing user interaction in the UI, the resource limitations of the computer the test is being run at could be limited to smaller numbers of concurrent users (instead of tens of millions). Further work on Jmeter testing would be to take out the UI portion and focus more on API specific hits.

In terms of validating the recommendation engine itself, We assess the performance of our approach from three angles. 1) Is the system capable of recommending packages that are deliberately removed from a project? 2) Is the system capable of recommending a package based on

the package's own dependencies? 3) Is the system able to make recommendations that are at least on par with a more traditional search engine?

First, we take a popular project and remove one package at a time from its package.json. If our recommendation engine recommends each missing package back to us, it's a good sign that our system recognizes which packages are commonly used together. We use express 4.17.1, a web framework for Node.js that has 30 dependencies. As shown in Figure 22, our system is able to recommend each missing package except qs@6, safe-buffer@5, and serve-static@1, with scores between 4 and 6.

| **Package** | **Score** | | **Package (cont…)** | **Score** | | **Package (cont…)** | **Score** |
|---|---|---|---|---|---|---|---|
| accepts@1 | 6 | | escape-html@1 | 5 | | qs@6 | - |
| array-flatten@1 | 6 | | etag@1 | 6 | | range-parser@1 | 5 |
| body-parser@1 | 6 | | finalhandler@1 | 4 | | safe-buffer@5 | - |
| content-disposition@0 | 6 | | fresh@0 | 6 | | send@0 | 5 |
| content-type@1 | 5 | | merge-descriptors@1 | 6 | | serve-static@1 | - |
| cookie@0 | 5 | | methods@1 | 5 | | setprototypeof@1 | 5 |
| cookie-signature@1 | 5 | | on-finished@2 | 6 | | statuses@1 | 5 |
| debug@2 | 6 | | parseurl@1 | 5 | | type-of@1 | 6 |
| depd@1 | 5 | | path-to-regexp@0 | 6 | | utils-merge@1 | 5 |
| encodeurl@1 | 5 | | proxy-addr@2 | 5 | | vary@1 | 6 |

Figure 22: Ability of system to recommend packages that are purposely removed from express.js

To verify this result against a baseline, we choose a completely unrelated project with a completely non-overlapping set of dependencies, retrieve scores for all 30 of express's dependencies, and see

how the scores compare. We use cucumber.js 6.0.5, an automated test framework for Node.js. As shown in Figure 23, scores are 1-2 points lower for all packages except qs@6, safe-buffer@5, and serve-static@1. This is what we would expect if the context of a package has an impact on its PkgPkr Score.

| **Package** | **Score** | | **Package (cont…)** | **Score** | | **Package (cont…)** | **Score** |
|---|---|---|---|---|---|---|---|
| accepts@1 | 4 (-2) | | escape-html@1 | - | | qs@6 | 4 |
| array-flatten@1 | 4 (-2) | | etag@1 | 4 (-2) | | range-parser@1 | 4 (-1) |
| body-parser@1 | 5 (-1) | | finalhandler@1 | 4 (-0) | | safe-buffer@5 | 4 |
| content-disposition@0 | 4 (-2) | | fresh@0 | 4 (-2) | | send@0 | 4 (-1) |
| content-type@1 | 4 (-1) | | merge-descriptors@1 | 4 (-2) | | serve-static@1 | 4 |
| cookie@0 | 4 (-1) | | methods@1 | 4 (-1) | | setprototypeof@1 | 4 (-1) |
| cookie-signature@1 | 4 (-1) | | on-finished@2 | 4 (-2) | | statuses@1 | 4 (-1) |
| debug@2 | 4 (-2) | | parseurl@1 | 4 (-1) | | type-of@1 | 4 (-2) |
| depd@1 | 4 (-1) | | path-to-regexp@0 | 4 (-2) | | utils-merge@1 | 4 (-1) |
| encodeurl@1 | 4 (-1) | | proxy-addr@2 | 4 (-1) | | vary@1 | 4 (-2) |

Figure 23: Comparison of PkgPkr Scores using cucumber.js as a baseline

Second, we see if our recommendation engine is capable of recommending a package based on that package's dependencies only. Because Package Picker takes a peer-dependency approach, we wouldn't necessarily expect this to work unless it is common for a project's dependencies to also be

peers with it in a parent application that relies on the project. We pick five unrelated projects and see if they recommend themselves and each other, and if so, what the scores are. As shown in Figure 24, our system was only capable of self-predicting express@4, cucumber@6, and mathjs@6. However, in each case the self-prediction score (in bold) was either the sole or highest score, which is what we would expect if context matters.

| | express@4 | cucumber@6 | serverless@1 | johnny-five@1 | mathjs@6 |
|---|---|---|---|---|---|
| express@4 | **8** | - | - | - | - |
| cucumber@6 | 5 | **4** | - | - | - |
| serverless@1 | 5 | - | - | - | - |
| johnny-five@1 | 4 | - | - | - | - |
| mathjs@6 | 4 | - | - | - | **3** |

Figure 24: Ability of recommendation engine to self-predict different packages

Third, we compare our recommendation engine to the authoritative search for npm and PyPI packages, npmjs.com and pypi.org, respectively. We use “express” for npm and “django” for PyPI, as both are popular packages that are also web frameworks, so the results should be similar in terms of domain. As shown in Figure 25, the quality of search results varies greatly. The top three results from pypi.org are django itself and some test packages, while the rest seem like specific django extensions. In contrast, the results from npmjs.com seem to do a better job of picking out packages that are relevant to web development in general while happening to also be compatible with express. In both cases, the results from Package Picker seem similarly focused on the domain of web development for django and express, respectively.

| **npmjs.com** | **Package Picker** | | **pypi.org** | **Package** |
|---|---|---|---|---|

| | | | | **Picker** |
|---|---|---|---|---|
| express | body-parser@1 | | Django 3.0.5 | asgiref@3 |
| path-to-regexp | morgan@1 | | Django-504 2.2.9 | sqlparse@0 |
| cors | cookie-parser@1 | | django-503 0.1 | whitenoise@5 |
| http-proxy-middlewar e | express-session@ 1 | | django-scribbler-django2.0 0.9.3 | gunicorn@20 |
| morgan | cors@2 | | django-filebrowser-django13 3.0 | pytz@2019 |
| is-regex | passport@0 | | django-jchart-django3-uvm 0.4.2 | requests@2 |
| express-handlebars | serve-favicon@2 | | Django-tracking-analyzer-dja ngo2 0.3 | six@1 |

Figure 25: Comparison of top search results and recommendations

As shown above, the results from Package Picker indicate that it is capable of making relevant recommendations for a particular package context. It can fill holes in an existing application's dependencies, recommend a parent package based solely on its own dependencies in many cases, and also deliver results on par with a good search engine. The relative ease with which we were able to add Python support indicates that our approach is extensible to other ecosystems, opening the possibility of a universal recommendation engine that is not language specific.

# Project risks

**Abuse of our service**

We provide a public-facing UI and API that anyone can use. Our production instance uses an AWS-generated SSL certificate to encrypt traffic between the user and our load balancer, so the user's data is not at risk of being read. With the exception of the load balancer, all of our AWS

resources are on private subnets and thus cannot be accessed from the outside. However, we have not put the load balancer itself behind a firewall or API gateway, which means we have no rate-limiting in place to protect against overuse or denial of service attacks.

**Vendor lock-in**

Our system is dependent on GitHub for repository data, npmjs.com for npm metadata, and pypi.org for PyPI metadata. We chose GitHub because it has the largest collection of open source code. We chose npmjs.com and pypi.org because they are the largest package registries for npm and PyPI packages, respectively. If any of these data sources moved to a proprietary model where access was eliminated or restricted to paying customers, our system would be unable to function. We think this risk is small due to the community pressure to keep open source projects accessible to the public. In fact, npmjs.com was recently acquired by GitHub with the promise of staying free forever. If any of these sites did become proprietary, it is almost certain that an open source alternative would become available (e.g. GitLab), in which case we would need to port our system to the new data provider but likely with only minimal changes.

**Unvalidated quality of our output**

After Milestone 2, we discovered that we weren't fetching all of the JavaScript repositories available on GitHub. By switching from a topic search to a language search, we were able to fetch 10 times as many repositories, giving us 350,000 repositories to work with. With this data at our disposal, our follower requirements no longer need to be so lax. We had changed the follower requirement from >100 followers in Milestone 1 to >1 follower in Milestone 2. We can now afford to be more strict about the packages we use, potentially reinstating the original requirement to filter out packages with fewer than 1,000 downloads per month.

With this increased amount of data, the sparsity of our similarity matrix is no longer a concern. Given a baseline repository we've been using to get recommendations (express), we went from ~1,500 recommendations in Milestone 2 to >10,000 recommendations in Milestone 3. In fact we cap the number of recommendations returned to 10,000 for performance reasons, so the true number is much higher than this. We still only provide recommendations based on major package versions, as this still enables users to get recommendations that don't break their existing code. We have also evaluated the effectiveness of our approach by entering individual packages and seeing what recommendations come up. For example, when pandas v0 is provided, our top

recommendations are for scikit-learn, scipy, numpy, matplotlib, and seaborn, which seems very promising as these packages are frequently used together. We did notice some bad packages in our recommendations, as some projects seem to use malformed dependencies (e.g. _better-assert@1.0.2@better-assert@1.0.2), but these could be easily filtered out at the time we fetch repositories from GitHub.

**Rate limiting on dataset collection**

We thought rate limiting would be an issue during data fetch that would prevent us from collecting a large enough dataset. It is no longer an issue, as GitHub allows us to fetch 100 repositories in a single query using their GraphQL API. The entire fetch process takes about 50 hours across two ecosystems (npm and PyPI), so we may want to parallelize this in the future, but for now it is not a limitation.

**Adversarial attacks**

We thought people might game our recommendation engine. We haven't observed this yet. Because our recommendations are based on usage patterns in public projects, people may game the system to skew our results. For example, a person could create a number of repositories with a meaningless collection of dependencies, then create a bunch of dummy accounts which they could use to star the repositories and get them above our inclusion threshold (currently >1 star). To mitigate this, we could increase the follower count requirement on repositories we fetch from GitHub, but it's not a robust solution and reduces the amount of data we can capture. We should consider more robust spam detection measures in addition to just changing our filter criteria. In fact, we have noticed bad recommendations in our results from repositories that appear to have malformed dependency names (e.g. _better-assert@1.0.2@better-assert@1.0.2). These malformed dependencies could be easily filtered out by using a regex that only recognizes valid package names.

**Assumptions around how GitHub repositories are structured**

Our ML pipeline assumes that every repository on GitHub has its package.json or requirements.txt (if they exist) in the root folder. While we have managed to fetch enough data in spite of this assumption, it should be noted that our own project doesn't have its requirements.txt files in its root directory, so our own ML pipeline wouldn't be capable of seeing what dependencies we're using. There is also a common pattern, known as the monorepo, in which a single repository

contains multiple applications, each with their own package.json or requirements.txt file in separate subfolders. Our ML pipeline is not capable of dealing with this pattern either.

**Legal liability**

We thought there might be legal liability for us if we recommended packages that turned out to be malicious. Alternatively, if people use recommendations with copyleft licensing (e.g. AGPL) in their production code, they could be compelled to either open source their own code or become open to a lawsuit. These aren't a factor yet because we haven't started to promote our tool. That said, we could mitigate this by putting a legal disclaimer on the site and including vulnerability and licensing information for each recommendation. It would be the user's responsibility to verify that they will not be impacted by any vulnerabilities or licensing issues in the recommendations they use.

**Skewing the Open Source ecosystem**

We thought our tool had the potential to have a bad influence on the open source ecosystem. For example, it is conceivable we could bury packages that have potential but aren't used with enough other packages to be picked up by our recommendation engine. Our project isn't yet used widely enough for this to be a risk we need to address, but one way to mitigate it would be to create a feedback system in which people who feel like their project is being unfairly overlooked can provide input to the system so it does a better job of incorporating these underutilized packages.

# Development Process & Lessons Learned Reflection

Refer to the Appendix for the requirements table.

We measured our work in *person-days* which represents about 2 hours of development work. This measure was decided upon by the team as a way to capture the amount of work a person can complete in a day, accounting for full-time jobs, family obligations, other extension school class commitments, etc. On weekends there might be 10+ hours of work, on Thursday lecture nights there might be 0 hours of work, but in the long run 2 hours per day is the balance.

At the start of our estimates we had 38 requirements that would take an estimated 244 person-days to complete. These 244 person-days divided up by 6 team members left around 41 days or about two months total to complete the project (which was the length of the remaining semester). These

41 days were the base amount of time to work, but we also factored in additional buffer time for things like bugs that were taking too long to solve, and other unexpected factors.

As shown in Figure 26 in the Appendix, we completed 35 of the 38 requirements, and decided to save the 3 remaining requirements for later work. In particular, we chose not to create Swagger API docs, write tests to validate the recommendation engine as we continue to make changes, and haven't yet presented at a conference. However, we have submitted our abstract to the International Conference on Software Reuse (ICSR) and will know by August 17 if our paper was accepted for their conference on November 9-11, 2020.

For the 35 completed requirements, their estimated time to complete was 218 person-days, and we finished them with an actual time of 171 person-days. The 27% overestimate makes sense because our estimates included a generous buffer for our work. We used this extra time to add more features and tools not originally captured in the requirements. Some of these tools included Docker, AWS Elastic Container Service (ECS), AWS Fargate, AWS Step Functions, and GitHub Actions. The additional features included a "select a branch" feature, additional metrics, PyPI support, a public API, and additional UI improvements.

We completed the requirements in this list by working in agile one-week iterations. We had bi-weekly meetings where we met as a team and discussed our progress on tickets, gave demos, practiced for presentations, and figured out what to do next through agile ceremonies.

There were several good key decisions we made in our process. These included:

- *Solid communication* - Using Slack, Jira, Confluence, and Zoom that made us productively organized and interconnected with timely response times.
- *Frequent useful, productive meetings and updates* - We met as a team at minimum every Tuesday and Saturday with additional meetings as needed. These made it easy to check-in and see what's going on. As a team we felt them appropriate and not excessive. We would go over on time if there were a lot of things to discuss, and ended early if next steps were obvious and clear.
- *Pair programming* - For certain parts of the project, two or more developers worked together over Zoom to develop code together. This made for faster development than a developer working alone and also allowed for simultaneous design discussion.

- *Automation* - we've automated most aspects of CI/CD to make it easy to hand-off the project, including the provisioning of AWS resources with Terraform.
- *Dividing of tasks and defining interest areas* - Early in the semester everyone noted what their interests were e.g. whether they want to work more on frontend or backend, cloud, recommendation engine, testing, journal paper. By establishing these team norms, it was easy to have each member work on the part that was the most interesting to them and thus produce better work.
- *Build the risky parts first* - Using the agile approach, having a minimum viable product ready by Milestone 2 made with proper CI/CD and testing gave us a base project from which we could confidently get feedback on early and spend more time polishing and refining the final result.
- *Good technical decisions* - Because we followed an agile methodology, it was easy to make adjustments as we went along
    1. Using containers and AWS ECS for CI/CD (highly automated, identical environment for both development and production).
    2. Using PostgreSQL to store the output from our recommendation engine (fast queries, no need to fetch new results from S3 periodically).
    3. Using GitHub for authentication (easy to set up, integrates well with our user flow).
    4. Using AWS so it allowed for the proper environment to build the cosine similarity matrix.

There were also some not-so-good decisions. These included:

- *Too many moving parts* - Using S3 as a staging area for the generated cosine similarity matrix was overly complicated.
- *Over-engineering the ML pipeline* - Using PySpark was overkill for the amount of data we were processing. Python pandas or numpy would have sufficed.
- *Not parallelizing the ML pipeline* - A full run of the ML pipeline takes ~50 hours. We should have used AWS Lambda to make it easy to scale out horizontally.
- *Tight coupling* - We originally planned to decouple the ML pipeline and recommendation engine into separate instances, but ended up running them sequentially on the same

instance using the same trigger. However, in all other ways they are still separate processes, so we could split them out into multiple instances later if we wanted to.

We made the following changes in how we approached our work:

- *Regular note-taking* - Started taking more notes during meetings to help better understand the other areas teammates were working on.
- *Regular async communication* - Unlike other general school projects, we kept communicating with teammates and getting feedback and approvals to make progress - this way allowed for better collaboration and a better result.
- *Pair programming* - We often found it faster to team up than to individually churn on the same task for a significant part of the sprint.
- *Bi-weekly planning meetings* - We found that holding regular planning meetings from one to two hours were necessary to accommodate our knowledge sharing needs and allow for demos/QU of work that was done since the prior meeting.
- *Agile with one week sprints* - we started out with two-week long sprints, but because of the two-month timeframe of development, this would have been too long to be useful, so we switched to one-week sprints.
- *Thorough testing* - in the early phases, we didn't write as many tests, but as we progressed we found that testing kept changes from being brittle, giving us increased confidence as we kept adding features.

Proactively identifying problems and negotiating them with the customer:

So far we have discussed problems with teammates and fixed/changed those issues. We are still working on getting feedback back from our customer. We anticipated issues early with GitHub API limitations and prioritized the work accordingly. We anticipated issues supporting multiple package formats so negotiated to start with npm, and only added support for PyPI near the end of the project.

Risks (unavoidable and intentional) for the software engineering aspects (not the product):

Aside from the added stress, COVID-19 surprisingly gave us more time to work on our project because people's commutes and weekend activities were eliminated. We stayed positive and made sure to take care of ourselves and talk about things in our meetings and in Slack.

# Appendix

## Development tasks and estimation

Figure 26 - grey means *completed*, yellow means *to do*

Estimates were in *person-days* i.e. 2-3 hours of development work.

| **Epic** | **Tasks** | **Requirement addressed** | **Estimate person -days** | **Actual** |
|---|---|---|---|---|
| Cross-component activities/dependencies | Document the database schemas for the ML pipeline | 0 | 4 | **2** |
| Cross-component activities/dependencies | Document the high-level architecture diagrams | 0 | 2 | **4** |
| Cross-component activities/dependencies | Write MS1 paper | - | 8 | **6** |
| Cross-component activities/dependencies | Create MS1 presentation | - | 4 | **4** |
| Cross-component activities/dependencies | Journal Research MS1 | - | 8 | **8** |
| Cross-component activities/dependencies | Write MS2 paper | - | 8 | **8** |
| Cross-component activities/dependencies | Create MS2 presentation | - | 4 | **3** |
| Cross-component activities/dependencies | Journal Research MS2 | - | 8 | **1** |

| | | | | |
|---|---|---|---|---|
| Cross-component activities/dependencies | Write MS3 paper | - | 8 | 8 |
| Cross-component activities/dependencies | Create MS3 presentation | - | 4 | 4 |
| Cross-component activities/dependencies | Journal Research MS3 | - | 8 | 4 |
| Cross-component activities/dependencies | Create docs using Sphinx, hosted on GitHub pages | 0 | 4 | 5 |
| Cross-component activities/dependencies | Enrich and finalize Sphinx documentation | 0-12 | 8 | 2 |
| Cross-component activities/dependencies | Publish (or present at the conference) the work | - | 10 | To do |
| Cross-component activities/dependencies | Set up team tools (Slack, etc) and invite team members | 0 | 2 | **1** |
| GitHub/AWS Infrastructure | Finalize deployment flow with GitHub Actions | 0 | 6 | **6** |
| GitHub/AWS Infrastructure | Document provisioning flow for recommendation engine | 0 | 4 | **1** |
| GitHub/AWS Infrastructure | Create infrastructure provisioning scripts for recommendation engine | 0 | 8 | 6 |
| GitHub/AWS Infrastructure | Document provisioning flow for the ML pipeline | 0 | 4 | **4** |
| GitHub/AWS Infrastructure | Create infrastructure for the ML pipeline | 0 | 8 | **6** |

| | | | | |
|---|---|---|---|---|
| GitHub/AWS Infrastructure | Document provisioning flow for web service | 0 | 4 | **1** |
| GitHub/AWS Infrastructure | Create infrastructure for web service | 0 | 8 | **12** |
| ML pipeline | Research and document GitHub API queries to support the ML pipeline | 1-4 | 4 | **4** |
| ML pipeline | Design and document ML pipeline process flow | 1-4 | 2 | **2** |
| ML pipeline | Design DB structure to persist the data | 1-4 | 2 | **1** |
| ML pipeline | Build a script to fetch results on demand (takes 50+ hours, so not doing daily for now) | 1-4 | 8 | **11** |
| Recommendation engine | Design and document recommendation engine re-train process flow | 5, 6 | 8 | **4** |
| Recommendation engine | Document the score inputs, features, and approach | 5, 6 | 4 | **2** |
| Recommendation engine | Create basic recommender system in Jupyter notebook | 5, 6 | 8 | **6** |
| Recommendation engine | Create a script to rebuild the similarity matrix on a cadence | 5, 6 | 6 | **2** |
| Recommendation engine | Keep Refining score to produce final results | 6 | 20+ | 10 |
| Recommendation engine | Create tests to monitor score | 6 | 8 | To do |

| | | | | |
|---|---|---|---|---|
| | performance | | | |
| WebServer | Document API specifications and backend | 7-12 | 4 | **2** |
| WebServer | Create a Django server | 7-12 | 10 | **10** |
| WebServer | Create UI pages as per wireframes with JS | 7-12 | 10 | **9** |
| WebServer | Create unit tests | 1-12 | 6 | **8** |
| WebServer | Create UI tests | 7-12 | 4 | **4** |
| WebServer | Add Swagger API | 5, 7-12 | 8 | To do |

## System installation manual

You can either run the system locally or provision AWS with the necessary services. The following instructions show how to use Terraform to provision the system on AWS.

1. First, fork our repository at https://github.com/pkgpkr/Package-Picker.

2. Next, clone the repository you just forked and check out the master branch.

```
git clone https://github.com/<you>/Package-Picker.git
cd Package-Picker
git checkout master
```

3. Install Terraform and initialize it within the `terraform/` folder.

```
cd terraform
terraform init
```

4. Spin up the requisite AWS resources with Terraform.

   ```
   terraform apply
   ```

   Don't provide a value for `DOMAIN_NAME`, just hit 'enter'

   `DB_PASSWORD` is the desired password for your database

   `DB_USER` is the desired username for your database

5. You will also need to create a new GitHub app so you can authenticate.

   a. Open your GitHub settings
   b. Create a new OAuth App
   c. Set the callback URL to `<load balancer DNS>/callback`
   d. Note the client ID and secret. You will need these in the next step.

6. Next, set the following secrets on your forked repository.

   ```
   AWS_ACCESS_KEY_ID     # Your AWS account's access key
   AWS_SECRET_ACCESS_KEY # Your AWS account's secret access key
   CLIENT_ID             # ID of your GitHub app
   CLIENT_SECRET         # Secret for the GitHub app
   DB_HOST               # Database URL
   DB_PORT               # Database port
   DB_DATABASE           # Database name
   DB_PASSWORD           # Database password (must not be empty)
   DB_USER               # Database user
   DOMAIN_NAME           # e.g. https://pkgpkr.com
   MONTH                 # How many months of data to scrape
   SELENIUM_TEST         # Set to 1 if running Selenium tests
   GH_TOKEN              # GitHub API token
   ```

7. Instantiate your AWS RDS instance with some data to get you started.

   ```
   wget https://pkgpkr-models.s3.amazonaws.com/database.dump.gz
   ```

```
cat big_data.dump.gz | gunzip | psql -h <DB host> -U <DB user>
<DB name>
```

8. Finally, make a test commit to both the `pipeline/` and `webserver/pkgpkr/` folders. This will trigger GitHub Actions to deploy the ML pipeline and web server containers to AWS, respectively.

9. Once the deploy has completed, open your browser to the load balancer DNS to view the app.

## Developer installation manual

We've included instructions on our documentation site for getting started with local development: https://pkgpkr.github.io/Package-Picker**.**

## Demo

https://www.youtube.com/watch?v=C3gwLmHGsh4&feature=youtu.be

## API as a Service

To allow programmatic access to the recommendations service, the public API endpoint has been added to allow passing a number of packages via RESTful API call and getting the recommendations for languages supported, while specifying the language and the max number of recommendations to return.

**API Specifications**

**Method:** POST

**Endpoint:** /api/recommendations

**Parameter content type:** application/json

**Parameters:**

- body:

```
{"language": string("Javascript"|"Python"),
```

```
"dependencies" : list
"max_recommendations": integer }
```

**Response:**

- 200 - Successful:

```
{
  "language": "string",
  "recommended_dependencies": [
      {
          "forPackage": "string",
          "recommendedPackage": "string",
          "url":"string",
          "pkgpkrScore": integer,
          "absoluteTrendScore": integer,
          "relativeTrendScore": integer,
          "boundedPopularityScore": integer,
          "boundedSimilarityScore": integer,
          "categories": [
              "react"
          ],
          "displayDate": "string",
          "monthlyDownloadsLastMonth": integer
      },
  ]
}
```

- 400 - Bad Request (Not parsable JSON, incorrect language, malformed dependencies, etc.)
- 405 - Method not allowed (e.g. GET)

**Example for Javascript:**

Payload:

```
{"language": "javascript",
 "dependencies" : ["lodash@4.17.15","react@16.13.1",
              "express@4.17.1", "moment@2.24.0"],
```

```
"max_recommendations":1}
```


Response:


```
{
    "language": "javascript",
    "recommended_dependencies": [
        {
            "forPackage": "react@16",
            "recommendedPackage": "react-dom@16",
            "url": "https://npmjs.com/package/react-dom",
            "pkgpkrScore": 9,
            "absoluteTrendScore": 2,
            "relativeTrendScore": 2,
            "boundedPopularityScore": 9,
            "boundedSimilarityScore": 9,
            "categories": [
                "react"
            ],
            "displayDate": "2020-04-09",
            "monthlyDownloadsLastMonth": 33497683
        }
    ]
}
```

# Screenshots

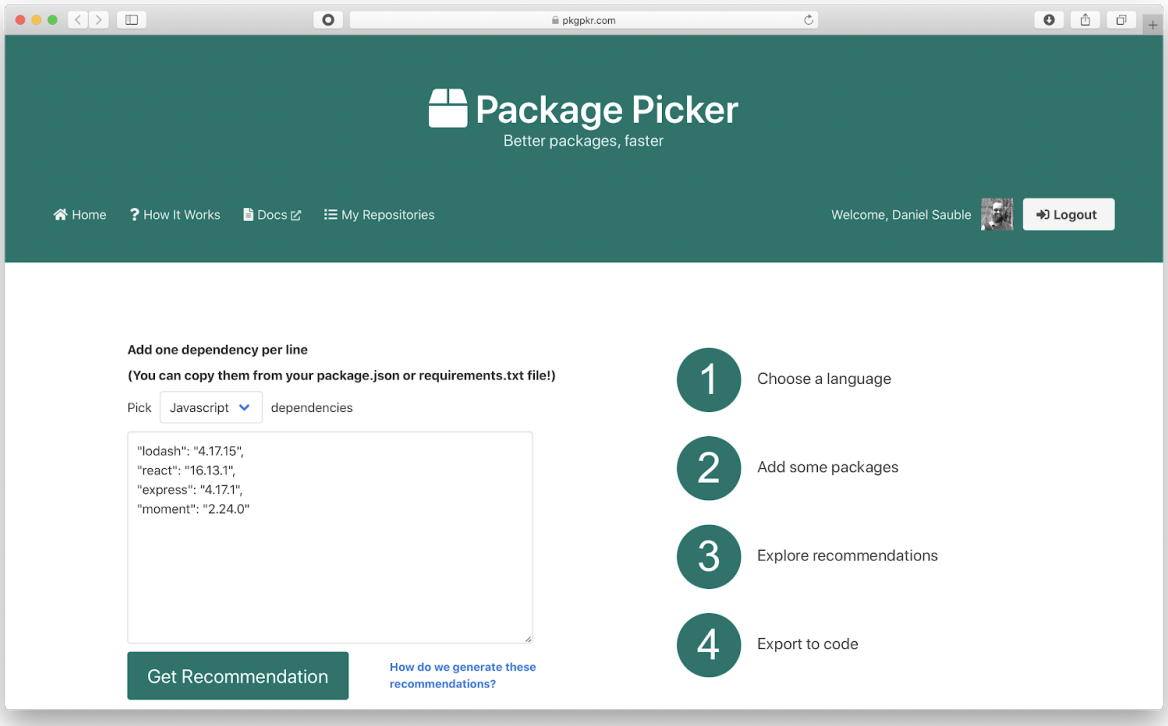


Figure 27: Home page of Package Picker

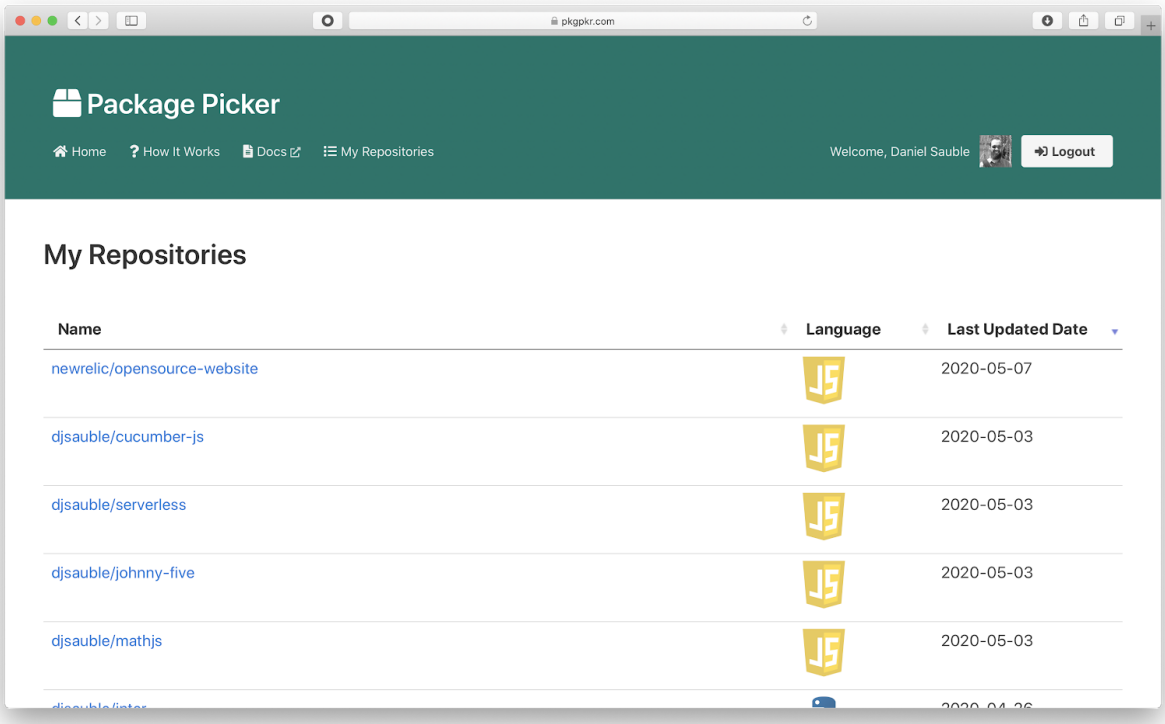


Figure 28: List of a user's repositories

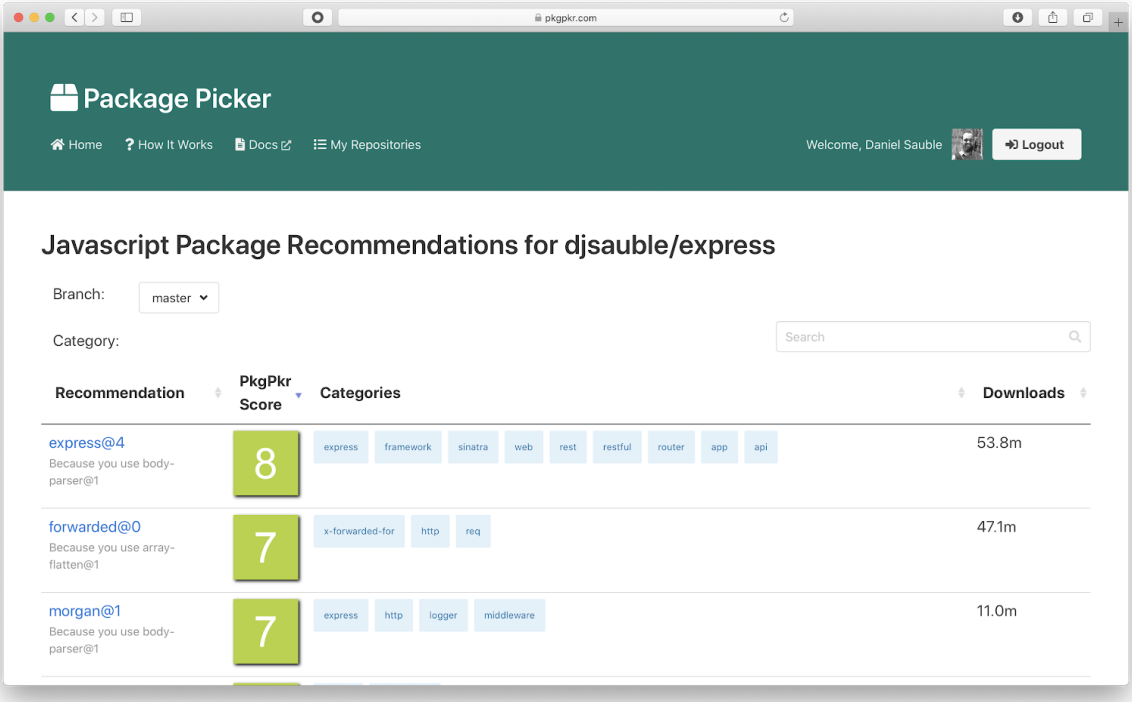

Figure 29: List of recommendations for express

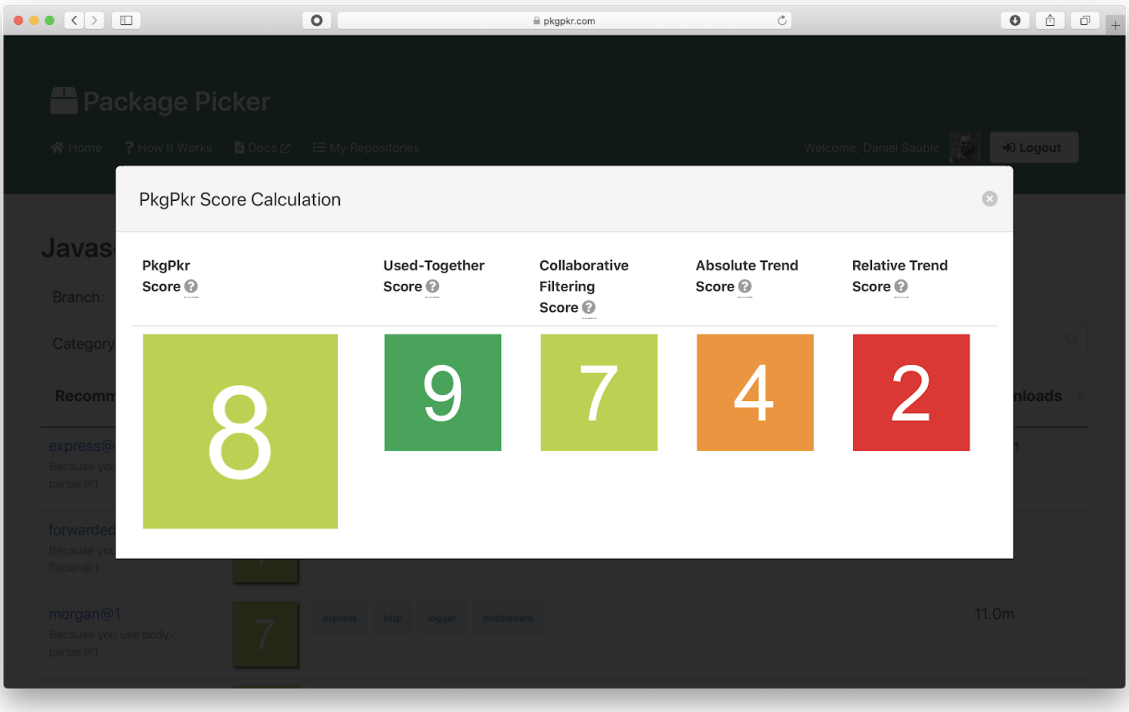


Figure 30: Component scores behind a PkgPkr Score