Kristianstad University
SE-291 88 Kristianstad
Sweden
+46 44 250 30 00
www.hkr.se

Pineapple Planner

Agile Development Project Report

Max Sellick, Varvara Aladyina, Deinoras Krasauskas, Azhaf Kahn, Simon Ostini

March 2025

Title
Pineapple Planner

Subtitle
Agile Development Project Report

Programme
Software Development

Authors
Max Sellick, Varvara Aladyina, Deinoras Krasauskas, Azhaf Khan, Simon Ostini

Keywords
Agile, Scrum, Project idea

Contents

Plots

Listings

Tables

1 Introduction

PineapplePlanner is a Windows application designed to serve as an intuitive to-do list integrated within a calendar. PineapplePlanner enables users to create and manage tasks efficiently by specifying a start and finish date, selecting a priority level (high, medium, low, or no priority), and providing a descriptive task name before submitting it to the calendar. With a user-friendly interface and AI-powered assistance, PineapplePlanner simplifies task creation by allowing users to generate tasks through natural language input—simply describing what they need to do, and the AI will automatically create and categorize the task for them. For those who prefer a more hands-on approach, PineapplePlanner also allows users to manually input their tasks, giving them full control over their scheduling. This makes the process even more seamless for individuals who feel overwhelmed by a heavy workload or numerous responsibilities, offering a structured and organized approach to task management. By visually mapping out tasks within a calendar, users can prioritize effectively, track deadlines, and stay on top of their commitments with greater ease and clarity. Whether users need to focus on urgent tasks or simply keep track of general activities, PineapplePlanner provides the flexibility to accommodate different levels of importance.

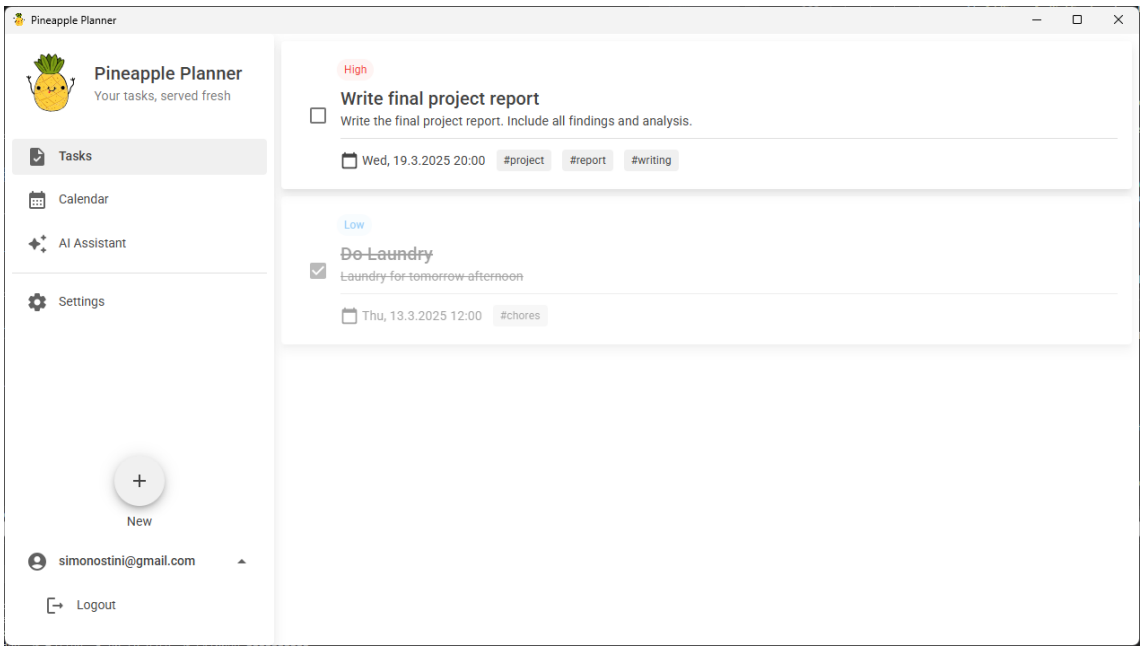

Figure 1: Pineapple Planner home page screenshot

1.1 Restrictions

1.2 Enhancements

The possible extra features we planned to implement or that has been implemented was custom usernames for users which was already implemented, dark-mode option in settings which transforms the application to dark-mode for users who would prefer to have it that way. AI was another feature that got implemented to help users create tasks faster. Not yet implemented features were Notification features

in settings that would allow users to enable/disable notifications, Language option in settings for users whom do not understand english or would rather prefer have it in another language.

2 Requirements

Nr	Req. Name	Req. Description	EMH	AMH	Prio
1	Log in	User is able to log in to see their personal tasks.	5	-	10
2	Log out	User is able to log out to secure their information	1	-	10
3	Task creation	The project should create a task with a name, description and a deadline	7	-	10
4	Edit task	The task should be editable and could be marked as completed.	2	-	8
5	Task prioritization	Task should have priorities: low, medium and high.	3	-	6
6	View tasks in the calendar	Task should be visible in calendar view.	10	-	10
7	Add tasks in the calendar	There should be a representation of creating a task in the calendar	5	-	6
8	Tags	Tasks should have tags for easier management.	4	-	5
9	Dark theme	There is an ability to set dark theme in the settings	5	-	3
10	Tags	Task can have tags for easier search	8	-	4
11	AI	There is an ability to ask Gemini to create a task or event	10	-	3

Table 1: Requirements

3 Design and Implementation

The application implementation is far to large to list and explain every single class. As the application is built with the object oriented language C# almost everything is a class. Thus, an explanation class-per-class does not really add any value. Some crucial classes with different use cases are explained in the following sections.
The entire project is contained in one .NET solution. This solution is divided into different projects for organization purposes. For instance, we have a class library project for all user interface related code, a class library for communication with the database and a class library for all entity classes and enums.

3.1 PineapplePlanner.Domain.Entities.Entry

The `Entry` class is our base entity class for user entries. There are two child classes `Task` and `Event` that are derived from `Entry`.

The class itself has a `[FirestoreData]` attribute which enables instances to be stored in the Firebase Firestore database. All properties with a `[FirestoreProperty]`

are marked to be stored in the Firestore.

```
1 using Google.Cloud.Firestore;
2 using PineapplePlanner.Domain.Enums;
3 using PineapplePlanner.Domain.Interfaces;
4 using PineapplePlanner.Domain.JsonConverter;
5 using System.Text.Json.Serialization;
6
7 namespace PineapplePlanner.Domain.Entities
8 {
9   [FirestoreData]
10  public class Entry : IBaseFirestoreData
11  {
12    [FirestoreProperty]
13    public string Type { get; } = nameof(Entry);
14
15    [FirestoreProperty]
16    public required string Id { get; set; }
17
18    [FirestoreProperty]
19    public required string Name { get; set; }
20
21    [FirestoreProperty]
22    public string? Description { get; set; }
23
24    [FirestoreProperty("Priority")]
25    public string? PriorityString
26    {
27      protected get => Priority?.ToString();
28      set
29      {
30        if (!string.IsNullOrEmpty(value) && Enum.TryParse<
    Priority>(value, true, out var parsedPriority))
31        {
32          Priority = parsedPriority;
33        }
34        else
35        {
36          Priority = null;
37        }
38      }
39    }
40
41    [JsonConverter(typeof(PriorityEnumConverter))]
42    public Priority? Priority { get; set; }
43
44    [FirestoreProperty]
45    public DateTime CreatedAt { get; set; }
46
47    [FirestoreProperty]
```

```
48      public DateTime? DeletedAt { get; set; }
49
50      [FirestoreProperty]
51      public List<Tag> Tags { get; set; } = new List<Tag>();
52
53      [FirestoreProperty]
54      public string? UserUid { get; set; }
55
56      public Entry() { }
57
58      protected Entry(string type)
59      {
60        Type = type;
61      }
62    }
63 }
```

Listing 1: PineapplePlanner.Domain.Entities.Entry

3.2 PineapplePlanner.Application.Repositories.BaseRepository

`BaseRepository` is a generic class that implements the `IBaseReposito`r according to the common Repository pattern. All other Repository are derived from the `BaseRepository` which has shared CRUD implementations. The `BaseRepository` accepts a generic argument which decides what Firestore collection should taken into account.

```
1 using Google.Cloud.Firestore;
2 using PineapplePlanner.Application.Interfaces;
3 using PineapplePlanner.Domain.Interfaces;
4 using PineapplePlanner.Domain.Shared;
5 using PineapplePlanner.Infrastructure;
6
7 namespace PineapplePlanner.Application.Repositories;
8
9 public class BaseRepository<T> : IBaseRespository<T> where T
      : IBaseFirestoreData
10 {
11   protected readonly string _collectionName;
12   protected readonly FirestoreService _firestoreService;
13
14   public BaseRepository(FirestoreService firestoreService)
15   {
16     _firestoreService = firestoreService;
17     _collectionName = typeof(T).Name + "s";
18   }
19
20   public async Task<ResultBase<List<T>>> GetAllAsync()
21   {
```

```
22      ResultBase<List<T>> result = ResultBase<List<T>>.Success
        ();
23
24      try
25      {
26          QuerySnapshot snapshot = await _firestoreService.
        FirestoreDb
27              .Collection(_collectionName)
28              .GetSnapshotAsync();
29          List<T> documents = snapshot.Documents.Select(doc =>
        doc.ConvertTo<T>()).ToList();
30
31          return new ResultBase<List<T>>(documents);
32      }
33      catch (Exception ex)
34      {
35          result.AddErrorAndSetFailure(ex.Message + ex.
        StackTrace);
36      }
37
38      return result;
39  }
40
41  public async Task<ResultBase<T?>> GetByIdAsync(string id)
42  {
43      try
44      {
45          DocumentReference docRef = _firestoreService.
        FirestoreDb.Collection(_collectionName).Document(id);
46          DocumentSnapshot snapshot = await docRef.
        GetSnapshotAsync();
47          T? document = snapshot.Exists ? snapshot.ConvertTo<T
        >() : default;
48
49          return ResultBase<T?>.Success(document);
50      }
51      catch (Exception)
52      {
53          return ResultBase<T?>.Failure();
54      }
55  }
56
57  public virtual async Task<ResultBase<T>> AddAsync(T entity
        )
58  {
59      ResultBase<T> result = ResultBase<T>.Success();
60
61      try
62      {
```

```
63          entity.Id = Guid.NewGuid().ToString();
64          DocumentReference docRef = _firestoreService.
        FirestoreDb.Collection(_collectionName).Document(entity.
        Id);
65          await docRef.SetAsync(entity);
66
67          result.Data = entity;
68        }
69        catch (Exception ex)
70        {
71          result.AddErrorAndSetFailure(ex.Message);
72        }
73
74        return result;
75      }
76
77      public virtual async Task<ResultBase<T>> UpdateAsync(T
        entity)
78      {
79        ResultBase<T> result = ResultBase<T>.Success();
80
81        try
82        {
83          DocumentReference docRef = _firestoreService.
        FirestoreDb.Collection(_collectionName).Document(entity.
        Id);
84          await docRef.SetAsync(entity, SetOptions.Overwrite);
85
86          result.Data = entity;
87        }
88        catch (Exception ex)
89        {
90          result.AddErrorAndSetFailure(ex.Message);
91        }
92
93        return result;
94      }
95
96      public async Task<ResultBase> DeleteAsync(string id)
97      {
98        try
99        {
100         DocumentReference docRef = _firestoreService.
        FirestoreDb.Collection(_collectionName).Document(id);
101         await docRef.DeleteAsync();
102
103         return ResultBase.Success();
104       }
105       catch (Exception)
```

```
106     {
107         return ResultBase.Failure();
108     }
109   }
110 }
```

Listing 2: PineapplePlanner.Application.Repositories.BaseRepository

3.3 PineapplePlanner.Domain.UnitTests.Shared.ResultBaseTests

The `ResultBaseTests` class contains unit tests for our custom `ResultBase` class that is used for error handling throughout the entire project. The unit tests are implemented using the .NET testing library XUnit. Each unit test is declared with a `Fact` attribute.

```
 1 using PineapplePlanner.Domain.Shared;
 2
 3 namespace PineapplePlanner.Domain.UnitTests.Shared
 4 {
 5   public class ResultBaseTests
 6   {
 7     [Fact]
 8     public void ResultBase_Success_ShouldSetIsSuccessTrue()
 9     {
10       ResultBase result = ResultBase.Success();
11
12       Assert.True(result.IsSuccess);
13     }
14
15     [Fact]
16     public void ResultBase_Success_ShouldContainNoErrors()
17     {
18       ResultBase result = ResultBase.Success();
19
20       Assert.Empty(result.Errors);
21     }
22
23     [Fact]
24     public void ResultBase_Failure_ShouldSetIsSuccessFalse()
25     {
26       ResultBase result = ResultBase.Failure();
27
28       Assert.False(result.IsSuccess);
29     }
30
31     [Fact]
32     public void ResultBase_Failure_ShouldContainNoErrors()
33     {
34       ResultBase result = ResultBase.Failure();
```

```
35
36          Assert.Empty(result.Errors);
37      }
38
39      [Fact]
40      public void
        ResultBase_AddErrorAndSetFailure_ShouldAddError()
41      {
42          ResultBase result = ResultBase.Success();
43          string errorMessage = "Test error";
44
45          result.AddErrorAndSetFailure(errorMessage);
46
47          Assert.NotEmpty(result.Errors);
48          Assert.Contains(errorMessage, result.Errors);
49      }
50
51      [Fact]
52      public void
        ResultBase_AddErrorAndSetFailure_ShouldSetIsSuccessFalse
        ()
53      {
54          ResultBase result = ResultBase.Success();
55
56          result.AddErrorAndSetFailure("Test error");
57
58          Assert.False(result.IsSuccess);
59      }
60
61      [Fact]
62      public void
        ResultBase_DefaultConstructor_ShouldInitializeEmptyErrors
        ()
63      {
64          ResultBase result = new ResultBase();
65
66          Assert.Empty(result.Errors);
67      }
68  }
69
70  public class ResultBaseTTests
71  {
72      [Fact]
73      public void ResultBaseT_Success_ShouldSetIsSuccessTrue()
74      {
75          ResultBase<int> result = ResultBase<int>.Success();
76
77          Assert.True(result.IsSuccess);
78      }
```

```
79
80      [Fact]
81      public void ResultBaseT_Success_ShouldContainNoErrors()
82      {
83          ResultBase<int> result = ResultBase<int>.Success();
84
85          Assert.Empty(result.Errors);
86      }
87
88      [Fact]
89      public void ResultBaseT_SuccessWithData_ShouldSetData()
90      {
91          string testData = "test";
92          ResultBase<string> result = ResultBase<string>.Success
        (testData);
93
94          Assert.Equal(testData, result.Data);
95      }
96
97      [Fact]
98      public void
        ResultBaseT_SuccessWithData_ShouldSetIsSuccessTrue()
99      {
100         string testData = "test";
101         ResultBase<string> result = ResultBase<string>.Success
        (testData);
102
103         Assert.True(result.IsSuccess);
104     }
105
106     [Fact]
107     public void ResultBaseT_Failure_ShouldSetIsSuccessFalse
        ()
108     {
109         ResultBase<int> result = ResultBase<int>.Failure();
110
111         Assert.False(result.IsSuccess);
112     }
113
114     [Fact]
115     public void ResultBaseT_Failure_ShouldContainNoErrors()
116     {
117         ResultBase<int> result = ResultBase<int>.Failure();
118
119         Assert.Empty(result.Errors);
120     }
121
122     [Fact]
```

```
123     public void
        ResultBaseT_ConstructorWithData_ShouldSetData()
124     {
125         int testData = 42;
126         ResultBase<int> result = new ResultBase<int>(testData)
        ;
127
128         Assert.Equal(testData, result.Data);
129     }
130
131     [Fact]
132     public void
        ResultBaseT_AddErrorAndSetFailure_ShouldPreserveData()
133     {
134         int testData = 42;
135         ResultBase<int> result = ResultBase<int>.Success(
        testData);
136
137         result.AddErrorAndSetFailure("Test error");
138
139         Assert.Equal(testData, result.Data);
140     }
141   }
142 }
```

Listing 3: PineapplePlanner.Domain.UnitTests.Shared.ResultBaseTests

4 Test Results

Table 2 below contains the current status of implemented and tested requirements.

Nr	Req. Name	Test result
1	Log in	Not Implemented/Passed/Failed
2	Log out	Not Implemented/Passed/Failed
3	Task creation	Not Implemented/Passed/Failed
4	Edit task	Not Implemented/Passed/Failed
5	Task prioritization	Not Implemented/Passed/Failed
6	View tasks in the calendar	Not Implemented/Passed/Failed
7	Add tasks in the calendar	Not Implemented/Passed/Failed
8	Tags	Not Implemented/Passed/Failed
9	Dark theme	Not Implemented/Passed/Failed
10	Tags	Not Implemented/Passed/Failed
11	AI	Not Implemented/Passed/Failed

Table 2: Requirements

5 Summary and Conclusion

This chapter contains a summary and conclusion of the work that was carried out in this project as well as reflections and thoughts about working methods and challenges.

5.1 Weekly Progress

Below is a short summary of what was done each week.

5.1.1 Week 1

In week 1, we made significant strides in setting up the foundation for our project. We established the GitHub repository, integrated Jira for streamlined project management, and configured Firebase for backend support. Additionally, we implemented user authentication and conducted a thorough code refactor to enhance efficiency and maintainability

5.1.2 Week 2

In Week 2, we focused on refining the user experience and adding key features. The UI was designed and implemented, email verification was integrated for secure access, and a custom logo was created and applied. Furthermore, essential features such as the calendar, settings panel, and to-do list were developed and implemented to enhance functionality and usability.

5.1.3 Week 3

In Week 3, we focused on improving functionality and stability. Unit tests were added to ensure reliability, minor issues were fixed for a smoother experience, and dark mode was implemented to enhance usability. We also introduced a tagging system for better organization and navigation.

5.1.4 Week 4

In Week 4, we made improvements to both functionality and project organization. The AI assistant was enhanced, and Git branch protection was set up to maintain code integrity. Additional settings features were implemented, localization was introduced, and various codebase warnings were cleaned up. Finally, we started working on the final report.

5.2 Difficulties and challenges

Below is a list of notable challenges that came up during this project and that took a long time to solve.

5.2.1 Blazor WPF wrapper

One significant challenge we faced several times was the fact that we decided to implement our Blazor SSR web application within a WPF wrapper to be ran as a desktop application. Having a web based application is very comfortable for development, especially when it comes to building user interfaces. Also you can

always fallback to use JavaScript as the app runs in the browser. On the other hand, having a WPF wrapper limited us when implementing certain features.
For instance, redirects that were necessary for the authentication with Firebase turned out to be rather difficult.

5.2.2 Dependency Injection issues

Blazor and WPF both have their own dependency injection frameworks and ensuring proper integration between the two required some trial and error. Some services that worked seamlessly within Blazor did not behave as expected when instantiated inside the WPF wrapper. For example, singleton services shared between Blazor and WPF sometimes led to unintended issues. The whole services setup took quite some time.

5.2.3 Lack of console outputs for debugging

Another major difficulty we encountered was the lack of direct console output when running the Blazor application within the WPF wrapper. Usually, web applications benefit from browser developer tools, in particular the console outputs. However, within the WPF environment, there was neither a .NET console nor a browser console available making debugging significantly harder. We often ended up rendering output to the user interface for testing.

5.2.4 Firebase API Quota Exceeded

During development and testing, we occasionally exceeded Firebase's API quota. This typically happened when we testing extensively in a short period or when our application accidentally fell into an endless request loop. We had to be extremely careful not to get stuck in infinite render loops. Otherwise we had to wait one day for the firebase cooldown to continue development.

5.3 Correctness of time estimates

5.4 Priority decisions

5.5 Conclusion

6 References