# TraderAuto+

## Connecting Car Buyers with Subprime Lenders

### Project Overview & Design Document


**Jia Hao Choo, Ameen Parthab, Sophie Sun, Elizabeth Li, Daniel Xu**

# Table of Content

*You may click on each heading to navigate to that specific section.*

## Problem Domain Summary

With the current user-journey, consumers with lower credit scores aiming to purchase used cars struggle to both communicate and obtain loans from subprime lenders. The existing gap between subprime lenders and consumers acts as an obstacle in the formation of mutually beneficial partnerships.

## Detailed Problem Statement & Solutions

Car buyers with lower credit scores often struggle obtaining loans they require to support their purchase. They are frequently labeled as “high risk, low return” by major banks, drastically increasing their loan application’s chance of rejection. Fortunately, subprime lenders, such as AutoCapital Canada, work to provide affordable yet fair loans while still offering consumers important features such as car trade-in. However, the problem lies in that there are thousands of subprime lenders out there, each with tens to hundreds of potential loan offers. This makes buyers feel overwhelmed and lost which ultimately results in the disconnection between lenders and their potential customers.

Given these consumers lack important knowledge on the loans they are likely to receive, they also have a much harder time finding a desirable car - a car that not only suits their needs but a car that they are financially conscious and capable of affording. As it exists, the car-buying experience is incredibly inefficient and unfriendly to lower credit consumers as they are only truly informed of their purchase’s financial details much later on in the process, typically at a car dealership’s finance office. However, the finance office will always try to upsell them by offering unnecessary services, such as rust proofing, which prevents consumers from acquiring the optimized deals they need. In short: car buyers with lower credit scores need to be more informed in two ways: potential loans they are likely to receive and, based on the loan, the cars most appropriate for their needs.

Through partnering with AutoCapital Canada, TraderAuto+ is bridging the gap between car buyers with low credit scores and subprime lenders. TraderAuto+ solves these existing problems by providing car buyers with more knowledge of the different subprime loan options available to them and a range of cars that they can afford. This revolutionary change expedites the whole car buying process - consumers no longer need to do extensive online research about subprime loan options available to them, as the product directly provides these different loan options and how likely they are to get approved for each loan. Additionally, by shopping only from a range of cars that they are more likely to receive financial assistance for, the likelihood of restarting the whole car buying journey if their car loans are rejected is greatly reduced.

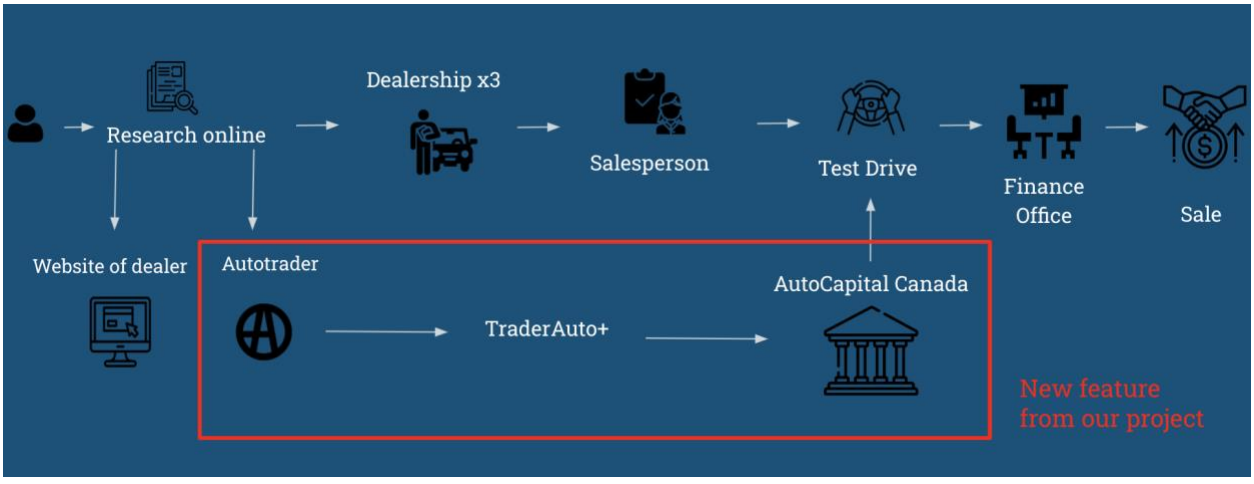
By decreasing the amount of times typical consumers need to restart the car-buying process, this simultaneously decreases the time dealerships spend on trying to sell these consumers a car. TraderAuto+ disrupts this cycle of wasted time by informing consumers of loans they can realistically be approved for so they do not end up getting rejected after going to a dealership. This allows dealerships to more commonly interact with consumers who have a higher likelihood of receiving the financial support they need for their purchase and thus helps dealerships move cars out of their lots faster.

In the subprime lenders' perspectives, TraderAuto+ would be able to help them demonstrate their loan options to a wider audience as consumers get exposed to their loan options much earlier in the customer journey. This helps reduce the existing stigma surrounding subprime lenders as being seen as "loan sharks" or people looking to "scam" the consumer as consumers are more financially aware of what they can reasonably expect from subprime lenders.

TraderAuto+ is designed to create a smooth and convenient purchasing experience for car buyers with lower credit scores by providing them information on loan options while recommending suitable cars, and the probability of getting approved for the loans. This product has the potential to greatly improve customer convenience while expanding the market share and valuation of both AutoTrader and AutoCapital Canada.

**Press Release:** [https://docs.google.com/document/d/1o_RSNlPj0oy91-Sg8GDQcCElFiWZqYdP4yYvevVRC8A/edit?usp=sharing](https://docs.google.com/document/d/1o_RSNlPj0oy91-Sg8GDQcCElFiWZqYdP4yYvevVRC8A/edit?usp=sharing)

## Updated User Journey Map & User Process




1. A consumer, likely with a low credit score, goes online to find potentially cheap, used cars nearby. Many of these individuals will end up on AutoTrader.

2. These users will then press on the “TraderAuto+” tab.
3. A brief description of what TraderAuto+ does will be displayed. If they decide to use this feature, they press continue and proceed.
4. The user will be prompted to report specific information including their credit score, what down payment they can afford, and their monthly budget. They have the option to become an Advanced User, which would require them to input more financial information, but can provide a more realistic loan calculation.
5. They will then be provided the opportunity to filter out the type of car they are browsing for.
6. A list of cars with possible loans from AutoCapital Canada and the likelihood of being approved for these loans will be generated.
7. If the user thinks that a loan result fits their requirement, they can call AutoCapital Canada and show them the result generated. AutoCapital Canada will then verify whether the loan can be finalised.
8. Finally, the customer can also book and attend a test drive. They then will be directed to the finance office where a deal is struck, based on the finalised loan offer with AutoCapital Canada. Given they are purchasing a used-car, there will generally be no upselling.

## Assumptions

The following assumptions are made for our project:

- We have established a partnership with both AutoTrader and AutoCapital Canada.
- AutoTrader does not have an incentive to work with multiple or other subprime lenders; there will be no conflict of interest.
- Our customer base consists of only people shopping on Autotrader, hence we are mostly targeting people looking for used cars and needing subprime loans.
- All used cars listed on AutoTrader are available at dealerships.
- Senso’s API has entries for used cars; ie. condition, kilometers driven, etc. that allows us to provide accurate loan estimates on these cars.

*Note:*
We have met with AutoCapital Canada to learn how they calculate loan offers. We made sure that our loan calculator has taken into account what AutoCapital

Canada's official loan calculation uses, but we have simplified the calculation for our MVP.

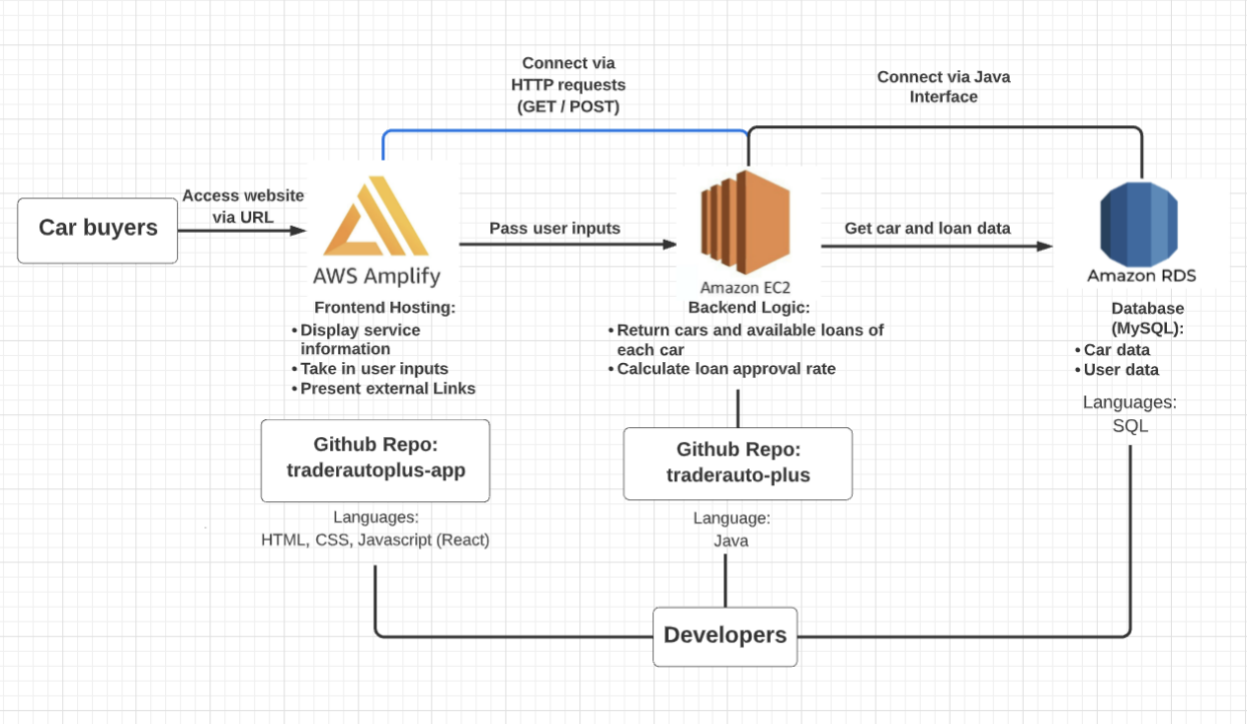
Loan Approval Calculation Table (based on suggestions from AutoCapital Canada): https://drive.google.com/file/d/1qr5RzeDyxVwNudx000ah5GJ9-wtdez1O/view?usp=sharing

## Project Specifications

- Provide users a list of loans for specific cars, and the likelihood of getting approval for each loan based on monthly income, credit score, employment status, homeowner status, monthly budget, car price and a variety of other factors.
    - Loans options are calculated based on the user's credit score, monthly budget, down payment using Senso Rate API
    - The basic loan approval rate are calculated based on senso prepayment score, user's credit score, monthly budget, downpayment,
    - A more advanced loan approval rate takes into account the user's monthly income, monthly debt as well as homeowner and employment status
- Allow users to select car preferences so that the loans are only calculated for the cars that they are interested in buying.

## Tech Framework

## Links to Repositories, Web




## App and Class Diagram

**Backend Repo:** https://github.com/autotrader-plus/traderauto-plus

**Front-end Repo:** https://github.com/autotrader-plus/traderautoplus-app

**Web app:** https://main.d2c0mw2w964w11.amplifyapp.com/

**Class Diagram:**
https://drive.google.com/file/d/1UE2gQkopkWQvWYY3Hs8xKkpvmlwoSWIC/view?usp=sharing

## Clean Architecture

- As seen in our packaging system, we've separated all our code into layers and ensured that outer layer code does not depend on inner layer code. This is further explained in our packaging section and dependency inversion section. For example, BasicLoansCreator, which is a use case, depends on CarList, which is our entity, but CarList does not depend on it, or other use cases.
- Furthermore, we've implemented interfaces to act as buffers between all our different layers. This helps to avoid high coupling between classes, and to ensure that use cases do not depend on controllers, presenters and gateways. For example, ConnectAutoTraderDB and ConnectSensoRateAPI are both

gateways to the outer entity. To allow our use cases classes like BasicLoansCreator and ReturnCarInformation to use outer entities like SensoAPI or Database, we have created interfaces called AutoTraderDBInterface and SensoAPIInterface that our gateways classes (ConnectSensoRateAPI & ConnectAutoTraderDB) implement. By creating these interfaces, our use cases can depend on the interfaces, which are in the use case layer, rather than relying on the gateways, which would violate clean architecture.

- The data flow of our program as a whole also adheres to the clean architecture principles. First of all, we have a controller class ServerMainEndpointHandler that receives information from the front end, and parses that information to be passed into our use cases, LoanResponseConstructor through the LoanInfoInterface. Finally, the data is then formatted through our presenter class HttpResponseConstructor and later sent back to the front end. Essentially, the data passes through the layer one by one without  jumping through layers.

## SOLID

- **Single Responsibility Principle**
  We've ensured that every class has one (and only one) job. For example, when rushing to finish our phase 0 design, we had one class that dealt with returning every piece of information from the database which led to unnecessarily long and disorganized methods and classes. Eventually, we split this up into the 4 classes currently in our "database_info_manipulation" package, such that each class only does one job, which is to either return user information, car information, add user or authenticate user. Similarly, in our "http_parser_constructor", we have two classes: one of them functions as a parser for http request and the other one functions as a http request constructor. Even though the two classes deal with creating and parsing http requests/responses, we decide that it would be best to make them into separate classes, because they ultimately have different functionalities.

- **Open-Closed Principle**
  We've segmented our code very granularly, which means whenever we finish a small piece of code, we will never need to come back and fix it. For example, we have a class solely dedicated to formatting the response body of our HTTP requests. This class will not be modified, but if we needed a second way to parse and retrieve the information it would be easy to create a second class. Similarly, our LoanInfoInterface also demonstrates the open-closed principle in the sense that if in the future, we need a different way of calculating loans, we can directly create a new class that contains the new loan calculations

method and also implement the LoanInfoInterface. This way, other higher level classes can just call the LoanInfoInterface to access different loan calculation methods that we have. This allows us to avoid the need to go in to edit our existing loan calculation class to accommodate different ways of loan calculations.

- **Liskov Substitution Principle**
  There is no need for parent and child classes in our project.

- **Interface Segregation Principle**
  All classes that implement any one of our interfaces implement every method that is defined in the interfaces. We have ensured that our interfaces contain the minimum number of methods required for the interfaces to work. By doing that, we ensure that all the classes that implement interfaces depend on all the methods defined.

- **Dependency Inversion Principle**
  We have discussed earlier how we have created interfaces between use cases and gateways so that our use cases do not depend on higher level classes Furthermore, we have also created an interface (LoanInfoInterface, which is also acting as our Input Boundary) between the controller (ServerMainEndpointHandler) and the use cases (LoanResponseConstructor), so that both high level or low level classes depends on the interface abstraction. This avoids high coupling between classes in different layers, and allows changes to be made more easily in the future. We also adhere to this principle when we need lower level classes to access high level classes/entities, and this is elaborated in the Dependency Injection section in Design Patterns.

## Design Patterns

- **Facade Design Pattern**
  When the server receives a POST request, there are a lot of things that need to be done such as calling the database, pinging the Senso API for the loan information, parsing and returning the request body. So, instead of having multiple endpoints that do all these tasks separately, which would increase the coupling between classes, we have decided to create a facade with our ServerMainEndpointHandler, which delegates all these separate jobs to other classes. This facade serves as an entry point for the clients without the clients needing to worry about which individual classes to call.

- **Factory Design Pattern**

We have used the factory design pattern in our UserFactory class so that we can abstract the creation process of User objects. This is because we have two constructors for User class depending on the amount of information we have from the user, so we need to use "if...else..." statements to handle the cases. By abstracting this process using a factory, it helps to make any class that needs to instantiate a User object much cleaner instead of writing repetitive if...else... statements in the class.

- **Dependency Injection**
  Our LoanInfoInterface adheres to Dependency Injection, in the sense that our use case class LoanResponseConstructor implements the interface, which declares the method to supply the dependency. Essentially, our injector (LoanResponseConstructor) uses the interface to supply information to our clients. Furthermore, when the inner layer classes like BasicLoans and LoansScoreCalculator need to access the classes or entities in the outer layer like Senso API, we use dependency injection to achieve that. Essentially, we provide an interface, SensoAPIInterface in the use case layer that our gateway classes like ConnectSensoRateAPI implements. This allows our use cases to depend on the SensoAPIInterface (which is in the use case layer) to access the Senso API, instead of having them depending on classes in the Interface Adapter layer.

- **Strategy Design Pattern**
  Our project needs access to both Senso API endpoints. However, we realise that while both endpoints require different information and depend on different classes, they have some very similar code, especially those that create a POST request to the endpoint. So, we have two "strategies" of doing things, depending on what we need, but we do not have to have huge chunk of repeating codes to avoid code smells. Hence, we have created an interface "SensoAPIInterface" that acts as a delegate to call different strategies (different endpoints) depending on what is needed.

## Packaging Strategy

We've organized our packages by functionality. Here's how it works:

1. **backend_logic** contains classes that tabulate relevant loans and such, as well as our entities classes like User and Car;
2. **connect_api_db** contains classes that retrieve information from Senso API or our database;
3. **database_info_manipulation** contains classes that parse information from database and turn it into data that we can use;
4. **exceptions** contains exception classes that we defined for our program;

5. **http_parser_constructor** contains classes that parse http requests and construct http responses;
6. **server_setup** contains classes that set up CORS application and endpoints for our server, as well as those that handle HTTP requests sent to our EC2 server;

We find that this packaging strategy works well because we can quickly identify where each Java class is, which makes it easier for us to locate files and structure our import statements.

## Github Features

We have made use of the Projects feature in Github to create a Kanban board to track our progress throughout the semester.We also used the pull requests and issues on Github to flag any issues that need to be fixed and allow peer reviews before it is merged to the main branch. We have restricted merging so that we need at least 2 peer reviews for each pull request, otherwise the pull request cannot be merged.

We have used predefined templates ".gitmessage" and "pull_request_template.md" for our commits and pull requests. Every time someone needs to make a commit or pull request, the respective template will automatically show up and we just need to fill out the appropriate information relating to the individual commits/pull requests.

We have also finalized our README.md that clearly communicates what the project is about, how to run the program, and other basic information about our program. Lastly, we have used Github Actions to automatically build Gradle projects that are triggered by all pull requests. We also integrated "CodeFactor" into our Github that will do a basic code review like checking code style, code repetition, etc. As the integration will not be able to catch all errors, we only use it as our "sanity check tool" once we have manually reviewed all our codes.

## Testing

We have a test class for every class, including exceptions classes and classes that configure spring boot server. Furthermore, we have achieved 100% classes and methods coverage from our test cases. The test class for exceptions class tests whether the exception can be thrown and whether the correct error message that we defined will return once the specified methods are called. For the test cases for classes that set up the spring boot server, they test whether the server can be started and work. While we have test classes for those that configure spring boot servers or set up endpoints, we have also manually tested our code and server on localhost every time before we merge a pull request.

We have also used Mockito to support unit testing for classes that directly access the database (those in the database_info_manipulation packages). Mockito allows us to mock the connection with the database, so we can do actual unit testing that makes sure that we are returning information in the correct format without actually connecting to the database.

## Frontend-Specific Information

- **Naming Convention**
    - For the naming convention for our JavaScript code, we have followed the guideline stated in this webpage, and we have decided to keep it similar to our back end code.
        - For our function names, we have camelCase;
        - For our component, class and file names, we have use PascalCase.

- **Single Responsibility Principle**
    - As each function is only responsible for rendering one (and only one) specified section of a webpage, this demonstrates the single responsibility principle. Each js file is only responsible for one page of a website and only contains behaviour functions such as handleClick, and handleChange, etc for elements of that page.
    - CSS style sheets are also separate for each specified component and webpage, for clarity, easier management and edits later on.

- **Inheritance/Composition**
    - For inheritance, React does not have inheritance, instead it has composition. We tried applying it in such a way as dividing the rendering of a webpage into multiple parts, multiple functions and calling the smaller one inside the larger one inside the actual class responsible for that page. Like how Car.js is responsible for rendering the browse page and it calls CarFilter and Caritem functions for specific elements/components. Cars class and Signup class are both called in UserInputs class so the inputs can be collected in one (sorta) parent class as state variables to make the post request to the back end with needed inputs information.

- **Packaging**
    - We packaged our front-end code by layers. We identified three primary packages.
        - Firstly, we have all incoming information under the “client info” as this is the package that receives information from the client.

        - We then had a package to parse and display this information under web-design. This class was split up into numerous components that each contained one specific part of the webpage.
        - Finally, we had a client input page that recorded and sent client input to our backend.
        - We have decided to put the test files together in a folder called “test” within individual packages instead of separating all the test files out into another folder.

- **Design Patterns**
    - Given most of our main code was in our backend, our frontend primarily dealt with receiving and displaying information. Hence we personally did not feel the need to implement any design patterns outside of what already exists when using React, HTML and CSS.
    - For example, React itself organizes functions using the Chain of Responsibility design pattern and we’ve also implemented the State Design Pattern to output information in a clean manner when switching between car options.

- **Testing**
    - Firstly, we tested rendering, just to ensure that everything renders correctly regardless of the information that was sent in from the client or the inputs the user makes.
    - Secondly, we tested functionality. This can be split up into two parts for each package we had on our front end.
        - Firstly we tested whether or not the information we received from our backend was consistent and worked with the code we had.
        - Then we tested whether or not the different aspects of our web design functioned correctly ie. whether or not dropdowns worked and if links sent to the correct page.
    - We have omitted the testing class for Button.js since that class because it is only used for rendering button objects with specified look associated with Button.css. We have manually tested the style of the buttons, hence the functionality of the class, by running the website.
    - We have also omitted the testing class for Signup.js, because it is a helper class that combines Signup_Step1, Signup_Step2, and Signup_Step3 together. By testing Signup_Step1, Signup_Step2, and Signup_Step3 as well as UserInputs individually, we have made sure that the entire sign up process works. We have also manually tested the sign up process on the local host.

## Problems Encountered

- A big issue we struggled with at the beginning of the project was connecting every different layer to one another. We had to use external APIs such as JDBC and edit gradle/POM files which none of us had any experience using/doing previously. We also didn't have a very good understanding of the interface segregation principle which led to terrible architecture that took a very long time to eventually fix.
- Another big issue that we faced as a group was figuring out what Amazon EC2 is, and how we can build a server on it. It took us a long time to figure out how to set up an EC2 server, how security groups on AWS works, and how we can keep our java server running all the time on the EC2 instance. We eventually figured everything out, and the process of deploying to EC2 and accessing EC2 becomes easier.
- One smaller issue was the trouble we had with environment variables. Since some people use macOS, while others use Windows and Linux, it was hard for all of us to properly set up the env variables. As you will read in the Individual Contributions & Credit section, some of us had trouble testing our code that requires the database due to issues with not having the environment variables properly set up. This caused a lot of frustration and it took us a very long time to realize that there wasn't an error with our code, but rather accessing the database.

## Accessibility Report

For our MVP, we have implemented several features in our web app that align with some of the universal design principles:

- Low Physical Effort
    - We have implemented drop down menus so that users can simply pick from existing options instead of having to type their options.
    - Using our website is pretty intuitive, and the form does not take too long to complete, so it will not be too physically demanding for the users. We've also allowed users to choose whether or not they'd like to input more detailed information depending on whether or not they have the time to do so.

- Perceptible Information
    - The text and font size of our website are of sufficient size so that everything is perceptible. Also, most of our web pages use white on dark color backgrounds, which provides a nice contrast for everything to be easily readable.

To improve our web apps accessibility to align more with the universal design principles, we would include:

- Equitable Use
    - We will implement text-to-speech features for our website to accommodate people with reading needs.
    - When the user first lands on the site, we plan to include a demo video to walk the user through the features available on our web app so that every user, regardless of their technical abilities, can still use our website with ease.

- Flexibility in Use
    - We plan to implement a “Help” option in the navigation to provide more information of the inputs of the form and the step-by-step process of how to use our web app.
    - We also plan to make our web app more responsive across devices of different sizes, so people can have more choices with how they access our application.

- Simple and Intuitive Use
    - We plan to include optional popup windows to detail definitions of each user input to ensure the user understands what information they need to provide.
    - Given our webpage is directly on a used-car selling website, it’s very easy to locate and head to.
    - We plan to include more detailed error messages when user input or actions are invalid.

- Perceptible Information
    - Overall, I think we have done everything pretty well to make sure that most of our information is perceptible. However, in the future, we plan to improve our HTML code to make sure that it fully implements and supports ARIA features, so that the web content can be made more accessible and perceptible in screen readers and text-to-speech applications.

- Tolerance for Error
    - We have implemented fail safe features for when the user inputs incorrect values, such as when users accidentally input a letter in responses that require only numerical values, the form will automatically delete/ignore the letter.
    - The next steps would be to create limits for how much a user can input for certain fields and add this capability to postal codes.

- Low Physical Effort
    - Overall, I think that our web app does not require a huge amount of physical effort. However, we could implement sliding scales for inputs that take in numeral values, so users do not need to physically type and they can just use the sliding scale to input values.

- Size and Space for Approach and Use
    - Right now, our web app is only responsive on certain devices. So, in the future, we would want to make sure that our web app is responsive across all devices, so that people who are using any kind of device can access our application with ease regardless of their posture and mobility.

Our program would be marketed towards car buyers with suboptimal credit scores. TraderAuto+ is intended to be used by car buyers who have lower credit scores as its purpose is to calculate possible loans along with their likelihood of approval for the user in order to help them gain a better understanding of their financial status in the context of purchasing a car.

As TraderAuto+ is a web application and we currently don't have any features that help the visually impaired interact with our application, it's less likely this demographic of people would use our web app. As we have a lack of customization options regarding the visual components of our website, such as background colour and text size, TraderAuto+ may also be less used among people who have dyslexia. Additionally, as our website presents all information in English, those who don't understand English would be less likely to use our product.

## Individual Contributions & Credits

**Sophie Sun:**

- My contribution is focused on the frontend. I set up the website and connected it to AWS Amplify and coded in Javascript and CSS along with using framework React to create the web application. I set up functions that send https requests to the end point of our web server and save + parse the response to be displayed on the website.
- Two significant pull-requests:
    - https://github.com/autotrader-plus/traderautoplus-app/pull/13
      This pull request is writing Loan.js which displays the car details page when users clicks on any filtered car image on the browse page. I learned how to do this by using the React Router library and passing responses received in different fetch requests as component props in

Route components. This eliminates the idea of hard coding every single car detail page for every car in our database.
    - https://github.com/autotrader-plus/traderautoplus-app/pull/21 This pull request is made during the final stage of the project where I fixed bugs detected by Elizabeth who was doing front end testing for my codes and cleaned up, restructured and also documented my codes.
- *When before I was working alone on frontend I did not use Github properly and saved all of my progress in one master branch, which is completely wrong and messy. That branch is now deleted and as a team we now work collaboratively on it and we maintain a much more organized Git repo, properly using different branches to work on different parts of the code and writing meaningful commit messages + pull request comments. But also as a result, my major commits and pulls on the development of the website before are not saved.

**Ameen Parthab:**

- I had two main contributions. Firstly, I set up the connection to the database and made most of the code to connect to it to retrieve and send data. Secondly, I worked on the frontend to package and clean up code along with improving the website design.
- Two significant pull-requests:
    - https://github.com/autotrader-plus/traderauto-plus/pull/9. This pull-request is the construction of the package that manipulates the data retrieved from the database. It deploys clean architecture, was thoroughly tested (primarily by Elizabeth), and efficiently connected to the database, which was one of the first steps we needed to do for this project. I'm also proud of the detail that went into each commit message.
    - https://github.com/autotrader-plus/traderautoplus-app/pull/11. As I began transitioning to the frontend, my PRs became more granular, but I'm proud of this PR because I went all out, learning HTML and CSS over reading week to begin working on the frontend. Furthermore, the PR also required hours of studying existing code which allowed me to build on it, repackage it, and implement clean architecture overall.

**Daniel Xu:**

- I've been working on the backend business logic, writing tests, and handling the responses from Database, LoanTableCalculator, and SensoScore/Rate APIs.
- Two significant pull requests that I've made for this project are:
    - https://github.com/autotrader-plus/traderauto-plus/pull/10. This pull-request is when I finished the skeleton version of our backend business logic. It builds the foundation for clean architecture and SOLID for

future updates, and provides our group with an idea of how our phase 2 backend code is going to be. The test file contained the necessary code for the completion of the individual backend tests (UserTest, CarTest, LoanTest, etc) in the future PRs. This PR is also when we first received a return info that incorporates both the database and SensoAPI.
    - https://github.com/autotrader-plus/traderauto-plus/pull/73. Although JiaHao was the one who made this pull-request, this was mainly due to an issue I had with environment variables so I couldn't test my code locally. Hence, I had JiaHao test and make the PR incorporating his packaging changes and my basic version of the backend code for phase 2 submission. In this PR, the new backend code processes the return info from the 2 Senso APIs and retrieves the first month installment and SensoScore, and incorporates the 2 with the LoanTable into a final score, which is the loan approval likelihood score with multiple loans. The final score is essentially the packaged version of the info that we will ultimately be displaying in the frontend website, and this PR is the proof of concept for the backend logic for phase 2 (final) submission.

**Jia Hao Choo:**

- I have been working on setting up new endpoints for our web server, as well as setting up Github Action for our project. Also, I have been doing refactoring and testing for backend and frontend code.
- Two significant pull requests that I have made throughout the term are:
    - https://github.com/autotrader-plus/traderauto-plus/pull/12, where I set up the HTTP server and connect the front end with the backend. I think that this is a significant contribution, because making sure that everything is connected provides a foundation for the team to build new features and make sure that we are able to work with a properly functioning web application.
    - https://github.com/autotrader-plus/traderauto-plus/pull/73, where I did major refactoring and test updates across our backend code. I made sure that all test cases cover 100% of the methods and classes, which is very important to ensure that each part of our program can work without errors. I also did some refactoring for this PR to make sure to reduce code smell and make our code more easily understandable.

**Elizabeth Li:**

- I've been working on testing the backend and frontend code, user authentication, as well as creating the loan table calculator
- Two significant pull requests:
    - https://github.com/autotrader-plus/traderauto-plus/pull/72, where I created the loan approval calculator which calculated a user's loan approval score using their credit score, PTI, DTI, employment and

homeowner status, and SensoScore as well as the testing. This score is important in helping generate appropriate loans for the user which directly serves the main output of our web app.
    - https://github.com/autotrader-plus/traderauto-plus/pull/60, where I worked with Jia Hao to refactor most of the code in the backend making sure the code was properly documented, naming conventions were respected, and removing the use of static variables and methods where we didn't need them. This was significant because it helped clean up our code to make it easier for members of the team to understand the code and keep our code consistent.

Credit to the following organizations for providing us with guidance and tools for our project:

- Senso.AI for providing their Senso API to calculate loans and prepayment score.
- AutoCapital Canada for providing us guidance on some important factors to take into account when calculating loan approval rates.

## Future Improvements (Icebox Tasks)

- Right now, we are passing user information (ie. username and password) directly into the server backend. This is not secure, and it might be worth looking into encryption or OAuth.
- Right now, when users sign in they are brought straight to the car filter page. However, for our MVP, we have not set up a way to retrieve existing user information to the front-end, after they fill out their information, they are automatically brought to the Browse Cars page if the authentication is successful. So, in the future, we will need to find some ways to get user information from our database.
- We may adjust our loan calculation so more weight is put on credit score and monthly budget.
- We may need to deal with the case where the user has no monthly income but is in debt, because right now, if the user doesn't input their monthly income, they are considered "Basic User", so we will not take into account any extra information. This is not reflective of the real world, because a user can have no monthly income but be in debt, which may affect their loan approval result.
- We may consider making our domain, though it might not be necessary if we are just a web app plug in to dealership websites.
- Make our website responsive to improve accessibility on a range of devices.
- Provide opportunities through embedded links to directly connect with AutoCapital Canada to move forward with the purchase journey.