ExpenseFlow

Ella Berglas 47425381
Donghyug Jeong 47430004
Lucas Hicks 47440085
Ruidong Ding 49125443
Aaditya Yadav 48268790

Abstract

ExpenseFlow is a comprehensive expense management tool which empowers users to gain financial clarity and control. With group expense splitting, automated document scanning and generating of financial reports, ExpenseFlow empowers users to gain financial clarity and control. Expensflow's main quality attributes considered for architecture design include security, extensibility and reliability. The best suited architecture for these quality attributes was a microkernel architecture distributed by technical partitions. This system has a static plugin system which follows the independent plug-in and standard interface principles. Architectural design decisions and trade-offs were considered in particular for static vs dynamically loaded plug-ins, use of object relational mapping and authentication. A test plan and test results were developed along with other methods of evaluation to highlight how the architecture is suited to deliver the project's functionality and satisfy the architecturally significant requirements.

Table of Contents

Changes

As per the proposal marking, ExpenseFlow had an “ambitious” scope. The delivered application also had some additional functionality with the inclusion of report generation. There were only 5 team members to complete the project (down from the usual 6-7) and we had some challenges with course time constraints including the late release of the github repositories, AWS accounts and assessment being delayed a week by cyclone Alfred. Despite this, the majority of the project's initial scope was delivered. However, we did run into issues with finishing all the requirements for the evaluation. Despite this, we still managed to justify and show testing results for all of the quality attributes and ASR’s with very strong test coverage. There are a few items from the proposal that are missing from the developed project. At the end of this section is a revised evaluation plan. We kindly ask that you consider the ambitious nature of our initial MVP and the challenges we faced when marking our evaluation. Thank you.

Removal of Business Functionality

The proposal mentions functionality for businesses including an approval process and budget. The exact features around this were not very specific and are not directly mentioned in the scope of the proposal. Instead, for the MVP, ExpenseFlow is made for general users with some business specific features that have been implemented app wide (including an approval process, and report generation).

Reasoning

- The business specific features were not well defined in the proposal.
- Time restrictions (particularly with late repository release and AWS account release) and resource restrictions (5 team members)
- There are other parts of the project that offer more complexity.
- Adding this feature did not add much complexity or features and instead created confusion within the group.
- Does not contribute to projects quality attributes.

Addition of an Approval Process

The proposal mentioned an approval process within a business application. However, we decided to add an approval process for all users as an extra step for security. When splitting a payment, all users must agree to the amount they are to pay before anyone transfers any money. This prevents miscommunications (e.g., some people disagree on the split of the bill but others have already paid, creating confusion) and allows all users to have (and feel) more control over their money and spending.

Reasoning

- Improves security and user experience
- Allows for cleaner transactions
- Helps with extensibility for an approval process if business functionality were to be developed.

Simplification of a Budget Managing System

It was decided that the app is to be used to split expenses rather than to track all of a user's expenses. It did not make sense to introduce full budgeting capabilities into the app as we do not know all of a user's individual transactions. A more detailed budgeting system perhaps makes more sense when implemented for businesses (which we have removed) which may have a budget per team or department etc. Instead, it is more fitting for the app to generate reports for a user to track their spending (more on this later). It should be noted; however, that a user still has the ability to set a monthly budget.

Reasoning

- Unnecessary and did not add much complexity
- Could create miscommunications around the feature due to lack of clarity for which budgeting features to incorporate.
- Aforementioned time and resource restrictions
- Does not affect any of the quality attributes.

Addition of Report Generation

Since budgeting features were removed, it was decided to instead create functionality for users to track their spending through reports. As there are categories for expenses and items, it is possible to show how much a user is spending in each category and reports can be generated on this. This can be done for expenses between groups and friends.

Reasoning

- Additional complexity for the project and a replacement for “budgeting”
- Makes more sense from a user perspective
- Potential extensibility for business functionality where business may require detailed reports of spending.
- It is also possible to extend this feature to allow for users to upload their bank statements and generate reports of all of their expenses.

Updated Scope

The scope now includes:

- Users - including MFA when logging in with google
- Groups - can create a group to split expenses between members
- Friends - To split an expense with and make friend requests
- Expenses
    - Each expense has a category and description
    - Split expenses with friends/groups with custom amounts/percentages.
    - Utilise data extraction to autofill expense information (item costs, quantities, etc.) from provided attachments using AI image recognition.
    - Approval process. When an expense is created amongst multiple users, all users must approve the expense before it can be processed further.
- Budgeting

    - Basic expense statistics on the app
    - A monthly budget feature
    - Can generate a more detailed pdf report

Changes to Evaluation

Removal of MFA for Approvals

This change had been discussed early on in the project. MFA has been implemented with Auth0. This was a complex process to integrate and additional MFA functionality is blocked behind a paywall imposed by Auth0. The team found it satisfactory that MFA was accomplished when logging in and using it for approvals was out of scope of the MVP. Initially, the idea of approvals was also removed from the scope as it was associated with business functionality however, this was eventually integrated into the app.
Reasoning

- Blocked behind a paywall
- Time restrictions
- Does not drastically impact the effect on the quality attributes as MFA is still implemented for user login.

Addition of Fuzz Tests

Fuzz testing is a technique where faulty, invalid or potentially dangerous data is used for testing to ensure the system does not crash and is not easily compromised. This made a lot of sense to add to the testing and evaluation plan for ExpenseFlow as we allow users to upload their receipts to autofill expense information.

Removal of a Code Smells / Coupling software

Ideally SonarQube would have been used to detect code smells and high coupling. The implementation of this would have been quite simple, as it just requires SonarQube to have access to the ExpenseFlow codebase, allowing the SonarQube server to read through and evaluate changes to the codebase. However, due to our restricted permissions on the CSSE6400 GitHub organisation, we were unable to give SonarQube (a 3rd party) access to our repository. As such, it has been taken out of the scope for the evaluation.

Updated Evaluation Plan

Security

- Fuzz Tests
- Measure success and failure rates for login attempts
- MFA to log in
- Vulnerability scanning tool
- Security Logging/ Checking / Scanning tool

Reliability

- Conduct regular team code reviews to assess any new features added to the system
- Sentry - Used for tracking and debugging errors, crashes and performance issues
- Large testing coverage

Extensibility

- Plug-in interface
- Implementation of wide range tests to ensure new additions do not break existing code.
- Create CI pipelines to ensure that the above tests are run frequently with every system change

Architecturally Significant Requirements (ASRs)

ExpenseFlow must be **extensible**. This is due to the various extra features that are out of scope such as budgeting and catering towards business specific requirements.

ExpenseFlow must be **secure**. We are not only dealing with user's private information but also sensitive financial data. Furthermore, there are several laws in place (in Australia) to ensure privacy, security and transparency when handling user data (Privacy Act 1988, Consumer Data Right, etc).

ExpenseFlow must be consistent, **reliable** and maintainable. Data inconsistencies and/or lost data can cause misreported financials for users.

Architecture Options

Whilst simplicity is not considered to be a main quality attribute of our system, security does favour simplicity (KISS principle) and so do our time and resource constraints. An ideal architecture would be a simple, extensible and reliable architecture that is easy for testing but is complex enough to handle the project's functionality and support the quality attributes of extensibility, security and reliability. A variety of architectures are discussed with corresponding pros and cons associated with them. Items that are bolded in these pros and cons lists are considered to be more important for the project.

Monolith

A monolith is not very extensible nor reliable. It has high coupling and is difficult to test. ExpenseFlow is too complex for a monolithic architecture and using one would result in higher complexity compared to other architecture options.

Pros

- Simple and easy to understand
- **Secure in the sense that it is has less components and a smaller attack surface**
- Fast and easy communication between components

Cons

- **Not extensible. Changes to functionality and/or the addition of features may require a large number of changes to the existing codebase**
- **Single point of failure (low reliability)**
- **High coupling**
- **Harder to test (due to high coupling)**
- Becomes much more complex and difficult to deal with as the project grows
- Not scalable (in terms of computing power and load)

Layered Architecture

This architecture was heavily considered. This layered architecture was followed to a certain extent with technical partitioning by having a presentation/front-end, business, persistence and database layer. However, each layer can be highly coupled. As extensibility and reliability are the main quality attributes, the selected architecture should be more suited to those.

Pros:

- Simple
- **Follows technical partitioning**
- Modular (by layer)
- Reusable

Cons

- **Not extensible**
- Can become very complex as the system grows
- **Layers can be tightly coupled, making it not very extensible.**
- Can have high coupling as it becomes more complex

Event Driven Architecture

An event driven architecture allows for scalability and extensibility. Although an extensible architecture is needed, an event driven architecture sacrifices data consistency for high performance. ExpenseFlow must have data consistency as per the ASR's and does not have such a need for high performance. Furthermore, an event driven architecture has unnecessary complexity.

Pros

- **Extensible**
- Quick/ Responsive
- Loosely coupled, which can be easier to maintain
- Scalable (due to loose coupling)

Cons

- **Complexity**
- **Data inconsistency**
- Difficult for testing
- **Doesn't make the most sense for ExpenseFlow, everything does not need to be very responsive and there are too many sacrifices for that.**

Microservice Architecture

The benefits of a microservice architecture are mostly for scalability and are more beneficial for larger and complex systems. A microservice architecture does not really support security and is more complex than what the system needs.

Pros

- Scalability
- Fault isolation
- Independent deployment
- Improved code maintainability
- Data isolation

Cons

- Increased complexity
- **Data management and consistency issues.** Can be hard to keep data consistent across all services.
- Debugging and testing can be more complex across all the services
- **Security complexities**

Microkernel Architecture

A microkernel architecture is extensible via the use of plug-ins, secure due to a smaller kernel size and attack surface area, reliable due to faults in plug-ins not affecting the kernel, and is relatively simple compared to event-driven and microservice. This satisfies all of our quality attributes and is the most suitable architecture forExpenseFlow.

Pros

- Modular

- **Extensible, can add plugs-ins making additional features easy. This is very relevant for the ability to cater to business specific requirements.**
- Can distribute and incorporate both domain and technical partitioning
- Simple (in comparison to event driven and microservice)
- **Secure as smaller kernel/core reduces risks of attacks**
- Can be more reliable as faults in services do not affect the kernel
- Maintainability
- Testability

Cons

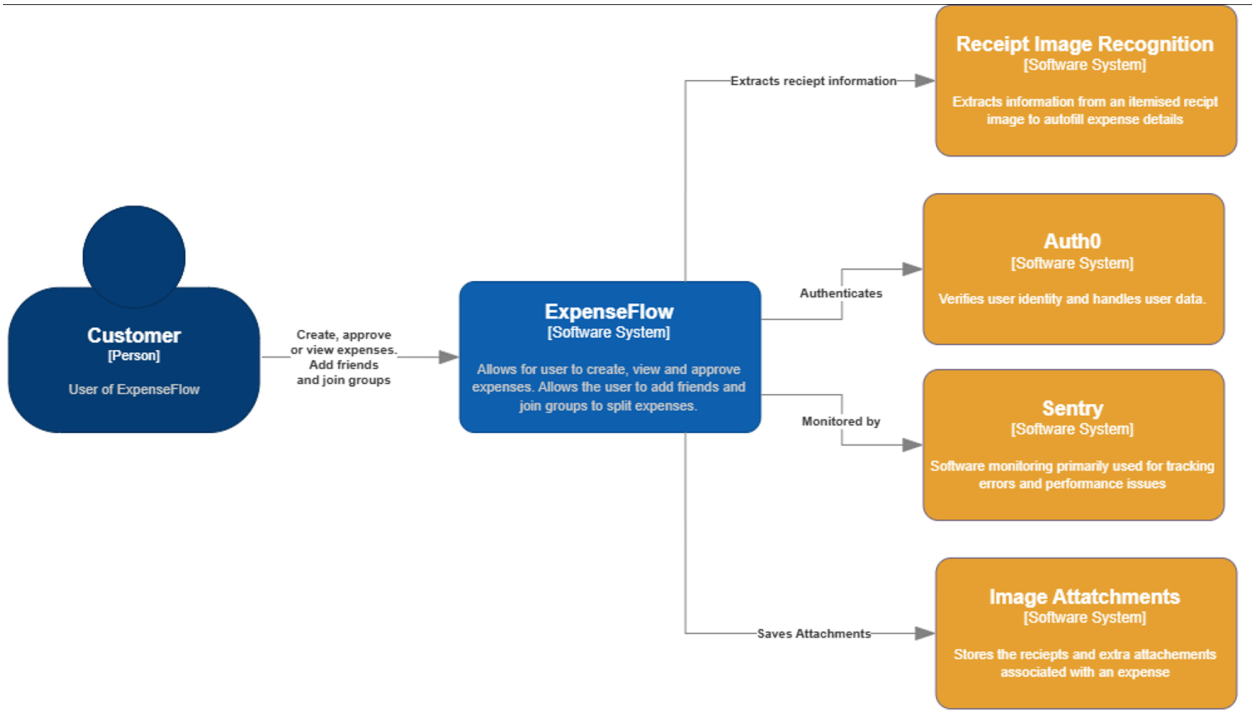
- Performance overhead
- Additional complexity than layered (but still relatively simple)

Architecture

Architecture Description

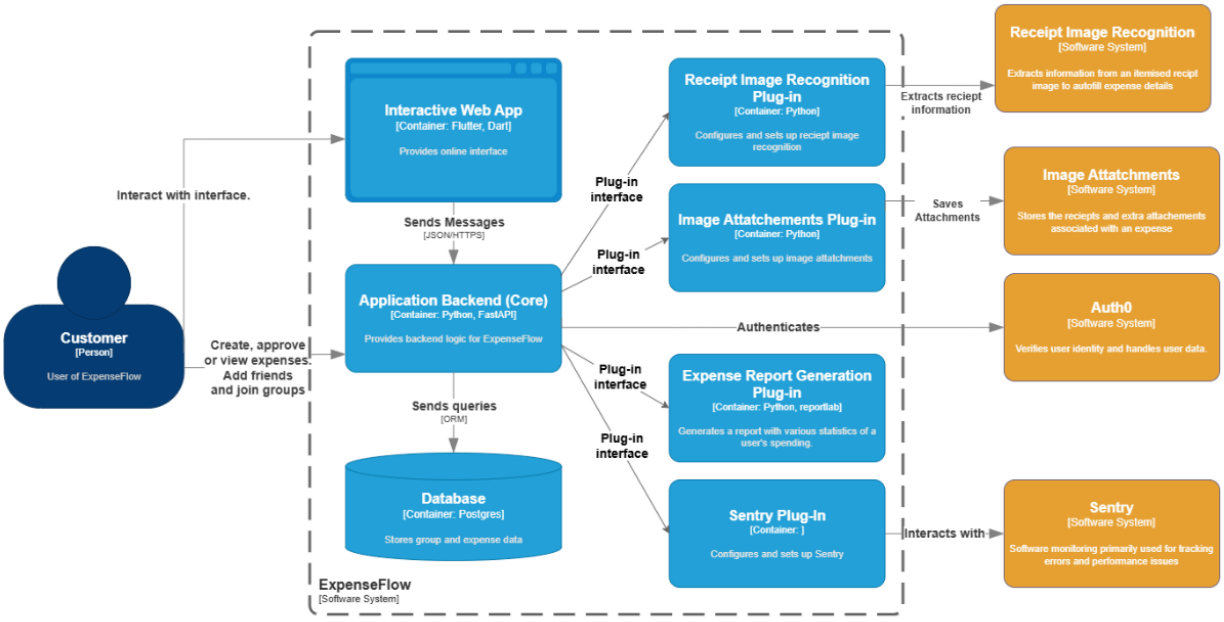
ExpenseFlow uses a Microkernel Architecture distributed over technical partitions.

Users interact with ExpenseFlow to create, approve or view expenses as well as be able to add friends and join groups to split expenses with. Users are authenticated through an external service: Auth0, which stores their personal details securely. This service will also audit and log user login attempts. Sentry is used to monitor system errors and performance. There are also external services for image recognition of a receipt to automatically create an expense as well as an image attachments service to store images used for creating an expense.

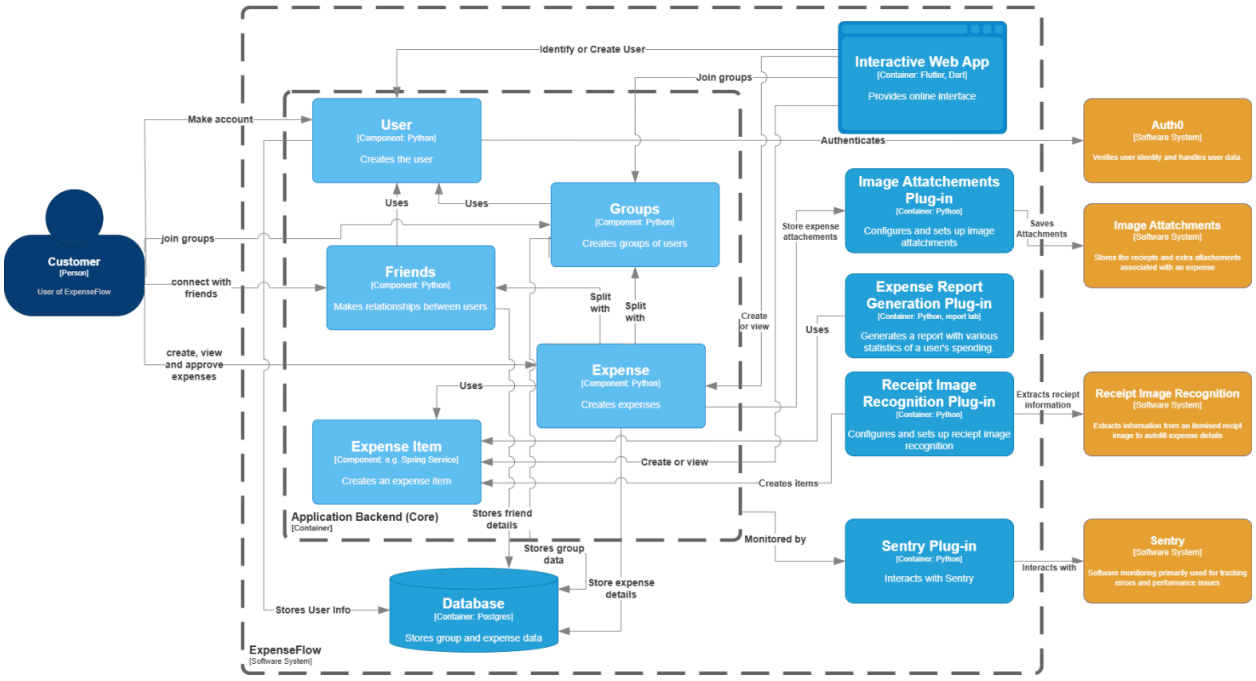


This C2 diagram highlights the plug-ins and how they interact with external services. The architecture is also distributed by technical partitions. ExpenseFlow must have a user friendly interface and the ability to store expense, friend and group data. As such, technical partitioning made the most sense for the project. The presentation layer is a single-page application (SPA), which uses the Flutter framework, and is communicated with over HTTPS. The application backend (RESTful API) contains the business and persistence layers. This backend then uses an object relational mapper (ORM) to communicate with the database layer. Since we are using an SPA as our presentation layer, the communication between the presentation and business layer is handled by the client's browser, and therefore, all calls to the business layer (API) are done over HTTPS.

The use and tradeoffs of using HTTPS and an ORM and how they support the quality attributes are discussed in the tradeoff section.

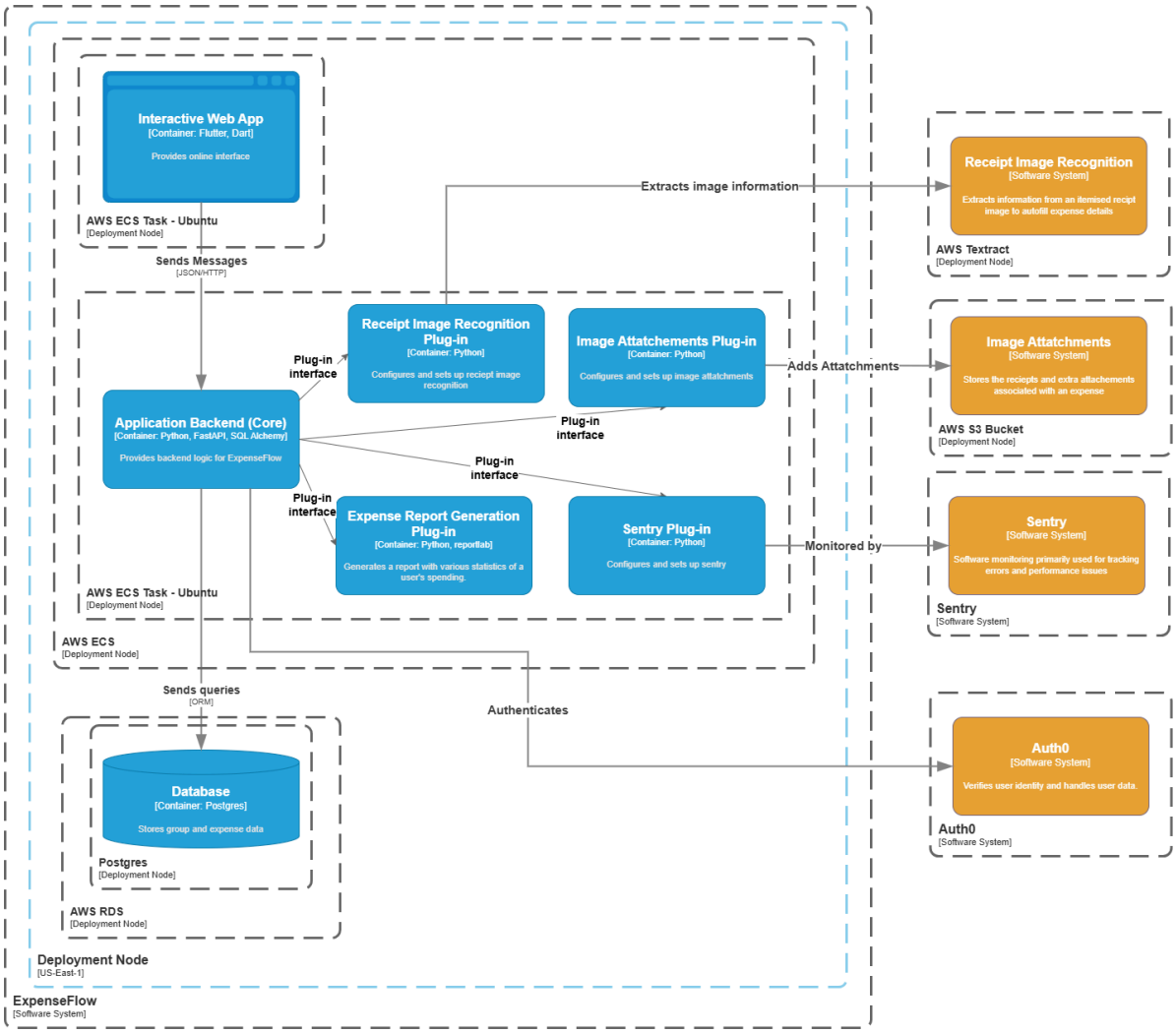


The kernel of the system handles the core functionality which is crucial for the app to function. The kernel is minimal and stable. Additional functionality can be added via a plug-in. Each plug-in shares a common interface (more on this later). Currently, the plug-ins are used for auto filling expenses from an image of a receipt, attaching various attachments to expenses, generating expense reports and for setting up Sentry (monitoring tool). Each of these have their own plug-in that interacts with the external service. A deployment diagram has been produced to highlight this.



This is a more detailed view of the architecture which highlights different components of the application backend. We chose to separate the code with domain partitioning. Each domain is a main component or

feature of ExpenseFlow including Users, Friends, Groups and Expenses. Each of these domains have their own business and persistence layer within which integrate with the backend. This choice was made for several reasons. Firstly, adding or changing functionality to the backend often only required changes to one folder/module. This is different to technical partitioning where the addition of a feature (e.g., friends) may require changes in a routes, services and schema folders, which is not extensible. Additionally, having all of the code in one place for a single feature makes it easier for developers to navigate the system enhancing simplicity.



The data layer is a single AWS Relational Database System (RDS) instance which uses PostgreSQL as its database engine. Both the web app (presentation layer) and application backend (business and persistence layer) are deployed as tasks within AWS ECS services. Separating them out into different services and containers allows for stricter traffic ingress and egress rules, reducing the app's attack surface, improving security. The plug-in interface and plug-in system are a part of the application backend. The receipt image recognition and image attachment plug-in use AWS Textract and AWS S3 Buckets respectively. Communication between these plug-ins and AWS services are done through the AWS network and are authenticated using IAM roles to increase security.

All components in the application's infrastructure are deployed within a virtual private cloud (VPC) network and access is managed with security groups. To ensure that application data is secure, public

network access to the database is disabled, with access restricted to the backend API only (done through security groups). The API and user interface (UI) are deployed on ECS, with their containers only accessible through their respective load balancer on container ports 8080 and 3000 respectively. Access to each load balancer is restricted to port 443 over HTTPS. If connections are made to port 80 (default HTTP Port), requests will be redirected and prompted to use HTTPS instead. HTTPS ensures that communication from the browser to both the UI and API is encrypted. This prevents eavesdropping and ensures that no one can alter the data in transit. HTTPS also ensures that users know that they are communicating with the correct API and UI instead of an imposter server.

How the architecture addresses ASRs

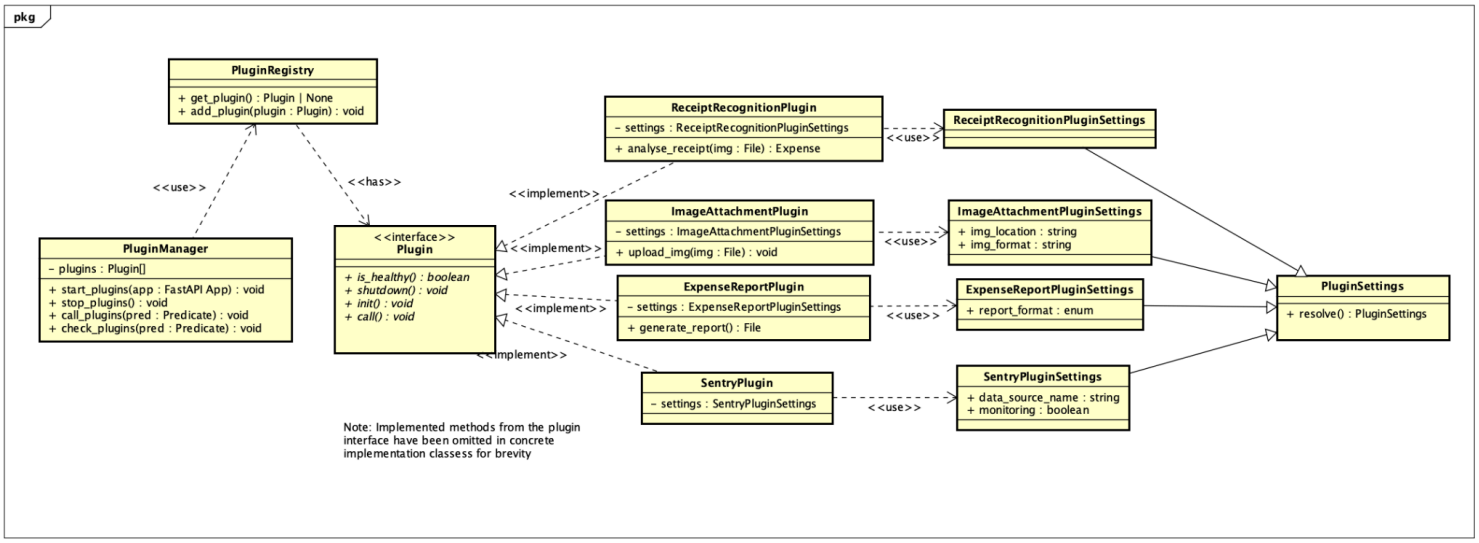
The microkernel architecture supports extensibility by allowing new features to be added as plug-ins. Each plug-in is modular and self-contained, making it easier to extend the system without disrupting existing functionality. This is also beneficial for features that could be added to ExpenseFlow including payment processing with a tool such as stripe, better budgeting tools, letting users upload bank statements to be used for budgeting etc.

Regarding reliability, the microkernel isolates faults within plug-ins so that failures in one part do not bring down the whole system. Furthermore, modularity between features enables more focused testing for each functionality, which helps maintain consistent and error-free operation of the system. Data is also more consistent in comparison to some other architectures such as Event-Driven.

Microkernel architecture also provides high interoperability, which is not a main ASR, but helps to enhance overall system security and extensibility by utilising trusted platforms such as AWS S3, Textract, and Auth0 for storage and authentication. Therefore, security as a main ASR is addressed through integration with secure third-party systems.

Plug-ins

Here is a class diagram of the plug-in system:

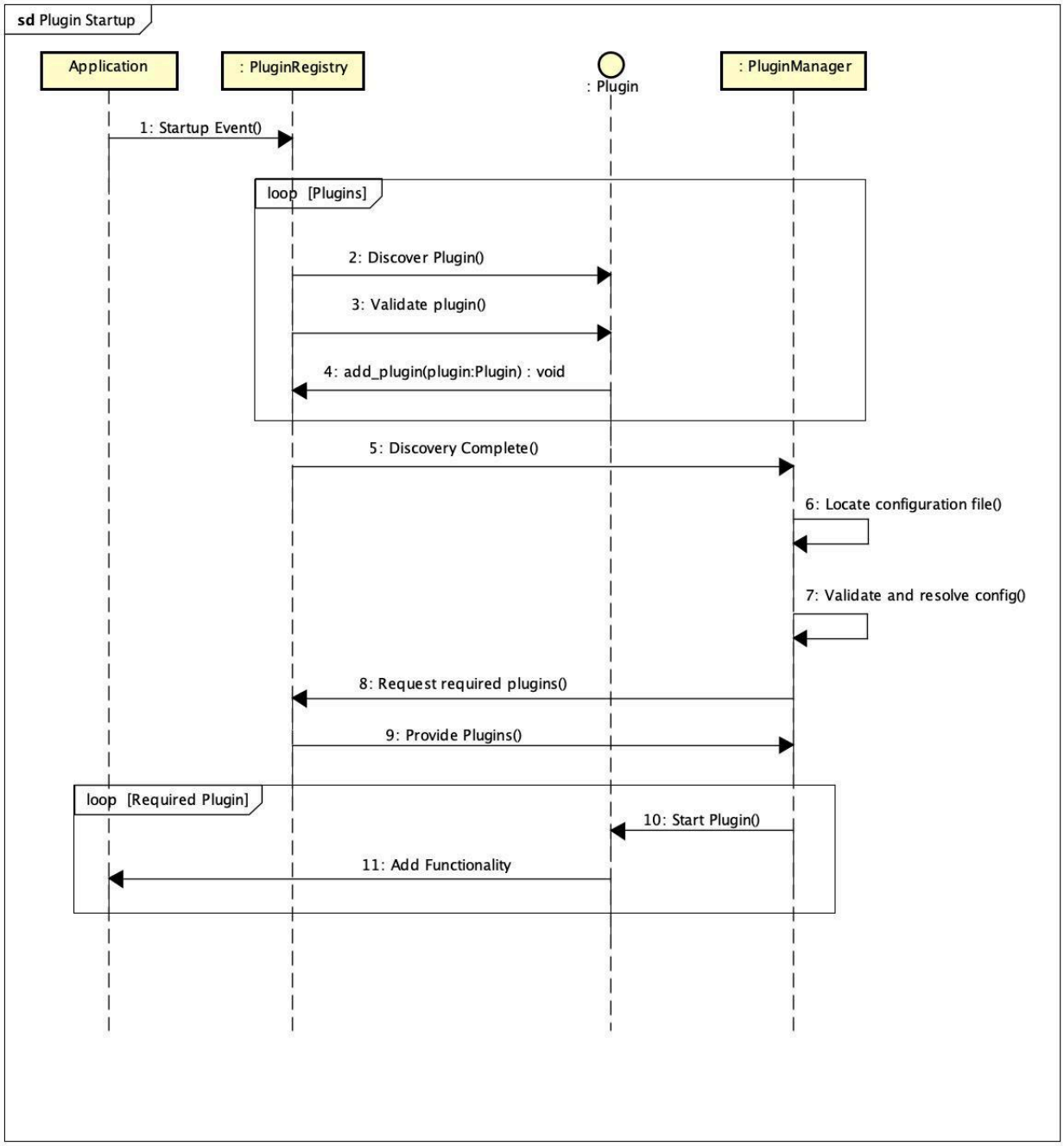


The plug-in interface defines the methods that a plug-in should define for it to be considered a plug-in. This includes: is_healthy(), shutdown(), init() and call(). Using one standard interface reduces complexity

for any additional plug-ins. Each plug-in has its own settings, a subclass of the plug-in settings. There is also the plug-in registry which is a passive catalogue of that plug-ins are available. The plug-in manager searches through this registry. The plug-in manager also manages the lifecycle of the loaded plug-ins with methods including start_plugins(), stop_plugins(), call_plugins() and check_plugins().

All plug-ins are statically loaded. This means that the plug-ins are loaded at application start-up and are included in the program's binary.

A sequence diagram of the plug-in system is provided:

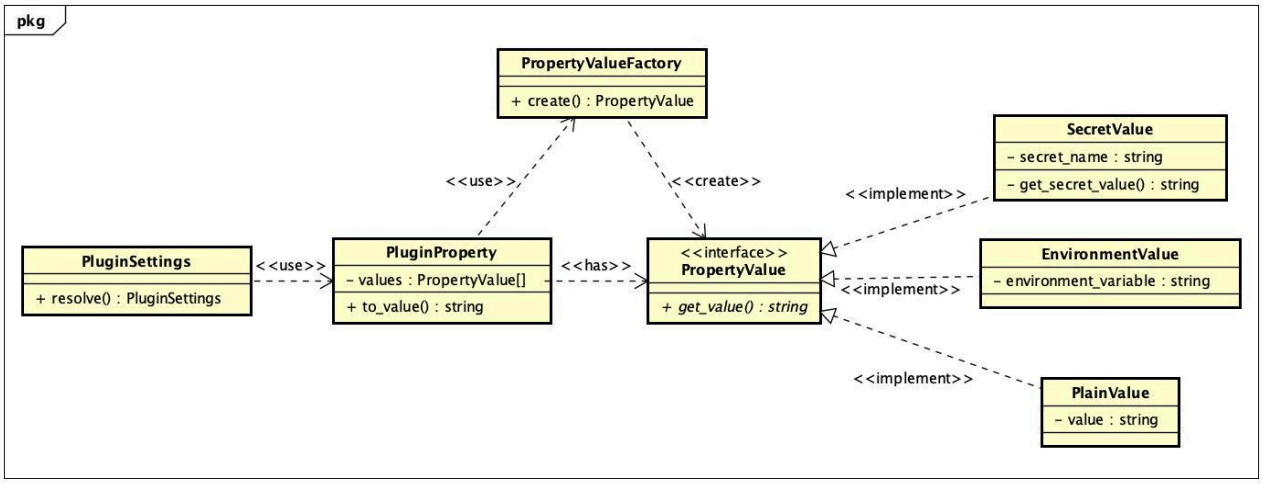


On application start-up, the plug-in registry begins. It then discovers and validates the plug-in ensuring that it satisfies the interface and has a unique identifier. The plug-in manager will then locate a plug-in config file. Once contents of this file are validated and resolved, the manager will query the registry for the specified plug-ins. Once the plug-ins are collected, they can be started and the functionality is added to the app.

As of completion of the code, the config file is:

```
1    textract:
2    sentry:
3      sentry_dsn: "{{env_SENTRY_DSN}}"
4    report_gen:
5    s3_attachment:
6      bucket_name: "expenseflow-attachments"
```

There are static settings such as the s3_attachment bucket_name. However, this is not ideal for settings involving secrets as they would be stored in plain text. Dynamic plug-in settings have been implemented to allow for values to be retrieved from environment variables and a secrets manager.. To specify this, the value must be in double curly braces and prefixed with “env” (as shown above) or “secrets”. This reduced the risk of secret information being leaked enhancing security. This is also more extensible as the system can display different characteristics based on the deployment environment (would include different settings for deployment and production). Note that this is not the same as dynamically loading plug-ins which are discussed in the tradeoffs section.



Each field in the plug-in settings class is a PluginProperty. That PluginProperty is composed of PropertyValues (strings), PluginProperty uses a factory to create these values from the string. The PluginProperty decouples the property from the logic to retrieve the PlainValue, EnvironmentValue or SecretValue.

Secrets

In our application backend, secrets are stored in environment variables. This is normally acceptable however storing these secrets in a secrets manager such as AWS secrets manager is considered more secure. This also allows for secrets to be rotated when necessary without having to redeploy the

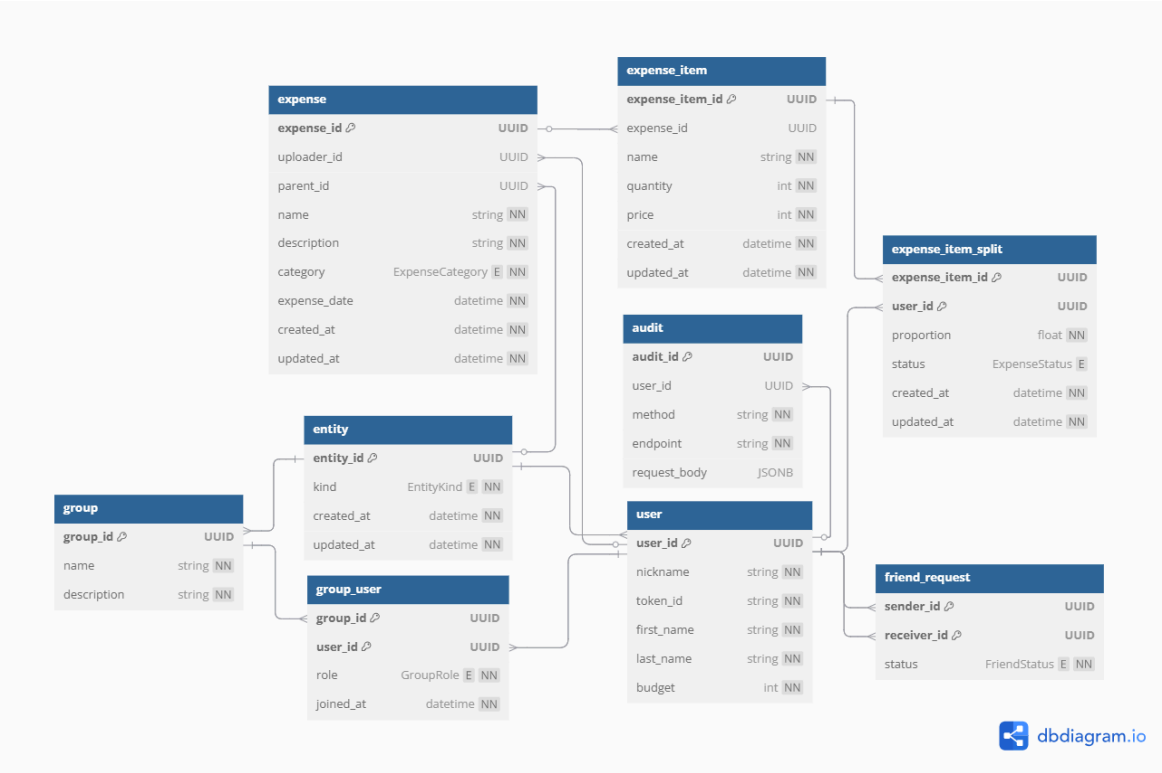
application. However, this creates a large dependency on AWS and changing to a different secrets manager would change the logic about how the secrets are fetched.

If manual intervention steps are required to connect to the db or make changes to Auth0, their respective secrets are stored in AWS secrets manager. This is done during the deployment process.The below snippets show the information being stored in the secret manager.

```
resource "aws_secretsmanager_secret" "auth0_details" {
  name = "auth0-details"

  lifecycle {
    prevent_destroy = true
  }
}

resource "aws_secretsmanager_secret_version" "auth0_details" {
  secret_id = aws_secretsmanager_secret.auth0_details.id
  secret_string = jsonencode(
    {
      domain        = var.auth0_domain
      client_id     = data.auth0_client.expenseflow_ui_client.client_id
      client_secret = data.auth0_client.expenseflow_ui_client.client_secret
      identifier    = auth0_resource_server.expenseflow_api.identifier
    }
  )
}
```

```
resource "aws_secretsmanager_secret" "db_secret" {
  name = "db-secret"

  lifecycle {
    prevent_destroy = true
  }
}

resource "aws_secretsmanager_secret_version" "db_secret" {
  secret_id     = aws_secretsmanager_secret.db_secret.id
  secret_string = var.db_password
}
```

Database Design



The above database schema has been designed with extensibility in mind. Groups and users are subtypes of entities. Reasons for this have been discussed in the tradeoffs section but essentially, this makes the database more extensible as adding say a business can be still treated as an entity. Each expense entry is composed of expense_item entries and an expense item entry is composed of expense_item_split entries. The expense table stores the expense name, the creator of the expense, description, category, and the parent of the expense (i.e., whether it belongs to a user, or group, or business, etc.).

Each expense_item entry is a particular item as per an itemised receipt. If an expense has no items it is considered to be an expense with a single expense_item entry. Each expense_item_split entry within an expense_item shows who is paying for a proportion of an item. This means that an expenses can be split by an overall percentage between group members (e.g., Person 1 owes 20% of the full expense, they would actually owe 20% of each item in the expense), a per item basis (with proportion as 1) or even customise how each item is split between particular group members. There is also a status on each expense_item_split entry. When a user approves an expense, all the items associated with them in that expense will be approved. Although this functionality could be achieved with a separate expense_approval table which would map a user's approval status to an expense, the outlined approach is more extensible, as it could enable users to approve or reject expenses at the item level. This database design ensures that there is a level of granularity to allow flexibility in the design (like splitting the expenses) and also does not contain any duplicate information. This ensures the data is consistent and reliable.

Trade-Offs

General Trade-offs with Microkernel Architecture

There are some general trade offs that were made with the choice of a microkernel architecture. This includes some performance overhead with inter-process communication between components, and additional complexity with the system compared to a layered architecture. The microkernel architecture can introduce challenges in data consistency and reliability between distributed nodes. There can also be some general tradeoffs between extensibility and security, this can include issues with sharing secrets to the plug-ins and the plug-ins can require extra security measures and validation checks.

Static Loading Plug-ins

The plug-ins for ExpenseFlow are statically loaded. This is different to dynamic loading where the plug-ins can be loaded as needed during runtime. This tradeoff is considered in [model/adrs/adr_03_plugin_system.md].

Why have Static loading plug-ins

Statically loaded plug-ins are much more simple and easier to implement. Dynamically loading plug-ins are much more complex and must implement some sort of standard for loading and fetching plug-ins during runtime. Statically loading plug-ins are also more secure than dynamic. Dynamic plug-ins could create vulnerabilities for code injection attacks. Furthermore, malicious plug-ins are forced to be evaluated during compile time (rather than runtime). Statically loading plug-ins are also faster to start up and load and are safer to compile. This is because errors within the plug-in are caught during the execution startup which makes the system more reliable.

Why have Dynamic loading plug-ins

The benefits of using dynamic plug-ins means that they have more flexibility with removing or updating plug-ins without recompiling the app. This also makes the system more modular and the plug-ins have less coupling resulting in better modularity.

Result

Statically loading plug-ins were ultimately chosen. This was mainly due to being easier to implement and time constraints with the project. Further, dynamically loading plug-ins is not really well justified for ExpenseFlow. ExpenseFlow does not require flexibility and changes during runtime. Dynamically loading plug-ins are much more suited to applications such as IDE's, browser and game engines which need to be extensible without recompiling the module.

Object Relational Mapping (ORM)

ExpenseFlow uses an ORM for data access with the database. This decision is documented in [model/adrs/adr_02_db_access_layer.md]

Advantages of an ORM

Using an ORM abstracts SQL and feels more similar to standard python programming rather than changing to the declarative nature of SQL. An ORM also means that developers do not need to have a strong knowledge of SQL. The use of an ORM is considered more secure as it automatically parameterised queries internally which helps enforce protection against SQL injection attacks.

Drawbacks of an ORM

An ORM has slightly more overhead and under the surface it converts to SQL anyways to interact with the database. An ORM can be slightly less flexible with complex queries, particularly those evolving complex joins. Using an ORM can also be harder to debug and it creates higher coupling between the back end and database.

Result

There are minimal differences between the two options. ExpenseFlow does not need to worry about the small additional overhead that an ORM can create and primarily basic CRUD operations are used so the flexibility of SQL is not really beneficial. The benefits of using an ORM are that it enforces tighter security and code readability. Furthermore, more group members are familiar with ORM than SQL.

Authentication

Authentication is very important to ExpenseFlow. Users must be identified and validated before they can make any actions. Users should have limited scope of the system such as only being able to see their own expenses. This follows the principle of least privilege and ensures that user information and sensitive financial data is not compromised. For this we have chosen to use Auth0. This is discussed in [model/adrs/adr_01_auth.md].

Benefits of Auth0

Using an authentication provider is much easier than building a comprehensive system yourself. It is more secure with JWT support. Auth0 abstracts away the storage of sensitive user data from ExpenseFlow such as email and passwords. Instead, we only store the username and name of the user. Further, Auth0 offers integration with google MFA enhancing user security. Auth0 can be managed with terraform deployment making it less complex to set up.

Drawbacks of Auth0

THe main drawback of using auth0 is that when catering to business specific requirements, those businesses may want to use a different or their own authentication provider such as Asure Active

Directory. THe use of an auth provider is more work to implement and set up (compared to minimal or not authentication provider). Further building your own authentication system can offer more flexibility however, this can be error prone and is very time consuming for the mvp.

Use of role based authentication

It is ideal to have different levels of role based authentication with Auth0 such as group roles (allowing only users in the group to see group data etc). Although storing this in Auth0 would be more secure, it also creates a stronger dependency on auth0 and would make changing authentication providers much more difficult, decreasing modularity and extensibility. As such, we have decided to implement this locally for the mvp and the group roles are stored in the database.

Other

Choice of Tooling

Tools such as Mypy, Ruff Lint, Ruff Format and UV have been used in developing ExpenseFlow ensuring higher code quality leading to a more reliable system. This is shown in the [model/adrs/adr_04_backend_tooling.md].
The use of the tooling can take time to set up however, the benefits of each tool outweigh this. These benefits include keeping the code readable and up to various PEP standards, reducing typing errors, managing local dependencies.

Entities

The [model/adrs/adr_others.md] talks about how we make groups and users a subclass of entity. Modelling users and groups separately is perhaps more simple and easy but it is less flexible as more constructs are added. This would have created more db tables for expenses potentially leading to inconsistencies and querying the database could require several joins which may impact the reliability of the system if bugs are present. Modelling them as entities is slightly more complex to set up the database but no joins are required, making faster queries and new constructs such as a business account can be easily integrated by also treating it as an entity.

Choice of Front End Interface

Both a web based and mobile app front end were considered for ExpenseFlow. Ideally, for general users (not business), ExpenseFlow will probably be more useful as a mobile app. However, this is more difficult to implement. A web app was chosen for simplicity and time and team constraints for the MVP.

Testing Plan

This testing plan aims to provide information on the tests completed on the project code, as well as tests that would have been completed, had the project been matured for full, professional deployment. The testing suite focused on testing the core functionality using unit tests, as stated in the initial proposal. The testing suite tests both correct/expected input and incorrect/invalid input, to allow the testing to be more rigorous.

How to run the tests

1. In the project root directory, run ./test.sh (this requires docker compose to be installed)

Description of Tests

The tests section is laid out like the following:

Layer/Component

Functionality/Module

- Test details/functionality being tested
    - Tests that test it (if one has been implemented).

Unit Tests & Integration Tests

This section heavily utilises isolated unit testing to verify that features work as intended, and that some new dependency introduced would not break every other feature. As there were some parametrised tests, there are more tests listed in pytest than the number listed as implemented. While implementing unit tests, some tests were written as integration tests, by testing multiple components with dependencies together. These integration tests are coloured in orange, and can be found in `*_routes.py` files of the test suite.

Frontend

Auth

- Correct user can access the data.
- The service retrieves the correct user.

Data display

- Data being displayed is accurately transferred to and from the backend.
- Entered data is being delivered correctly to and from the backend.

UI

- Buttons and widgets correctly render on screen.
- Buttons and widgets correctly link to its respective handler.
- Navigation between pages using links or menus are as intended.

Backend

Audit

- Getting the audits.
  - test_get_audits_empty
  - test_get_audits_with_data
  - test_get_audits_user_isolation
  - test_get_audits_multiple_for_same_user
  - test_get_audits_response_schema
  - test_get_audits_with_none_request_body
  - Test_get_audits_for_user
  - Test_get_audits_empty_for_user
  - test_get_audits_user_isolation
- Making an audit.
  - Test_create_audit
  - Test_create_audit_with_none_request_body
  - test_create_multiple_audits_same_user

Auth

- Getting the correct user token with an identifier.
  - Test_get_user_token_identifier_success
  - Test_get_user_token_identifier_invalid
- Getting the current user.
  - Test_get_current_user_success
  - test_get_current_user_not_found
- JWTClient correctly throws an error.
  - Test_verify_jwk_client_error
- Miscellaneous Verifications.
  - Test_verify_missing_sub
  - test_verify_invalid_signature

Database

- Persistence
- Inserting data
- Correct storage of data according to schemas
- Data integrity

Expense

- Creating an expense
  - Test_create
  - **Test_bad_parent_create**
  - **Test_bad_expense_create**
  - **Test_bad_user_create**
- Updating an expense
  - Test_update
  - Test_nonexistent_update

    - Test_nonexistent_user_split_update
    - Test_bad_split_update
- Seeing an overview of an expense
    - Test_get_empty_overview
    - Test_get_non_empty_overview
- Seeing the status of an expense
    - Test_get_status
    - Test_get_empty_status
    - Test_get_all_status
- Updating the status of an expense
    - Test_update_nonexistent_status
    - Test_update_dup_user
    - Test_update_status
- Change to expense's status is valid
    - Test_is_valid_expense_change

Friend

- Getting friends
    - Test_get_friends
    - Test_get_friend_by_id
- Getting friend requests
    - Test_get_requests
    - Test_accepted_request
- Adding friends
    - Test_create_by_nickname
    - Test_create
    - Test_add_friend
    - Test_fuzz_create_friend
- Removing friends
    - Test_delete
    - Test_remove_friend

Group

- Get groups
    - Test_get_groups
    - Test_get_group (there are two of the same name, one for routes, one for services - i.e. unit and integration testing was completed)
    - Test_get_group_not_found
    - Test_get_user_groups
    - Test_get_user_groups_empty
    - Test_get_user_groups_other_user
    - Test_get_group_users_empty
    - Test_get_group_no_group
- Create groups

    - Test_create_group (there are two of the same name, one for routes, one for services - i.e. unit and integration testing was completed)
    - Test_create_group_no_name
    - Test_create_group_no_description
    - Test_create_group_owner_created
- Update/manage groups
    - Test_update_group (there are two of the same name, one for routes, one for services - i.e. unit and integration testing was completed)
    - Test_update_group_not_found
    - Test_delete_user_not_found
- See users in group
    - Test_get_group_users
    - Test_get_group_users_only_one
    - Test_get_group_user
    - Test_get_group_user_none
- Create roles in groups
    - Test_create_user_role
- Update roles in groups
    - Test_update_user_role
    - Test_update_user_role_invalid_group
    - Test_update_user_role_invalid_role
    - Test_create_update_group_user_role

User

- Get user
    - Test_get_me
    - Test_get_user
    - Test_get_user_invalid_uuid
    - Test_get_user_not_found
    - Test_get_user_by_id
    - Test_get_user_by_id_not_found
- Create user
    - Test_create_user (there are two of the same name, one for routes, one for services)
    - Test_create_user_invalid_body

Plug-ins

Core plug-in interface and registry

- Test_environment_dynamic_value
- Test_environment_dynamic_value_missing
- Test_dynamic_value_factory_env
- Test_plugin_registry_register_and_get
- Test_plugin_registry_duplicate
- Test_plugin_registry_missing_settings_type
- Test_plugin_property_static
- Test_plugin_property_mixed

Core plug-in manager functionality

- Test_plugin_manager_lifecycle
- Test_plugin_manager_strict_missing_plugin

S3 Attachment Plug-in

- Attach and retrieve
    - Test_attach_and_retrieve_valid
    - Test_attach_and_retrieve_invalid
- Integration with core

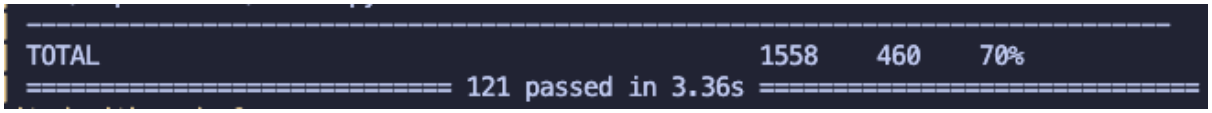
Textract Plug-in

- Integration with AWS Textract service
- Correct recognition
- Handling invalid images
- Handling invalid files
- Transfer to database
- Integration with core

Report generation plug-in

- Correct data
- Correct format
- Integration with core

The listed tests with actual test names instead of test functionality were all passed.



Manual end-to-end Testing

For this section, personas would be created to simulate the user experience. Scenarios would be formulated using use case diagrams, activity diagrams or sequence diagrams, then those diagrams would be followed to simulate the expected behaviour of the persona. From there, we would compare the expected output to the real output. This section mostly focuses on frontend, and the frontend-backend integration. Due to time constraints and scope, manual end-to-end testing was completed without the creation of personas, following expected use cases.

Summary with Requirements

Functional Requirements

- Users
    - MFA, when logging in with google
        - Tested in the *Auth* module of the backend testing.
        - Also tested during manual testing.
- Groups

    - Make groups to split expenses
        - Tested in the *Group* module of the backend testing.
        - Tested in the *Expense* module of the backend testing.
- Friends
    - To split an expense with
        - Tested in the *Expense* module of the backend testing.
    - Includes friend requests
        - Tested in the *Friend* module of the backend testing.
- Expenses
    - Each expense has a category. Can add a description to expenses
        - Tested in the *Expense* module of the backend testing.
    - Split expenses with friends/groups with custom amounts/percentages.
        - Tested in the *Expense* module of the backend testing.
    - Utilise data extraction to autofill expense information (item costs, quantities, etc.) from provided attachments.
        - Tested during manual testing.
        - Also planned - see *Textract Plug-in* section of *Plug-ins* module.
    - Approval process. When an expense is created amongst multiple users, all users must approve the expense before it can be considered “paid”
        - Tested in the *Expense* module of the backend testing.
- Budgeting
    - Basic statistics on app
        - Tested during manual testing.
        - Planned to be tested in the *Frontend - Data Display* section and *Backend - Database* section.
    - A monthly budget
        - Tested during manual testing.
        - Planned to be tested in the *Frontend - Data Display* section and *Backend - Database* section.
    - Can generate a more detailed report
        - Tested during manual testing.
        - Planned to be tested in the *Plug-in - Report Generation* section.

Non-functional Requirements

Security

- Fuzz Tests
    - All tests with input utilise both valid input and fuzzed (invalid) input.
- Auth0
    - Tested in the *Auth* section.
- MFA to log in
    - Tested in the *Auth* section.
- Conduct regular team code reviews to assess any new features added to the system
    - This was not testable through automation, but was completed through Pull Requests on GitHub with reviews mandatory before merging.
- Vulnerability scanning tool
    - Integration with Prowler was completed to scan vulnerabilities, to test for vulnerabilities in the code.
- Security Logging/ Checking / Scanning tool
    - All api actions within the system will be logged to a user
        - Tested in the *Auth* section.
        - Tested during manual testing.
    - Auditing and logging for potential data inconsistencies.
        - Tested in the *Auth* section.
        - Tested during manual testing.
    - Auditing user login and authentication
        - Tested in the *Auth* section.
        - Tested during manual testing.

Reliability

- Sentry - Used for tracking and debugging errors, crashes and performance issues
    - Was made part of the testing pipeline.
- Large testing coverage
    - While could be done through automation, was done manually due to pytest's coverage malfunctioning with async code.

Extensibility

- Plug-in interface
    - Tested in the *Plug-in* section, and all unit tests help to detect if any plug-ins break the entire codebase.
- Implementation of wide range tests to ensure new additions do not break existing code. By running a bash script.
    - All unit tests help to detect if any plug-ins break the entire codebase. The bash script was integrated into GitHub Actions to automate the entire unit/integration testing process.
- Create CI pipelines to ensure that the above tests are run frequently with every system change

Integration to GitHub Actions was completed to automate the entire continuous integration system, and run the unit/integration tests automatically. The results of the tests were used to validate the non-functional requirement of CI pipelines.

Critique and Evaluation

The architecture of ExpenseFlow was chosen and designed to support not only the MVP but the full system. This critique/evaluation assesses the architecture, microkernel with technical partitions, in terms of delivering both functional and non-functional requirements, specifically the three identified ASRs i.e. Security, Reliability, and Extensibility.

The proposed scope for the full system includes support for a wide range of features including incorporating custom business requirements, payment integration with a service (such as stripe), notification alerts, etc. The architecture that has been chosen considers these features that are not in the MVP. As explained previously, the microkernel architecture supports addition of new functionality without any modifications to the core system, due to the system's adherence to the independent plug-in principle and standard interface principle.

ExpenseFlow has security, extensibility and reliability as the main quality attributes of the system. These have been addressed and evaluated through various architectural decisions and tests. Note that many of the tests have already been summarised in the testing plan.

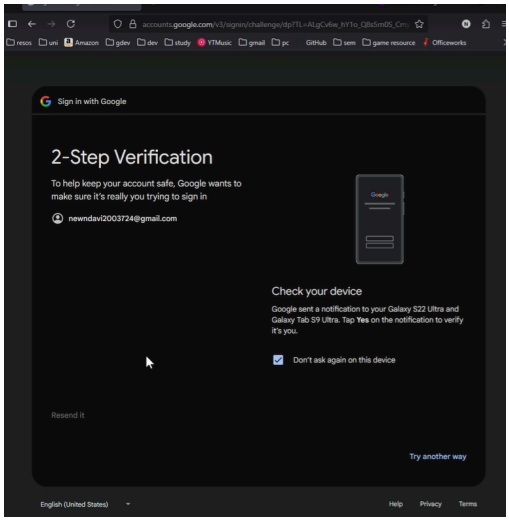
Security

- Fuzz Tests (described in the testing plan)
- Measure success and failure rates for login attempts
- MFA for log in
- Vulnerability scanning tool (Prowler)
- Security Logging/ Auditing

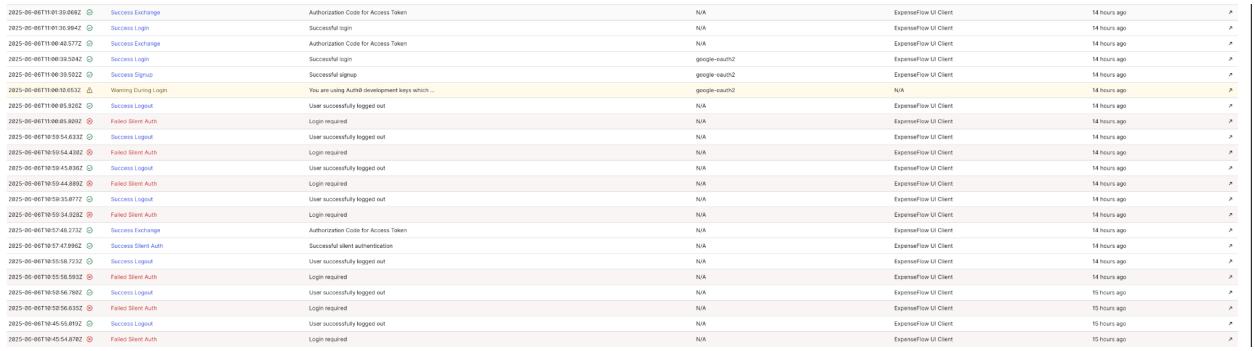
Security is the most critical quality attribute for our system as ExpenseFlow handles sensitive financial information about the users. The microkernel architecture plays a key role in addressing this risk as it helps enforce security by maintaining a minimal core which reduces the system’s attack surface, in line with the defence-in-depth and KISS design principles, and limits the potential impact of any exposure of vulnerabilities to individual components, rather than the whole system.

Auth0

From an implementation perspective, the user authentication and session management was achieved through Auth0, which also supports multi-factor authorisation. To ensure the successful implementation, during development and testing Auth0 audit logs were reviewed to ensure that login failures and unauthorised or suspicious login attempts were correctly recorded and rejected. This increases our confidence in a secure authentication environment. This ensures only the user can see their potentially sensitive financial data enhancing user security and minimising potential attacks. Further, auth0 stored the users personal data such as email and password. This enhances security as user data is not stored in the app.



Auth0 also provides auditing and logging of user login and authorisation attempts. This detects suspicious behaviour, unauthorised access and monitor account misuse. These are crucial to ensuring the app is secure and user data is not mishandled.

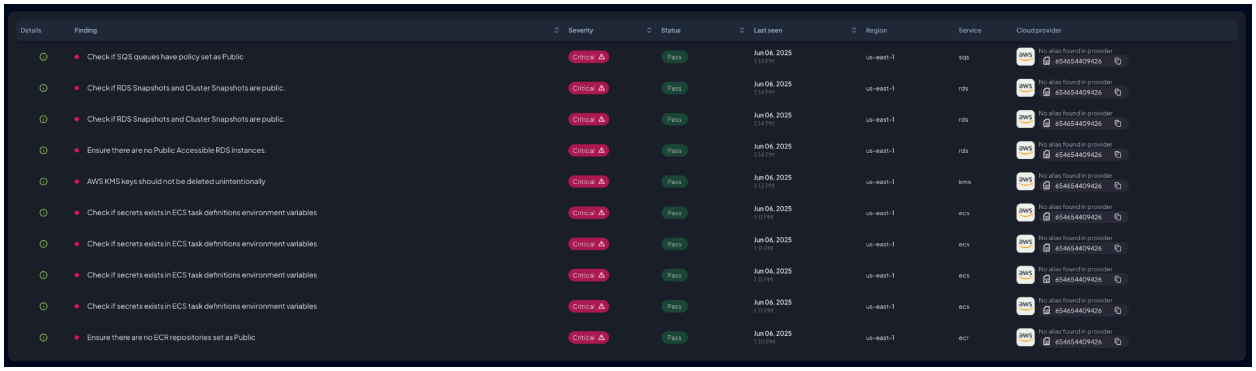


Prowler

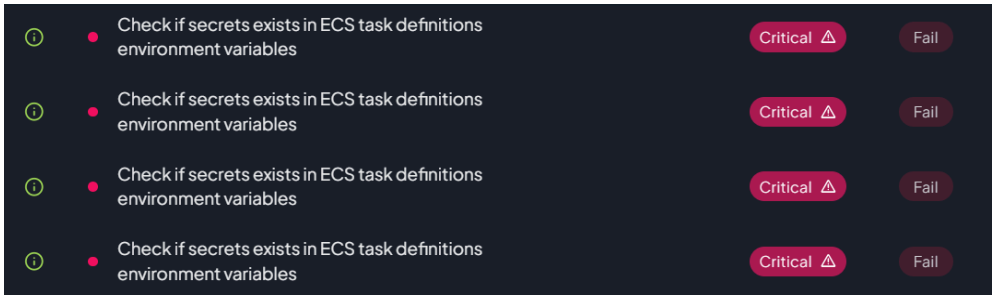
Prowler is a security tool used to identify security vulnerabilities on cloud resources.

Note that the findings here are the number of tests it runs (not the number that fail).

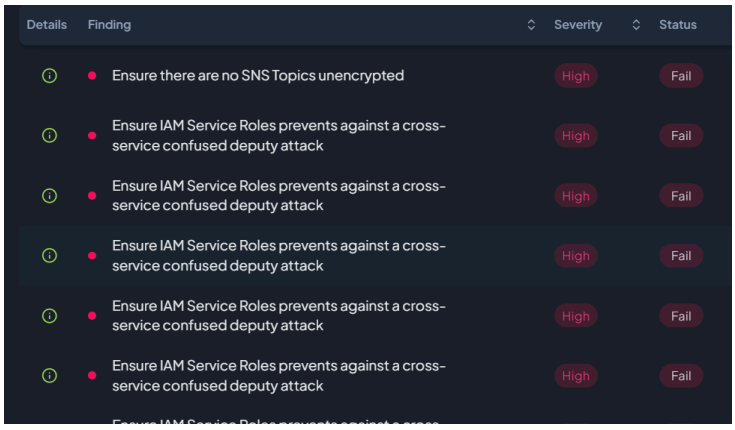
We pass almost all of the 52 critical tests:



There are however 4 critical tests that fail in relation to secrets. This is because our secrets are stored in environment variables. The reason for this has been explained earlier in the report (in the architecture section).



There are various issues that are high-priority but are to do with the AWS account and are out of our control to fix. There is a lot of other stuff in the aws account that prowler raises as vulnerabilities that has nothing to do with this project. There are also some limitations with prowler as we cannot automate or schedule it as we are unable to create IAM roles and users to give to prowler. At this stage we have been using our own session tokens which expire.



Regardless, we still pass the ones that are within our control:

(more not pictured here)

Prowler supports that the ExpenseFlow infrastructure is secure and passes most of the security tests.

Logging/Auditing

We implemented our own logging/ auditing system. This monitors user actions and interactions with endpoints in the api. This includes making expenses, forming groups, approving expenses etc.

The use of an auditing/logging system can help with security and unusual behaviour can be identified.

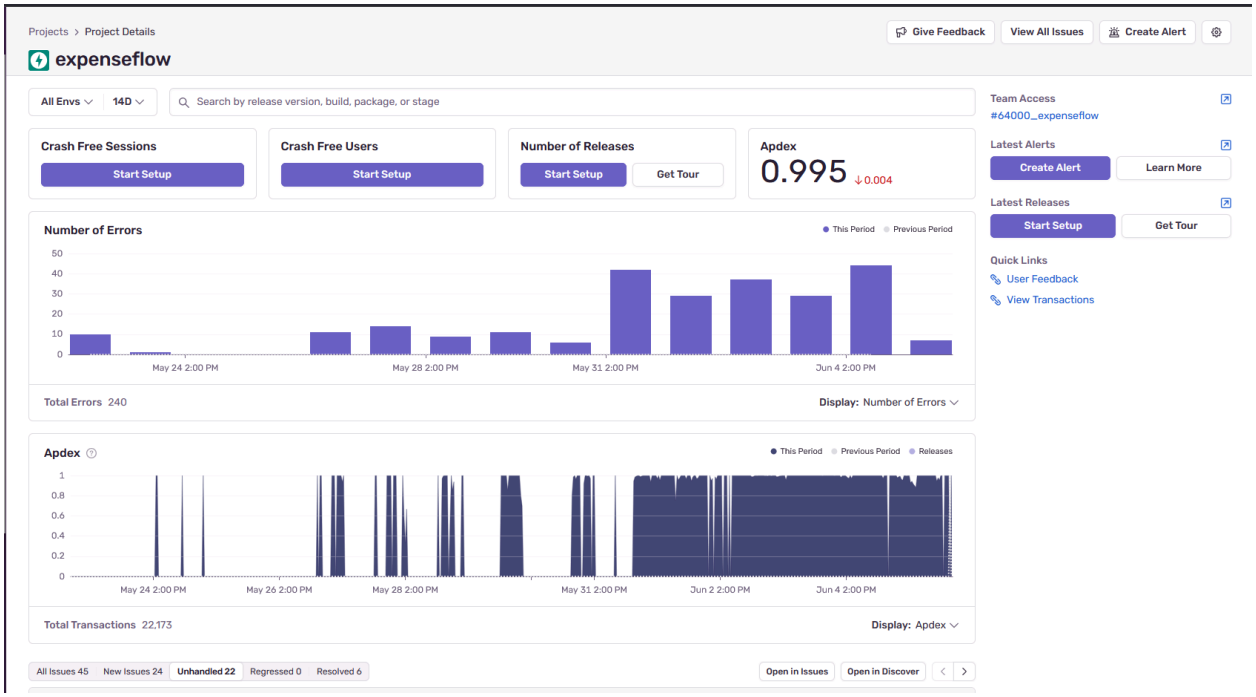
Reliability

- Sentry - Used for tracking and debugging errors, crashes and performance issues
- Conduct regular team code reviews to assess any new features added to the system
- Large testing coverage

The microkernel architecture supports reliability of the system by implementing different features using independent plug-ins. This is one of the key benefits of microkernel as it supports fault isolation which ensures that a failure of a single plug-in does not affect the core system or other plug-ins. In addition to this, the use of a standardised interface helps reduce likelihood of error due to incompatibility between components. Reliability is also improved with technical partitioning separating capabilities into different layers thereby facilitating modular testing.

Sentry

ExpenseFlow uses sentry for runtime error tracking and range of tests to ensure and verify that the system performs consistently for normal and edge-case scenarios.

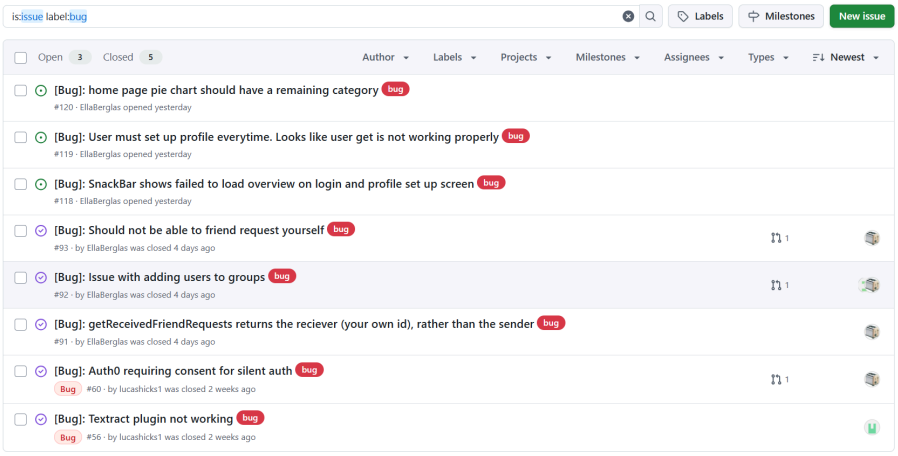


Sentry shows when something in the code base has gone wrong. This can help identify bugs and errors and keeps ExpenseFlow more reliable.

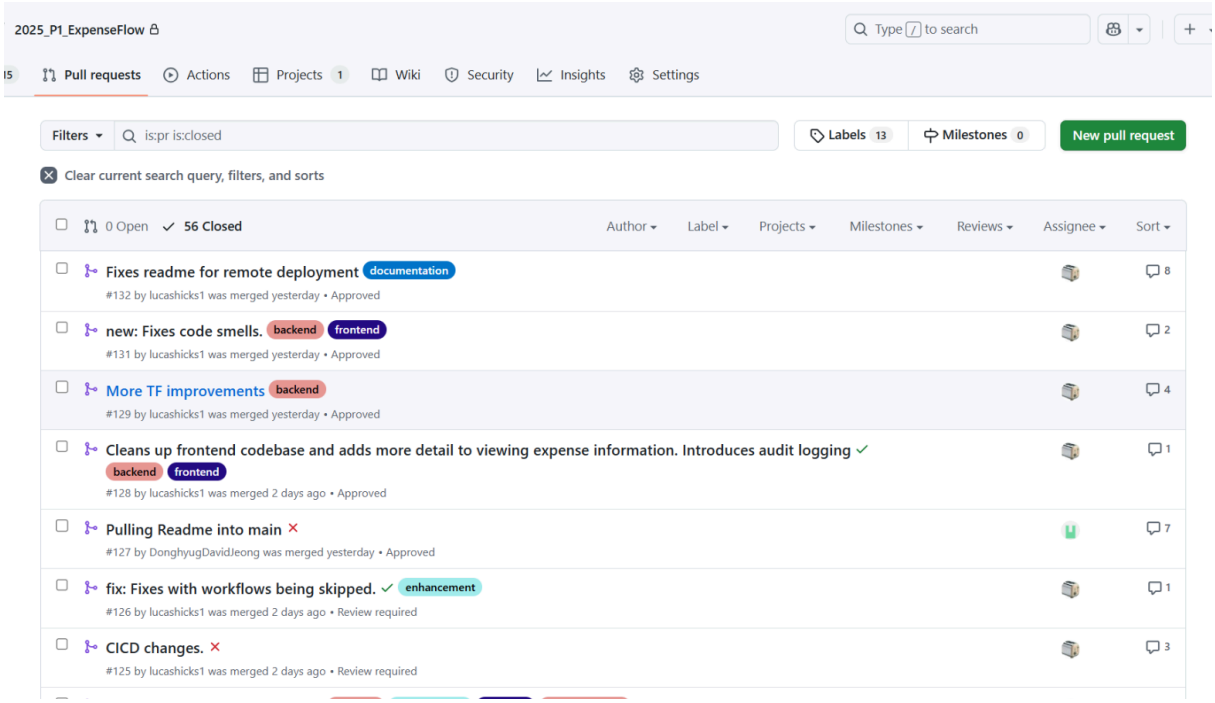
Code Reviews

Through the development of ExpenseFlow, we made effective use of branches, pull requests and issues. Pull requests allowed for the code to be reviewed by everyone before pushing to main. This was enforced through modifying the rules for github pull requests.. This ensures that multiple people have reviewed and run code before it has been pushed to main. This helps reduce the risk of bugs.

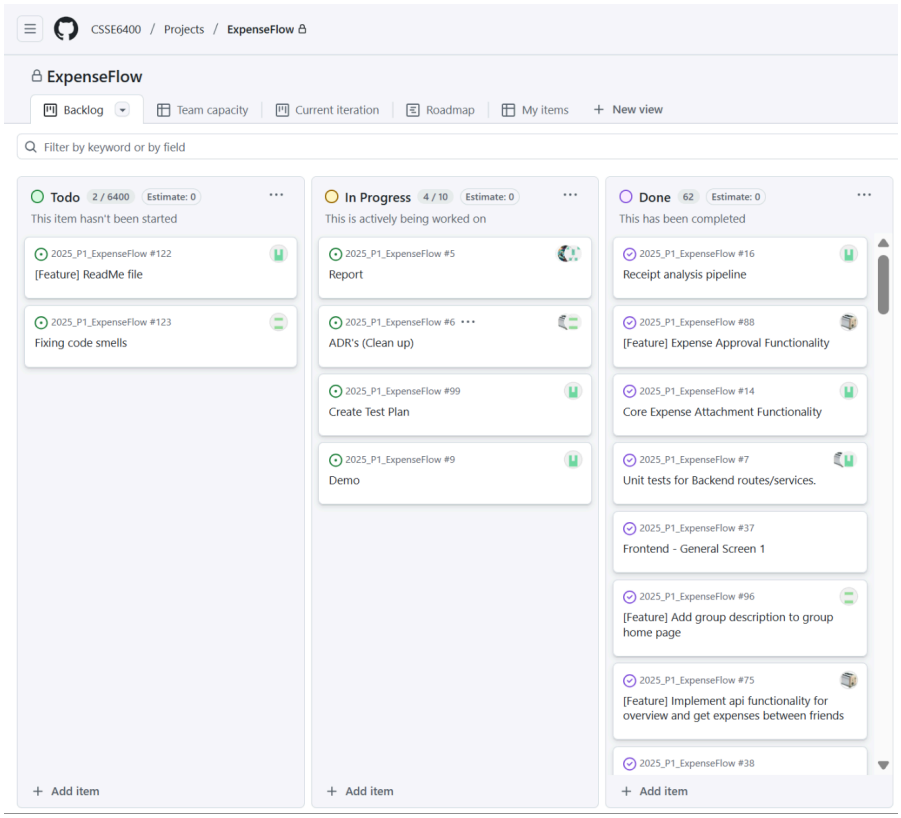
Documenting Bugs on main through GitHub Issues



We had 56 pull requests.



We also used a kanban board and made github issues to assign work to team members.



Testing Coverage

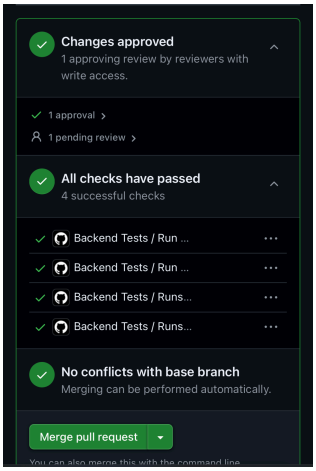
We had a very large test coverage for the code base. Note that this pytest code coverage analysis has a minor bug where it does not detect coverage for some lines of code in async methods after an await has been called. So the testing coverage is actually a bit higher than the image suggests. There is almost 100% text coverage across all of the files. There are 121 tests all of which pass, More details on these can be found in the testing plan.

```
=== 121 passed in 3.36s ====
```

```
Name                                                Stmts   Miss  Cover   Missing
---------------------------------------------------------------------------------
src/expenseflow/__init__.py                             0      0   100%
src/expenseflow/audit/__init__.py                       0      0   100%
src/expenseflow/audit/models.py                        14      0   100%
src/expenseflow/audit/routes.py                        10      0   100%
src/expenseflow/audit/schemas.py                       12      0   100%
src/expenseflow/audit/service.py                       13      0   100%
src/expenseflow/auth/__init__.py                        0      0   100%
src/expenseflow/auth/deps.py                           34      2    94%   55-56
src/expenseflow/auth/service.py                        34     12    65%   38-39, 44-61
src/expenseflow/config.py                               9      0   100%
src/expenseflow/database/__init__.py                    0      0   100%
src/expenseflow/database/base.py                       19      2    89%   23-24
src/expenseflow/database/core.py                       17      9    47%   35-43
src/expenseflow/database/deps.py                        4      0   100%
src/expenseflow/database/mixins.py                      7      0   100%
src/expenseflow/database/service.py                    11      0   100%
src/expenseflow/entity/__init__.py                      0      0   100%
src/expenseflow/entity/models.py                       13      1    92%   27
src/expenseflow/entity/schemas.py                       6      0   100%
src/expenseflow/entity/service.py                       5      0   100%
src/expenseflow/enums.py                               35      0   100%
src/expenseflow/errors.py                              11      0   100%
src/expenseflow/expense/__init__.py                     0      0   100%
src/expenseflow/expense/models.py                      41      0   100%
src/expenseflow/expense/routes.py                      75     28    63%   45, 52, 66, 77-91, 106, 119-124, 133-138, 147-155, 170-177
src/expenseflow/expense/schemas.py                     65      0   100%
src/expenseflow/expense/service.py                    136     55    60%   55, 69-70, 73-123, 182, 212-236, 299-316, 353-358, 395, 417-436, 495, 502
src/expenseflow/friend/__init__.py                      0      0   100%
src/expenseflow/friend/models.py                       14      0   100%
src/expenseflow/friend/routes.py                       53     13    75%   50-55, 63-71, 80-85, 93, 105
src/expenseflow/friend/schemas.py                       7      0   100%
src/expenseflow/friend/service.py                      44      5    89%   46-78
src/expenseflow/group/__init__.py                       0      0   100%
src/expenseflow/group/models.py                        26      0   100%
src/expenseflow/group/routes.py                        83     38    54%   74, 81-86, 101-106, 115-122, 131-153, 162-188, 197-203
src/expenseflow/group/schemas.py                       26      0   100%
src/expenseflow/group/service.py                       76     31    59%   47-75, 148-149, 152-153, 158-161, 192-215
src/expenseflow/main.py                                37      8    78%   25-33, 57, 63
src/expenseflow/middleware.py                          15      4    73%   31-36
src/expenseflow/plugin/__init__.py                    168     15    91%   33, 107-108, 225-228, 231, 234-235, 254, 276, 284-287
src/expenseflow/plugin/report_gen.py                  151    122    19%   35, 46, 50-53, 59-67, 75-98, 109-111, 115-167, 176-303
src/expenseflow/plugin/s3_attachment_plugin.py         87     57    34%   42-72, 83, 90-94, 103-111, 117-136, 140-151, 165-198
src/expenseflow/plugin/sentry.py                       19      2    89%   31, 52
src/expenseflow/plugin/textract.py                     64     41    36%   33-34, 47, 61-127
src/expenseflow/schemas.py                              7      0   100%
src/expenseflow/user/__init__.py                        0      0   100%
src/expenseflow/user/models.py                         18      1    94%   13
src/expenseflow/user/routes.py                         36      8    78%   38-39, 47-48, 56, 77-78, 87
src/expenseflow/user/schemas.py                        21      0   100%
src/expenseflow/user/service.py                        29      6    79%   20, 45, 49-50, 67-68
src/expenseflow/utils.py                                6      0   100%
---------------------------------------------------------------------------------
TOTAL                                                1558    460    70%
```

Extensibility

- Plug-in interface
- Implementation of wide range tests to ensure new additions do not break existing code.
- Create CI pipelines to ensure that the above tests are run frequently with every system change



Extensibility is one of the main benefits for choosing microkernel architecture. As described above, the plug-ins support addition of new features without requiring additional changes to the existing codebase. This ensures that a new plug-in can be developed and tested in isolation. To ensure any new additions to the casebase didn't break any existing code, a CICD pipeline was created to run a test workflow after every pull request to main. More details on these can be found in the testing plan section of the report.

It is difficult to test for extensibility but many details of the plug-in system and plug-in interface have been provided in the Architecture section of this report. It is by design of the microkernel architecture that it is extensible.

Reflection

No one in the team had experience with microkernel architecture before the project. As such, it was not really known how to best implement the kernel and the plug-in system. There is a considerable jump between understanding a diagram of a microkernel architecture and the independent plug-in and standard interface principles and actually implementing it all. There were many more considerations involved including with static vs dynamically loaded plug-ins, the various settings and integrating the environment variables and secrets, as well as implementing the plug-in registry and manager. Much of the group has a much deeper understanding of a microkernel architecture as a result of this project. The architecture was the perfect balance of complexity and learning for the project while also being simple enough to implement for the mvp.

Upon some reflection, it may have been beneficial to have more considered or used a microservice architecture (especially for the full implementation of the scope). The microkernel architecture was the best suited for our selected quality attributes of extensibility, reliability and security. However, as specified in the proposal, there are a few more quality attributes that probably could have been considered. These include:

- Availability - users (especially business users) may need to use the system across many different times zones and devices, demanding the tool to be available with minimal downtime
- Interoperability - many companies use multiple different planning tools and therefore, for ExpenseFlow to be usable in business environments, it should be able to be easily integrated. This was not deemed to be as important as it is primarily focused on business use cases.
- Scalability - As more and more users join the platform, the system must be able to handle high amounts of load.

If these attributes were to be more heavily considered, it is possible a microservices architecture should have been chosen.

Unlike systems such as IDE's, browsers or game engines, a microkernel architecture and the use of plug-ins (especially dynamically loaded plug-ins) is not essential for ExpenseFlow. As per the scope of the MVP, there are only 4 plug-ins currently used. It is possible that more of the kernel could have been extracted out into their own plug-ins. This would have reduced the attack surface and had benefits with security and reliability. However, it was decided that the functionality currently residing in the kernel is crucial for the main system functionality.

The project would have benefitted from more planning and catering to each team member's strengths. To start, we did not have a very clear idea of what the scope of the mvp was. To exacerbate this, the mvp changed multiple times throughout the project. This led to some confusion between team members as to what the scope of the work was and the code became inconsistent (e.g. approvals implemented in back end but not the front end etc). This also led to some delays in the coding cut-off date which led to delays in the documentation etc. It would have been more cohesive to have pre planned the scope of the project (and had it well documented). It would have also been beneficial to have established the end points that would be needed for the front and back end integration which could have provided a guide for the front end developers.

There also should have been more communication at the beginning of the project as to the strengths and weaknesses of each team member and for where they were best suited to contributing to the project. This is also relevant for the tools and frameworks. For example, flutter was chosen for developing the front end as one group member was familiar with it, however, more group members were actually familiar with react. This resulted in more group members having to learn flutter instead of just one person learning react.