Project Information:

Project: TaskServer II – Making Task Synchronization Effortless

Organization: CCExtractor

Google Summer Of Code 2025

Time Estimation: 175 hours

Mentor: Mabud Alam

Student Detail:

Name: Nileshkumar Yadav

College/University: Shree Lr Tiwari College of Engineering/University of Mumbai

Country: India

State: Maharashtra

City: Mumbai

Lat/Long: 18°57'N / 72°50'E

Timezone: Indian Standard Time (GMT +05:30)

Preferred Language: I'm proficient in English for communication, both spoken and written

Email: yadav.nilesh9172@gmail.com

GitHub: https://github.com/Ni1esh-Yadav

LinkedIn: https://www.linkedin.com/in/ni1esh-yadav/

Zulip: https://ccextractor.zulipchat.com/#user/866756

Contents:

Personal Introduction:

I am Nileshkumar Yadav, a 4th-year Computer Engineering undergrad at Shree LR Tiwari College of Engineering, Mumbai, India. My interest lies in exploring new technology and diving deep into it. I like to learn new things, and I am not afraid of trying something new. I am always ready to learn and build whatever comes my way.

I started coding in the 2nd year of my college and explored various technologies. I found data structures and algorithms very interesting, along with databases. I came across open-source contributions in my 3rd year and thought of giving it a try. However, due to a lack of knowledge, I was not able to make much of an impact, so I put it on hold to get more familiar with it.

I like open-source contributions because they allow me to work on large projects that people use, solving real-world problems. This thought excites me about contributing to open source. Over the past few months, I have tried my best to do so.

Why This Project Matters:

To Me:

Open-source development excites me because it offers the opportunity to contribute to real-world solutions. Being part of Google Summer of Code (GSoC) is a long-held aspiration, and I'm committed to giving my best to complete this project successfully. While I haven't had the chance to work on a large-scale practical project before, I'm now ready to dedicate myself entirely to this challenge, learn from it, and grow as a developer.

To the Organization:

This project will provide users with an efficient and cost-free task management solution. By ensuring seamless task synchronization across devices and platforms, it will significantly enhance productivity for individual users and teams. Additionally, a well-structured web UI and robust API will simplify task management and offer a flexible alternative to paid task management tools. With improved performance, security, and mobile integration, CCSync will become a reliable choice for the open-source community and beyond.

BUG FIXES AND CONTRIBUTION:

I am actively contributing to various repositories and continuously working to improve functionality, documentation, and user experience.I've made it a point to stay engaged with the community, quickly responding to opportunities to contribute and proactively suggesting improvements. These efforts have allowed me to make meaningful contributions across various aspects of the projects.My contributions so far include:

Contributions to CCExtractor Repository:

- **[Issue]** No Captions Found Using GUI and Command-Line on Windows.
- **[Issue]** Broken Links for "Documentation for Subtitle Downloader" and "User Documentation for Activity Extractor" on the Downloads Page **#1654**.
- **[MERGED]** Fixed the Dockerfile **#1659**.

Contributions to CCSync Repository:

- **[MERGED]** Fixed GitHub icon navigation in Navbar **#86**.
- **[MERGED]** Ensured the modal closes automatically after clicking the "Yes" button **#81**.
- **[MERGED]** Resolved incorrect toast message appearing on task creation failure **#61**.
- **[Closed]** Deleted Tasks Reappear After Syncing #96

Contributions to Ultimate-Alarm-Clock Repository:

- **[MERGED]** Updated Contribution Guidelines by adding a Troubleshooting section **#702**.
- **[MERGED]** Fixed timer input overflow issue **#611**.

Contribution to Taskwarrior-flutter Repository:

- **[Issue]** UI Issue: Project List Text Overflow and Misalignment **#466**
- **[Pr]** fix:Project List Text Overflow and Misalignment **#470**
- **[Issue]** Improve TaskWarrior-flutter App Sync Setup with CCSync

About My Qualification for GSoC and CCExtractor:

This is my first time applying for GSoC, and I meet all the eligibility criteria for participation. I am highly motivated to contribute and gain experience through this program.

For CCExtractor's qualification, I have successfully completed two qualification tasks:

1. **MyFitnessPal Data Visualization with Grafana**
   a. This project provides a simple web application that allows users to upload MyFitnessPal CSV exports and visualize the data using Grafana. Since manually analyzing CSV files can be tedious, this project automates the process by integrating with Grafana for clear and interactive visualizations.
   b. **Task Link:** MyFitnessPal Project

2. **Rust: Fixing the Build System on GitHub**
   a. The CCExtractor build system was failing due to a bug in the FFmpeg crate used for Rust. After analyzing the issue, I determined that the problem was related to package dependencies. I worked on identifying and resolving the issue, ensuring that the build system now functions correctly.
   b. **Fixed Issue:** FFmpeg Build Error for ffmpeg-sys-next in Rust #101

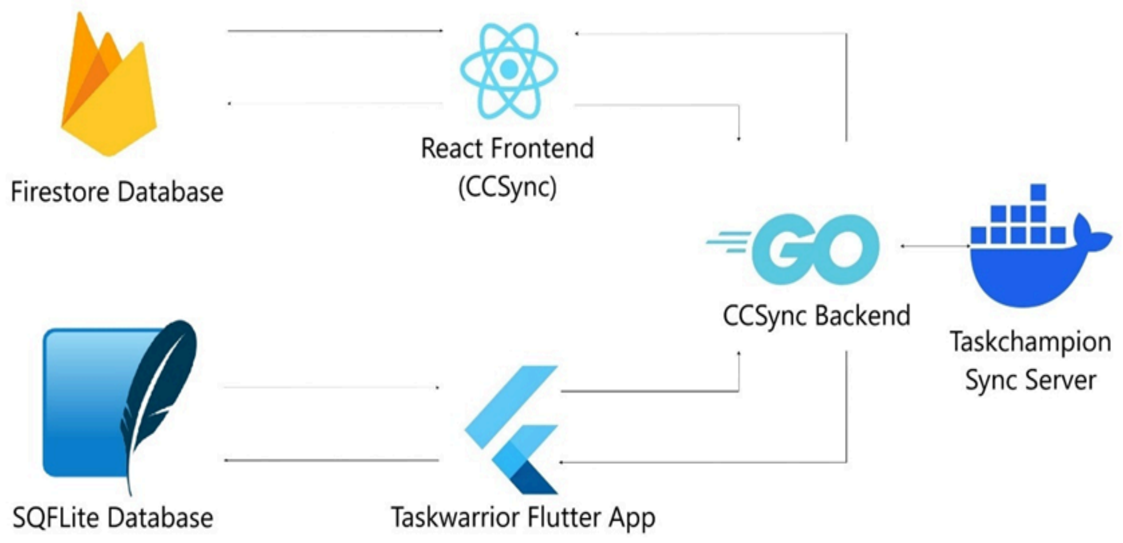
Introduction of Project:

CCSync is a web UI and API solution designed to facilitate the retrieval and synchronization of tasks from a taskchampion-sync-server container. It provides a seamless experience for managing tasks across all Taskwarrior 3.0 (and higher) clients, whether using the Taskwarrior CLI, the web frontend, or the Taskwarrior Flutter app.

Existing Architecture of the project:

CCSync comprises three main components:

1. **Backend**: The main server-side component that interfaces with Taskwarrior clients, performs operations, and provides a RESTful API.
2. **Web Frontend**: A user-friendly web interface built for task management, and credentials retrieval.
3. **Taskwarrior Flutter App**: The mobile app that allows users to manage tasks on the go, fully integrated with the CCSync API.



Problem Statement:

Task management tools are critical for productivity, but existing open-source solutions often lack seamless synchronization across platforms (CLI, web, mobile) and self-hosting flexibility, limiting their adoption by users and teams. CCSync, built on TaskWarrior's ecosystem, aims to solve these challenges. However, the current implementation faces critical gaps that hinder its production readiness:

1. Scalable Authentication & Security

Problem:

- The existing session-based authentication is stateful, leading to scalability issues in distributed environments.
- API endpoints lack rate limiting and token refresh mechanisms, exposing the system to abuse and security risks.

Impact:

- Users face session loss during server restarts, and manual API management complicates integration with third-party tools.

2. Unreliable Cross-Platform Synchronization

Problem:

- Sync failures and conflicts (e.g., simultaneous edits on mobile and CLI) disrupt workflows.
- The mobile app lacks real-time updates and offline sync support, reducing usability.

Impact:

- Users manually resolve task conflicts, leading to frustration and data inconsistency across devices.

3. Limited Self-Hosting & Performance

Problem:

- Firebase dependency restricts self-hosting flexibility, forcing users into vendor lock-in.
- The backend struggles with large datasets (e.g., 100k+ tasks), causing slow sync times and timeouts.

Impact:

- Organizations cannot deploy CCSync privately, and power users experience degraded performance.

4. Underdeveloped Web UI & Documentation

Problem:

- The Web UI lacks advanced features like bulk editing and task prioritization, limiting adoption.
- Poor documentation hinders user onboarding and developer contributions.

Impact:

- Users resort to CLI for complex operations, and developers struggle to extend CCSync's functionality.

5. Manual Testing & Deployment

Problem:

- No automated testing leads to regression bugs and slows development.
- Deployment relies on manual Docker configurations, increasing setup complexity.

Impact:

- Critical bugs reach production, and non-technical users struggle to self-host CCSync.

Areas of Improvement:

The primary goal is to finalize CCSync for production by implementing essential features, including:

- Authentication System: Enhance the existing secure authentication mechanism for user access.

- Secure API Layer: Ensure safe and reliable communication through API enhancements.
- Self-Hosting Support: Enable users to host CCSync on their own servers with ease.
- Data Synchronization Across Devices: Improve the reliability and accuracy of task synchronization across all connected devices using the Taskchampion Sync Server.
- Integration with TaskWarrior Mobile App: Strengthen the connection between CCSync and the TaskWarrior Flutter app for a seamless mobile experience.
- Enhancing the Web UI: Refine the web interface for better usability and user experience, leveraging IndexedDB for offline support.

While significant progress was made last year, there is still room for improvement in terms of performance, architecture, and scalability. This year, the focus will be to:

- Optimize the API Architecture to ensure maintainability and performance with synchronous operations.
- Develop a Production-Ready Web UI with a polished, user-friendly interface and offline support via IndexedDB.
- Enhance Sync Stability for consistent and error-free task updates across devices using the Taskchampion Sync Server.
- Improve Server Performance by optimizing resource usage and increasing reliability.
- Ensure Seamless Mobile Integration by refining the connection with the TaskWarrior Flutter app.

Proposed Deliverables:

1.Authentication System (High Priority)

- Enhance the existing Google OAuth-based authentication.
- JWT tokens for secure API access, with improvements in token management.

2.Session Management

- Provide API Key and Sync Token generation for TaskWarrior clients to authenticate with the Taskchampion Sync Server.
- Implement token refresh, session expiration, and rate limiting (already partially implemented with RateLimitMiddleware).

3.Data Models and Syncing

- Refine the existing data models for Tasks, Projects, and Tags (models/task.go).
- Enable bi-directional data sync between the Taskchampion Sync Server and TaskWarrior clients, leveraging the existing utils/tw package.

4.TaskWarrior Mobile App Integration (High Priority)

- Ensure seamless task syncing between the Taskchampion Sync Server and the TaskWarrior Flutter app.

- Implement background sync for offline tasks in the Flutter app, with support for syncing when the app comes online.

5.Self-Hosting Capability

- Provide Docker-based self-hosting options for easy deployment of the CCSync backend and Taskchampion Sync Server.
- Offer clear configuration options for public or private hosting (e.g., environment variables for OAuth and JWT secrets).

6.Web UI for Task Management

- Enhance the React.js-based web interface with features like filtering, sorting, and bulk editing.
- Leverage IndexedDB for offline support, ensuring tasks are cached locally and updated only on user-initiated sync.

7.API Layer (Production-Ready)

- Refine the existing REST API with better error handling, input validation, and rate limiting (build on the existing RateLimitMiddleware).
- Document the API using Swagger for developer usability.

8.Testing and Quality Assurance

- Ensure 90%+ code coverage through unit, integration, and end-to-end tests (build on existing tests in controller_test.go and taskwarrior_test.go).
- Set up a CI/CD pipeline for automated testing and deployment using GitHub Actions.

9.Cross-Device Sync (Critical)

- Enable task synchronization across all TaskWarrior clients via the Taskchampion Sync Server.
- Implement retry mechanisms for failed sync attempts and improve sync stability (enhance utils/tw/sync_task.go).

10.Infrastructure Setup

- Provide Docker containers for the CCSync backend and Taskchampion Sync Server for easy deployment and scaling.
- Implement monitoring and error logging using tools like Sentry.

11.Documentation (Must-Have)

- Create clear, complete documentation of the project.
- Provide detailed guides for deployment, API usage, and troubleshooting.
- Post weekly blog updates on the project’s progress on Medium or another platform.

Problem-Solution Mapping:

1.Authentication & Security

Problem:

- The current authentication system can be enhanced for better token management and scalability.

Solution:

- **Deliverable 1:** Improve the existing Google OAuth-based authentication with better JWT token management (e.g., token refresh, expiration).
- **Deliverable 2:** Enhance API security with rate limiting (already implemented in middleware/rate_limiter.go), token expiration, and refresh mechanisms.
- **Deliverable 7:** Document API endpoints with Swagger and enforce input validation in handlers (e.g., AddTaskHandler, EditTaskHandler).

2.Cross-Platform Sync

Problem:

- Sync failures between the Taskchampion Sync Server and clients can disrupt user workflows, especially with offline scenarios.

Solution:

- **Deliverable 3:** Refine the task data model (models/task.go) to ensure consistency during sync operations.
- **Deliverable 4:** Add support for background sync in the TaskWarrior Flutter app, ensuring offline tasks are synced when the app comes online.
- **Deliverable 9:** Implement retry mechanisms for failed sync attempts in utils/tw/sync_task.go and improve error handling for sync operations.

3.Self-Hosting & Scalability

Problem:

Users need an easy way to self-host CCSync without external dependencies.

Solution:

- **Deliverable 5:** Provide a Docker Compose setup for the CCSync backend and Taskchampion Sync Server, with clear environment variable configurations (e.g., CLIENT_ID, JWT_SECRET).
- **Deliverable 10:** Optimize TaskWarrior command execution in utils/tw (e.g., reduce temporary directory creation overhead) and add Sentry logging for performance monitoring.

4.Usability

Problem:

- The web UI lacks advanced task management features and needs better offline support.

Solution:

- **Deliverable 6:** Enhance the React.js frontend with features like drag-and-drop task prioritization, filtering, and sorting, while leveraging IndexedDB for offline task caching.
- **Deliverable 11:** Write user guides with screenshots and video tutorials for task management and sync workflows.

5.Stability & Testing

Problem:

- Manual testing slows development and may miss edge cases.

Solution:

- **Deliverable 8:** Automate testing with GitHub Actions, expanding unit tests (controller_test.go, taskwarrior_test.go) and adding integration tests for the full sync flow.

Purpose:

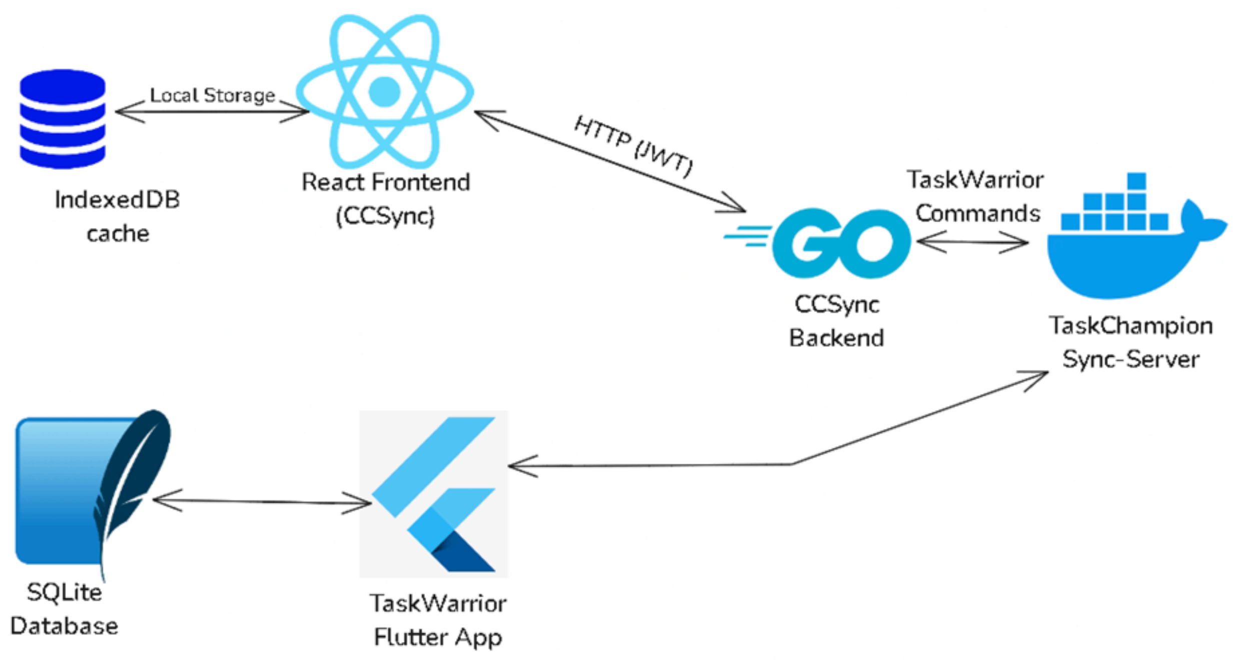
My primary focus is to make TaskServer production-ready — ensuring that users can self-host it, use it as a public service, and integrate it smoothly with the existing TaskWarrior mobile app.

I also want to significantly improve performance, efficiency, and usability based on the learnings from last year.

Implementation Detail:

The system architecture is modified as our goal is to make system cost effective because some users will have 100k tasks in their system so it should handle it efficiently:

Updated System Architecture:



Flow of Interactions

The arrows between components represent the flow of data and interactions:

1. **React Frontend (CCSync) ↔ CCSync Backend (Go)**

- Description: The frontend communicates with the backend via HTTP requests:
    - Authentication: /auth/oauth, /auth/callback, /api/user, /auth/logout for Auth login and user info.
        - The OAuth flow uses Google's OAuth2 endpoint (not shown in the diagram, as it's an external service).
        - After login, the backend returns a JWT token containing user info (email, encryptionSecret, uuid).
    - Task Operations: /add-task, /edit-task, /modify-task, /complete-task, /delete-task to perform task operations synchronously.
        - Requests include a JWT token in the Authorization header.
        - Responses include taskUUID (for /add-task) or success messages (e.g., "Task edited successfully. Click Sync to view the changes.").
    - Sync: /sync to fetch the full list of tasks from TaskWarrior when the user clicks "Sync."

Subflow (within Frontend):

- React Frontend → IndexedDB: On /sync, the fetched tasks are stored in IndexedDB.
- IndexedDB → React Frontend: On page load or after /sync, tasks are retrieved from IndexedDB to update the UI.

2. **CCSync Backend (Go) ↔ Taskchampion Sync Server**

- Description: The backend uses the utils/tw package to interact with TaskWarrior:
    - Sets up TaskWarrior config (SetTaskwarriorConfig).
    - Performs task operations (AddTaskToTaskwarrior, EditTaskInTaskwarrior, etc.).
    - Syncs tasks (SyncTaskwarrior).
    - Fetches tasks (FetchTasksFromTaskwarrior for /sync).

3. **TaskWarrior Flutter App ↔ Taskchampion Sync Server**

- Description: The Flutter app syncs tasks with the Taskchampion Sync Server independently.

4. **TaskWarrior Flutter App ↔ SQLite Database**

- Description: The Flutter app uses SQLite for local task storage.

Self-Hosting Capability (Public & Private Hosting)

Implementation Plan

Docker-Based Deployment:

Overview: The project already includes a Dockerfile for the CCSync backend. To simplify deployment, we will enhance this by adding a Docker Compose setup for a one-click deployment that includes both the CCSync backend and the Taskchampion Sync Server. No external database is required, as task management is handled by the Taskchampion Sync Server.

Goal: Enable users to run the entire CCSync system (backend and sync server) with minimal effort.

Implementation Steps:

1.Create a docker-compose.yml File:

- Define two services: ccsync-backend for the CCSync backend and taskchampion-sync-server for the Taskchampion Sync Server.
- Use environment variables to configure OAuth credentials, JWT secrets, and sync server settings.
- Expose ports for both services (e.g., 8000 for the backend, 8080 for the sync server).
- Include a volume to persist Taskchampion Sync Server data.

2.Here's an example

Docker-compose.yml:

```
version: '3.8'
services:
  ccsync-backend:
    build: .
    env_file: .env
    ports:
      - "8000:8000"
    depends_on:
      - taskchampion-sync-server
  taskchampion-sync-server:
    image: taskchampion/sync-server:latest  # Official Taskchampion Sync Server image
    restart: always
    ports:
      - "8080:8080"
    environment:
      - SYNC_SERVER_PORT=8080
      - SYNC_SERVER_DATA_DIR=/data
    volumes:
      - sync-data:/data
volumes:
  sync-data:
```

3.Update Documentation:

Provide clear instructions for users to start the services:

```
·     docker-compose up -d
```

Include guidance on creating and configuring a .env file (see Configuration Management below).

Public Hosting Support for CCExtractor:

Overview: CCExtractor can host a public instance of CCSync on a cloud server (e.g., AWS, DigitalOcean) using the same Docker Compose setup, with additional configurations for security and accessibility.

Goal: Enable a scalable, secure public deployment with minimal setup complexity.

Implementation Steps:

1. Use .env for Configuration:
   a. Store sensitive information like OAuth credentials, JWT secrets, and sync server settings in a .env file.
   b. Example .env:

```
CLIENT_ID=your_google_oauth_client_id

CLIENT_SEC=your_google_oauth_client_secret

REDIRECT_URL_DEV=http://localhost:8000/auth/callback

JWT_SECRET=your_jwt_secret

CONTAINER_ORIGIN=http://taskchampion-sync-server:8080

FRONTEND_ORIGIN_DEV=http://localhost:5172
```

2. **Add a Reverse Proxy for HTTPS:**

   Use a reverse proxy (e.g., Nginx) to handle HTTPS traffic for the public instance.

   Sample Nginx configuration (to be included in documentation):

```
server {
    listen 443 ssl;
    server_name yourdomain.com;
    ssl_certificate /path/to/cert.pem;
    ssl_certificate_key /path/to/key.pem;
    location / {
        proxy_pass http://ccsync-backend:8000;

        proxy_set_header Host $host;

        proxy_set_header X-Real-IP $remote_addr;

    }

}
```

3. **Provide Deployment Scripts:**

- Create shell scripts or Ansible playbooks to automate deployment on cloud servers. Example tasks:

    1)Pull the latest code from the repository.

    2)Build Docker images.

    3)Start services with docker-compose up -d.

- Sample shell script:

```
#!/bin/bash

git pull origin main

docker-compose build

docker-compose up -d
```

Configuration Management:

- **Overview:** Allow users to customize their CCSync setup without modifying the source code by using environment variables.
- **Goal:** Ensure flexibility and ease of configuration for different hosting scenarios.

Implementation Steps:

1. **Update** config.go **to Read from Environment Variables:**

   Modify the backend configuration file (e.g., config.go) to load settings dynamically from environment variables.

   Example in Go:

```
package config
import "os"
var (
    FrontendOrigin  = os.Getenv("FRONTEND_ORIGIN_DEV")
    ClientID        = os.Getenv("CLIENT_ID")
    ClientSecret    = os.Getenv("CLIENT_SEC")
    RedirectURL     = os.Getenv("REDIRECT_URL_DEV")
    JWTSecret       = os.Getenv("JWT_SECRET")
    ContainerOrigin = os.Getenv("CONTAINER_ORIGIN")
)
```

2. **Document Environment Variables:**

- Provide a comprehensive list in the documentation, including:
    - **Required:** CLIENT_ID, CLIENT_SEC, JWT_SECRET, CONTAINER_ORIGIN
    - **Optional:** FRONTEND_ORIGIN_DEV (default: http://localhost:5172), REDIRECT_URL_DEV (default: http://localhost:8000/auth/callback)
    - Include descriptions and examples for each.

Implementation of GUI:

I have a strong grasp of GUI development, so I am not overly concerned about it. Additionally, there are not many GUI-related tasks in this project. My primary focus is on working intensively on the backend to make it robust and production-ready.

Current GUI Task:

- Implement a **Bulk task editing, filtering, and sorting**.
- Data Export/Import options.
- Ability to close accounts or reset data.

While the frontend work is minimal, ensuring seamless integration with the backend authentication system (JWT-based login, session management, etc.) is a priority

Also, there are here and some error in Gui which I will mend as I progresses in the project and I found them while using it

Testing & Quality Assurance Implementation Plan:

To ensure the reliability, scalability, and maintainability of CCSync, the project will implement **unit testing, CI/CD integration, and load testing**. The testing strategy will cover both the **React-based Web UI** and **Go backend**.

1. Unit Testing

- **Frontend (React + TypeScript)**
    - I will be using **Jest** and **React Testing Library** for component testing.
    - Ensures UI elements function correctly with user interactions.
    - Example test cases: Button clicks, form validation, API call mocks.
- **Backend (Go)**
    - Uses **Go's built-in `testing` package** and **Testify** for API and logic validation.
    - Covers critical functionalities such as authentication, task synchronization, and API responses.

2. CI/CD Pipeline

- **Automated testing** using **GitHub Actions** to trigger tests on every commit/pull request.

- Ensures code quality before merging/deployment.

3. Load Testing

- **Frontend Performance**:
  - Uses **Lighthouse** to audit page load times, accessibility, and SEO (not sure I will discuss with the mentor).
- **Backend Performance**:(Not sure I will discuss with the mentor).
  - Uses **k6** to simulate concurrent user requests and measure API response times.
  - Ensures the system can handle **high traffic loads** without degradation.

Sample Implementation of Backend:

Test Code (controllers/add_task_handler_test.go):

```
package controllers
import (
      "bytes"
      "ccsync_backend/models"
      "ccsync_backend/utils/tw"
      "encoding/json"
      "errors"
      "net/http"
      "net/http/httptest"
      "testing"

      "github.com/stretchr/testify/assert"
)
func mockAddTaskToTaskwarrior(email, encryptionSecret, uuid, description, project,
priority, dueDate string, tags []string) error {
      if description == "" {
            return errors.New("description cannot be empty")
      }
      if dueDate == "" {
            return errors.New("due date cannot be empty")
      }
      return nil
}
func TestAddTaskHandler(t *testing.T) {
      tw.AddTaskToTaskwarrior = mockAddTaskToTaskwarrior
      tests := []struct {
            name           string
            requestBody    models.AddTaskRequestBody
            expectedStatus int
```

```
}{
	{
		name: "Valid Task Addition",
		requestBody: models.AddTaskRequestBody{
			Email:            "test@example.com",
			EncryptionSecret: "secret123",
			UUID:             "1234-abcd",
			Description:      "Complete unit testing",
			Project:         "Testing",
			Priority:        "H",
			DueDate:         "2025-04-10",
			Tags:            []string{"test", "golang"},
		},
		expectedStatus: http.StatusAccepted,
	},
	{
		name: "Missing Description",
		requestBody: models.AddTaskRequestBody{
			Email:            "test@example.com",
			EncryptionSecret: "secret123",
			UUID:             "1234-abcd",
			Description:      "",
			Project:         "Testing",
			Priority:        "H",
			DueDate:         "2025-04-10",
			Tags:            []string{"test"},
		},
		expectedStatus: http.StatusBadRequest,
	},
	{
		name: "Missing Due Date",
		requestBody: models.AddTaskRequestBody{
			Email:            "test@example.com",
			EncryptionSecret: "secret123",
			UUID:             "1234-abcd",
			Description:      "Complete unit testing",
			Project:         "Testing",
			Priority:        "H",
			DueDate:         "",
			Tags:            []string{"test"},
		},
				expectedStatus: http.StatusBadRequest,
			},
		}
		for _, tt := range tests {
			t.Run(tt.name, func(t *testing.T) {
				requestBodyBytes, _ := json.Marshal(tt.requestBody)
				req, err := http.NewRequest(http.MethodPost, "/add-task",
bytes.NewBuffer(requestBodyBytes))
				assert.NoError(t, err)
				req.Header.Set("Content-Type", "application/json")
				rr := httptest.NewRecorder()
				handler := http.HandlerFunc(AddTaskHandler)
				handler.ServeHTTP(rr, req)

				assert.Equal(t, tt.expectedStatus, rr.Code)
			})
		}
		t.Run("Invalid Request Method", func(t *testing.T) {
			req, err := http.NewRequest(http.MethodGet, "/add-task", nil)
			assert.NoError(t, err)
			rr := httptest.NewRecorder()
			handler := http.HandlerFunc(AddTaskHandler)
			handler.ServeHTTP(rr, req)
			assert.Equal(t, http.StatusMethodNotAllowed, rr.Code)
		})
}
```

Explanation

1. **Mocking**: I have mocked tw.AddTaskToTaskwarrior to simulate the expected behavior.
2. **Test Cases**: checked for valid task addition, missing description, missing due date, and invalid HTTP method.
3. **Assertions**: I have used testify/assert to compare expected vs. actual results.

Test Code (controllers/complete_task_handler_test.go):

```
package controllers
import (
	"bytes"
	"ccsync_backend/models"
	"ccsync_backend/utils/tw"
	"encoding/json"
	"errors"
	"net/http"
	"net/http/httptest"
	"testing"
	"github.com/stretchr/testify/assert"
)
func mockCompleteTaskInTaskwarrior(email, encryptionSecret, uuid, taskuuid string)
error {
	if taskuuid == "" {
		return errors.New("taskuuid is required")
	}
	return nil
}
func TestCompleteTaskHandler(t *testing.T) {
	tw.CompleteTaskInTaskwarrior = mockCompleteTaskInTaskwarrior
	tests := []struct {
		name           string
		requestBody    models.CompleteTaskRequestBody
		expectedStatus int
	}{
		{
			name: "Valid Task Completion",
			requestBody: models.CompleteTaskRequestBody{
				Email:            "test@example.com",
				EncryptionSecret: "secret123",
				UUID:             "1234-abcd",
				TaskUUID:         "task-5678",
			},
			expectedStatus: http.StatusAccepted,
		},
		{
			name: "Missing TaskUUID",
			requestBody: models.CompleteTaskRequestBody{
				Email:            "test@example.com",
				EncryptionSecret: "secret123",
				UUID:             "1234-abcd",
				TaskUUID:         "",
			},
			expectedStatus: http.StatusBadRequest,
		},
	}
	for _, tt := range tests {
		t.Run(tt.name, func(t *testing.T) {
			requestBodyBytes, _ := json.Marshal(tt.requestBody)
			req, err := http.NewRequest(http.MethodPost, "/complete-task",
bytes.NewBuffer(requestBodyBytes))
			assert.NoError(t, err)
			req.Header.Set("Content-Type", "application/json")
			rr := httptest.NewRecorder()
			handler := http.HandlerFunc(CompleteTaskHandler)
			handler.ServeHTTP(rr, req)

			assert.Equal(t, tt.expectedStatus, rr.Code)
		})
	}
	t.Run("Invalid Request Method", func(t *testing.T) {
		req, err := http.NewRequest(http.MethodGet, "/complete-task", nil)
		assert.NoError(t, err)
		rr := httptest.NewRecorder()
		handler := http.HandlerFunc(CompleteTaskHandler)
		handler.ServeHTTP(rr, req)
		assert.Equal(t, http.StatusMethodNotAllowed, rr.Code)
	})
}
```

Explanation

1. **Mocking:**
   - mockCompleteTaskInTaskwarrior simulates the function to return errors for invalid inputs.
   - This avoids making real external calls during testing.

2. **Test Cases:**
   Valid Request → Ensures task completion is processed correctly.
   Missing taskuuid → Ensures request validation fails.
   Invalid Request Method → Ensures GET requests are rejected.
3. **Assertions:**
   - I have used testify/assert to compare the expected and actual HTTP status codes.

Similarly I will test each and every endpoint while performing the testing.

Sample Implementation of Frontend:

The existing frontend already includes unit tests and CI/CD integration. My focus will be on **enhancing test coverage, improving reliability, and optimizing performance** to ensure a seamless user experience.

1. Expanding Test Coverage

- Identify gaps in the existing test suite by measuring **test coverage percentage**.
- Add missing test cases for:
    - **Error handling** (e.g., API failures, network timeouts).
    - **Task synchronization edge cases** (e.g., duplicate tasks, offline mode).
    - **Bulk operations** (e.g., bulk task edits, deletions, completions).

2.Improving Test Reliability

- Address **flaky tests** by analyzing intermittent failures in CI.
- Optimize mocking of API requests to reduce test execution time.
- Introduce **integration tests** for API calls using MSW (Mock Service Worker).

3.Performance Benchmarking

- Conduct **Lighthouse & WebPageTest audits** to analyze frontend performance.
- Identify and optimize slow-rendering components in large task lists.
- Implement **code-splitting** to improve initial load times.

4.Enhancing CI/CD Workflows

- Optimize **GitHub Actions** to:
    - Cache dependencies to **reduce build time**.
    - Parallelize test execution for faster feedback.
    - Implement **automated performance tests** in CI pipeline.

These improvements will ensure the frontend remains **fast, scalable, and resilient**, improving the overall user experience for CCSync users.

Documentation Plan:

Clear and well-structured documentation is essential for CCSync to be easily understandable and maintainable. As part of my implementation, I will focus on writing detailed documentation that covers the following key areas:

- **API Endpoints**: I will document all available API endpoints, including request and response formats, authentication methods, and practical usage examples. This will help developers integrate with the system efficiently.
- **Deployment Setup (Docker/Self-hosted)**: Since Docker is already part of the project, I will refine and expand the documentation to include step-by-step instructions on setting up CCSync in a self-hosted environment, along with details on environment variables and configuration options.
- **Web UI Usage**: I will provide a straightforward guide on how users can navigate and utilize the web interface, covering login, task management, and synchronization with Taskwarrior.

- **Mobile App Integration Guide**: Since Taskwarrior can be used on mobile devices, I will document how to integrate CCSync with mobile clients, ensuring smooth task synchronization.

To make onboarding easier, I will also create a **"Getting Started"** guide that provides a quick overview of installation, setup, and basic usage. This will help both developers and end-users get up to speed with CCSync efficiently.

Features to implement if project is completed before GSoC time period:

If I complete the core tasks ahead of the GSoC timeline, I will focus on implementing additional features to enhance the functionality of CCSync. These may include:

- **RSS Feeds for Upcoming Tasks**: Implementing an RSS feed system that allows users to subscribe to their upcoming tasks and receive updates in their preferred feed reader.
- **Push Notifications for Task Reminders**: Adding support for push notifications to alert users about upcoming or overdue tasks, improving task management efficiency.
- **Task Archive and History Functionality**: Providing a way to archive completed tasks and maintain a detailed history, allowing users to track progress and revisit past tasks when needed.

These features will further enhance the usability of CCSync, making it more user-friendly and efficient for task management.

BRIEF TIMELINE:

- **Phase 0 (Till May 7):** Pre-GSoC period.
- **Phase 1 (May 8 - June 1):** Community Bonding.
- **Phase 2 (June 2 - July 14):** Coding Phase 1.
- **Phase 3 (July 14 - July 18):** Midterm Evaluations.
- **Phase 4 (July 14 - August 25):** Coding Phase 2.
- **Phase 5 (August 25 - September 1):** Final Work Submission & Evaluations.
- **Phase 6 (September 1 - September 8):** Final Mentor Evaluations.
- **Phase 7 (September 1 - November 9):** Extended Coding Period (if applicable).
- **Phase 8 (November 10):** Final Deadline for Extended Projects.

Tentative Project Timeline for GSoC - CCSync:

Phase 0 (Till May 7): Pre-GSoC Period

Goals:

- Finalize project scope with mentors.
- Clarify architectural decisions.

Tasks:

- Review plans for removing Firebase and integrating Taskchampion's Sync Server.
- Draft Proposal and getting reviewed by the mentor
- Submitting the proposal

Phase 1 (May 8 - June 1): Community Bonding

Goals:

- Engage with the CCSync community and mentors.
- Deepen understanding of Taskchampion Sync Server.

Tasks:

- Finalize technical requirements and design decisions.
- Familiarize with the existing codebase.
- Contribute to minor issues or documentation.

Phase 2 (June 2 - July 14): Coding Phase 1

Week 1-2 (June 2 - June 15)

- **Deliverable 1:** Authentication System
  - Implement JWT-based authentication using Google OAuth.

Week 3-4 (June 16 - June 30)

- **Deliverable 7:** API Layer
  - Build REST API endpoints (/add-task, /edit-task, /sync).
  - Implement error handling and validation.
- **Deliverable 2:** Session Management
  - Add API key generation, token refresh, and rate limiting.

Week 5-6 (July 1 - July 14)

- **Deliverable 4:** Mobile App Integration
  - Implement real-time sync with the TaskWarrior Flutter app.
  - Add conflict resolution for task updates.
- **Deliverable 9:** Cross-Device Sync
  - Design and implement bi-directional sync logic using Taskchampion Sync Server.

Phase 3 (July 14 - July 18): Midterm Evaluations

By Midterm:

- Authentication system fully functional.
- Core API endpoints ready for integration.
- Initial Docker Compose setup for self-hosting completed.

Phase 4 (July 14 - August 25): Coding Phase 2

Week 7-8 (July 15 - July 31)

- **Deliverable 3:** Data Models and Syncing
  - Ensure seamless integration between Taskchampion Sync Server and frontend.
  - Test bulk task imports with large datasets (10k+ tasks).

Week 9-10 (August 1 - August 14)

- **Deliverable 6:** Web UI
  - Develop React.js frontend with task filtering, sorting, and bulk edits.
  - Integrate JWT-based authentication into the UI.
- **Deliverable 5:** Self-Hosting Capability
  - Finalize Docker Compose setup for easy self-hosting.
  - Create deployment guides for public and private hosting using Oracle Cloud Free Tier.

Week 11 (August 15 - August 25)

- **Deliverable 10:** Infrastructure Setup
  - Implement monitoring and logging using Sentry.
  - Optimize system performance and resource usage.
- **Deliverable 8:** Testing and QA
  - Achieve 90%+ test coverage using unit, integration, and end-to-end tests.
  - Set up a CI/CD pipeline using GitHub Actions.
- **Deliverable 11:** Documentation
  - Publish API documentation using Swagger.
  - Provide deployment guides, tutorials, and user manuals.

Phase 5 (August 25 - September 1): Final Work Submission & Evaluations

- Submit project for final evaluation.
- Address mentor feedback if required.

Phase 6 (September 1 - September 8): Final Mentor Evaluations

- Submit project for final evaluation.
- Address mentor feedback if required.

Phase 7 (September 1 - November 9): Extended Coding Period (If Applicable)

- Continue refining and optimizing the project.
- Implement additional features if time permits.

Phase 8 (November 10): Final Deadline for Extended Projects

- Submit the final version of the project.

Key Milestones

- **Midterm Evaluation (July 18):** Complete authentication, API endpoints, and initial sync implementation.
- **Final Evaluation (August 25):** Deliver a production-ready CCSync with functional Docker deployment, robust API, and complete documentation.

Note1: I have a working understanding of Golang and have already begun contributing to the backend. While I continue to deepen my expertise, I am confident in my ability to effectively implement and improve features as part of this project. With guidance from my mentor and through consistent hands-on experience, I aim to further refine my Go proficiency while delivering high-quality contributions.

Note2: Above all are the tentative timeline it can be changed if needed according to the requirement of the project under the guidance of mentor.

Availability:

As I am on my last semester of computer engineering which will be completed on may I will be devoting my full time to this project and I do not have any other summer internship or Job

I reviewed the whole GSOC timeline and I would be able to devote approx. 40-50 hours every week to GSOC. If the project demands me to work overtime, I am more than happy to work as my main goal is to complete this project. I will be reporting my work to my mentor on a daily basis.

I would strive to be regular, and sincere with my scrum and daily updates as I understand that selection in this project will require a serious commitment and 100% devotion from my side.

Why CCExtractor?

When I first started my open-source journey and began exploring GSoC, CCExtractor was the first organization I came across. Out of curiosity, I started looking into its projects and decided to solve a few issues. What began as a simple exploration soon turned into something meaningful—I found myself deeply engaged with the organization and its work.

CCExtractor, by coincidence, became the place where I took my first steps in open source, and that connection has only grown stronger. The welcoming community, the challenging yet exciting tasks, and the opportunity to contribute to real-world projects have motivated me to continue working with CCExtractor. Now, as I apply for GSoC, choosing CCExtractor feels less like a decision and more like a natural continuation of my journey. I am eager to deepen my contributions and be part of its growth while improving my own skills along the way.

Other GSoC Organization:

I have not applied to any other organization except CCExtractor. This is also my first GSoC.

Being part of GSoC would be an exciting opportunity to take my contributions to the next level, work alongside experienced mentors, and make a meaningful impact on the project. I am eager to apply my skills, continue learning, and contribute to CCExtractor in a way that benefits both the organization and the open-source community.

Thank you for considering my proposal. I look forward to the opportunity to contribute and grow through this experience.