The problem description, motivation, and survey of related work as in the project proposal, but more detailed and refined.

The open course project chosen by our group was the creation of the Duke Textbook Marketplace, which offers a centralized location for all Duke students to buy and sell textbooks.

The marketplace can be accessed at

http://duketextbookmarketplace.herokuapp.com/#/

Duke students currently use a variety of different services to buy and sell textbooks for their courses. The most popular include the Duke Textbook Exchange (Facebook Group), Amazon, eBay, and the Duke Bookstore.

Duke Textbook Exchange - As a Facebook group, the Duke Textbook exchange provides a quick way for students to create posts regarding textbooks they are willing to buy or sell. Because almost all Duke students use Facebook, this service provides a centralized marketplace specific to Duke. However, many problems still exist. Only recent posts are easily accessed/displayed, so posts that are over a few days old are rarely seen. In addition, the status of books are typically unknown, so there is no way of knowing whether a posted book has already been sold. Lastly, there is no way to filter posts by department, course, etc.

Amazon - Amazon offers textbook delivery, and has access to almost any textbook a student would ever need. However, the delivery methods can often be unreliable, as ordering without Amazon Prime results in long, and often unknown delivery times. In addition, fees for shipping and handling significantly increase the price of buying textbooks.

Duke Textbook Store - The textbook store located in the Bryan Center offers every book that is necessary for courses at Duke. However, used books are often difficult to purchase, and new books are extremely overpriced. In addition, the store will only purchase books back from students at a significant discounts.

The Duke Textbook Marketplace aims to solve the issues of other forms of textbook exchange. Specifically, we provide a secure and centralized location for Duke students to buy, sell, and exchange textbooks.

A few features of the Duke Textbook Marketplace are listed below

- Authentication by Netid - Though we were unable to properly integrate Duke's Shibboleth tools within our web application, we do provide a means by authenticating Duke users through their netid. This provides, at least at a basic level, a marketplace platform that is limited to the Duke community.

- Textbook Search - After logging in, a user can easily search through an extensive textbook list. The marketplace allows users to simply search for textbooks by department name, or through a fully functional search bar.
- Add / Delete Textbook Listings - A user can easily add a textbook listing, and remove the listing once the book is sold.
- Textbook Watching - Users can specify textbooks that they are 'watching.' In the future, this will allow for users to be notified when any watched textbooks are listed in the marketplace.

An in-depth discussion of your system, including the design choices you made.

We chose to implement the following schema for our database with stated key constraints:

- class_to_book(class_id, isbn (foreign key to Textbook *isbn*))
- course(class_id, *dept_id* (foreign key to Department *dept_id*), course_num, course_name)
- department(dept_id, dept_name)
- listings(listing_id, *netid* (foreign key to User's *netid*), *isbn* (foreign key to Textbook *isbn*), date, statusOfBook, conditionOfBook, price)
- score (tid, score)
- textbooks(isbn, title, author, description, edition)
- transaction(tid, netid (foreign key to User's *netid*), listing_id, type, status, date_posted, date_paid)
- users(netid, firstName, lastName, major, phoneNumber)
- watching(netid, isbn)

We developed with AngularJS/JavaScript, HTML5, CSS3, Python, PHP, and MySQL.

On the backend, we have a Duke Co-Lab Bitnami VM set up to host the webapp. The VM comes with a MySQL storage engine and Apache2 already set up. The VM also come with a visual GUI to work with our MySQL database which provides an easy access to look through our data and check for possible errors. There are some limitations using this VM as co-lab only supports up to a certain amount of data, however, for this stage in our project, the co-lab VM is good enough to handle the amount of data that we have.

As for the database component of this project, we ultimately wrote PHP query scripts to access our MySQL DB. These will be the functions that the front end will call to populate whatever data the front end needs to display. We formatted the JSON string result of the query calls to match our front end's needs before passing it on. As for the Python component, we ran into a bit of trouble installing the module MySQLdb which would allow us to make calls to the MySQL Database. Instead of wasting time to install the MySQLdb module, we wrote PHP script that will do the same purpose. In order to connect Python with the PHP script, essentially, Python is just executing a get request for the query that it needs. The PHP script will return the result query in JSON format and the rest of the Python code would parse the query into a JSON format that is used by our frontend view component. However, since Python only acts as a "messenger" between PHP and Javascript, we figure to use the direct link between PHP and Javascript instead.

Evaluation of your system, and if applicable, comparison with competing systems. Be clear about what your evaluation metric is. If you have experimental evaluation, describe the experimental setup in enough detail so that others can repeat your experiments.

Our current system was successful for our PHP and Angular communication, in terms of running and testing SQL queries with little overhead, much to the execution speeds on the Co-Lab Bitnami VM, as well as seeing updated schemas in the Bitnami VM upon testing out features on the Angular client side. After writing SQL queries to the local database on the VM for each of the commands (Insert, Delete, Update, Select), the results would be dumped onto the VM, which would then be accessed by the HTTP GET requests.

Our system could have been tested and developed with more ease had we neglected to use PHP and replaced it with Django. The maturity of Django's architecture and its RESTful framework has many advantages that could have been utilized with many benefits such as:

- extremely simple API calls for our Angular guru to dynamically update our database and reflect those changes on the Marketplace pages
- an ORM layer that divorces us from the trouble of converting Django model instances into something serializable on the Angular API endpoint, as well as petty considerations such as PostgreSQL vs MySQL
- Separation of processing and presentation: processing in Django view functions that construct rules and presenting our template by rendering the results as HTML/CSS
- Relatively little code in that we would not have had to write so many complicated transactions in Python (and when testing this code showed faulty HTTP requests in our browser very quickly once we had our server running the Django module on a local port such as 127.0.0.1:8000)

Unfortunately our backend team experimented with Django to a limited degree: we completed a RESTful API with hyperlinked relations but faced complexities in running a server instance of Django every time:

- OperationalError: unable to connect to MySQL server through socket when sending a GET request
- Trouble with virtualenvwrapper installation
- Import errors on basic Django classes such as Views.py and Model.py due to an incomplete understanding of the libraries being used in different tutorials
- Linking API to boilerplate functions on Angular
- Incorrect URI mappings in the patterns function (never resolved)
- A lack of resources for direct debugging due to complicated meta request tracebacks. We thereby mismanaged the settings file, URL mappings, and the API application that we made.

It basically came down to a difficulty of completely grasping the ORM layer (separated from the other layers) which led us to flawed over-generic models based on SQL and not enough of an object oriented mindset. In addition, there is no doubt a more efficient way of managing each session on the virtualenv and to better control our workflow, we could have reverted back to a working standalone.

Any open issues or directions suitable for future work.
We have several limitations to our product. While users can sign in and register based off of their netid, there is no integration with Duke's OAuth service, that allows us to user their API to sync up with a user's password. The API that we use to access the textbook data is very messy and unorganized, which can be a minor annoyance to read for users, but is otherwise negligible. By pulling so much data on courses and their associated textbooks, there can be a slight latency when the textbooks are first loaded in. This latency, however, is usually < 1 second on most computers and can therefore be considered negligible. This can be improved by pushing filtering on other parameters (such as by department) on the database side so that we don't load all textbook data per page by default. Another limitation is that a user with malicious intent can easily figure out our public Streamer API calls making our database potentially vulnerable to SQL injection attacks. If we had integrated with OAuth, this would not be an issue. Finally we were limited by sheer design: none of us are graphic designers and we don't have extensive experience with UI/UX design. Our interface can be lacking in terms of aesthetic appeal, but in terms of functionality, our application is solid.

To further develop our Marketplace website in the future, we can improve our basic Net ID login by replacing it with Shibboleth user authentication for each session such that we can validate user specific SERVER arrays that have already been populated. We can also use a variety of external API's instead of writing our own (in the case of a Django REST API). In order to fully implement transactions, there must be at least two other API's used. The first is Venmo OAuth authentication, in which both the buyer and seller must have Venmo accounts for the transaction to go through. Once a textbook of choice is found upon querying our database, the buyer may wish to offer a price to the seller in the public listing. If this price is accepted (another feature that is yet to be implemented) then the seller must either charge the amount via a Venmo link, which would have the proposed number parametrized into the URI already. Another way to set up these negotiations would be a text-based input module for which negotiations over prices and Payout Links (in String form) can be sent, which would entail calls to Mandrill's transaction messaging API or a separate (limited by token expiry) realtime P2P chat application with the Node webkit, Angular, and Firebase.