

CS 501 Final Project Report: apt.
Alice Han, Tiffany Liu, Wyatt Napier

Problem Definition and Solution

Living with other people is hard. Sometimes they're your close friends, sometimes they're strangers, but it is still difficult to juggle spending that much time in close proximity and balancing your living styles. As a result, we created this application to try to smooth out the experience of living with other people, particularly for college students who are in apartments for the first time and may need the added structure.

There are a few main points of contention in every living situation: payment, chores, and household events. Our app addresses these individually to ensure that co-living is a smooth experience. By integrating Google calendar to create a shared household calendar that is displayed to all users, we've increased transparency behind upcoming events in the household. We've also tackled the issue of payments by setting up recurring payments and direct linking to Venmo to simplify each transaction. On top of that, we also clearly show payment history to help combat confusion over what has been paid and when. Finally, for chores, we've added a similar set of customizable recurring chores and automated chore assignments to users. The key feature for chores is that a user must upload a photo to demonstrate chore completion to discourage future conflicts over the existence and quality of chore completion in the past.

While there are some solutions to all of these problems individually, creating one centralized hub simplifies all the annoying tasks of living together so users can enjoy living with their roommates in harmony.

Team engagement and process

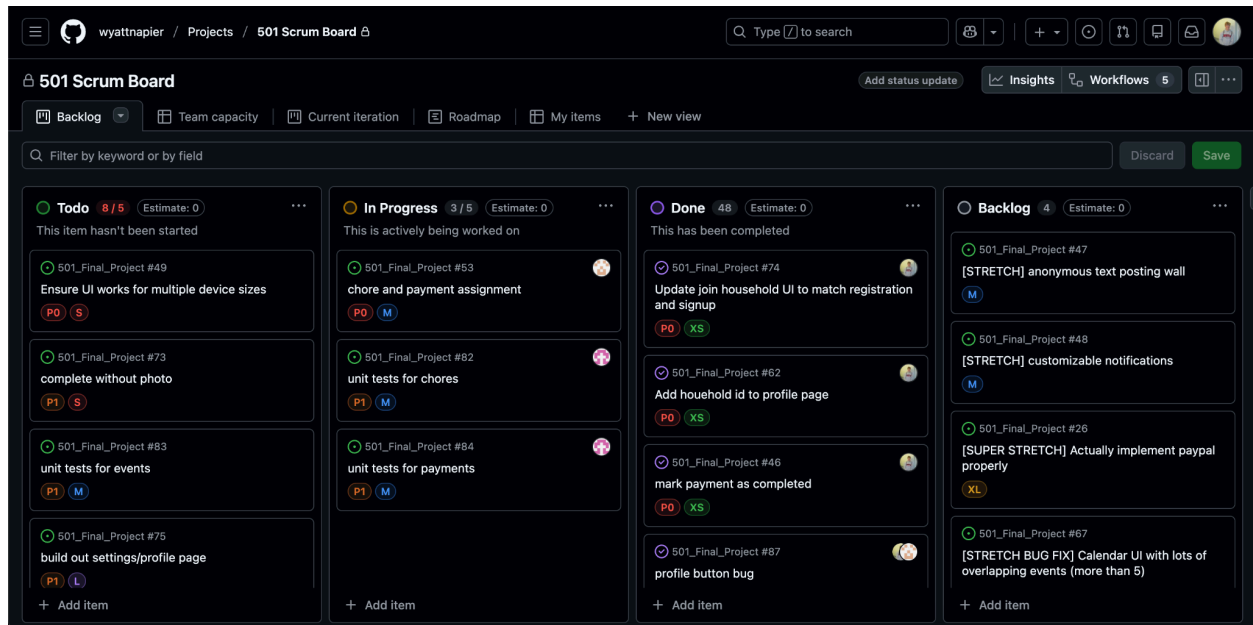
Given that we created an app about how hard it is to live with other people and avoid conflict, we were very conscious about how to set up our workflow to keep the entire team engaged and spread the workload evenly. The key to our success in this undertaking was GitHub!

Before we even got around to coding, we all collectively brainstormed to ensure that our app would be something that we'd all be invested in and excited about creating. We went through various ideas and eventually landed on something to make the process of living with people easier. Once we arrived at that idea, we continued to meet to draft up exactly what we wanted this app to look like and what features it would have. After we'd finally come up with a strong proposal and gotten it approved, then we hit the ground running with code.

Back to GitHub. Our team worked in loose sprints to ensure that the workload was evenly distributed and tasks were all completed in a timely manner. At the start of each sprint we would meet in person to discuss the progress we made on our individual tickets – highlighting the key

successes, blockers that we had hit, and whatever tips we had generated. Once we concluded our reflection, we turned our eyes to the future and self-assigned our next tickets based on priority.

We used GitHub issues in tandem with GitHub projects (essentially just a sprint board populated with GitHub issues) to create tickets, assign them, and track their progress. We would assign priorities and t-shirt sizes to issues in order to evaluate which ones should be tackled next. All together, this kept our team highly organized without needing constant communication and ensured that we tackled the most important issues first.



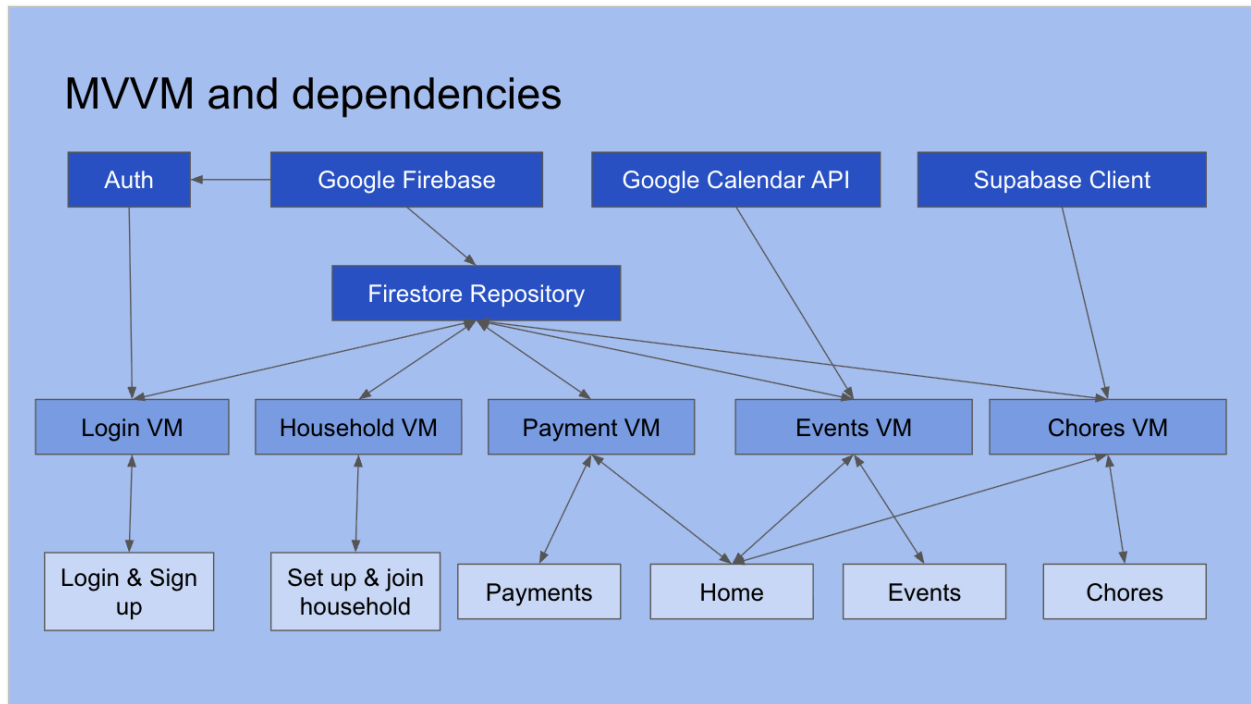
When it came to actually merging code, we would make a new branch for our issue and then create a Pull Request (PR) when it was time to merge, sometimes even a draft PR while we were still completing with checkboxes to enhance transparency about progress on that specific issue, and then request review from our teammates when the PR was ready. To merge, we required at least 1 review on each PR.

Outside of our formal sprint planning and review meetings, we would also text frequently to keep each other updated. This was particularly helpful for requesting review on PRs, asking for suggestions on how to tackle common build or emulator issues, and celebrating the little wins like new features or bug fixes. Again, this constant communication kept everyone engaged and enthusiastic about the project.

Technical implementation

As for technical implementation, our application follows a Model-View-ViewModel (MVVM) architecture, which is a common architectural practice across Compose applications. At the Model level, we defined various custom data classes to best represent the data our application uses. For example, for our household, we have a LocalResident data class to model residents of

the household. For chores, we have a data class to model recurring chores, which just represent a type of chore, and a data class to model an individual instance of a chore. Similar data modeling was applied for recurring payments and payments. These models allow us to define and contain the necessary data and logic to be used cleanly elsewhere in the code.



Between our Model and ViewModel layers, we have a data layer, which is where our external database sits. This layer abstracts the interactions between our code and our external database, which is a Firebase Firestore. In this FirestoreRepository, we define all the methods needed to directly update and access our data, so these methods can be exposed to our ViewModels. These methods include functionality to mark chores and payments completed, update active chores and payments, get all the users in the household, and retrieve household information, which are all operations that require direct interaction with the Firestore.

To connect the Model and View layers is the ViewModel layer, which is where all of the business logic sits. The ViewModels can perform the necessary business logic using the abstracted Firestore repository methods, and thus do not have to focus on the details of the data. For our application, we have one ViewModel per page in our application in order to keep code clean, separate concerns, and promote maintainability. In our app, the functionality of handling payments should have no effect on the functionality of chores, as well as no effect on household events. Moreover, the login and household set up processes, though related, are still separate concerns. For this reason, the different ViewModels store just the necessary business logic relevant to each page in the application, instead of one large ViewModel that is less organized

and more error prone. The View level encompasses all of our user interface components. This level provides us with the necessary composable functions to display information to and capture information from the user in the simplest way possible. These components should not handle any of the business logic or data fetching, and mainly serve as a visual representation of application data for the user.

To authenticate users in our application, we use Firebase Authentication provided by Google. Firebase Authentication provides backend services and UI components for easier integration and authentication. To provide calendar functionality to users, our application makes use of the Google Calendar API. Since we use Firebase authentication, we already have the necessary permissions to access a user's Google Calendar. Each household gets its own calendar created, and using the ID of this calendar, we can query the Google Calendar API for this calendar object. If a user has multiple calendars, then we filter the calendars by calendar name to obtain the events from only the household's Google Calendar object. This ensures that the information in the application is only relevant to the household and only gives users what they need to know. With the Google Calendar API and the calendar's ID, we can easily add events to the calendar to provide greater functionality than just displaying upcoming events. To actually manage access to the household's calendar, the Google Calendar API allows us to modify the Access Control Lists of the calendar. As a result, if User A creates the household and thus the household's calendar, whenever a new user joins the household, they are added to a pending users list. Once the household's creator signs in, the pending users are then automatically granted access to the calendar, meaning that we do not have to manually share access across users each time.

Within a household, there are many payments that roommates need to keep track of: rent, utilities, shared groceries, or even just a dinner. In order to allow the users to easily make payments to one another, we connected our application to the third-party payment service, Venmo. Since this is already a very widely used and trusted payment platform, we allow users to pay directly through the Venmo app. In the backend of our application, we calculate the amount each user owes for recurring payments, such as rent, as well as keep track of any additional payments users want to add. Upon clicking a button to make the payment, we use deeplinking to bring the user to the venmo app with the amount and recipient's username loaded in.

Another common struggle that we noticed in our own apartments is lack of accountability for chore completion and the quality of it. We saw this as a perfect opportunity to integrate the camera as our sensor. Our app uses the camera to allow users to snap a picture of their completed chore, and this photo then serves as proof of completion. These photos can help resolve any potential conflicts between roommates and keep each other accountable. As for implementation of the camera, our application first requests permission if permission has not already been granted, then launches the device's camera to take the photo. When the photo is taken, it is first stored in our app's private storage using a temporary URI, and then the chore is marked as

completed and the photo is associated with this chore. Since our Firestore database does not allow for the storage of large binary files, like these images, our photos are sent to an external Supabase database and stored there. These files are stored under the household id and chore id, allowing us to easily access them in the future. When our application needs to retrieve the photo for display, we get the signed url for the photo from Supabase as they are stored in a private database.

For the actual user interface development, we declaratively built out UI components using Jetpack Compose. This allowed us to build custom composables that made use of provided components and containers to format our UI to be intuitive and visually appealing. An important requirement of an application's UI is not only to present information to the user, but to actually react to changes in data. Our application undergoes constant changes and needs the most up-to-date information. To handle this, we use Kotlin's StateFlow to manage the state of data in the code. By using MutableStateFlow, we are able to emit the most current states of the data to collectors of the data, who observe changes with collectAsState(). This then allows for our UI to react to changes in the data. Moreover, our application deals with constant database updates, meaning that asynchronous operations should be properly handled, which StateFlow allows for.

Together, these features allow us to confidently present the accurate data to the user and keep the most up to date information in our database. The architecture of our application helps enforce this by separating data management, business logic, and UI state, clearly tracking the flow of data throughout the application.

Challenges and Lessons Learned

As this was the first android we have developed from scratch for all of us, there were many bumps that we ran into along the way. One of the main issues that we ran into was having to restructure the architecture of the app as we were developing. We had originally started with developing the front end, which allowed us to get a good idea of what the app would look like. However, since we were simply using mock data, with a rough idea of how we would implement the database, we had to change the way that we parsed and displayed the data many times. Additionally, we had originally planned to only use one view model across the entire application, as we believed that the logic would be simpler that way. However, it quickly became clear, as we began building out the backend, that we would need separate view models for each of the pages in order to keep the logic clean.

Another challenge we ran into was designing the database. Our app keeps track of many different moving parts; even just looking at the chores page, we need to know information pertaining to the chores created by the users, who the chore is currently assigned to, and past instances of each chore. When trying to determine how all of this information could be concisely stored in order to allow for simple and quick queries, we felt uncertain about where to begin. Through many

discussions, we realized that it would be easiest to store it in a noSQL database so that we could include many subdocuments with all of the information relating to these different objects we would need to keep track of. Going back to the chores example, we stored an array of subdocuments in our household document that represent the recurring chores of a household, which allow us to implement the logic of assigning chores to members of the household. We also stored an array of subdocuments for all of the chores, past and present, pertaining to the household. By using a noSQL database, we are able to more flexibly store this information for a single household, making queries more efficient and data more easily accessible.

Finally, we also ran into the challenge of facing business decisions regarding the logic of certain features of our app, especially in relation to how we were assigning chores and payments. For instance, when it came to chores, we needed to make a decision about when and how they should be assigned. Since chores are created with a cycle, it felt like the chores should simply be rotated between the residents in the household on completion of the cycle. However, more complicated scenarios arose as we were discussing that concept – what if there are more chores than roommates, what if different chores have different cycles, what happens if a chore is overdue and not yet completed? While this was less of a technical issue, it felt difficult to make a decision regarding what would be best from a user’s perspective. Ultimately, we decided that chores should simply be reassigned when completed and assigned to the user with the least chores. This allows for a more balanced load for all of the residents of a household and ensures that no more than a cycle will occur between each time a chore is completed.

AI use in Development

In 2025, you can’t go on YouTube or walk down the street without seeing ads about AI. As we all approach the actual working world, it is important that we know how to use tools like this for development and on top of that, to acknowledge the successes and failures of these tools.

Our team was able to integrate AI use into our development in many ways. The most direct and simple way that we used AI, particularly early on in our project, was with advice in structuring the overall application and user flow. For these design questions, it was helpful to essentially add a fourth engineer’s advice to the conversation, particularly because the AI tools we were querying (namely Claude and ChatGPT on the web browser at this stage) are aware of the technical APIs and tools that can be used to implement some of our more specific architectural choices. One key example of this was when we were designing how many view models to use and which pages they’d be linked to. We realized that they would all be querying the database separately for similar data, which is when we started looking for a filetype that would act as our single point of contact with the database. Using AI, we came up with a Repository which has been essential to our data flow and overall application design.

Once we'd progressed past initial design, AI was quite useful for development as well. Once again, it is a fantastic tool for explaining new technical concepts or technologies, especially since it can provide examples of implementation. One downfall that we came across within this regard is that it sometimes provides overly complicated examples or those that lack the context of the greater app so they're harder to learn from. Here, we would either query in the browser or in Android Studio.

Another core strength of these AI tools is debugging and quick fixes. For example, basic additions or simplifications like using the elvis operator rather than an if statement will quickly be suggested by Gemini in Android Studio. My personal favorite use of AI tools though, is to take an error log and paste it in so that I can understand what is causing the error and where – sometimes I even get an immediate fix just from that!

Sometimes, such as when in agent mode, the Android Studio AI tool will provide large chunks of code that are updated in multiple places and just require a single approval for the changes to be implemented. Additionally, the agent iteratively keeps working on the code until all errors are gone, so it can very quickly change the codebase in a very large way. For this reason, we avoided using the Android Studio AI in agent mode, or copying large swatches of code from any AI tools because it is much harder to review with a high attention to detail. In these cases, security issues or flaws in Kotlin Compose idioms could slip through unnoticed much more easily. Instead, we focused on applying smaller incremental changes from AI tools and using them to understand how to approach new problems.

Another particularly helpful use case for AI tools was boilerplate and testing code. It performed well to create basic file structure and boilerplate because it is so simple and free of logic. It also was good for generating testing code and ensuring high test coverage because lots of the testing code was similar with minor variations.

We did find that there were some more limitations to using AI tools though. One example is that it would get very stuck in one solution to an error, even if it wouldn't work. In these cases it could even end up cycling between two fixes; issue A would need fix A, but fix A creates issue B and fix B would create issue A again. Another point of failure would be with imports and dependencies. There would frequently be version misalignment which would dramatically slow down development as we fight to untangle the dependency issues. The central issue with these AI tools is that they have a limited capacity for context. In some cases that means that the file version in the Android Studio AI tool is out of date, or even that it can't track the flow of data or logic across the entire file structure because it is too complex.

Given that AI tools, despite aiming to be helpful, do have their limitations, our team developed an AI usage strategy. First, the author of the branch who is using AI must engineer their prompts

to AI tools in a way that emphasizes core Kotlin development practices and security. Then, after AI code is generated, the author of the branch is the only person to implement it as well as the first reviewer, and then the subsequent PR reviewers are the additional reviewers. Adhering to this development process added more resiliency and ensured that AI created errors didn't slip through the cracks.