Distributed REsearch Apprenticeships for Master's (DREAM) Program Final Paper

Introduction

One definition of insurance is "[a] thing providing protection against a possible eventuality".[1] With more detail, insurance becomes "[a] practice or arrangement by which a company or government agency provides a guarantee of compensation for specified loss, damage, illness, or death in return for payment of a premium."[2] The first definition captures the idea that insurance wants to protect against something in the future. The second definition captures the idea that this "eventuality" is something bad (i.e., a "specified loss"). It also captures that such protection comes at a price (i.e., "a premium"). If there were infinite resources then, you could protect against all bad "eventualities" compensating for all "specified losses". As there are not infinite resources, maximizing the good of insurance means making best use of the resources available to protect people. One way to maximize the value of insurance is to make sure payments are only made for true "losses". In low information environments, it can be hard to tell whether or not a loss as envisaged by the provider and purchaser of insurance has actually occurred. Our summer work on Decision Engine for Socioeconomic Disaster Risk (DESDR), an open source insurance toolkit, seeks to improve information available to decision makers and those insured in the context of farm index insurance.

In the next sections of this paper, we first cover what farm index insurance is in greater detail and the information problems it faces. Next, we outline one approach to solving this problem (i.e., an open source insurance toolkit). We then proceed to discuss possible existing solutions that meet DESDR's purpose. Finally, we explain the advantages of pursuing the open source insurance toolkit and our work towards DESDR's completion.

DESDR is co-developed between the WuLab[3] and the Financial Instruments Sector Team (FIST[4]) of the International Research Institute for Climate and Society at the Columbia Climate School at Columbia University.[5] We worked as part of the WuLab under the direction of Professor Eugene Wu to translate the technical requirements of the FIST team into a workable toolkit.[6]

[1] https://www.lexico.com/en/definition/insurance

[2] Ibid

[3] https://cudbg.github.io/lab/

[4] https://iri.columbia.edu/our-expertise/financial-instruments/

[5] We note that proofs of concept for the open source insurance toolkit were completed prior to the start of our work this summer. Additional work was also completed on initial how-to guides for the toolkit as well as extensive work to translate prior R code from the FIST team into DBT code.

[6] Overall there are around four to five contributors on a regular basis to the development of the toolkit from the WuLab. We focus this paper on our contributions as part of the DREAM program.

Background

Disasters cause billions in crop losses each decade, and climate change may disrupt how and where crops are produced.[7] Whether or not such disruption occurs, the possibility that it might occur makes it harder to plan for investments and build savings. This in turn means farmers are less likely to be able to weather extreme disruptions and more likely to be forced into selling off assets. Then as the chance of disruption to crops (i.e., risk) increases, the importance of a hedge against such risk also increases. Insurance could act as one mechanism to counteract this risk. However, to buy insurance you have to have the option available to you; this has not been true everywhere historically.[8] For example, smallholder farmers in Africa have not had much access in the past to insurance products despite suffering from climate related events. Index insurance has arisen as one way to increase insurance availability to people without access to more traditional loss-based indemnity insurance.[9]

Conventional indemnity insurance is based on direct claims assessments, but this prohibitively expensive for developing countries. It is also hard to do at scale. To get around these issues, index insurance uses some proxy for loss that is cheaper to implement. In agricultural insurance, one common proxy is satellite measurements of weather and vegetation conditions (i.e., satellite climate data). Evidence also suggests that index insurance is effective when it is available and utilized. Research on the impact of index insurance in East Africa found that households with insurance were “45-50% less dependent on food aid” in one study, and insured farmers “on average, increased their savings and the number of oxen they owned relative to uninsured farmers” in the case of one index insurance program studied.[10]

The problem we observe is that although index insurance can increase protection to previously uncovered farmers, its impact is limited due to the imperfect information embedded in the index. The index is an abstraction of facts on the ground, and collecting the satellite rainfall and vegetation data typically used in agricultural insurance is difficult. This can lead to misalignment between suggested payouts from the index and the actual losses of farmers (i.e., basis risk). To increase subscription to index insurance among the most vulnerable, two things need to happen. First, the ratio between premiums paid to actual losses covered needs to be kept as low as possible, and second, people need to trust how index insurance works.

This raises two related questions. How do you increase information available to calculate the index, and how do you enable participation in assessment of the index by different stakeholders (e.g., farmers, policy makers, insurers, and economists)? Our team’s project proposes to answer these questions via an open source insurance toolkit called DESDR. DESDR’s goal is not only to make gathering more data possible but also to help transform that data. In the long run, DESDR will include tools to address three requirements: (1) to collect data

[7] https://climate.nasa.gov/news/3124/global-climate-change-impact-on-crops-expected-within-10-years-nasa-study-finds/ and https://www.fao.org/resources/digital-reports/disasters-in-agriculture/en/

[8] One could also argue that even if insurance is available, it does not make sense to purchase. Instead a farmer could save the money spent on insurance themselves in a rainy day fund. However, insurance companies can build large risk diversified portfolios in a way a farmer cannot. See the following source for more details: https://link.springer.com/content/pdf/10.1007/978-94-007-4839-2.pdf (page 4)

[9] https://link.springer.com/content/pdf/10.1007/978-94-007-4839-2.pdf (page 2)

[10] https://www.mdpi.com/2072-4292/10/12/1887 (page 4)

about farmers' experiences, (2) to clean, manage, and post-process data, and (3) to choose and assess insurance policies.

To do this, we built a technology stack based on DBT, Flask, D3.js and Svelte. DBT is built in Python, and it turns SQL models (i.e., select statements) into a dependency graph. This allows you to break up larger queries into a modular form.[11] Besides select statements in DBT models, macros are available in DBT, and they allow for recycling of often repeated code. Macros are more general than models. They permit a user to recycle any amount of code by placing the code in a macro file. A macro file may contain one or many named macro templates. Whenever the macro name is invoked within a DBT model, the name will be replaced with the section of applicable code in a macro file. Macros can also take arguments, which means you can use them to hold functions.[12]

A user controls DBT through a command line interface. For example, executing *dbt run* will lead DBT to look through the dependency graph it built executing models in order. You can also control certain configurations and use variables with DBT via a *dbt_project.yml* file. DBT loads the previously mentioned *dbt_project.yml* into Python dictionaries and then uses Jinja to render the final SQL queries to be run. DBT constructs a directed acyclic graph (DAG) before then executing a form of modified topological sort to run models in the correct order.[13] As per the comments in the source code, DBT adapts the networkx package topological sort such that "ties are grouped together in lists".[14] This enables some degree of parallelism in executing models at a similar depth in the DAG which in conjunction with DBT's implementation of a priority queue helps to improve the performance of *dbt run.*

In the current DESDR architecture, *dbt run* is executed using calls from a Python server which connects Docker containers running DBT to a front-end Svelte application. The server uses Flask routes and the REST API to receive a parameter dictionary passed in from the Svelte application. This parameter dictionary includes things like the administrative region of interest and the time periods of interest. This information is logged from users in the Svelte application via input boxes, dropdown menus and sliders. The parameter dictionary can be added to the *dbt run* call in the Python server to override the default values in the *dbt_project.yml* file thus updating the outputs of model execution.

Svelte and D3.js are the two tools used for the front-end design. Svelte is a front-end JavaScript framework similar to Vue and React. It combines HTML, CSS, and JavaScript files into one single .svelte file and is able to pass JavaScript code and logic into the HTML section, which is more effective than pure JavaScript. For data visualization and displaying graphs (i.e., charts) in DESDR's dashboards, we decided to use D3.js with Svelte. D3.js is "a JavaScript library for manipulating documents based on data. D3 helps you bring data to life using HTML, SVG, and CSS."[15] Basically it helps us to convert the data from a database to plots or graphs we

[11] https://docs.getdbt.com/reference/commands/run and https://docs.getdbt.com/docs/building-a-dbt-project/building-models
[12] https://docs.getdbt.com/docs/building-a-dbt-project/jinja-macros
[13] https://github.com/dbt-labs/dbt-core and https://www.getdbt.com/blog/how-we-made-dbt-runs-30-faster/
[14] https://github.com/dbt-labs/dbt-core/blob/main/core/dbt/graph/queue.py
[15] https://d3js.org/

want to render in a web browser. We decided to use pure D3.js instead of some other libraries built on top of D3.js, for example, Vega-Lite, because D3.js allows more flexibility.

Related Work

The primary alternative to DESDR is Open Data Kit (ODK), which satisfies some, but not all of DESDR's requirements. ODK is an open source toolkit to build forms for data collection. It also includes features to download your data and export it to other platforms like Excel or R for live updating dashboards. ODK satisfies requirements (1) and (2), but its dependency on other platforms such as R for data visualization have proven to be unreliable in testing. The FIST team for example does use ODK for collecting information from farmers on historical bad years and planting times. Without DESDR, the FIST team has had to rely on Shiny, an R package, for building interactive data dashboards from the data collected. However, Shiny in testing has had stability issues and lacks a degree of UI flexibility that the FIST team would like to achieve. Then ODK fully satisfies requirement (1) listed as a specification for DESDR. Theoretically, ODK when taken together with its ability to export to R satisfies requirement (2). However, as noted above due to issues with Shiny, ODK cannot be viewed as a viable solution for requirement (3) .

Furthermore, implicit in requirement (3) is the belief that the ability to assess different insurance policies effectively will lead to improved data collection (i.e., requirement (1)) and better data manipulation (i.e., requirement (2)). In other words, there is a feedback loop between assessing the outcomes of data collection and manipulation and the types of data collected and transformations done on such data. Given that current solutions do not satisfy requirement (3) and because a good implementation of requirement (3) plays into if requirements (1) and (2) are met, we believe it is necessary to develop a new open source toolkit to power DESDR. Our work as outlined below satisfies requirements (2) and (3), and in the future it will be extended to requirement (1).

Technical Details

Requirement (3) needs data to be loaded, then transformed according to some default parameters, which is handled by requirement (2), and then the transformed data needs to be displayed on a data dashboard. The dashboard also needs to be interactive; that is the dashboard needs ways for a user to adjust the parameters that go into the data transformations. These transformations should be re-run before then re-rendering the dashboard. Examples of typical data transformations include averaging rainfall over a time period for a given geographic region or computing expected payouts of insurance using windowed satellite climate data. Also the dashboard needs to be hosted somewhere easily accessible to the intended audience (e.g., economists, policy makers, farmers, and insurers). Finally, the dashboard should be reliable and customizable, so that the FIST team could deploy a new instance of the dashboard within a few weeks if needed. Customizations means the FIST team could add or take out several types of standard charts like grouped bars or lines along with corresponding parameter controls via things like sliders or text boxes. Relatedly, they should also be able to control the raw data and the transformed data that goes into each chart.

To meet these specifications, we developed DESDR according to the below in Figure 1. At the highest level, DESDR is split between the server-side and client-side. A dashed line shows this split. Anything on the server-side deals with inputting raw data into a database and then using SQL statements to transform raw data into a form that feeds a dashboard. The client-side represents everything related to this dashboard, including the users who interact with the dashboard. In short, the client side represents the end users of DESDR and how they interact with data. The server-side represents data loading and transformations carried out for users on the client-side.

Dashed boxes represent a logical grouping. A logical grouping may be either a set of similar things (users) or it may relate to an atomic program, and these groupings may be nested. An atomic program refers to a process running within a wider system (i.e., DESDR) which produces an output useful on its own.

For example, the Ubuntu Docker container and Postgres Docker container each produce their own useful outputs, but together these are part of the Docker logical grouping which outputs transformed data tables to the client-side of DESDR. On the other hand, DBT on its own cannot produce a useful output without substantial installation and setup. For that reason, we don't consider it an atomic program but rather part of the Ubuntu container logical grouping.

Solid blocks represent types of data or processes that when combined form a single logical grouping. Going back to the Ubuntu container, there are model and macro files which run on DBT to create the raw and transformed data tables and views that are visualized in the client-side of DESDR. A single solid block means there is only one instance of a process or type of data. A stack of solid blocks means there could be one or more than one instance of similar processes or types of data.[16]

Arrows show the flow of information between logical groupings. Flask with the REST API routes all information (i.e., arrows) between the server-side and the client-side. Information flow between logical groupings in the server side happens using existing Python packages to connect to Docker containers and Postgres databases. Information flow between the user and the dashboard happens via text inputs, list selections, and the users visual assessment of the output charts on the dashboard.

For example, the Svelte dashboard passes user tweaks to the arguments for DBT taken via sliders, text inputs, and list selections back to the Ubuntu container via the Python server. In turn, the Python server fetches updated data and passes it back to the Svelte dashboard. The dashboard then rerenders its applicable charts, which users can then use to choose and assess insurance policies.

[16] The block for DBT includes a reference to DBT client, which is part of our future work to make DBT models exportable to a JavaScript library which can directly run data transformations in a web browser. DBT client will also enable the removal of the Python server as all processes will be run on an in browser database.

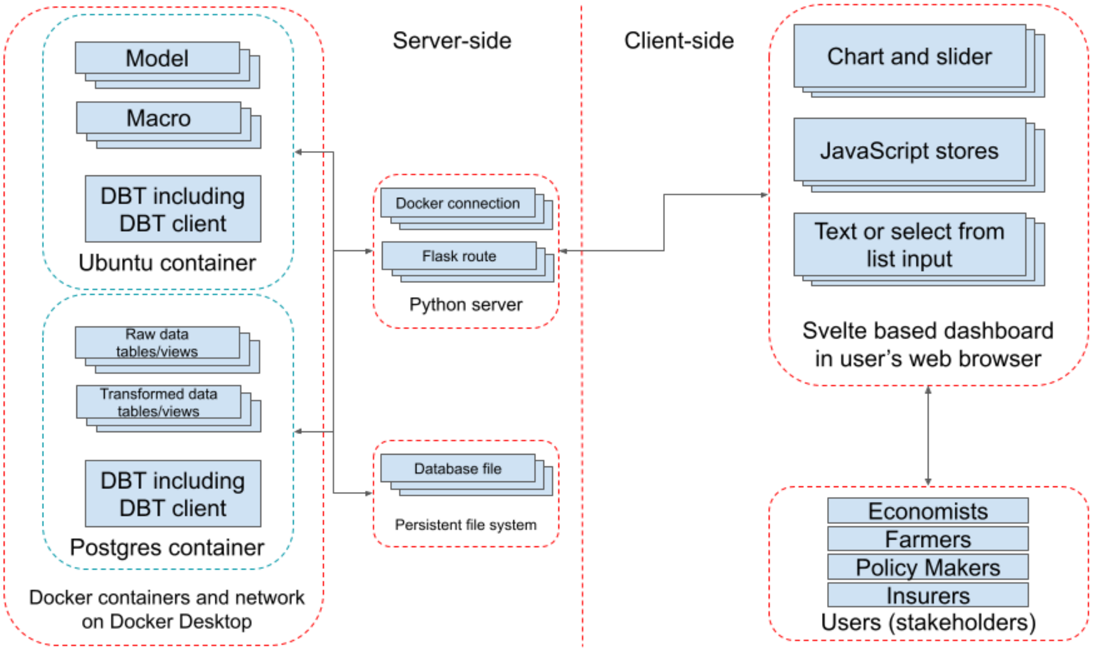


Figure 1

Figure 1 captures the modular nature of DESDR. This in turn allows non-computer scientists to make their own DESDR deployments. Consider for example an economist who wants to build a new data dashboard and the corresponding data transformations. First, they say they would like a grouped bar chart that shows different types of payouts based on rainfall data. Second, they would like to make a line chart displaying average rainfall by dekad (i.e., a ten day period). They also want sliders for each chart to control the timeframe, and they want highlighting on the chart to correspond to the time frame selected. Without DESDR, they would need to find a way to render each chart, but this could prove challenging with lower code graphics frameworks. With DESDR, they can recycle the chart components we have developed according to their needs. The main thing they need to do is specify the data set to be passed to the chart. The data set itself can also be designed in a modular way. Rather than writing a single long SQL query on the server-side, which can be error prone, the economist can reuse prior DBT models and macros. Each model and macro takes care of only one step in the data transformation process, which reduces visual complexity. This also makes it easier to isolate where the economist needs to tweak code if they want to change how the data transformation works.

Another advantage of modular design is that changes to a DESDR deployment after launch are easier. Consider the economist with their dashboard in hand who goes to meet with local policy makers and farmers. They ask for a new type of graph, perhaps one that shows a matching of when farmers thought years were bad for agricultural production versus when rainfall data says it was a bad year. If there is already the corresponding chart component, DBT models, and DBT macros, the economist can drop in the chart without impacting any of the other charts already in the dashboard. If the request is novel, the economist can on their own or with computer scientists work to develop a new chart component and corresponding data transformations. This also means over time the DESDR toolkit can grow to cover more use cases.

Server-side Data Pipeline

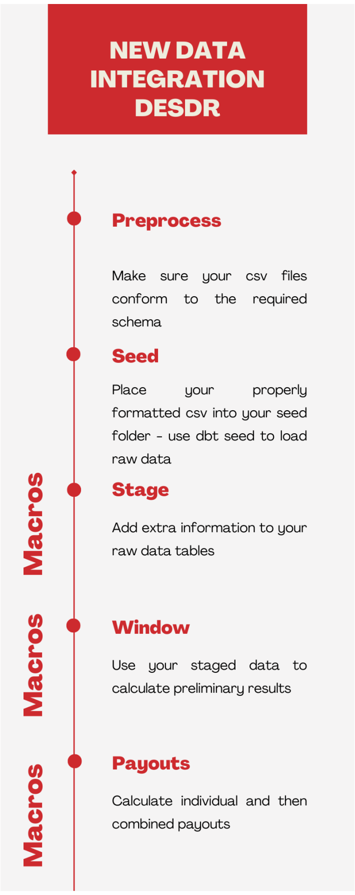
The server-side exists to intake raw CSV data and transform it into table data that can be visualized in charts. Figure 2 shows each step in the data pipeline from pre-processing CSVs to calculating payouts for farmers based on satellite climate data. In the first step, raw satellite data by some time period and geographic identification (i.e., gid) are preprocessed to ensure they follow the schema for raw data tables stored in DESDR's database implementation. Next, *dbt seed* loads the satellite data along with other geographic administrative division data (e.g., village name and gid, region name and gid, etc.) into a database.[17]

The raw data we are working with includes both satellite data in a dekadal (i.e., ten day periods) and monthly format along with information on administrative regions. The specific datasets may grow more expansive in the future, and they depend on the individual implementations of the DESDR toolkit by country. We decided to enforce certain schema on the satellite data although not the administrative data we are uploading because this increases the recyclability of DBT models. Without enforcing such schema, we would start to lose the modularity we argued is useful in Technical Details. No matter what form the satellite data starts in, it must enter into the DESDR implementation with the following column headers of year, gid, value, and dekad. Users are expected to translate their data to conform with this schema, and we provide example scripts for dekadl CHIRPS rainfall and monthly EVI vegetation datasets. Once in this form, *dbt seed* loads data with the following data types based on our experience: year is of data type integer, gid is of data type integer, value is of data type double precision, and dekad is of data type integer.

After loading raw data into a database, the stage, window, and payouts steps in the data pipeline transform data using DBT models and macros. The stage step combines raw satellite data with the raw geographic administrative data to create more informative tables. The window step then uses a macro to compute rainfall statistics such as the frequency of total rainfall in a given dekad range across years of interest. Multiple macros are maintained for calculating different types of statistics over different time periods. This allows DESDR users to see how different statistics compare and how time period selection impacts a given statistic. After windowing, a similar process is used to compute payouts based on trigger values (usually the percentile of rainfall level). Payouts are also combined using a macro to create a combined payout. Combined payouts are some form of weighted average. All payouts are computed

[17] *dbt seed* is not recommended for loading raw data sets but rather things like abbreviations and definitions. We use it due to its ease of use. We only need to load data periodically, and future work should lead to not needing *dbt seed*. For more information on *dbt seed* see: https://docs.getdbt.com/docs/building-a-dbt-project/seeds and https://docs.getdbt.com/reference/commands/seed

between zero and one, and if used, they are applied as a scalar to whatever is the maximum monetary amount a farmer could receive for insurance.[18]



Make sure your csv files conform to the required schema

Place your properly formatted csv into your seed folder - use dbt seed to load raw data

Add extra information to your raw data tables

Use your staged data to calculate preliminary results

Calculate individual and then combined payouts

Figure 2

The modular form of DBT allows the folder and file structure within a given DESDR deployment to mirror Figure 2. As was mentioned earlier, the FIST team has employed Shiny for their data dashboards. They also have done the underlying data analysis in R for each deployment of their data dashboards. Scripts for analysis mixed different elements of the data pipeline and without the clear references between analyses (i.e., model references) present in DBT. This makes the data pipeline harder to understand and more difficult to isolate reusable versus custom code. In contrast, DBT is modular by design making it easy to create a common data pipeline that can be quickly modified for custom deployments.

[18] Insurers decide the maximum monetary payout however they choose to do so. DESDR itself does not provide tools for such an estimate.

Client-side Visualizations

Toolsets (2) and (3) also require data visualizations and the aforementioned user inputs passed to DBT via a parameter dictionary. Data visualization and user interaction are the primary functionality of our front-end (i.e., client-side) design. For DESDR, the client-side renders charts, which help users to design and assess the insurance policies based on customized user inputs. These inputs include the region, the starting and ending year, the start and end window, and many other different parameters. Users can adjust the parameters passed to DBT via sliders, input boxes, and dropdown menus. After each modification of parameters, the charts rerender.

During the development process, there were a few suggestions the FIST team gave us to make the client-side design more user-friendly. The first suggestion is that users should be able to change the parameters to their desired values and the charts should be updated according to these new values. This will also be the main feature for this section. The second suggestion is that the charts should align with the sliders tracking timeframes to allow users to better understand how their selections will impact the server-side data pipeline. The last suggestion is that we will need to try to make our graph responsive so that the graph size could adjust to fit for different devices (PC, cellphone, etc.).

To complete the first task as suggested by the FIST team, we first pass in the default parameters and store them as a JavaScript dictionary. We then create some range sliders by using a package called "svelte-range-slider-pips". These sliders are bound with the values in the dictionary we previously created so users can change the values in these sliders, and whenever there is a change in the dictionary, the dictionary will pass these new values back to the server-side, and re-do the calculation and get the updated data for D3.js to display.

For the alignment issue, since we have many different types of modular chart components (bar graph, grouped bar graph, and line graph) and they all have different margins in D3.js, sliders will need to adjust their width according to the type of chart. We solved the need for width resizing using Svelte Stores. A Svelte Store is "simply an object with a subscribe method that allows interested parties to be notified whenever the store value changes."[19] We put writable-stores in a Svelte store.js file, which contains the coordinates of the first and last elements of the D3.js charts. We use these written values to calculate the width for the sliders. Whenever the coordinates change, the width of the sliders will change with it. By default, graphs created by using D3.js are not responsive, meaning the size of the graph will remain the same even if we change the browser window size.

In order to make our design fully responsive and be able to change the graph size according to the screen resolution, we used Svelte built-in elements called "svelte:window" and "watchResize". The first one binds with the inner width of the browser window, and the second one detects if the screen size is changed. After detecting there is a change in the screen size, we use the inner width of the browser window to set the width of the charts, which also affects the width of the sliders, so all charts will change their size according to the inner width of the browser window, which makes our design fully responsive.

[19] https://svelte.dev/tutorial/writable-stores

As briefly mentioned in Technical Details, our D3 charts are designed as Svelte components, which means we can re-use our code for the charts across DESDR deployments. All a user needs to do is to insert the pre-created chart components into the HTML section of the main Svelte app in a DESDR deployment, and pass in some necessary parameters. If the FIST team or other users want to add new charts, they can simply insert these components and the new chart will be automatically generated.

Discussion and Future Work[20]

In the future, the work done this summer will need to be compiled into a base toolkit repository. This will serve as the complete version of DESDR, which we aim to train the FIST team to manage. They can deploy DESDR implementations by country where they support index insurance schemes. Furthermore, we have begun work to make it possible to use DBT with duckDB web assembly (i.e., duckDB-WASM), which should remove the server layer and allow offline use of DESDR implementations.

This work has involved modifying DBT source code to add a *dbt client* command which compiles models to a target folder with globally unique identifiers for variables from the *dbt_project.yml* file maintained. We then have begun testing an inliner script that converts these models into a JSON file that can be exported into a JavaScript library to be executed against data loaded via duckDB-WASM in a web browser.[21] To make this possible, we need to further test the inliner and write the appropriate updates to user guides and our Svelte application to enable deployment of this more flexible (i.e., load data once then work offline) DESDR toolkit. We also believe this implementation will greatly improve the speed of query processing and thus graph rerendering based on proofs of concept. We also will seek in the future to implement logging of user inputs in the parameter dictionary. In the case of DESDR deployments for index insurance, this should enable the FIST team and their partners to further study and improve index insurance schemes as well as providing the opportunity to better understand how rainfall might differentially impact farming communities.

Creating DESDR was a team effort that included many people beyond those of us working under the DREAM program. The above sections then are common but the following section is unique between final papers submitted as conclusion to the DREAM program.

For the first few weeks of the program, I worked with my fellow DREAM recipient Yuhao to learn more about how DBT and Svelte work. This involved creating a test application, and we exchanged notes with each other on how the various tools worked. We also spent a fair amount of time getting up to speed with Docker and developing setup guides that could also be used by the FIST Team.

[20] While the majority of the work described in prior sections was completed by us under the DREAM program, we do note that contributions being done towards the future work is substantially done by others in the WuLab. We have made some and may make further more substantial contributions towards this work going forward as well.

[21] We are still working to understand how duckDB-WASM will be used in future versions of the DESDR toolkit. Prior work done in the WuLab for DESDR showed a proof of concept for how it can be employed. If you are interested in duckDB-WASM, you can see the following link. We note however that while we have skimmed this link, we are not claiming an in-depth understanding of duckDB-WASM or how it will be employed at this time. That will be part of future work. See here for the link: https://duckdb.org/2021/10/29/duckdb-wasm.html

My primary contributions to the project were related to everything covered under the server-side of Figure 1. I developed the skeleton of our Python server, and I set up the necessary Docker containers to run the system described in Figure 1. I worked with Yuhao and the FIST team to develop an example data pipeline for Ethiopia. This included working to develop Python preprocessing scripts and incorporating Yuhao's idea to use DBT seed to load raw data. I also developed further how-to guides and video walkthroughs for modifying data pipelines in the future to allow the FIST team to develop their own analytics. As we look towards future work, I have started to help look at the DBT source code with other members of the WuLab and to think about how we can use DuckDB-WASM to replace the server-side based on past proofs of concept and other research projects.