# Rideshare: Connecting Travelers

EE 364: Requirements Specification

Yoni Ben-Meshulam  
Garrett Cooper

Team Venture

Dr. Bruce McCann

November 7, 2007

**TABLE OF CONTENTS (Unedited)**

1. Introduction 2
2. Needs Analysis 3
3. Deliverables 6
4. Principles of Operation 7
5. Inputs and Outputs 8
6. User Interface Requirements 10
7. Key Operational Specifications 13
8. Key Environmental Specifications 14
9. Prototype Cost 15
10. References 16

**LIST OF FIGURES (unedited)**

F-5.1 Input/Output Diagram 8

F-6.1 Google Maps Application Example 10

F-6.2 Rideshare Sequencing Diagram 12

# Introduction (unedited)

Our product aims to connect drivers who want passengers with passengers who want drivers. It will do so in an efficient manner, which minimizes travel time and distance, as well as taking into account the preferences of customers.

In a time when energy and fuel efficiency pressuring people to leave their car at home and seek alternatives, our rideshare application can help people coordinate rideshare programs and plan trips efficiently and effectively. People who share rides regularly will significantly reduce their ecological footprint and save on travel expenses.

The target audiences can be divided into three types:

* Travelers who are interested in a regularly-scheduled rideshare to their place of work or school, or for their children.
* Travelers who seek one-time partners for longer trips (i.e. cross-state).

Many people travel alone in their cars, with much excess capacity wasted. This waste is compounded by the increasing price of gasoline. People who seek alternatives frequently find mass transit to be unreliable and underdeveloped, and must continue to waste their own and the world's resources.

Our solution will enable individuals to communicate their travel needs and find others who are compatible with those needs. It will eliminate much of the coordinating tasks from the rideshare paradigm, and subsequently increase the likelihood that people will choose rideshare. Our hope is that this product will help facilitate a paradigm shift from single passenger travel to a ride sharing culture.

In order to accomplish this, our project must meet certain basic criteria. These criteria will ensure its proper functionality, reliability, and maintainability. In the following sections, we establish this set of criteria, which will guide us in architecting and developing the application.

# Needs Analysis

In order to successfully implement our rideshare application, it will be necessary for each facet of our system to adhere to a set of requirements. The following are what we consider to be basic requirements for a number of crucial elements of the system.

# Matching

When matching riders and drivers, we must implement an algorithm which generates the most efficient results, where efficiency is a measure of a number of parameters. The foremost of these is maximal matching, i.e. the highest number of customers receiving rides. Maximal matching is crucial for the results to be effective and useful for our customers. Route lengths and times must also be minimized by the algorithm, and suggested rideshares should be viable, intelligent solutions. Timeliness of results is also important, since an effective result which doesn't complete in a timely manner is useless. This last requirement can be quantified as an algorithm which runs in polynomial-time, as opposed to exponential-time, which means it will complete some time tolerable by humans. Exponential-time algorithms can potentially take longer to complete than the history of the universe.

The algorithm should also utilize external inputs efficiently. This can be quantified as an algorithm which minimizes route requests to Google-Maps by reducing the search-space, i.e. the number of possible valid matches. For example, if a driver is going from Austin to Houston, the algorithm would refrain from evaluating routes for which the riders' beginning and end points aren't in Texas. Those that are in Texas can further be reduced to riders' who are in central Texas, etc. Once this pre-processing has been completed, we would evaluate only those combinations within the limited search space which has been generated.

2.2 Server

Our final product will need a server that can support the projected bandwidth and storage requirements. The server must have an application server, preferably Apache, installed on it. Apache can run our implementation, which uses an SQL database and the programming language Ruby. We will need an application development environment such as Eclipse, which supports web application development, including Ruby, SQL, and JavaScript. Our choice must be evaluated on its reliability, effectiveness, usability, and cost. Within this environment, we will need to enforce good programming practices, including a secure wiki and file server for collaboration, naming conventions, revision control, and modularization of programming tasks.

2.3 Software

The final product will also need to meet certain reliability and availability parameters. Reliability is the measure of how stable the program is. An unstable program will crash or generate erroneous results, whereas a reliable one will perform with minimal interruptions. The program should also be free of bugs. Availability is the measure of how much down-time, measured as the percentage of time during which our website is unavailable, we can expect to have. These will both ultimately depend on our ability to test for problems early on, and be able to quickly fix any bugs.

Rideshare needs to implement a solid user interface. This includes designing an intuitive menu and a friendly graphical interface with usability in mind. We must minimize the number of menu levels, as well as the number of operations required by users.

It is also imperative that we establish application-level security in order to protect our users' privacy, their personal computer, and the application server from malevolent entities. This requires that we use a secure web server and design all incoming and outgoing communications with security concerns taken into account. We must evaluate the web server and implementation methods based on this requirement.

A good testing platform will be necessary in order to evaluate each module of the application. To accomplish this goal we must design the software with testing in mind, also called design-for-test. Each testing requirement must be quantified in terms of our final software architecture. We have included an extensive testing description in section 11.

All of these requirements will need to be incorporated into a high-level application architecture and module-level implementation. At each point in the design and implementation phases, we will need to assess our adherence to these requirements.

Assuming these needs are met, we will need a broad user-base in order to ensure that all users actually find ride-share solutions. Given our application, the more people we have in our user base, the better the solution we will be able to provide in terms of cost, distance, and compatibility.

# Deliverables

At the culmination of the project, we will provide the following set of deliverables, which will encompass the totality of the project, and allow a peer to completely understand our application and implementation methods.

Foremost, we will provide the functional rideshare application running on a server, which users can utilize to find and offer rideshares. We will offer documentation of the high-level software architecture, as well as the low-level module implementation. The actual code will also document the programming structures and functions used for implementing the low-level modules. We will also provide a comprehensive demonstration of this application.

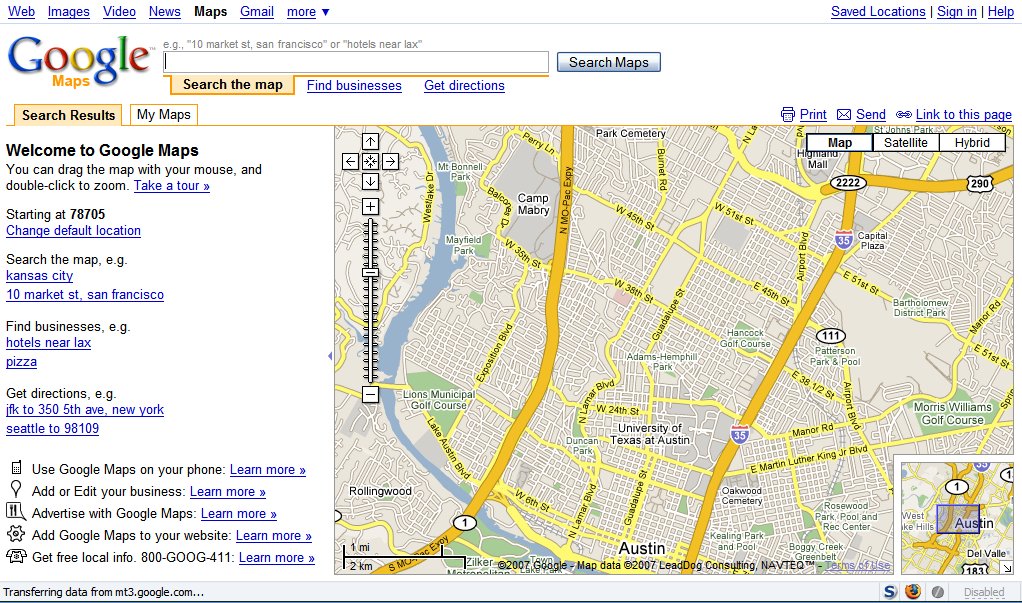
In addition, we will provide a use-case diagram for the application, defined by Wikipedia as "a type of behavioral diagram...designed to present a graphical overview of the functionality provided by a system in terms of actors, their goals — represented as use cases — and any dependencies between those use cases." [1] Later, we can find a more accurate definition for a use-case diagram than Wikipedia if necessary.

In order to demonstrate the practical functionality of the website, we will provide a site map, including the menu architecture and description, as well as a link graph showing the relationships between different pages of the site.

We will include a maintainability document to ensure the proper operation and maintenance of the site. This document will cover the information necessary to implement the application on a server, including a connectivity graph of application components, bandwidth and storage requirements, compatibility issues, and security requirements.

# User Interface Requirements

What sets our project apart from other websites such as eRideShare [2] is the user interface. By giving the map a central role in the user interactions, we create a unique user experience that will facilitate simple and effective ridesharing. We aim to display routes similarly to Google Transit [3], with menu options available as a side-bar. Features like zooming and turn-by-turn directions will be implemented in the same manner. Figure 4.1 shows a typical map-based application on Google’s own website.

**Figure 4.1: Google Maps Application Example**

**4.1 User Experience**

The following paragraphs describe a typical user experience on our rideshare website, though the exact mechanics may change.

The driver or passenger logs in to the secure website. The user then proceeds to create a new "looking for ride/passenger" rideshare, or revise a previous rideshare from his history tab. This entails setting the beginning and ending points, as well as designating a window of time for when he plans to leave. He may also designate the rideshare as public or private, indicating if matches may be found only within his "friends" list, or within the general user base. He may set preferences for how far out of the way he is willing to pick people up, as well as for the gender or age of his counterparts.

The user experience will now split into two cases. In the case of a rideshare which will occur on short notice, the user will receive either an immediate match or a message that no match has been found, given the time-frame requested. He can then either alter his rideshare information and search for a new match, or leave a "contact directly" message, so that potential riders can phone or text him to coordinate a pick-up.

In the case of a ride with relatively long notice, the user will receive a confirmation message and email for his request. He will then log out and wait for an email indicating a match, or check back with the service later. When a passenger is added to the driver's route or vice versa, an email or text message will be sent to both parties, and each user can go to the site and review their counterparts. If the user deems the match acceptable, they can accept the rideshare; otherwise they can reject the rideshare and move on to a new search, or review a new possible match if more than one users matched the criteria. This sequence of events is illustrated in Figure 4.2.

Once a rideshare has been accepted by both driver and passengers, a ride confirmation message will be sent to them. This will include all contact information, along with an annotated map of the rideshare, indicating the route and times for all passengers. If a user cancels a rideshare, the website will request they specify a reason to be sent to their counterparts. After a certain number of cancellations or negative reviews, a user's profile may be locked for adding new rideshares until it has been reviewed by a site administrator.

**Figure 4.2: Rideshare Sequencing Diagram**

4.2 Options

Users may also manage their profile and review or alter different options. These options may include the following: a user’s personal information, such as name, address, gender, and age; general preferences, such as the look and feel of the website, starting page, and communication methods; and rideshare preferences, such as gender, age, and driver ratings. The user may also take advantage of other rideshare tools to alter pending rideshares or set ratings/comments for previous rideshares.

# Key Operational Specifications

In order for our program to ultimately work, each of its pieces must perform to a certain standard.

To begin with, our user interface must be secure. Each user's login information and preferences must be saved, along with their route. We can test this using human beta testing as well as attempting to rudimentarily hack user accounts, changing user settings around and verifying that they have been saved.

Additionally, our data server must measure up to several standards. The database it contains must be secure, in that no one outside our program can query it for data, which we can quickly test by setting a password and trying to bypass it. The server's hardware must also perform well; we must be able to add and drop entries from the table quickly, we must be able to query data quickly, and the server overall must have a large data capacity in terms of hard drive space. Our connection to Google Maps needs to be fast as well.

Our algorithm must be efficient. This implies that it should generate a maximal solution in terms of the number of users' served and the number of passengers per car, while providing those users with the shortest possible routes. We also must be sure that our algorithm can intelligently compare routes, mostly in terms of length and time, doing all this in a reasonable amount of time.

Finally, one of the most important functions of our website is its user interface. The interface must be user-friendly and intuitive, which we can measure by timing how long people spend on each page and where their eyes are. Tester feedback will be essential for improving the user interface, as they can inform us exactly where the website's mechanics aren't working, as well as qualitative things such as if the user did not like the color scheme, or why they chose to leave the website.

# Key Environmental Specifications

On the hardware level, the environmental requirements for our ride sharing program are the same as any other computer: electricity and a cool, dry room to operate in, as well as a connection to the internet. As this is a software-based project, it will have to stand up to some electronic rigors. Our program will need to adhere to some capacity limitations, probably measured in total number of users and routes saved in our database. Because of hard drive space constraints, our database will only be able to hold so many such users and routes without a technician adding extra capacity.

Also, the server can only handle some finite number of requests in a short amount of time without causing long waits, because the server's processing power and communication capacity are bounded.

# System Design

In order to provide the functionality described in the previous sections, the Rideshare application includes several application-level modules. Each of these modules in turn includes sub-modules which implement lower-level functionalities.

**Figure 1.1:** Top-level software architecture

Figure 1.1 describes the top level view of the software architecture. Note the differences between the arrow heads and arrow colors. Information which flows locally, i.e. between modules of our application, to provide regular functionality will always be marked with orange arrows. Data flow which corresponds to testing is marked with red arrows. Information originating or terminating in external sources is marked with blue arrows.

For ease of viewing the application architecture, we have omitted the “internet cloud” from the following diagrams, though its presence is implied.

The main modules in our application, marked with orange outlines throughout the rest of this section, will be **rideshare control**, **user interface**, **secure session manager**, **optimizer**, and **testing suite**. These modules provide the core functionalities of the Rideshare Application, and coordinate all local and external communications.

Each module utilizes information supplied from local (other application-level modules) or external (users, third-party) sources. They also utilize an application-level database, denoted **rideshare database**, to store and retrieve information. Application-level modules will access only those tables relevant to them, as we will show in the module-level designs.

Descriptions of each module, sub-module, external source, and database tables will be provided in the following sub-sections.

## Rideshare Control

The rideshare control module can be viewed as the “conductor” of the rideshare application. It is in charge of running and maintaining all of the other application-level modules.

The following sub-modules comprise **rideshare control**:

### User Interface Control

This control sub-module will initialize and allocate resources for proper functionality of the user interface. The controller will be the program which “calls”, or in other words runs, the user interface.

Any maintenance required for the user interface will be provided by the user interface control. For example, if the user interface has surpassed its capacity, the control module will allocate more resources, or, if necessary, terminate the user interface module.

In addition, any requests for rideshare matchings from the user interface will be handled by the control module.

### Optimizer Control

The optimizer control sub-module “calls” the **optimizer module** when matching results are needed. It relates to the optimizer which sets of rides are to be optimized, and what time priority they have.

The controller will also initialize and maintain any resources necessary for the proper operation of the optimizer, similarly to what was described in the user interface controller.

### Server Control

This sub-module will perform all initialization and maintenance routines for the rideshare application. Any server-level interactions, besides normal internet communications, will be handled through the server controller.

For example, if the user interface and the optimizer both require additional resources, the server controller will be the arbiter, and reallocate the server resources according to some policy.

### Database Control

This controller will initialize and maintain the database. It will also set restrictions on database access at the application-level.

### Arbiter

There are potential cases of “critical sections” between the modules. This is a case when one module alters data which is being used by another module. The second module will subsequently be using “stale”, or old, data.

An example of this can be seen in the case of a user changing their ride-request details while the optimizer is running the corresponding request.

The arbiter will inform the optimizer of the critical section, and in some cases allow it to include the “fresh”, or new, data.

Wherever other such situations arise, the arbiter will perform similar duties.

### Backup Utility

This sub-module will backup all rideshare application information, including the database, log files, statistical data, and server log files.

Source-code files, i.e. our software development project, will be backed up separately.

## User Interface

The user interface control module can be viewed as the “agent” connecting the rideshare application and the users. It runs separate user interface sessions for each user, interacting with the user through a set of menus, forms, and maps. All information exchange and communication with the internet browser (i.e. Firefox, Safari) will be sent to the user interface via the secure session manager. Output and data storage tasks will be performed based on user input.

The following sub-modules comprise **user interface**:

### Input

# Form

All user form entries will be processed using a set of functions devoted to this task. This utility will also store form entries for each user. This will allow for automatically filling fields on similar forms.

# Interactive Map

Any user map inputs will be captured by this utility. Support for all Google Maps interactions necessary for the proper functionality of the map interface.

The map interactions utility will store statistical data on these interactions, as well as user-specific data. This will allow for site administration and testing, as well as a user-friendly map, i.e. one which “remembers” where the user’s usual locations are.

# Menu Interactions

Any user input to menus will be captured and processed by this utility. This includes all navigation and actions, i.e. links and clickable “buttons.”

The map interactions utility will store statistical data on these interactions, as well as user-specific data. This will allow for site administration and testing. In addition, it will allow for a better user experience, since the utility “remembers” the user’s favorite pages on the site, and thus saves the user repetitive navigation of the site to get to these pages.

### Output

# Form

The form utility will generate the relevant form according to the page-relevant form needed. If the user has previously filled similar fields, the utility will fill the fields for the user, according to the values which the user “prefers.”

For example, if the user’s history for the field “Name: \_\_\_\_\_” is “Garrett”,”Garrett”, and “Yoni,” then the field would read “Garrett” the next time the user sees it.

# Menu

The menu output utility will generate the relevant menus for the user, according to the page-relevant menu, as well as the user’s “favorite” links.

# Interactive Map

All map outputs will be handled by this utility. This includes map overlays and menus. Map outputs will be handled by this utility using the standard Google Maps API.

# Page Frame

The actual website that the user sees will be handled by the page frame utility. This utility can be viewed as a set of blueprints for the site, with specifications for the placement of content. Each blueprint corresponds to a different set of page configurations which are presented to the user.

The form, menu, and interactive map utilities will output their content to the user via the page frame utility.

The page frame utility will record statistical data for uses in site administration. It will also store user-specific information to regenerate user-specific “favorite” page-frames. For example, if a user were to usually open an “advanced options” version of the ride request form, then the user’s default page-frame for that page would already have advanced options visible.

### Statistical/History Utility

This utility provides the input and output sub-modules with the routines for storing and retrieving statistical and user-specific data. It will provide functionalities corresponding to the above-mentioned usability and administrative features.

## Optimizer

The optimizer module is responsible for generating rideshares which are both efficient and timely. It takes as inputs the sets of all ride offers and ride requests, and outputs the optimized results.

### Theoretical overview of the optimization problem

Some understanding of elementary graph theory is necessary in order to solve our optimization problem. We include this section to assist in understanding the optimization problem at hand, and thus gain an understanding of the optimizer module’s required functionalities. Definitions are marked with bullets.

Definition of a graph:

* A *graph* is defined as a set of *vertices* V={v1,v2,…,vn} and a set of *edges* E={e1,e2,…,em}, for some positive integers m, n.
* The vertices can be thought of as dots, and the edges as lines connecting those dots, as shown in Figure 7.3.1.

Our optimization problem can be viewed as a graph theory problem. In our case, the drivers (D) and riders (R) can be viewed as vertices (dots), where two vertices are connected by an edge (line), if they can possibly participate in a rideshare.

Figure 7.3.1: Bipartite graph of drivers and riders

* A *matching* is a set of vertices and edges. For a matching M, two vertices are said to be matched if M contains an edge which connects them.

Our optimization problem can be viewed as equivalent to the *maximal matching problem*. The maximal matching problem is the problem of finding a matching within a graph which includes the maximum number of vertices. Translated to our problem, this means finding the most matches between drivers and riders, and thus providing rideshares for as many people as possible.

In particular, our optimization problem can be viewed as equivalent to a *bipartite graph* optimization problem.

* A *bipartite graph* is a graph in which the vertices of the graph can be divided into two disjoint sets, where no two vertices in the same set can have an edge between them.

In our case, we have two sets of vertices, D (drivers) and R (riders), which can never have rideshares within the sets, i.e. no two drivers will be in a rideshare together, and the same holds for riders. This relationship can be seen in figure 7.3.1, where no edges are visible within the sets.

### Problem Reduction

In order to find a maximal matching, we first need to reduce the search space of the problem at hand. Search space reduction is the process of eliminating possibilities which are clearly invalid, and thus reducing the amount of work necessary for finding a matching.

# User Criteria

In this utility, user preferences will be used to eliminate possible matchings. For example, if a user specifies that they will ride only with females, any matching which includes males will be eliminated from the search space. In graph theory terms, the user criteria search space reduction removes edges between the vertices of incompatible users.

Similarly, users with conflicting time preferences or age preferences will not be considered for a rideshare with one another.

# Geographic Criteria

As with user criteria based problem reduction, it is possible to remove those solutions which are obviously incompatible on the basis of geographic distance. For example, if Jane is driving from New York to Maine, and John needs a ride from downtown Austin to North Austin, then Jane and John are obviously incompatible for a rideshare.

The output of the problem reducer sub-module will be a set of smaller problems which can then be processed by the optimization algorithm. This will look like a collection of sets, each of which includes some number of riders and drivers who have passed the initial search space reduction.

### Matching Algorithm

After we have reduced the search space in the problem reduction utility, we will need to employ an algorithm to find matchings for each of the sets generated. An algorithm is a set of consecutive operations, which will guarantee a certain result.

In our case, we have found a maximal matching algorithm which is designed specifically for bipartite graphs.

We plan to use the algorithm specified in [10]. Specifically, we will use the algorithm for bipartite graphs. This algorithm is proven to run in polynomial time, meaning it is possible to run it on a large amount of data, while completing the optimization in a reasonable amount of time.

We may also employ other optimization methods as we develop our specific matching algorithm.

## Secure Session Manager

In order to ensure the privacy of our clients, all communications between the application and users will be performed via a secure session. The secure session manager, Figure 7.4.1, will provide secure session services to all other modules requiring outside data or interaction.

It will do so by creating secure sessions for each module, and possible many sessions for a single module. For example, the user interface will require a secure session to be created and managed for each user who is interacting with the application.

## Rideshare Database

Each of the above mentioned modules has had one or more databases which are necessary for it to store and retrieve data, and ensure its proper functionality.

The rideshare database, Figure 7.5.1, will contain all of these databases. Each database represents a set of tables, which in turn contain a set of fields. We will utilize a MySQL database to create and maintain the rideshare database.

The specific tables for each database will be determined during the implementation phase of the project.

# User Interface Design

The quality of the user interface for our application is crucial, since the usefulness of our program will be directly correlated with the size of our user-base. The easier our website is to use, the more people will use it. The login, rideshare creation, and confirmation pages are outlined below.

Figure 8.1: Login Screen

The login screen is fairly basic, but it illustrates an important point. Every user will be required to log in, only entering the website through the secure session manager. If the user does not already have an account, they will be able to register through the link on the bottom right. Beyond this, the top of this page will have a logo and attractive heading for the website. The bottom left will have a link to an unsecured “about” page, where the user can read basic information about what the website does.

Figure 8.2: The New Rideshare Page

On this page, the user can input whether they are a driver or a rider, which will open new options, as shown by figures 9.3 and 9.4 below. Most importantly, the user will input their route, in terms of where they will be leaving from, and where they are going, as well as when they can leave.

The time window must be specified at least partially, but the user does not necessarily need to enter both. For example, if the user specifies only the departure time, then the application will be able to use the estimated drive time to specify the arrival time.

Figure 8.3: Driver’s New Ride Figure 8.4: Rider’s New Ride

Each user must enter whether they are a rider or a driver not only so the program can match them correctly, but also so they can enter these role-specific options. The algorithm needs to know how many passengers a driver can accommodate, as well as how many passengers and luggage the rider will be bringing.

Figure 8.5: Passenger Confirmation Page

Figure 9.5 shows how a passenger confirmation page will look. This page allows a user to view a potential passenger, their details, and how the combined route would look. The passenger’s name (in this case “John Rider” will be a link to their profile, where the driver can view their details. The stars by the name represent the user’s rating, like eBay. In this case the user has a rather good rating of 4 out of 5.

The example page indicates that the passenger has not yet accepted the driver, which is indicative of the double-handshake process in which both the passenger and the driver have to approve each other before the rideshare can take place. If the passenger had accepted the driver before he saw this page, it would say so. A similar page will be created for passengers approving drivers.

Notice that each of these pages has the home bar at the top. Once logged in, every page the user sees will have this bar visible at the top. Through it, the user can navigate through the website and view the following pages:

1. My Rides, which will display the status of the rides the user has created or been added to.
2. My Profile, where the user can change their details and set personal options.
3. New Ride, which would take the user to the new rideshare page, Figure 8.2.
4. My Friends, which will let the user easily communicate and set up rideshares with people they know.
5. Main Page, which will take the user back to the front page, where they can log out or read the website’s current news.

Add other communication, like email and text messaging (?).

Change heading styles.

Add sections

Look for technical information (like algorithm, software module), and add further explanation.

# Prototype Cost (Unedited)

We will likely need to purchase several tools to put our first run program together. Initially, we would need an internet-enabled server computer to run our programs, as well as crunch all the numbers. We could rent a server from several different companies for somewhere around $250 per month [4], or at least a virtualized portion of one server on which we could run just what we needed to. Virtualization allows multiple people to have access to the same computer as if they were separate physical machines.

Alternately, the University could furnish us with a virtual server for essentially no cost to us.

The server also requires a few programs to run our project. Tomcat, a program which runs a Java web server, is freely downloadable on Apache's website [5]. Additionally, we will require a structured query language, or SQL, database program such as Oracle [6], which is expensive, or MySQL [7], which is free. The database program will allow us easy storage and retrieval of hundreds of thousands of data entries, which is necessary to support a large number of users.

We may also need to acquire a Software Development Environment (SDE), such as Microsoft Visual Studio, which could cost us as much as $300 [8]. This compiler and development environment will allow us to actually put together and publish our code, and it would be very useful in debugging. Alternately, we can likely use University resources such that this cost would not be incurred, or use an open-source SDE such as Eclipse.

To communicate with Google Maps, we will need to use their application programming interface, or API. To obtain this, we will use an API key provided by Google, which is free so long as we do not charge for our service [9]. However, if we do charge for our RideShare program, we will have to purchase an Enterprise API Key from Google for $10,000.

All totaled, our costs could be over ten thousand dollars, or free, depending on the university.

# Project Cost and Schedule (new section)

The prototype cost for our project is almost completely overhead; as we are doing a software project, the product itself doesn’t cost anything, it’s the tools we require to create it. As a project, we will use free software, but to continually run the site, we need electricity to keep the server running.

The schedule, however, requires more consideration. The first week we come back to school, we’ll meet up with Patrick, who will be our third member for 464. We will review the schedule set forth on the Gantt chart in Appendix C and make changes as need be. The next week and a half or so will be for setting up the server, in terms of actually physically obtaining it and then setting up the various programs on it. After that, we’ll begin the first step of the actual website, the user login mechanism, or secure session manager.

The next week will be devoted to specifically and explicitly defining the module interfacing definitions. We will review what we know, and explicitly figure out which functions each module can call from the other functions, and what exactly these functions will be called. Once we have this information, we can specify what each of the tables will be called in the database, as well as hammer out any errors in the database left over from the initial bringup.

While Garrett and Patrick are working on the database, Yoni can start working on the optimization bringup. After that, we can work on communication with Google as well as basic control and matching. Once this is done, we have our first real milestone: all the basic functionality will be in place, so we can test the program as a whole.

The rest of semester we will work on the advanced features, as well as refinement or debugging of features we have already implemented. The last few weeks, we can work on documentation and aesthetics.

# Test Plan

In order to develop a stable and reliable application which fulfills the set of requirements outlined in previous sections, we will need a comprehensive testing utility. This utility will need to comply with all of the requirements mentioned in the needs analysis section (section 2) of this document.

The implementation of each module will include a set of functions specifically designed for testing that module. This method of testing is called unit-testing. These functions should include the following test options:

* Functionality  
  Each module will include methods for evaluating the inputs and outputs of that module and assessing their correctness.
* Performance  
  The testing suite should include methods for evaluating the timing behavior, i.e. how long it takes to complete, of the system under various circumstances.
* Stress  
  Each module will have a method for creating a large number of requests relevant to that module. This will allow us to test for and approximate the maximum number of users, server-requests, etc. that our application can handle.
* Security  
  Evaluates server-level security and application level security. This allows us to establish a definitive security framework within which we are operating. We will design a security testing module which attempts to bypass each security feature in the application.
* Usability  
  We can assess the usability of the application by having test-users enter and navigate the application, and comment on the menus, look and feel, options, and navigability. We can then alter or remove content based on these results.

A good testing system will be necessary in order to evaluate each module of the application. In order to accomplish this we must design the software with testing in mind, also called “design-for-test.” For each module, we will assign a set of test-cases relevant to that module’s functionality, and conduct a strict policy of adding to the application only those modules which have passed the necessary tests. Each module will also be assigned a test version, to indicate the level of testing for which it has been approved, and subsequently its readiness for launching the final product.

# Conclusion (New Section)

# References (Unedited)

[1] Unknown Author, "Use Case Diagram" [Online document], 2007, [cited 2007 Nov 6], Available HTTP: http://en.wikipedia.org/wiki/Use\_case\_diagram  
  
[2] Unknown Author, "Use Case Diagram" [Online document], 2005, [cited 2007 Nov 6], Available HTTP: http://www.erideshare.com/  
  
[3] Unknown Author, "Google Transit Beta" [Online document], 2007, [cited 2007 Nov 6], Available HTTP: http://www.google.com/transit  
  
[4] Unknown Author, "eApps Hosting Plans," [Online document], 2007, [cited 2007 Nov 4], Available HTTP: http://www.eapps.com/Docs/Documents.jsp  
  
[5] Unknown Author, "Apache Tomcat," [Online document], 2007, [cited 2007 Nov 5], Available HTTP: http://tomcat.apache.org/  
  
[6] Unknown Author, "Oracle, The World's Largest Enterprise Software Company " [Online document], 2007, [cited 2007 Nov 5], Available HTTP: http://www.oracle.com/  
  
[7] Unknown Author, "MySQL, The world's most popular open source database " [Online document], 2007, [cited 2007 Nov 4], Available HTTP: http://www.mysql.com/  
  
[8] Unknown Author, "Visual Studio 2005 Standard Edition" [Online document], 2007, [cited 2007 Nov 4], Available HTTP: http://msdn2.microsoft.com/en-us/vstudio/aa718672.aspx  
  
[9] Unknown Author, "Google Enterprise Website Search Solutions" [Online document], 2007, [cited 2007 Nov 3], Available HTTP: http://www.google.com/enterprise/maps/faq.html

[10] Santosh Vempala, “Matching Algorithms”, *Combinatorial Optimization Lecture Notes*, 07-Oct-2004, Available HTTP: [http://www.cs.dartmouth.edu/~ac/Teach/CS105-Winter05/Handouts/](http://www.cs.dartmouth.edu/%7Eac/Teach/CS105-Winter05/Handouts/)

# Appendix A: Relevant Standards (Not technically a new section, but…)

# Appendix B: Bill of Materials

< see section 9, prototype cost for reference >

# Appendix C: Gantt Chart

**--- TO BE ATTACHED FROM gantt.xls ---**

Revision control:

1. Added all of content.
2. Yoni’s second edits
3. Garrett’s Ton-O-Edits
   1. Added mentions of what each section needed by its heading
   2. Added correct numbering for each section
   3. Finished main UI section (section 8)

**Notes to you!**

* ~~None of the numberings are correct.~~
* The new section (7), currently just appended to the end of the document, has correct multi-level numbering format. You can copy the attributes (control-shift-c, or paintbrush) of this line, and give your section the same. Or just ignore it, and I’ll do them all at the end.
* Ask Kristin about:
  + Words in bold, used to allow reader to find modules within text.

Operational specifications