CS3354 Software Engineering
Final Project Deliverable 2

Budget Buddy

Safura Shafi
Salwa Sarwer
Mateilda Chahine
Kunuth Siddiqui
Vimal Sebastian
Terrence Li
Manab Sthapit
Eduardo Hernandez

1. **[5 POINTS]** Well described delegation of tasks, i.e. who did what in the project. Now that your project is complete, you are required to submit the delegation of tasks from beginning of the project until the end. Please make sure to fairly distribute tasks in the team and remember that in the end of the semester, each member of a team will receive the same grade. See grading policy below for more detail. If no/poor contribution by a member, please specify clearly so that we can grade each student fairly.

   Terrence Li - contributed to the design of the nonfunctional requirements, designed sequence diagrams and use cases, addressed feedback to designs, participated in the management of source control on GitHub, reviewed and edited deliverable drafts, collected and organized IEEE references.

   Vimal Sebastian – participated in managing and adding content to the GitHub, leaded how each screen on the User Interface should present, created the system architectural design pattern, made the Cost Effort Pricing Estimation using the Function Point (FP) method, created the basic database schema of how the app should be based on, estimated the costs of hardware, software products, and personnel, programmed the unit test source code in Java and JUnit.

   Eduardo Hernandez – tasked with researching the typical functionality of budgeting apps to determine what functionality our application should take on, and designing the User Interface accordingly

   Kunuth Siddiqui – designed the class diagram, created and worked on the presentation.

   Mateilda Chahine – designed Gantt chart, used the chart to plan out project scheduling timeline incorporating agile/scrum techniques.

   Manab Sthapit – participated in management of GitHub, created and submitted project_scope file to GitHub, conducted thorough comparison of our project with other similar budgeting applications, wrote conclusion to wrap up our findings/experience working on this project.

   Salwa Sarwer – designed the sequence diagrams, created and worked on the game User Interface, created the logo

   Safura Shafi – designed diagrams along with Kunuth and Salwa and worked on creating and maintaining the presentation.

2. [10 POINTS] Everything required and already submitted in Final Project Deliverable 1. Please specify this part as "Project Deliverable 1 content".

<u>Project Deliverable 1 content:</u>

1. [5 POINTS] Please attach here the Final Project draft description (that contains the instructor feedback). It is ok to include a picture of the original document. Address the feedback provided for your proposal by listing what you did / plan to do to comply with those proposed changes and or requests for additions to your project.

Below is the Final Project Draft:

I. Title of Project: Budget Buddy ™

II. Group members (first name and last name):

- Safura Shafi
- Salwa Sarwer
- Kunuth Siddiqui
- Mateilda Chahine
- Terrence Li
- Manab Sthapit
- Vimal Sebastian
- Eduardo Hernandez

III. What you will be doing: Creating an app that helps people budget their savings.

IV. A detailed description of your motivation (why you chose to do this particular project, where you expect your design to be used in real life):

We chose to do this particular project because as college students, managing our money can be overwhelming with school itself. With something automated we can have less stress in our day-to-day lives, knowing there's one less thing we must manage. With this tool we can expect it to help with balancing the funds we want to go to concerts, shop, eat with friends and our school funds such as loans, textbooks, transportation, and rent.

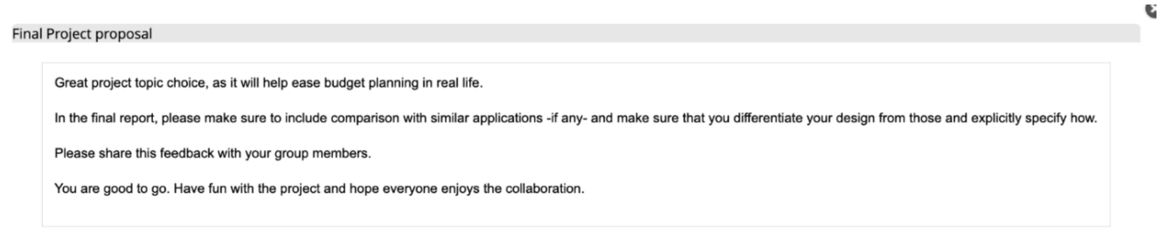
Although we are speaking from a college student's perspective, we expect our design to be used in the lives of any adult. A budgeting app is something that people outside of college can utilize as well. We plan to include features that will be inclusive of people with different lifestyles, such as someone who will be beginning a professional career, a student, or parents balancing their own funds with what they need to spend on their family.

V. Lists of tasks:

a. **Vimal –** I will be working on the GitHub repository, generating the working ideas of the functions involved in the app, and architecting how different processes will interact.
b. **Safura –** I will be working on making the diagrams and models to represent the functions of our application.
c. **Kunuth –** I will be working on creating the models that represent the functions of our application.
d. **Mateilda –** I will be working on leading the group, submitting all assignments, and keeping documentation on everyone's participation.
e. **Terrence –** I will manage source control and uploads to GitHub, designing and overseeing functional requirements.
f. **Manab** – I will be responsible for the upkeep and regular updates of our GitHub repository, in addition to contributing to the implementation of ideas for our application.
g. **Eduardo –** I will be working on documenting group research findings and progress reports of our application.
h. **Salwa –** I will be working on the diagrams and models to represent the functions of our application.

VI. Are you interested in writing a scholar paper in the end (no extra grade advantage): No

Instructor Feedback:


Final Project proposal

Great project topic choice, as it will help ease budget planning in real life.

In the final report, please make sure to include comparison with similar applications -if any- and make sure that you differentiate your design from those and explicitly specify how.

Please share this feedback with your group members.

You are good to go. Have fun with the project and hope everyone enjoys the collaboration.

To comply with the feedback provided, we will be incorporating a comparison with a similar budgeting application and explaining how our design differs from theirs.

Comparison with NerdWallet:
We conducted a comparative analysis with NerdWallet, a prominent budgeting application/resource. NerdWallet offers features like transaction tracking, budget planning, and goal setting, similar to our app. However, NerdWallet focuses on

tracking cash flow and providing budget guidelines, while our app works to enhance users' financial education through gamification.

Differentiation of Design:
Our app design offers a unique feature – gamification of financial education. While traditional budgeting apps like NerdWallet primarily focus on tracking expenses and providing budgeting tools, our app stands out by incorporating interactive elements such as articles, videos, and quizzes to educate users on personal finance topics. This gamification component incentivizes user engagement and sets our app apart from others in the market.

2. [10 POINTS] Setting up a Github repository. Please use your utdallas email accounts only for each group member.

Repository link: https://github.com/spevenexe/3354-MoneyBuddies

1.1. Each team member should create a GitHub account if you don't already have one.

1.2. Create a GitHub repository named 3354-teamName. (whatever your team name will be).

1.3. Add all team members, and the TA as collaborators. Our TA has already posted her GitHub info in EL: TA GitHub id: TA email:

1.4. Make the first commit to the repository (i.e., a README file with [team name] as its content).

1.5. Make another commit including a pdf/txt/doc file named "project_scope". If you choose a predefined topic (one of the 4 topics described in the "Project Topic Ideas" section of this document), the contents of the file should be identical to the corresponding project in this section. If you choose other topics, the contents should follow a similar structure.

1.6. Keep all your project-related files in your repository as we will check them. Include the URL of your team project repository into your project deliverable 1 report.

Important Note:

- Tasks 1.3 - 1.5 should be performed by different team members. We will check the commit history for these activities.
- Do not include credentials (e.g., UTD ID) in the repository.
- Only commits performed before the deadline will be considered. Do not forget to push your changes after you have done the work!

3. [5 POINTS] Delegation of tasks: Who is doing what. If no contribution, please specify as it will help us grade each group member fairly.

a. **Vimal –** I will be working on the GitHub repository, generating the working ideas of the functions involved in the app, and conducting the architectural design of the project.
b. **Safura –** I will be working on making the diagrams and models to represent the functions of our application.
c. **Kunuth –** I will be working on creating the models that represent the functions of our application.
d. **Mateilda –** I will be working on leading the group, functional requirements submitting all assignments, and keeping documentation on everyone's participation.
e. **Terrence –** I will manage source control and uploads to GitHub, designing and overseeing functional and nonfunctional requirements.
f. **Manab** – I will be responsible for the upkeep and regular updates of our GitHub repository, in addition to contributing to the implementation of ideas for our application (1.5 and instructor feedback section).
g. **Eduardo –** I will be working on documenting group research findings and progress reports of our application.
h. **Salwa –** I will be working on the diagrams and models to represent the functions of our application.

4. [5 POINTS] Which software process model is employed in the project and why.

Based on the scope of our application, we chose to employ an Agile process model with elements of Scrum. The development process benefits from flexibility and adaptability which could be necessary if the user base has changing needs. It also allows our team to deliver working increments per every sprint, essentially allowing us to push new updates to keep up with user demands effectively. The Scrum elements allow us to identify and address issues quickly via the frequent feedback loops. It also allows for continuous improvement to the application through the feedback loops. Overall, Agile methods allow us to effectively incorporate the users' evolving needs and demands in our development.

5. [15 POINTS] Software Requirements including

5.a.) [5 POINTS] Functional requirements. To simplify your design, please keep your functional requirements in the range minimum 5 (five) to maximum 7 (seven). (Ch 4)

1. Users

    1.1 Users should be able to log into the system

    1.2 Users should be able to sync bank information to account

    1.3 Users should be able to set personal budget

    1.4 Users should be able to set personal goals

    1.5 Users should be able to play financial education game

    1.6 Users should be able to track transactions, budget progress, and goals

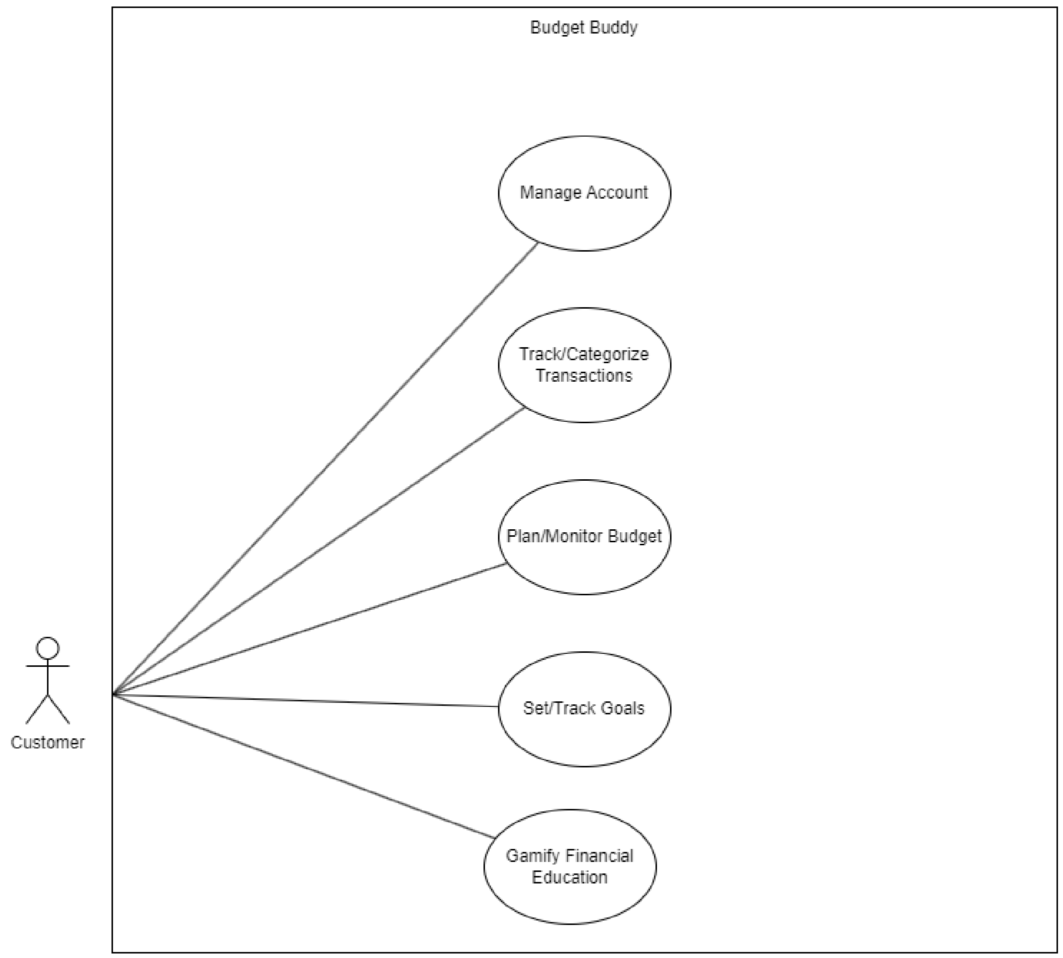
5.b.) [10 POINTS] Non-functional requirements (use all non-functional requirement types listed in Figure 4.3 - Ch 4. This means provide one non-functional requirement for each of the leaves of Figure 4.3. You can certainly make assumptions, even make up government/country based rules, requirements to be able to provide one for each. Please explicitly specify if you are considering such assumptions.)

Non-functional Requirements:

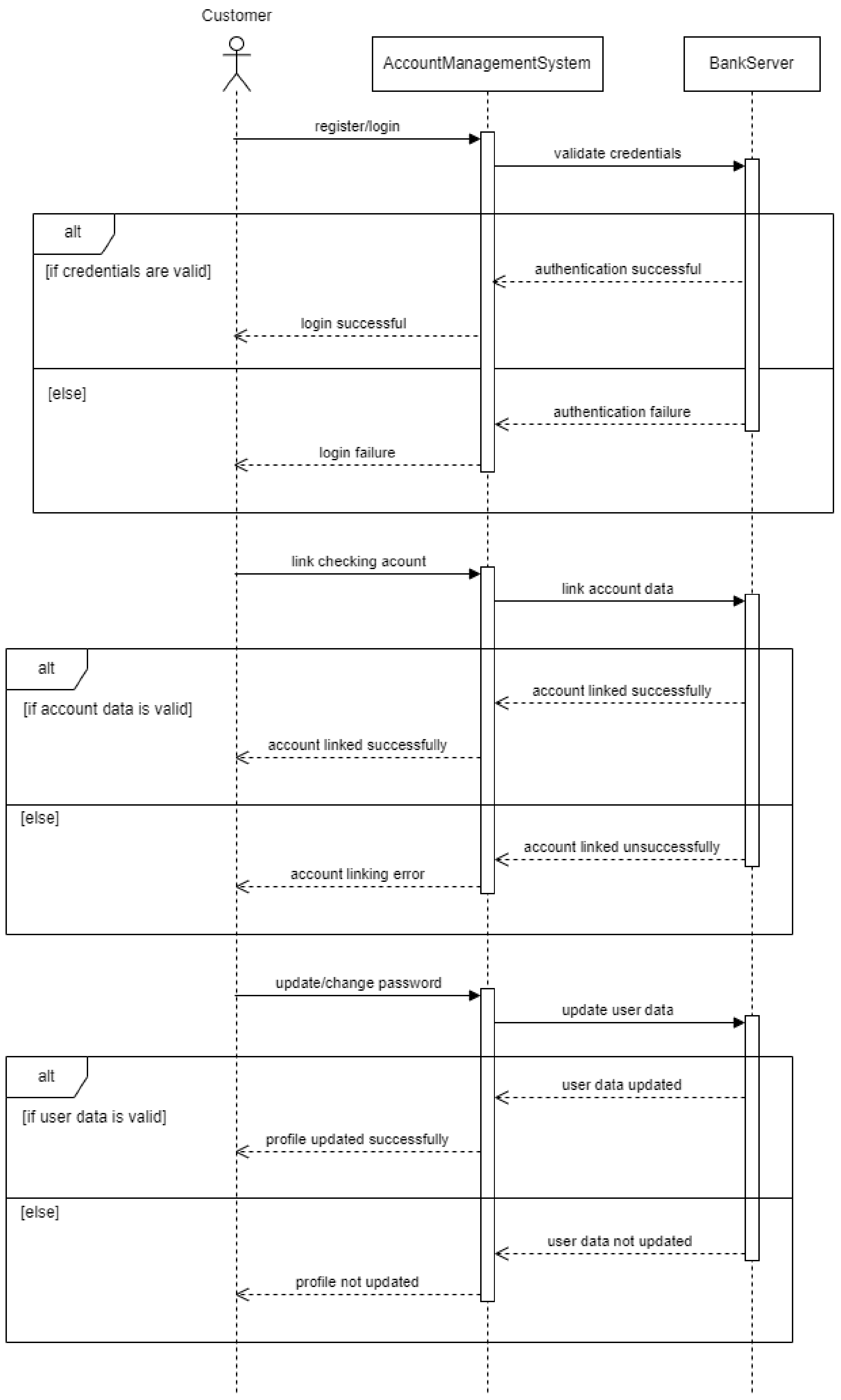
- Product Requirements
    - Usability Requirements – a user shall be able to use all systems after a five-minute tutorial. After the tutorial, the experienced user-error rate shall not exceed 3 per hour of application use.
    - Efficiency Requirements
        - Performance Requirements – the response time between user input (button press, transaction adding/editing, etc.) shall not exceed 30ms (the app should be efficient, so minimizing the delay while giving leeway for sending information across the internet are the factors considered when determining this requirement).
        - Space Requirements – the app shall not exceed 3 Gigabytes in size (the app should be lightweight so as not to be a burden on a computer system).
    - Dependability Requirements – the app shall have <5% of events cause failure. The app shall also maintain a backup of data in the case of data corruption.
    - Security Requirements – users shall only be able to use the app on a device after the user has been verified via email or phone (this app deals with sensitive information. It should make sure the user's identity is verified).

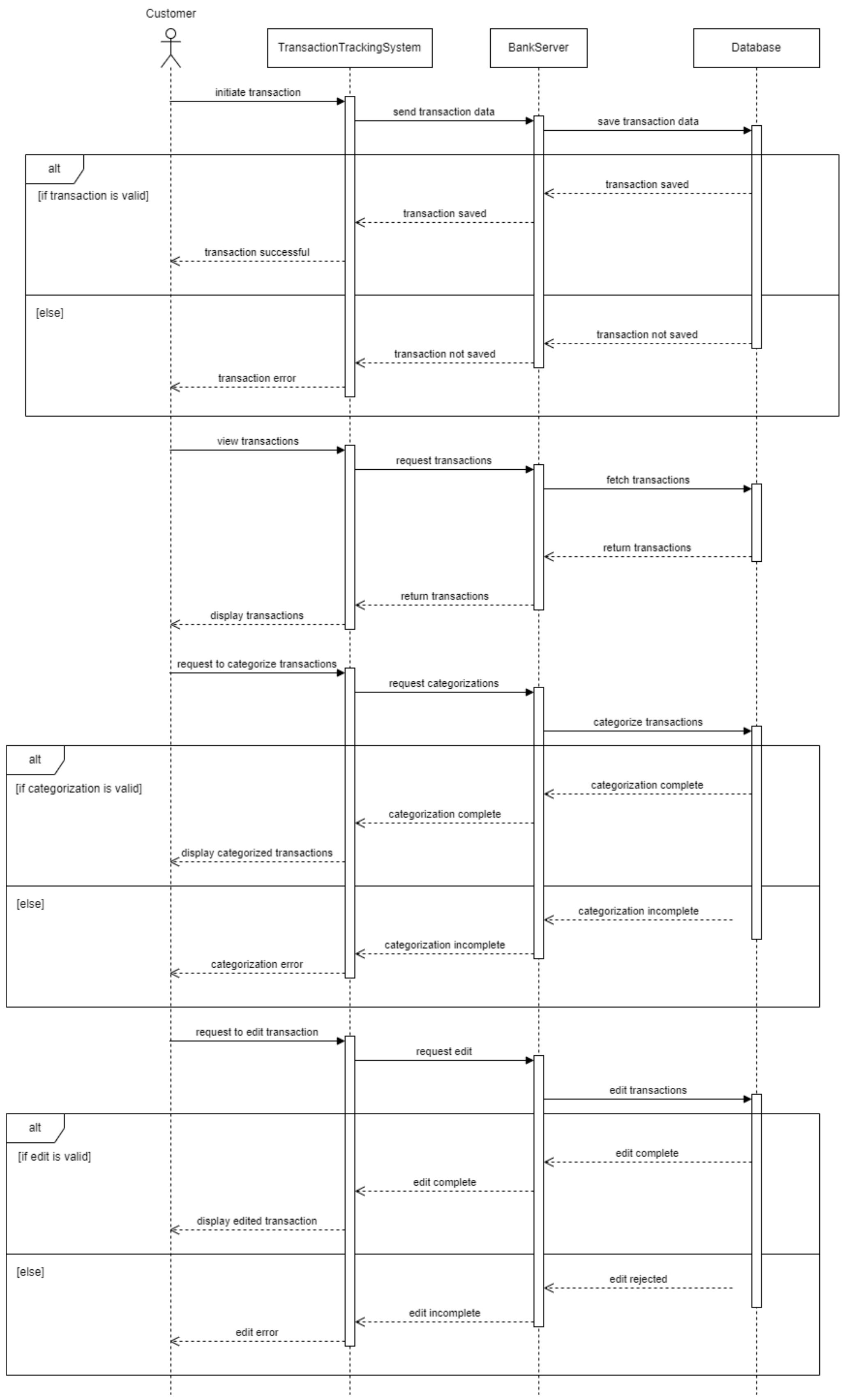
- Organizational Requirements
    - Environmental Requirements – the app shall be runnable from Mac, Windows, Linux, iPhone, and Android devices. If necessary, separate versions of the app shall be to accommodate the different device environments.
    - Operational Requirements – users shall request for a key from the bank that manages their checking account(s) before linking the account to their app (again, to maintain the security of the application, users should link sensitive information on their own terms).
    - Developmental Requirements – the app versions shall be maintained and controlled through a Git repository. Developers will be promptly added and removed from the repository as they enter and leave development (the repository will also allow the app to be rolled back in the case of severe development failures).
- External Requirements
    - Regulatory Requirements – the app shall follow PCI compliance, or payment card industry compliance. That is, the app shall maintain data encryption and other protective security standards when handling transactions to protect user data.
    - Ethical Requirements – the app shall not redistribute any user data. The app shall encrypt and remotely store user data to maintain user privacy.
    - Legislative Requirements (assuming the app is developed and published in America)
        - Accounting Requirements – the app shall not track any information that is deemed illegal to use per U.S. privacy laws.
        - Safety/Security Requirements – the app shall obey financial regulations as set forth by the Federal Trade Commission (FTC).

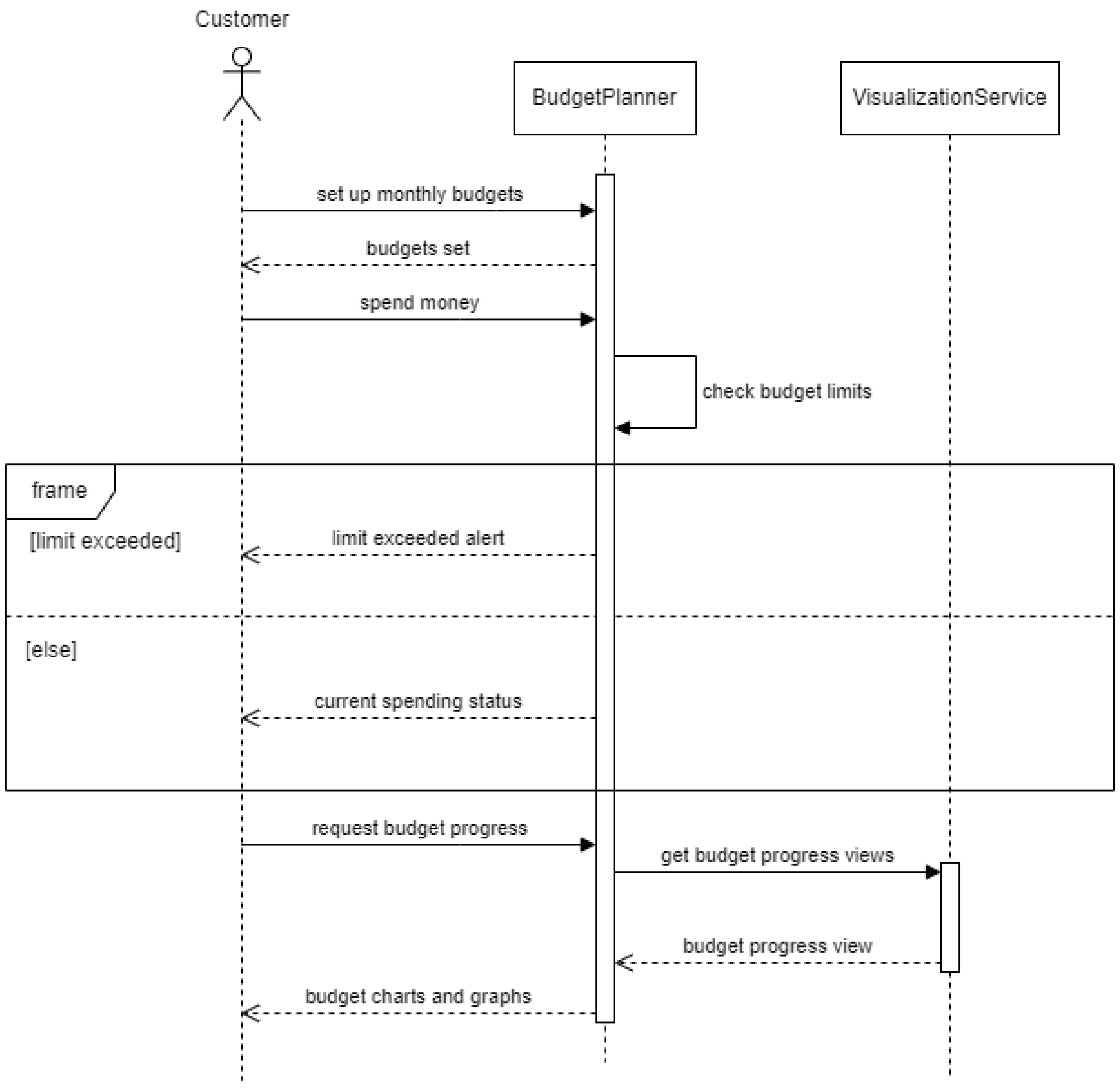
6. [15 POINTS] Use case diagram – Provide a use case diagram (similar to Figure 5.5) for your project. Please note that there can be more than one use case diagrams as your project might be very comprehensive.

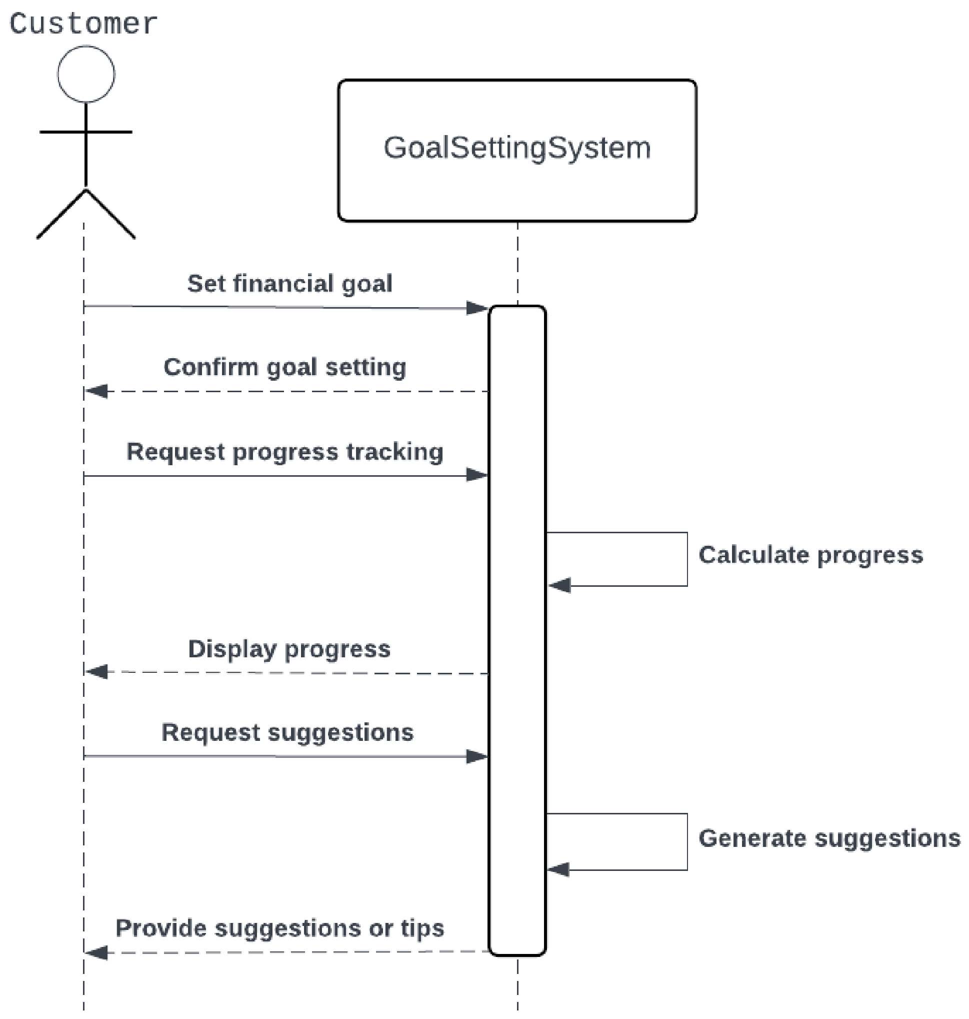


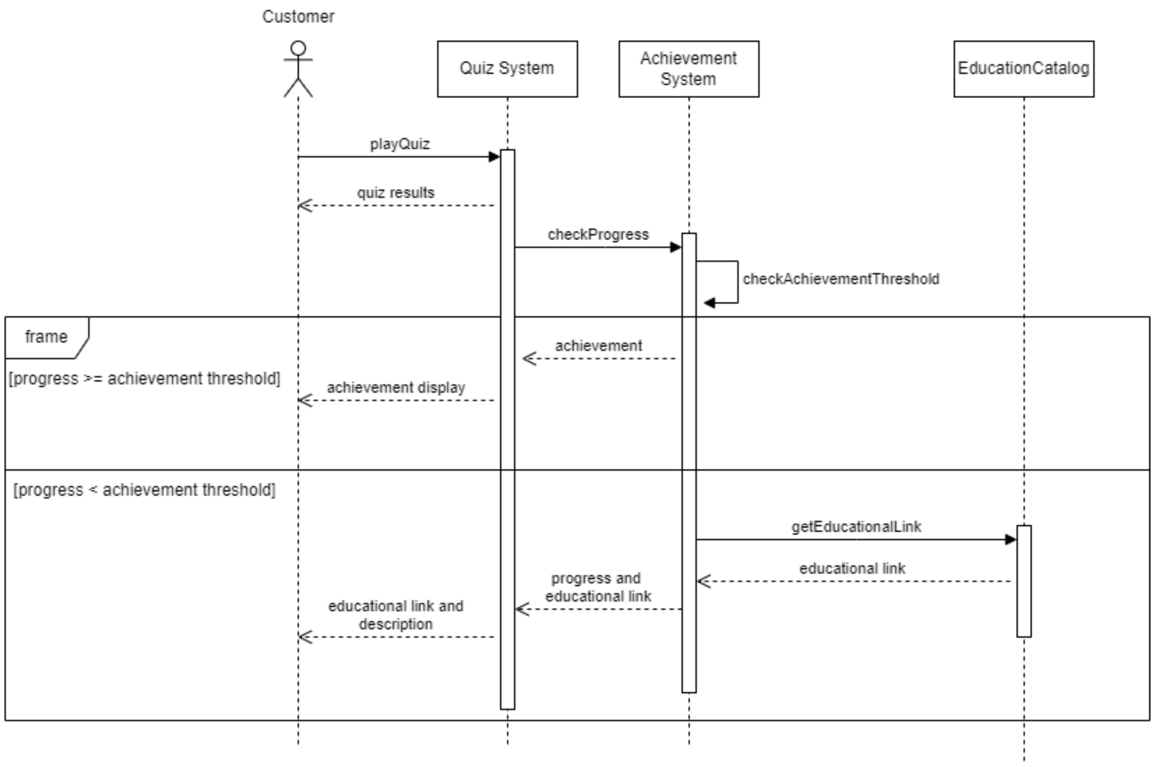
7. [15 POINTS] Sequence diagram – Provide sequence diagrams for each use case of your project. Please note that there should be an individual sequence diagram for each use case of your project.



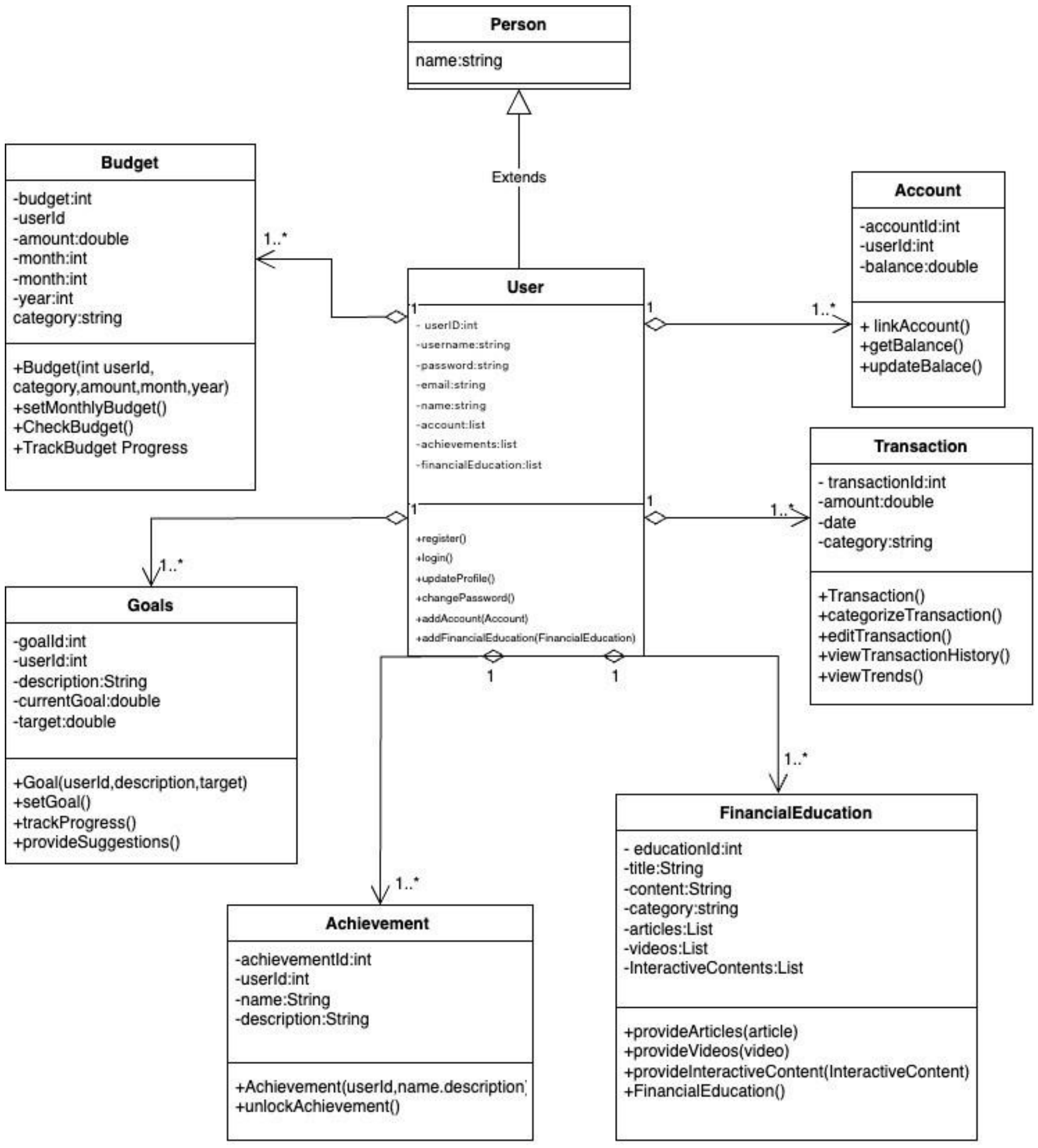

Customer
TransactionTrackingSystem
BankServer
Database
initiate transaction
send transaction data
save transaction data
alt
[if transaction is valid]
transaction saved
transaction saved
transaction successful
[else]
transaction not saved
transaction not saved
transaction error
view transactions
request transactions
fetch transactions
return transactions
return transactions
display transactions
request to categorize transactions
request categorizations
categorize transactions
alt
[if categorization is valid]
categorization complete
categorization complete
display categorized transactions
[else]
categorization incomplete
categorization incomplete
categorization error
request to edit transaction
request edit
edit transactions
alt
[if edit is valid]
edit complete
edit complete
display edited transaction
[else]
edit rejected
edit incomplete
edit error


Customer
BudgetPlanner
VisualizationService
set up monthly budgets
budgets set
spend money
check budget limits
frame
[limit exceeded]
limit exceeded alert
[else]
current spending status
request budget progress
get budget progress views
budget progress view
budget charts and graphs


Customer
GoalSettingSystem
Set financial goal
Confirm goal setting
Request progress tracking
Calculate progress
Display progress
Request suggestions
Generate suggestions
Provide suggestions or tips


Customer
Quiz System
Achievement System
EducationCatalog
playQuiz
quiz results
checkProgress
checkAchievementThreshold
frame
[progress >= achievement threshold]
achievement
achievement display
[progress < achievement threshold]
getEducationalLink
educational link
progress and educational link
educational link and description

8. [15 POINTS] Class diagram – Provide a class diagram (similar to Figure 5.9) of your project. The class diagram should be unique (only one) and should include all classes of your project. Please make sure to include cardinalities, and relationship types (such as generalization and aggregation) between classes in your class diagram. Also make sure that each class has class name, attributes, and methods named.



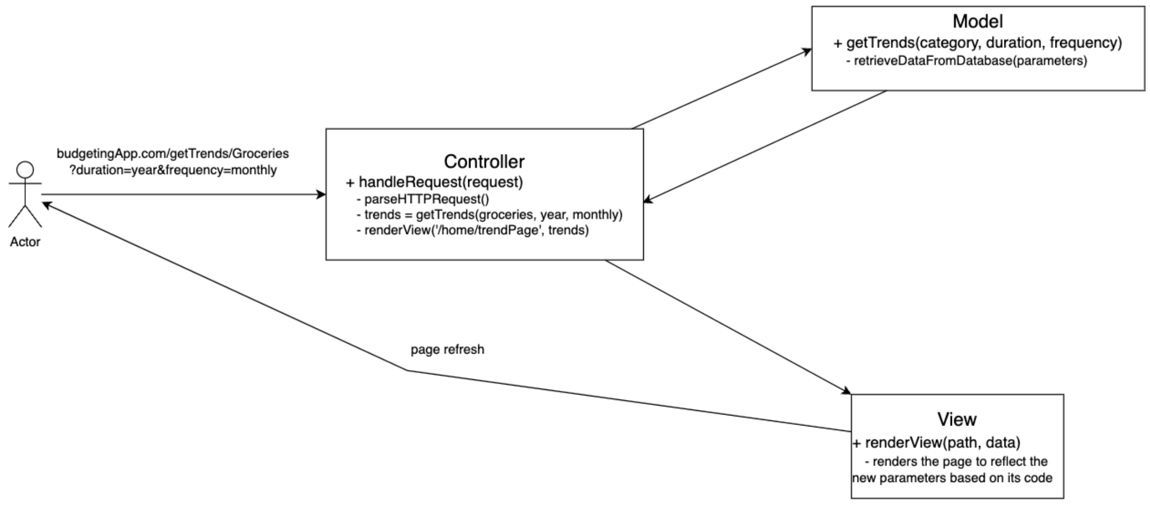
9. [15 POINTS] Architectural design – Provide an architectural design of your project. Based on the characteristics of your project, choose and apply only one appropriate architectural pattern from the following list:

<u>Model-View-Controller (MVC) pattern</u>

The Model View Controller architecture pattern best fits the software that we would be building in our budgeting app, because of its clear separation of concerns such as the View component being completely focused on developing good user interfaces which our app would benefit from as it uses visuals for charts/graphs. The MVC provides the necessary structure and organization that a budgeting app would need by separating all the financial data into a database that the Model component can quickly access, and the Controller can get and send the appropriate data or actions for the View to present.

The layered architecture would not be applicable since we are not developing layers of separate concerns of functionality. The repository model would not work as there is no single storage of universal data our app offers; every piece of data is individual to the client that uses it. The pipe-and-filter model does not make sense as there is no sequential processing of data, plus we rely on user-interaction. The client-server architecture is a suitable option, however the features that it offers like scalability and centralized data management distributed among a network of different servers, would not be necessary for our budgeting app.

For an example of our architectural design implemented in the context of our application, we can take the following action we would have implemented, such as viewing the spending usage trends of a particular category of transactions, and see it in a bar graph format with respect to the total duration of time you want to see it or the frequency of graph you want to see in that timespan. If we model this as a form of some network request, like a sample HTTP formulated and sent to our application, our Model View Controller architectural diagram below illustrates this function happening by the agent and shows what goes on inside our system.


Model
+ getTrends(category, duration, frequency)
- retrieveDataFromDatabase(parameters)
budgetingApp.com/getTrends/Groceries ?duration=year&frequency=monthly
Actor
Controller
+ handleRequest(request)
- parseHTTPRequest()
- trends = getTrends(groceries, year, monthly)
- renderView('/home/trendPage', trends)
page refresh
View
+ renderView(path, data)
- renders the page to reflect the new parameters based on its code

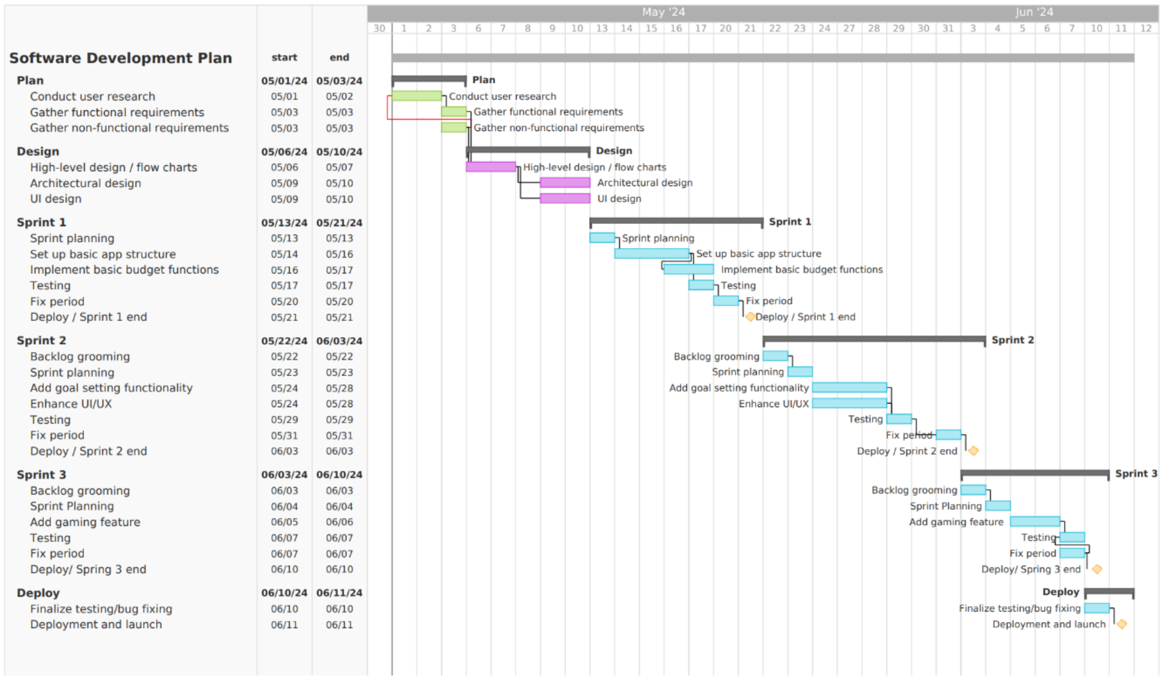
As illustrated by the diagram, the user sends an HTTP request which the Controller picks up and retrieves and parses this request and begins applying the internal logic of application to communicate with the other components and understand the request. After getting the right function to compute, it calls a request to the Model component to get the data given the parameters it sent. The Model receives these parameters, queries the Database with its built-in logic and returns the appropriate data back to the Controller. The Controller now saves this data in its component and sends it over to the View Component, calling its render function and using its logic to give the correct path page to render with that data. The View Component receives these parameters in its rendering along with the data and executes its code which causes a page refresh on the user's end, with the updated data, which terminates the action. Applying this similar design to other functions and procedures that comprise the application will successfully build a comprehensive architecture of this software that uses the MVC pattern.

(End of Project Deliverable 1)

3. **[35 POINTS]** Project Scheduling, Cost, Effort and Pricing Estimation, Project duration and staffing: Include a detailed study of project scheduling, cost and pricing estimation for your project. Please include the following for scheduling and estimation studies:

   3.1. **[5 POINTS]** Project Scheduling. Make an estimation on the schedule of your project. Please provide start date, end date by giving justifications about your estimation. Also provide the details for:
   - Whether weekends will be counted in your schedule or not
   - What is the number of working hours per day for the project

An estimation for the total time for our project would be about 6 weeks. This is determined through creating a calendar-based bar chart, showing the tasks, estimated duration, and task dependencies. The Gantt chart used as the calendar was applied using Agile and Scrum principles. A start date would be May 1st with an end date of June 11, in which this time was calculated not using weekends. The number of working hours per day would be around 6 hours, and we assumed a group size of 8 software engineers.
Project Scheduling Gantt Chart:



Software Development Plan	start	end
Plan	**05/01/24**	**05/03/24**
Conduct user research	05/01	05/02
Gather functional requirements	05/03	05/03
Gather non-functional requirements	05/03	05/03
Design	**05/06/24**	**05/10/24**
High-level design / flow charts	05/06	05/07
Architectural design	05/09	05/10
UI design	05/09	05/10
Sprint 1	**05/13/24**	**05/21/24**
Sprint planning	05/13	05/13
Set up basic app structure	05/14	05/16
Implement basic budget functions	05/16	05/17
Testing	05/17	05/17
Fix period	05/20	05/20
Deploy / Sprint 1 end	05/21	05/21
Sprint 2	**05/22/24**	**06/03/24**
Backlog grooming	05/22	05/22
Sprint planning	05/23	05/23
Add goal setting functionality	05/24	05/28
Enhance UI/UX	05/24	05/28
Testing	05/29	05/29
Fix period	05/31	05/31
Deploy / Sprint 2 end	06/03	06/03
Sprint 3	**06/03/24**	**06/10/24**
Backlog grooming	06/03	06/03
Sprint Planning	06/04	06/04
Add gaming feature	06/05	06/06
Testing	06/07	06/07
Fix period	06/07	06/07
Deploy/ Spring 3 end	06/10	06/10
Deploy	**06/10/24**	**06/11/24**
Finalize testing/bug fixing	06/10	06/10
Deployment and launch	06/11	06/11

3.2. **[15 POINTS]** Cost, Effort and Pricing Estimation. Describe in detail which method you use to calculate the estimated cost and in turn the price for your project. Please choose one of the two alternative cost modeling techniques and apply that only:

We used the Function Point (FP) method.

	Function Category	Count	Complexity			Count × Complexity
			Simple	Average	Complex	
1	Number of user input	22	3	4	6	66
2	Number of user output	20	4	5	7	100
3	Number of user queries	5	3	4	6	15
4	Number of data files and relational tables	5	7	10	15	35
5	Number of external interfaces	4	5	7	10	28

GFP - 244

Justifications:
First, based on searching online, the most common average number of base budget categories that is mainly used is 7, and these are: housing, utilities, insurance, food, transportation, entertainment, and miscellaneous. Therefore, we will use this assumption of 7 categories for our further calculations and justifications in the following sections.

For User input:
email (username), password, bank routing number, checking account number, name, category setup (7), monthly budget setup (7), overall monthly budget setup, notification opt-in, and goal. All of these would be of simple complexity, as there is not much complicated in inputting these values. This adds up to 22 inputs.

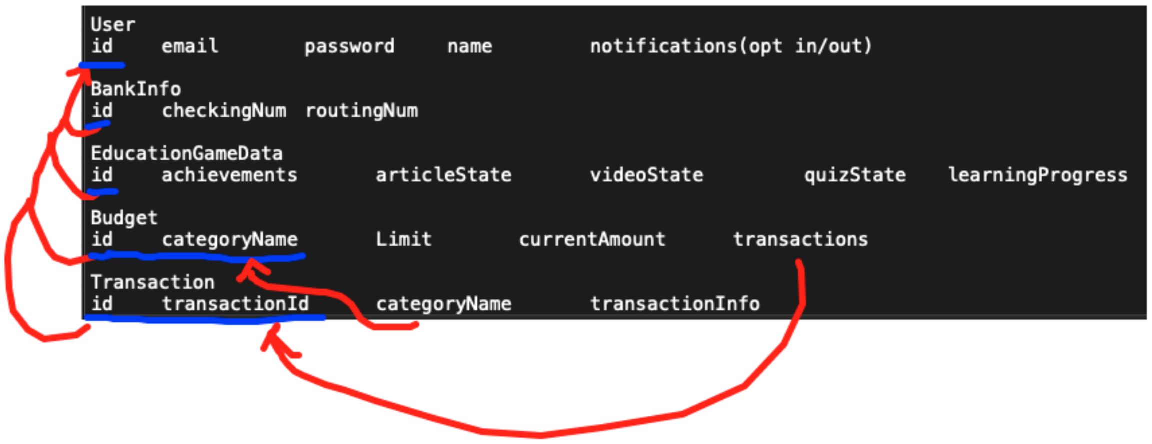
For User Output:
expense trend chart (overall and per category)
transaction list (overall and per category)
pie chart for category expenses
listing of current expense amount vs set limit + alert if exceeded
achievements
learning progress

We set these to have average complexity due to these outputs needing more complex implementation when generating trend and pie charts, alert functionality logic, and the computation for getting learning progress.

Once again we assume the first two outputs actually have (7 + 1) outputs each for each category + overall, giving us 8 + 8 + 1 + 1 + 1 + 1 = 20 outputs. The total count complexity is trivially calculated based on the complexities we set.

For User Queries:
User info
trends
transactions
category limit
category current amount
As these would comprise of all the essential types of calls to the database storage which the app would retrieve the data from and transform it into the appropriate outputs that the user requested, based on usage of canned transactions. These queries would be simple complexity as the overhead or type of data transfer for these types would be mainly numerical/text data.

For number of data files/relational tables, we set this to be 5 relational tables in our database schema, as we mapped the basic relational schema tables seen below:



```
User
id    email       password     name        notifications(opt in/out)

BankInfo
id    checkingNum  routingNum

EducationGameData
id    achievements       articleState       videoState        quizState    learningProgress

Budget
id    categoryName       Limit       currentAmount      transactions

Transaction
id    transactionId       categoryName       transactionInfo
```

Just for basic clarification, the red arrows represent foreign keys and blue lines represent primary keys of each table. We believe this design would be a simple and holistic way of storing the data types in our application in a modularized manner. We set the complexity of these tables to be simple as the design and implementation would not need to be very rigorous.

For number of external interfaces, we set this to be:
Chart/Graph GUI to display expense trends
Pie Chart GUI for category spending usage
Security Third Party API when connecting bank account information
GUI for displaying articles/videos/interactive quizzes for financial education

We set these complexities to be average as the interfaces would depend on the implementation of the GUIs to render proper data as financial charts and graphs, and the security API would involve more complexity than a simple approach since this involves people's sensitive Bank information.

Now for Processing Complexity:
(1) Does the system require reliable backup and recovery? PC = 4
(2) Are data communications required? PC = 3
(3) Are there distributed processing functions? PC = 2
(4) Is performance critical? PC = 5
(5) Will the system run in an existing, heavily utilized operational environment? PC = 3
(6) Does the system require online data entry? PC = 5
(7) Does the online data entry require the input transaction to be built over multiple screens or operations? PC = 5
(8) Are the master files updated online? PC = 4
(9) Are the inputs, outputs, files, or inquiries complex? PC = 2
(10) Is the internal processing complex? PC = 2
(11) Is the code designed to be reusable? PC = 1
(12) Are conversion and installation included in the design? PC = 3
(13) Is the system designed for multiple installations in different organizations? PC = 3
(14) Is the application designed to facilitate change and ease of use by the user? PC = 5
= 4 + 3 + 2 + 5 + 3 + 5 + 5 + 4 + 2 + 2 + 1 + 3 + 3 + 5 = 47

PCA = 0.65 0.01(47) = 1.12
FP = GFP PCA = 244 1.12 = 273.28
E = FP/Productivity = 273.28 / 30 = 9.10 9 person-weeks
With team size of 8, 9 / 8 = 1.125 1 week

3.3. **[5 POINTS]** Estimated cost of hardware products (such as servers, etc.)

To host our application, we decided to go with Amazon EC2 as it provides on-demand computing to service our app [1] - meaning we only pay server costs for how much is demanded at that current time, which can provide flexibility and save money for us. To store our actual data for this app, we went with Amazon EBS as it works well with EC2 [2]. Since we would be assigning data from our database hosted in cloud servers to users' devices via the internet, we would factor data transfer costs as well. EC2 provides built-in security features which we will not need to pay extra for [1]. Now for parameters, we will use the t3.medium instance type as it provides a balance between compute power and memory capacity for running a medium-sized app like ours. We went with 2 instances to distribute workload and avoid downtime if a large number of requests are made, thereby improving reliability. We will assume 10 GB data transfer in and 100 GB data transfer out monthly based on usage pattern estimates of other applications, as transfer in tends to be smaller than transfer out (which includes serving web page content along with other database data). Using AWS Calculator [3], with 2 Shared Instances + Linux OS + Constant Usage all month = $60.74/month for EC2 hosting our servers. Next, 100 GB of storage for 2 instances in a month = $16.00/month for storing our data on the cloud using EBS. Data Transfer In for AWS is free, Data Transfer Out for 100 GB = $9.00/month. The total cost then would be $60.74 + $16.00 + $9.00 = $85.74 per month.

3.4. **[5 POINTS]** Estimated cost of software products (such as licensed software, etc.)

For a team of 8 software engineers, we will decide to go with BitBucket [4] for cloud code hosting and repository control for a distributed version control system to develop software in an organized manner conducive to business development. This will be $24 monthly [5]. We plan on using JIRA [6] for project management and organization software, especially for providing an Agile workflow environment [7]. For a team of 8, JIRA software is free, so we won't factor anything in [8]. We will use MySQL as our Database Management System, and we can use it for free under the GNU General Public License (GPL) as it is an open-source software. We will also use Plaid [9] as a Third-Party Bank Aggregator for securely linking customer Bank Accounts and using their API to retrieve transactions for each user in our application. The enterprise version of this application would cost $500/month. The total cost therefore is $524 per month.

3.5. **[5 POINTS]** Estimated cost of personnel (number of people to code the end product, training cost after installation)

We would be accounting for a professional 8-person software engineering team to develop the software for our application. But to continuously work on bugs, improvements, user-feedback, etc. this software would have to be perpetually maintained, tested, debugged. We would assign each software engineer's labor costs and thus salary to be $60,000 yearly. We would set this to be constant before and after installation of the initial application. Therefore, the total cost of personnel would be $480,000 per year / $40,000 per month.

Since this application is designed for public end users, there is no particular training cost, as the usability will be built into the design.

4. **[10 POINTS]** A test plan for your software: Describe the test plan for testing minimum one unit of your software. As an evidence, write a code for one unit (a method for example) of your software in a programming language of your choice, then use an automated testing tool (such as JUnit for a Java unit) to test your unit and present results. Clearly define what test case(s) are provided for testing purposes and what results are obtained (Ch 8). Include your test code as additional document in your zip file submitted.

A test plan for our budget application is outlined below. Our method we developed was a function to calculate any excessive spending amounts for a particular user based on the budget limits they set for each category. This method takes in the user ID as an input and then scans all their budgets and transactions and computes the amount of money exceeded for each budget and sums them. If the user did not exceed any of their budgets, then the method will trivially return 0. This is labeled as 'public double checkBudgetExceeded(int userId)'. We wrote this code in Java so we can conveniently test cases using JUnit. Due to the somewhat different number of inputs for these overarching methods, such as each individual Budget limit, Budget current amount, each Transaction amount, user ID, and so on, we found it more prudent to use JUnit to test and validate the outputs of the method rather than the inputs as this strategy would make more sense and be more conducive for ensuring the correctness of our function. We decided to use Boundary Value Analysis for changing the outputs by testing a number one above and one below the correct output. We also decided to keep the input Budgets and Transactions constant and do Boundary Value Analysis for the 2 users we created. These test cases are summarized below:

Test 1: **BT_U1_V1**: This is a Correct Output Test Case for User 1's budgets and transactions exceeded amounts

Test 2: **BT_U1_I1**: This is an Incorrect Output Test Case for User 1's budgets and transactions exceeded amounts by modifying the output + 1.

Test 3: **BT_U1_I2**: This is an Incorrect Output Test Case for User 1's budgets and transactions exceeded amounts by modifying the output - 1.

Test 4: **BT_U2_V1**: This is a Correct Output Test Case for User 2's budgets and transactions exceeded amounts

Test 5: **BT_U2_I1**: This is an Incorrect Output Test Case for User 2's budgets and transactions exceeded amounts by modifying the output + 1.

Test 6: **BT_U2_I2**: This is an Incorrect Output Test Case for User 2's budgets and transactions exceeded amounts by modifying the output - 1.

5. **[10 POINTS]** Comparison of your work with similar designs. This step requires a thorough search in the field of your project domain. Please cite any references you make.

In comparing our budget planning application with similar designs in the field, we conducted a comprehensive analysis of NerdWallet [10], a widely used budgeting resource. NerdWallet offers various tools for budgeting, including transaction tracking, budget planning, and goal setting. Users can monitor their cash flow and adhere to the 50/30/20 budget guidelines [11]. Additionally, NerdWallet provides features such as net worth tracking, debt monitoring, and credit score monitoring, which are not explicitly included in our app design. While both applications share common functionalities such as account management, transaction tracking, budget planning, and goal setting, our app sets itself apart through its unique gamification of financial education component. Unlike NerdWallet, our app offers interactive articles, videos, and quizzes on personal finance topics, enriching users' financial literacy while making budget planning more engaging. This gamification element distinguishes our design from traditional budgeting applications and enhances user motivation to learn and manage their finances effectively. Through this comparison, it's evident that while NerdWallet and our app share fundamental budgeting features, our design introduces a novel approach to financial education, providing users with a more interactive and enjoyable budget planning experience.

Similarly, YNAB (You Need a Budget) [12] follows a zero-based budgeting system, emphasizing proactive financial planning. Like our app, YNAB allows users to link their financial accounts for comprehensive tracking and offers educational resources. However, our app stands out by providing a more gamified approach to financial education, aiming to make budget planning engaging and enjoyable for users of all levels of financial literacy. While YNAB requires a high level of commitment and active involvement from users [13], our app caters to those seeking a balance between guidance and autonomy in managing their finances, with a focus on making financial education accessible and enjoyable. Additionally, YNAB's availability across various platforms and its detailed budget planning features may suit certain users' preferences [14], yet our app aims to provide similar functionalities at a more affordable or competitive price point. Overall, our app offers a unique blend of user-friendly design, comprehensive budgeting features, and engaging financial education, positioning it as a valuable tool for individuals seeking to improve their financial literacy and manage their finances effectively.

6. **[10 POINTS]** Conclusion - Please make an evaluation of your work, describe any changes that you needed to make (if any), if things have deviated from what you had originally planned for and try to give justification for such changes.

Our project remained consistent with the initial vision, which was to design a unique budgeting application focused on both tracking spending and providing financial education through gamification. This differentiation sets our concept apart from other applications in the market because it offers users a comprehensive solution that blends practical budgeting tools with an engaging learning experience.

Throughout the design process, we encountered little to no deviations from our original plans. The core features and functionalities outlined in our initial proposal were successfully conceptualized and mapped out, including account management, transaction tracking, budget planning, goal setting, and the gamification of financial education. Our team collaborated effectively to ensure that these design elements were integrated logically and seamlessly into the app's interface.

Moreover, our comparative analysis with similar budgeting applications, such as NerdWallet and YNAB, provided valuable insights into the market landscape and helped us refine our design strategy. By understanding the strengths and weaknesses of existing solutions, we were able to tailor our design to address user needs more effectively and differentiate ourselves from competitors.

Our focus on meticulous planning and thoughtful design decisions has laid a solid foundation for future development efforts. Moving forward, we remain committed to refining the design based on user feedback and technological advancements to ensure that our app delivers a seamless and impactful experience for users seeking to improve their financial literacy and management skills.

7. [5 POINTS] References: Please include properly cited references in IEEE paper referencing format.
It means that your references should be numbered, and these numbers properly cited in your project report.

[1] Amazon, "Amazon EC2," *amazon.com*. [Online]. Available: https://aws.amazon.com/ec2/. [Accessed Apr. 19, 2024].

[2] Amazon, "Amazon Elastic Block Store," *amazon.com*. [Online]. Available: https://aws.amazon.com/ebs/. [Accessed Apr. 19, 2024].

[3] Amazon, "AWS Pricing Calculator," *calculator.aws*. [Online]. Available: https://calculator.aws/#/. [Accessed Apr. 19, 2024].

[4] Atlassian, "BitBucket," *bitbucket.org*. [Online]. Available: https://bitbucket.org/. [Accessed Apr. 19, 2024].

[5] Atlassian, "BitBucket Pricing," *atlassian.com*. [Online]. Available: https://www.atlassian.com/software/bitbucket/pricing. [Accessed Apr. 19, 2024].

[6] Atlassian, "Jira Software," *atlassian.com*. [Online]. Available: https://www.atlassian.com/software/jira. [Accessed Apr. 19, 2024].

[7] Atlassian, "Jira Software features," *atlassian.com*. [Online]. Available: https://www.atlassian.com/software/jira/features. [Accessed Apr. 19, 2024].

[8] Atlassian, "Jira Software Pricing," *atlassian.com*. [Online]. Available: https://www.atlassian.com/software/jira/pricing. [Accessed Apr. 19, 2024].

[9] Plaid, Inc., "Auth," *plaid.com*. [Online]. Available: https://plaid.com/products/auth/. [Accessed Apr. 19, 2024].

[10] NerdWallet, Inc., "NerdWallet," *nerdwallet.com*. [Online]. Available: https://www.nerdwallet.com/. [Accessed Apr. 19, 2024].

[11] B. O'Shea and L. Schwahn , "Budgeting 101: How to Budget Money," NerdWallet, Inc., Feb. 16, 2024. [Online]. Available: https://www.nerdwallet.com/article/finance/how-to-budget. [Accessed 2024 19 April].

[12] YNAB, "YNAB," *ynab.com*. [Online]. Available: https://www.ynab.com/. [Accessed Apr. 19, 2024].

[13] YNAB, "Why YNAB is different," *ynab.com*. [Online]. Available: https://www.ynab.com/why-ynab-is-different. [Accessed Apr. 19, 2024].

[14] YNAB, "Our App Lineup," *ynab.com*. [Online]. Available: https://www.ynab.com/our-app-lineup. [Accessed Apr. 19, 2024].