# Phase 1

1. Market Analysis & Product Definition

Competitor Analysis:  
Time-tracking applications like Clockify, TimeCamp, and Toggl Track offer attendance and productivity monitoring. However, they often require complex setup or subscriptions. This Java-based app aims to offer a simple and lightweight solution for time tracking — ideal for students, staff, and small offices.

User Personas:

User Journey Map:

1. User opens the web application

2. Registers with username and password

3. Logs in with credentials

4. Views time tracking dashboard

5. Clicks Clock In to log entry time

6. Clicks Clock Out to log exit time

7. Views time entries in a table showing:

- Date

- Clock-in time

- Clock-out timeValue

Poposition:  
A secure and easy-to-use Java app for logging work hours with basic login authentication and a visual time log. It reduces manual attendance tracking and promotes accountability.

Pricing Strategy:  
This is a free, open-source Micro SaaS application designed for internal or educational use. No subscription is required.

2. Technical Architecture

C4 Model Breakdown:

• Context: Web-based time tracking application

• Containers:

- Angular frontend

- Node.js/Express backend

- MySQL database

• Components:

- Authentication Module (JWT-based)

- Time Tracking Module

- Database Handler

- API Endpoints

Technology Stack:

* Frontend: Angular (not Java Swing)
* Backend: Node.js/Express (not Java)
* Database: MySQL (not local file storage)
* Authentication: JWT (not basic authentication)

Scalability Justification:  
Though the app is currently for desktop, it can later be expanded to a web version using Spring Boot and MySQL for centralized data access.

Infrastructure as Code (Future Scope):

* Dockerize the app for cross-platform deployment
* Use GitHub Actions for build/test automation

Security & Compliance Considerations:

# • JWT-based authentication

# • Password hashing using bcrypt

# • CORS enabled for API security

# • Input validation on both frontend and backend

# • SQL injection prevention using parameterized queries

# Phase 2

**1. Git Workflow Implementation**

**Branch Strategy:**

* **Main Branch:**

**Branch Protection Rules:**

* Only approved pull requests can be merged to main
* Requires successful build and code review before merge

**Commit Standards:**

* Uses **Conventional Commits**, e.g.:
  + feat: add clock-in feature
  + fix: correct time display bug

**Automated Tools (Future Integration):**

* GitHub Actions to automate build/test pipeline (for future scalability)

**Pull Request Templates:**

markdown

CopyEdit

### Summary

\_Explain briefly what this PR does\_

### Changes

- Added clock-in button logic

- Updated record storage format

### Checklist

- [ ] Code compiles

- [ ] Feature tested

- [ ] PR reviewed by team

**2. Project Management Integration**

**GitHub Project Board:**  
A Kanban board with the following columns:

* To Do (feature backlog)
* In Progress
* Code Review
* Done

**Issue Templates:**

* **Bug Report:**
  + Describe bug
  + Steps to reproduce
  + Expected and actual result
* **Feature Request:**
  + Feature title
  + Description and expected behavior

**SLA Monitoring:**  
Internal SLA:

* Bug fix turnaround: Within 48 hours
* Feature merge approval: Within 24 hours after review request

**Team Collaboration Protocols:**

* Daily commit updates
* Weekly checkpoint reviews via GitHub issues
* Shared documentation in project README and wiki

# Phase 3

**1. Automated Pipeline Configuration**

Since this is a Java-based desktop app, initial CI/CD setup focuses on code quality, testing, and maintainability. In future versions, this can extend to web-based deployment.

**GitHub Actions Setup:**

* Trigger on push and pull request to main branch
* Steps:
  + actions/checkout – Pull code from repo
  + setup-java – Set up Java environment (e.g., JDK 17)
  + build – Compile Java code
  + test – Run JUnit test cases (when added)
  + codeql-analysis – Static code scanning

**Sample Workflow YAML (Java CI):**

yaml

CopyEdit

name: Java CI

on: [push, pull\_request]

jobs:

build:

runs-on: ubuntu-latest

steps:

- uses: actions/checkout@v3

- name: Set up JDK 17

uses: actions/setup-java@v3

with:

java-version: '17'

distribution: 'temurin'

- name: Compile

run: javac -d bin src/\*\*/\*.java

- name: Run Tests

run: java -cp bin org.junit.runner.JUnitCore MyTests

**Future Enhancements:**

**Planned Features:**

**• User profile management**

**• Time entry editing**

**• Report generation**

**• Multiple time zones support**

**• Team management features**

**• Email notifications**

**• Mobile responsiveness improvements**

**2. Monitoring & Observability (Planned for Future Web Version)**

**Current Implementation:**

**• Console logging for:**

**- Login attempts**

**- Registration attempts**

**- Clock in/out events**

**- Database operations**

**- Error tracking**

This section is limited for desktop apps but useful when extending to a cloud version:

* **Logging:** Local file logs to track login and time events
* **Custom Metrics (Future):**
  + Daily login count
  + Average clock-in/out time
* **Alerting (Future):**
  + Email notification for missing clock-out
  + Late clock-ins beyond threshold time

**Incident Response Plan:**

* Log timestamp anomalies and alert user
* Maintain recovery logs to avoid data loss
* Enable manual record correction (admin access)

# Phase 4

**1. Infrastructure & Deployment**

**Deployment Strategy (Current):**

* Deployed as a **Java Desktop Application (.jar)** file
* Can run on any system with Java Runtime Environment (JRE)
* Simple double-click or command-line execution

**Deployment Instructions:**

1. Compile source: javac -d bin src/\*.java
2. Package app: jar cfe ClockTracker.jar MainClass -C bin .
3. Run app: java -jar ClockTracker.jar

**Multi-Environment Strategy (Planned):**

* **Dev Environment:** Local testing with test data files
* **Staging Environment (Future):** Web version hosted on localhost or Heroku (using Spring Boot)
* **Production Environment (Future):** Deployed with user database and live user access

**Blue-Green Deployment (Future Enhancement):**

* Maintain two versions (active and backup) of the app
* Roll back instantly in case of bugs during deployment

**Disaster Recovery (Planned):**

* Auto-backup .csv or database file daily
* Store logs for rollback or admin verification

**2. Documentation & Knowledge Base**

**Divio Framework:**

* **Tutorials:**
  + How to clock in/out
  + How to generate daily time report
* **How-To Guides:**
  + Add new user
  + Reset password (admin only)
* **Explanations:**
  + Why local storage is used initially
  + How time data is logged securely
* **References:**
  + Java classes used: LocalDateTime, Scanner, FileWriter, JTable
  + Folder structure, build instructions

**API Documentation (Future):**  
Current API Endpoints:

• POST /api/register - User registration

• POST /api/login - User authentication

• POST /api/time-entries/clock-in - Record clock in

• PATCH /api/time-entries/:id/clock-out - Record clock out

• GET /api/time-entries - Get time entries

**Architecture Decision Records (ADRs):**

* Use of Java Swing for fast development
* Chose file-based storage for simplicity and offline support
* Plan to migrate to RESTful backend and MySQL for web version