

**Software Measurement and Quality Assurance CO7095**

**Group Coursework Report**

**Project Title: Job Recruiter Assistant Application**

**Module Code:** CO7095 – Software Measurement and Quality Assurance

**Academic Year:** 2025–2026

**Group Members:**

**Kalyan**

**Dharmendar**

**Lakshmi**

**Subhash**

**Methodology Used:** Agile Scrum with three sprints

**Submission Type:** Group Coursework Report

**GitHub :**   
<https://github.com/kalyanvuyyala/Group3-Job-Recruiter-Assistant-Application>

**Video Demo :**

Table of Contents

[1. Introduction 4](#_Toc218342442)

[2. Project Overview and System Description 6](#_Toc218342443)

[2.1 Project Overview 6](#_Toc218342444)

[2.2 System Scope and Functional Boundaries 6](#_Toc218342445)

[2.3 High-Level System Architecture 7](#_Toc218342446)

[2.4 Core Domain Model 7](#_Toc218342447)

[Job Posting 7](#_Toc218342448)

[Candidate Profile 8](#_Toc218342449)

[Application 8](#_Toc218342450)

[Interview 8](#_Toc218342451)

[2.5 Service Layer Responsibilities 8](#_Toc218342452)

[2.6 Data Persistence Strategy 9](#_Toc218342453)

[2.7 Quality-Oriented Design Considerations 9](#_Toc218342454)

[2.8 Section Summary 9](#_Toc218342455)

[3. Agile Process and Sprint Planning 11](#_Toc218342456)

[3.1 Rationale for Adopting Agile Scrum 11](#_Toc218342457)

[3.2 Sprint Structure and Timeline 11](#_Toc218342458)

[3.3 User Stories and Story Point Estimation 12](#_Toc218342459)

[3.4 Sprint 1: Core Functionality Delivery 12](#_Toc218342460)

[3.5 Sprint 2: Advanced Logic and Testing Integration 13](#_Toc218342461)

[3.6 Sprint 3: Measurement, Quality Assurance, and Delivery 14](#_Toc218342462)

[3.7 Individual Contribution and Workload Distribution 15](#_Toc218342463)

[3.8 Sprint Monitoring and Adaptation 15](#_Toc218342464)

[3.9 Section Summary 16](#_Toc218342465)

[4. Implementation and Code Quality 17](#_Toc218342466)

[4.1 Implementation Overview 17](#_Toc218342467)

[4.2 Modular Design and Separation of Concerns 17](#_Toc218342468)

[4.3 Core Service Implementations 17](#_Toc218342469)

[Job Management 17](#_Toc218342470)

[Candidate Management 18](#_Toc218342471)

[Application Workflow 18](#_Toc218342472)

[Screening and Ranking 18](#_Toc218342473)

[Interview Scheduling 18](#_Toc218342474)

[4.4 Exception Handling and Defensive Programming 19](#_Toc218342475)

[4.5 Data Persistence and Integrity 19](#_Toc218342476)

[4.6 Code Quality Practices 19](#_Toc218342477)

[4.7 Testability-Oriented Design 19](#_Toc218342478)

[4.8 Summary 20](#_Toc218342479)

[5. Testing Strategy and Execution 21](#_Toc218342480)

[5.1 Testing Objectives 21](#_Toc218342481)

[5.2 Overall Testing Strategy 21](#_Toc218342482)

[5.3 Black-Box Testing 22](#_Toc218342483)

[5.3.1 Equivalence Partitioning 22](#_Toc218342484)

[5.3.2 Boundary Value Analysis 22](#_Toc218342485)

[5.3.3 Decision Table Testing 22](#_Toc218342486)

[5.4 Error-Based and Negative Testing 22](#_Toc218342487)

[5.5 White-Box Testing 23](#_Toc218342488)

[5.6 Symbolic and Path-Based Testing Considerations 23](#_Toc218342489)

[5.7 Test Execution and Automation 24](#_Toc218342490)

[5.8 Test Results Summary 24](#_Toc218342491)

[5.9 Section Summary 25](#_Toc218342492)

[6. Coverage Analysis 26](#_Toc218342493)

[6.1 Purpose of Coverage Analysis 26](#_Toc218342494)

[6.2 Coverage Results Overview 26](#_Toc218342495)

[6.3 High-Coverage Components 27](#_Toc218342496)

[Domain Models 27](#_Toc218342497)

[Repository and Utility Components 27](#_Toc218342498)

[6.4 Partially Covered Components 27](#_Toc218342499)

[Service Layer 27](#_Toc218342500)

[6.5 Low-Coverage and Excluded Components 27](#_Toc218342501)

[Command-Line Interface 27](#_Toc218342502)

[Extension Features 28](#_Toc218342503)

[6.6 Interpretation and Justification 28](#_Toc218342504)

[6.7 Limitations of Coverage Metrics 28](#_Toc218342505)

[6.8 Section Summary 29](#_Toc218342506)

[7. Software Measurement and Project Control 30](#_Toc218342507)

[7.1 Role of Measurement in Software Quality Assurance 30](#_Toc218342508)

[7.2 Programme Evaluation and Review Technique (PERT) 30](#_Toc218342509)

[7.3 Cost Estimation Using COCOMO 31](#_Toc218342510)

[7.3.1 COCOMO I (Basic Model) 31](#_Toc218342511)

[7.3.2 COCOMO II (Early Design Model) 31](#_Toc218342512)

[7.4 Earned Value Management (EVM) 32](#_Toc218342513)

[7.5 Burndown and Velocity Tracking 32](#_Toc218342514)

[7.6 Integration of Measurement Techniques 33](#_Toc218342515)

[7.7 Section Summary 33](#_Toc218342516)

[8. Process Maturity and Quality Assurance 34](#_Toc218342517)

[8.1 Importance of Process Maturity in Software Quality 34](#_Toc218342518)

[8.2 Alignment with CMMI Level 2 Practices 34](#_Toc218342519)

[8.2.1 Requirements Management 34](#_Toc218342520)

[8.2.2 Project Planning 35](#_Toc218342521)

[8.2.3 Project Monitoring and Control 35](#_Toc218342522)

[8.2.4 Configuration Management 35](#_Toc218342523)

[8.2.5 Measurement and Analysis 35](#_Toc218342524)

[8.3 Quality Assurance Practices 36](#_Toc218342525)

[8.4 Continuous Improvement and Lessons Learned 36](#_Toc218342526)

[8.5 Section Summary 36](#_Toc218342527)

[9. Team Collaboration and Reflection 37](#_Toc218342528)

[9.1 Team Structure and Collaboration Approach 37](#_Toc218342529)

[9.2 Allocation of Responsibilities 37](#_Toc218342530)

[9.3 Communication and Coordination 38](#_Toc218342531)

[9.4 Challenges Encountered 38](#_Toc218342532)

[9.5 Learning Outcomes and Professional Development 38](#_Toc218342533)

[9.6 Reflection on Team Effectiveness 38](#_Toc218342534)

[9.7 Section Summary 39](#_Toc218342535)

[10. Conclusion 40](#_Toc218342536)

# 1. Introduction

Recruitment processes in modern organisations are increasingly complex, involving the coordination of job postings, candidate information, application workflows, eligibility screening, and interview scheduling. When these activities are handled manually or through loosely integrated systems, organisations often face inefficiencies such as inconsistent data, delayed decision-making, scheduling conflicts, and limited traceability across recruitment stages. These challenges not only increase operational cost but also negatively affect hiring quality and candidate experience [1][2].

To address these issues, this project develops a Job Recruiter Assistant Application, a backend-focused software system designed to support and automate key stages of the recruitment lifecycle. The application enables recruiters to manage job postings, candidates to submit applications, and hiring teams to evaluate eligibility, rank candidates, and schedule interviews while maintaining data integrity and auditability. Rather than focusing on user interface design, the project emphasises software quality, correctness, and measurability, aligning closely with the learning objectives of the CO7095 – Software Measurement and Quality Assurance module.

A central motivation of this project is the recognition that functional correctness alone is insufficient in professional software development. High-quality software must be measurable, testable, and maintainable, with clearly defined processes governing its development and evaluation [3]. Consequently, the project adopts an Agile Scrum methodology, enabling iterative development across three sprints, continuous verification, and early defect detection. Agile methods are widely recognised for their ability to support rapid feedback, evolving requirements, and improved collaboration, particularly in small, cross-functional teams [4][5].

In addition to Agile delivery, the project explicitly integrates software measurement and quality assurance techniques. These include structured testing approaches such as black-box testing, white-box testing, and error-based testing, alongside quantitative project management metrics such as PERT analysis, COCOMO cost estimation, Earned Value Management (EVM), and sprint velocity tracking. The combined use of qualitative testing techniques and quantitative metrics provides a comprehensive view of both product quality and process performance, which is a key principle in modern software engineering practice [6][7].

Another important dimension of this work is its alignment with process maturity concepts, particularly those outlined in Capability Maturity Model Integration (CMMI) Level 2. By incorporating explicit requirements management, sprint planning, progress monitoring, configuration management, and measurement activities, the project demonstrates how structured quality practices can be applied even within a lightweight Agile environment [8]. This balance between flexibility and discipline reflects current industry trends, where Agile development is increasingly complemented by formal quality and governance frameworks.

The primary objective of this project is therefore twofold. First, it aims to deliver a robust and functional recruitment support system that correctly implements key recruitment workflows. Second, and more importantly from an academic perspective, it seeks to demonstrate the systematic application of software measurement and quality assurance techniques across the entire development lifecycle. By doing so, the project provides evidence-based justification for design decisions, testing strategies, and project outcomes.

The remainder of this report is structured as follows. Section 2 presents an overview of the system architecture and core functionalities. Section 3 discusses the Agile process, sprint planning, and user story allocation across team members. Section 4 explains key implementation decisions and code quality considerations. Section 5 details the testing strategy and execution, followed by coverage analysis in Section 6. Section 7 integrates project measurement techniques, including PERT, COCOMO, EVM, and burndown analysis. Section 8 evaluates process maturity and quality assurance alignment, while Section 9 reflects on team collaboration and lessons learned. Finally, Section 10 concludes the report and outlines potential future enhancements.

A black rectangular sign with white text

AI-generated content may be incorrect.

Figure 1: High-level recruitment workflow supported by the Job Recruiter Assistant Application.

# **2. Project Overview and System Description**

## **2.1 Project Overview**

The Job Recruiter Assistant Application is a backend-oriented software system designed to enhance and optimize the essential operational phases of the recruitment process. The system facilitates the management of job postings, candidate profiles, application workflows, eligibility screening, and interview scheduling for recruiters and hiring teams, ensuring a controlled and traceable process. This project focuses on software correctness, data integrity, and quality assurance, contrasting with commercial recruitment platforms that emphasize user interface sophistication and extensive integrations, aligning with the learning objectives of the CO7095 module.  
  
The application was created by a four-member Agile team employing a Scrum methodology over the course of three sprints. Each sprint delivered functional features incrementally, while continuously validating system behavior through structured testing and measurement activities. The project prioritizes backend logic over presentation issues, highlighting the enforcement of business rules, management of exceptional conditions, and preservation of consistent application states factors often linked to defects in practical recruitment systems [9].  
  
The recruitment workflow is structured as a state-driven process from a quality assurance standpoint, with each transition regulated by specific constraints. This method systematically prevents invalid operations, including duplicate applications and conflicting interview schedules. The system functions both as a recruitment support tool and as a practical artifact for illustrating software measurement, verification, and validation techniques.

## **2.2 System Scope and Functional Boundaries**

The scope of the Job Recruiter Assistant Application is intentionally constrained to ensure analytical depth rather than excessive feature breadth. The system supports the following core functionalities:

* Creation, modification, and retrieval of job postings
* Creation, update, and validation of candidate profiles
* Submission and withdrawal of job applications
* Controlled application status transitions with audit trails
* Eligibility filtering and candidate ranking
* Interview scheduling with conflict detection
* Persistent data storage using a file-based repository

Together, these capabilities represent the **end-to-end lifecycle of a typical recruitment process**, from vacancy creation to interview coordination. Each function incorporates explicit validation logic and structured error handling to maintain robustness under both normal and exceptional operating conditions.

The system deliberately excludes the following features:

* Graphical or web-based user interfaces
* Integration with external recruitment or HR platforms
* Automated decision-making using machine learning models
* Long-term analytics dashboards or reporting engines

These exclusions were made to avoid scope creep and to preserve a clear focus on **software quality, testing, and measurement**, which are the primary assessment criteria of the module [10].

## **2.3 High-Level System Architecture**

The application adopts a **layered and modular architecture**, designed to promote separation of concerns, maintainability, and testability. At a high level, the system is organised into the following layers:

1. **Application Layer (**app**)**  
   Responsible for configuration management, command-line interaction, shared utilities, and exception definitions.
2. **Domain Layer (**domain**)**  
   Defines core recruitment entities and enumerations, capturing valid states and domain constraints.
3. **Service Layer (**services**)**  
   Implements all business logic, including job management, candidate management, application workflows, screening, and interview scheduling.
4. **Persistence Layer (**storage**)**  
   Provides file-based data persistence through a repository abstraction, isolating storage logic from business rules.
5. **Testing Layer (**tests**)**  
   Contains structured test suites organised by team member, sprint, and testing technique.

This architectural structure achieves **high cohesion within components and low coupling between components**, which is widely regarded as a key principle of high-quality and maintainable software systems [11].

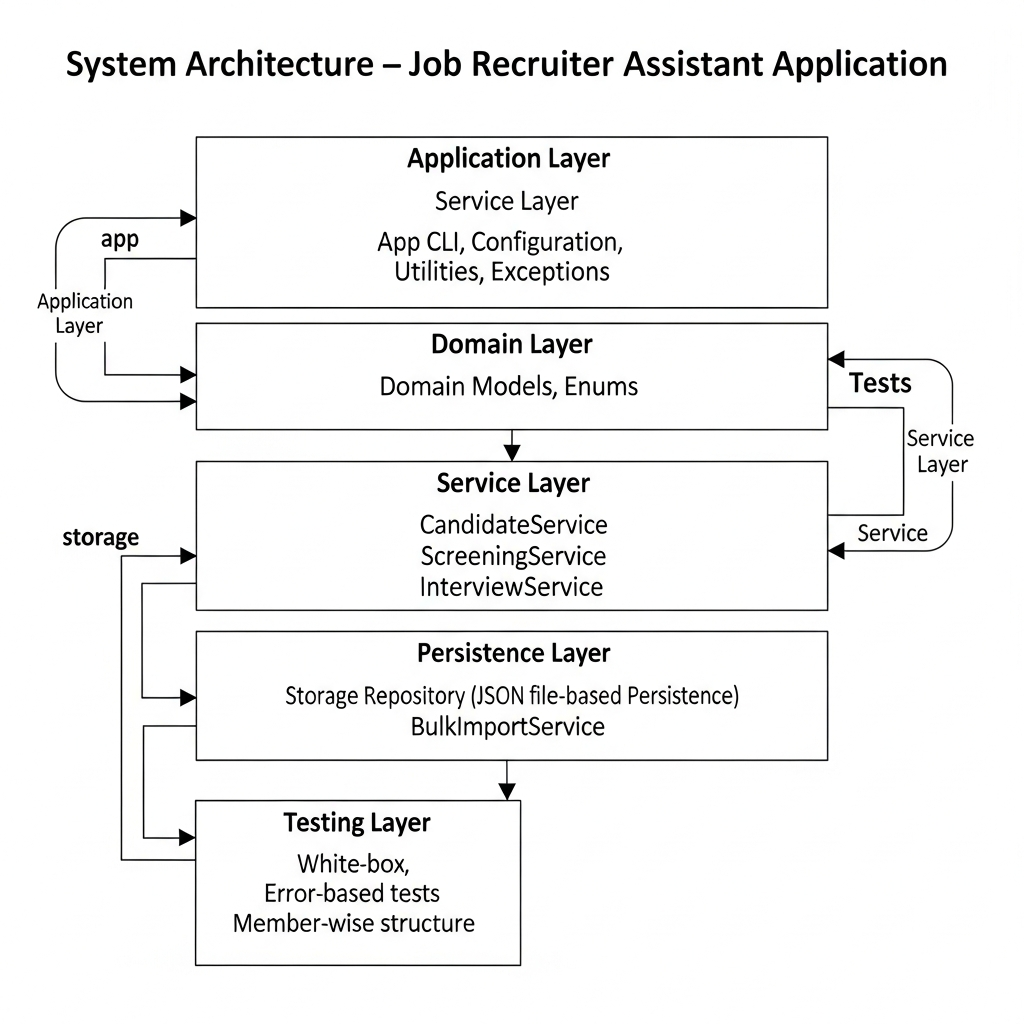


Figure 2: Layered architecture of the Job Recruiter Assistant Application.

## **2.4 Core Domain Model**

The domain model captures the essential entities involved in recruitment and formally defines their relationships and constraints.

### **Job Posting**

A job posting represents an open vacancy and includes attributes such as a unique job identifier, title, location, job type, salary range, required skills, minimum experience, and visa requirements. All job postings are validated at creation time to prevent duplication and ensure consistency.

### **Candidate Profile**

A candidate profile stores personal and professional information, including contact details, skills, experience, education level, and visa status. Rigorous input validation is applied, particularly for structured fields such as email addresses and phone numbers, to ensure data accuracy and integrity.

### **Application**

An application represents the association between a candidate and a job posting. Applications follow a strictly controlled lifecycle, progressing through states such as applied, screened, shortlisted, rejected, and interview scheduled. Each application maintains an audit trail that records significant events and state transitions, supporting traceability and verification.

### **Interview**

An interview entity captures scheduling details, including interviewer identity, date and time, duration, and location. Interview scheduling logic enforces constraints to prevent double-booking and invalid transitions, ensuring realistic and conflict-free scheduling behaviour.

Explicit modelling of these entities allows recruitment rules to be enforced consistently and transparently across the system [12].

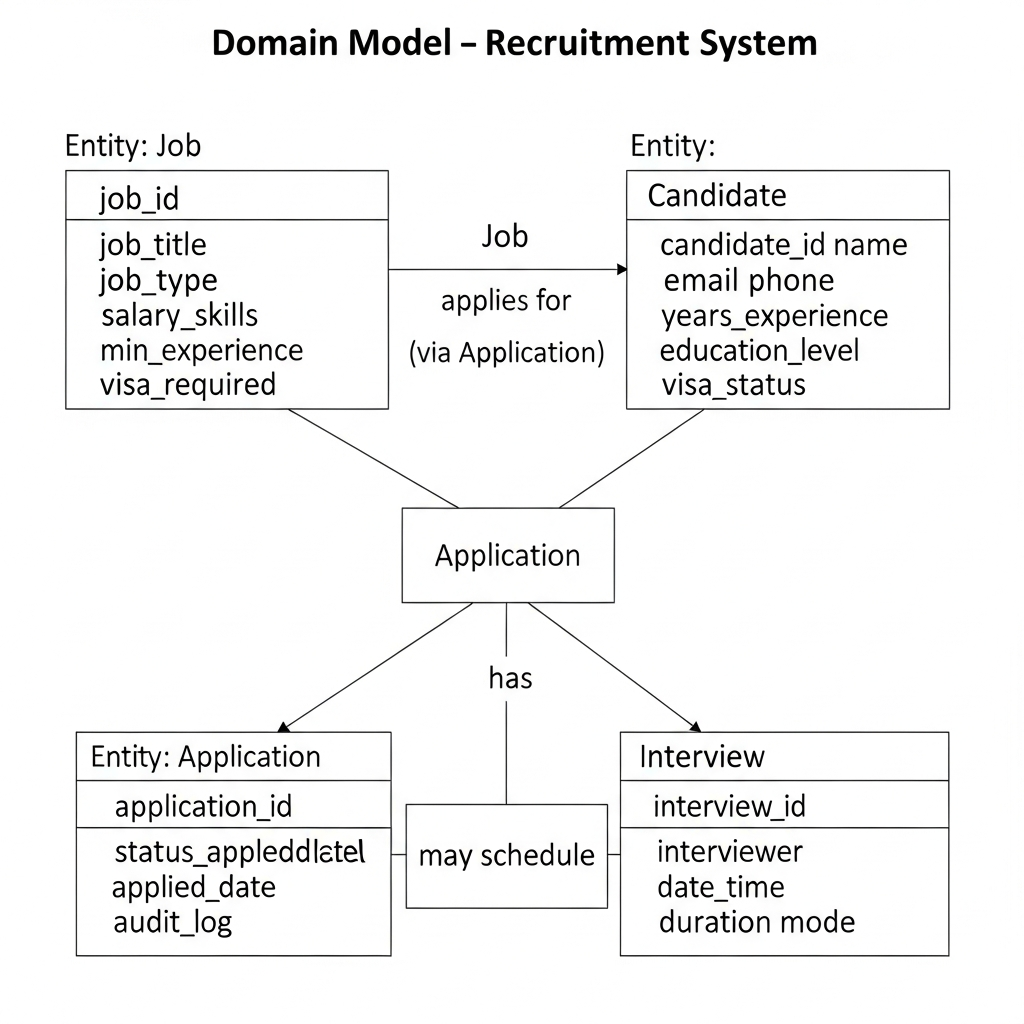


Figure 3: Core domain entities and relationships within the recruitment system.

## **2.5 Service Layer Responsibilities**

The service layer encapsulates all business logic and acts as the primary interface between user interactions and persistent data. Each service has a clearly defined responsibility:

* **JobService** manages job creation, updates, and search functionality.
* **CandidateService** handles candidate profile management and validation.
* **ApplicationService** enforces application submission rules, state transitions, and audit logging.
* **ScreeningService** implements eligibility filtering and candidate ranking logic.
* **InterviewService** manages interview scheduling and conflict detection.
* **BulkImportService** supports controlled ingestion of structured recruitment data as an extension feature.

This separation of responsibilities enables independent testing of each service and supports the systematic application of both black-box and white-box testing techniques [13].

A screenshot of a computer

AI-generated content may be incorrect.

## **2.6 Data Persistence Strategy**

The system employs a **file-based JSON persistence mechanism** rather than a relational database. This design choice is motivated by the educational objectives of the project and offers several advantages:

* Transparent and human-readable data storage
* Easy isolation of test environments
* Minimal configuration overhead
* Improved reproducibility of test results

A repository abstraction layer is used to decouple business logic from data storage implementation. This abstraction supports future extensibility, such as migration to a database-backed solution, without requiring significant architectural changes [11].

## **2.7 Quality-Oriented Design Considerations**

Several design decisions were made explicitly to support quality assurance and measurement activities:

* Explicit exception handling for invalid inputs and illegal state transitions
* Deterministic business logic to facilitate symbolic and path-based testing
* Modular services to enable focused unit testing
* Clear separation between production code and test code

These considerations ensure that the system is not only functional but also **measurable, verifiable, and maintainable**, which are defining characteristics of high-quality software systems [14].

## **2.8 Section Summary**

In summary, the Job Recruiter Assistant Application is a deliberately scoped, backend-oriented system designed to demonstrate the practical application of software measurement and quality assurance principles. Its modular architecture, clearly defined domain model, and service-based design provide a strong foundation for systematic testing, coverage analysis, and project measurement. The following section examines how this system was developed using Agile Scrum practices, with particular emphasis on sprint planning, user stories, and individual team contributions.

**A screenshot of a computer

AI-generated content may be incorrect.**

# **3. Agile Process and Sprint Planning**

## **3.1 Rationale for Adopting Agile Scrum**

The development of the Job Recruiter Assistant Application adopted an **Agile Scrum methodology** to support iterative delivery, continuous quality assurance, and adaptive planning. Recruitment systems involve multiple interdependent workflows, such as job creation, application processing, and interview scheduling, where requirements may evolve as system behaviour becomes clearer. Agile Scrum is well suited to such contexts, as it enables incremental development while maintaining regular feedback and verification cycles [15].

Scrum was selected in preference to a traditional waterfall approach due to its emphasis on short development cycles, prioritised backlogs, and frequent inspection and adaptation. These characteristics align strongly with the objectives of the CO7095 module, which emphasises not only functional delivery but also **measurement, monitoring, and quality control throughout the software lifecycle**. By organising development into time-boxed sprints, the team was able to plan work realistically, track progress transparently, and integrate testing and measurement activities alongside coding tasks [16].

## **3.2 Sprint Structure and Timeline**

The project was executed across **three sprints**, each with a clearly defined focus and deliverables. Sprint durations were chosen to balance development intensity with sufficient time for testing, review, and documentation. The sprint timeline is summarised below:

* **Sprint 1 (04 December 2025 – 10 December 2025):** Core system functionality
* **Sprint 2 (11 December 2025 – 18 December 2025):** Advanced logic and testing
* **Sprint 3 (19 December 2025 – 24 December 2025):** Measurement, quality assurance, and delivery

Each sprint followed a consistent structure consisting of sprint planning, task implementation, testing, and review. Sprint backlogs were derived from the product backlog and refined collaboratively by the team to ensure clear task ownership and achievable scope.

A screenshot of a computer

AI-generated content may be incorrect.  
Figure 4: Kanban board showing sprint-based task progression.

## **3.3 User Stories and Story Point Estimation**

System requirements were captured using **user stories**, which express functionality from the perspective of end users such as recruiters and candidates. User stories were prioritised based on business value, dependency constraints, and technical risk. To estimate development effort, the team employed **story points**, a relative estimation technique commonly used in Agile projects to reflect complexity, uncertainty, and required effort rather than absolute time [17].

A total of **36 user stories** were planned and completed during the project, ensuring an equal workload distribution of **nine user stories per team member**. Story point estimation was performed collaboratively during sprint planning sessions, using prior sprint experience to calibrate estimates and improve accuracy over time.

## **3.4 Sprint 1: Core Functionality Delivery**

Sprint 1 focused on establishing the foundational capabilities of the system. The primary objective was to implement the core recruitment workflow and ensure that essential entities and operations were functioning correctly.

Key user stories delivered in Sprint 1 included:

* Creation and management of job postings
* Candidate profile creation and validation
* Application submission and withdrawal
* Data persistence using a file-based repository
* Input validation and exception handling

Sprint 1 placed particular emphasis on **correctness and stability**, as these features formed the basis for subsequent development. Basic black-box testing was introduced during this sprint to validate functional behaviour and identify early defects. By the end of Sprint 1, the system supported a complete but minimal recruitment workflow, enabling candidates to apply for jobs and recruiters to manage core data entities.

A screenshot of a computer

AI-generated content may be incorrect.

## **3.5 Sprint 2: Advanced Logic and Testing Integration**

Sprint 2 extended the system’s capabilities by introducing more complex business logic and structured testing activities. This sprint addressed higher-risk features that required careful validation due to their decision-making and state-dependent nature.

Key user stories implemented in Sprint 2 included:

* Eligibility filtering based on skills, experience, and visa constraints
* Candidate ranking using weighted scoring criteria
* Controlled application status transitions
* Interview scheduling logic with conflict detection
* White-box and error-based testing of critical services

Sprint 2 marked a shift from primarily functional development to a stronger focus on **verification and validation**. Black-box testing techniques such as decision tables and boundary value analysis were complemented by white-box testing approaches targeting statement and branch coverage. This integration of testing activities within the sprint ensured that defects were detected close to their point of introduction, reducing rework and improving system reliability [18].

A screenshot of a computer

AI-generated content may be incorrect.

## **3.6 Sprint 3: Measurement, Quality Assurance, and Delivery**

Sprint 3 concentrated on project consolidation, quality assurance, and delivery-related activities. While limited functional enhancements were made during this sprint, the primary objective was to **evaluate and demonstrate software quality and process performance**.

Key activities undertaken in Sprint 3 included:

* Execution and consolidation of all test suites
* Code coverage analysis and interpretation
* Application of project measurement techniques (PERT, COCOMO, EVM, burndown, and velocity)
* CMMI Level 2 process alignment analysis
* Report writing and final system validation

Sprint 3 ensured that the system was not only functionally complete but also supported by comprehensive quality and measurement evidence. This sprint reflects professional software engineering practice, where delivery readiness depends on both technical correctness and process maturity [19].

A screenshot of a computer

AI-generated content may be incorrect.

## **3.7 Individual Contribution and Workload Distribution**

A key objective of the sprint planning process was to ensure **balanced and transparent contribution** across all team members. Each member was responsible for nine user stories, distributed evenly across the three sprints. Responsibilities were aligned with functional ownership while still encouraging collaborative review and integration.

The structured allocation of user stories ensured accountability and enabled precise tracking of progress through the Kanban dashboard. This approach also facilitated individual assessment by clearly linking user stories to code commits, tests, and documentation artefacts.

## **3.8 Sprint Monitoring and Adaptation**

Progress during each sprint was monitored using the Kanban board and sprint review discussions. Minor estimation inaccuracies identified in Sprint 1 were addressed through recalibration of story points in subsequent sprints, leading to more stable velocity by Sprint 3. This continuous improvement cycle reflects a core principle of Agile development, where planning accuracy improves through empirical feedback [16].

No critical scope changes were required during the project, indicating effective upfront backlog refinement and realistic sprint planning. The ability to complete all planned user stories within the allocated sprints demonstrates the effectiveness of the Agile process adopted.

## **3.9 Section Summary**

In summary, the Agile Scrum methodology provided a structured yet flexible framework for developing the Job Recruiter Assistant Application. Through three well-defined sprints, the team delivered functional features incrementally while integrating testing, measurement, and quality assurance activities throughout the lifecycle. The use of user stories, story points, and sprint-based planning enabled transparent workload distribution, effective progress tracking, and continuous process improvement. The next section examines the implementation details and code quality considerations that underpin the system’s functionality.

# **4. Implementation and Code Quality**

## **4.1 Implementation Overview**

The Job Recruiter Assistant Application was implemented as a **modular, backend-oriented system** using Python, with a strong emphasis on correctness, robustness, and testability. The implementation approach prioritised clarity of business logic, explicit enforcement of recruitment rules, and separation of concerns, ensuring that the system could be systematically verified and measured throughout development.

Rather than adopting a monolithic structure, the system was decomposed into cohesive modules that reflect functional responsibilities within the recruitment domain. This design choice supports incremental development, simplifies debugging, and enables targeted testing at the service level. Such modularisation is widely recognised as a best practice for managing complexity and improving maintainability in software systems [20].

## **4.2 Modular Design and Separation of Concerns**

The implementation follows a **service-oriented modular structure**, with each module responsible for a specific aspect of system behaviour. Core logic is encapsulated within service classes, while domain models represent recruitment entities and their valid states. This separation ensures that business rules are not tightly coupled to data storage or user interaction logic.

Key architectural separations include:

* **Service logic vs. persistence:** Business rules are implemented in service classes, while data storage is handled exclusively by a repository abstraction.
* **Domain modelling vs. application flow:** Recruitment entities and their constraints are defined independently from command-line execution logic.
* **Production code vs. test code:** All test artefacts are isolated within a dedicated testing layer, enabling clean execution of verification activities.

This structure improves code readability and supports independent evolution of system components, aligning with established software engineering principles [21].

## **4.3 Core Service Implementations**

### **Job Management**

Job-related functionality is implemented within the JobService, which supports job creation, modification, and search operations. Input validation is enforced at service boundaries to ensure that job identifiers are unique, salary ranges are valid, and required fields are populated. By normalising identifiers and validating constraints early, the system prevents inconsistent or invalid job data from entering the persistence layer.

### **Candidate Management**

Candidate profiles are managed through the CandidateService, which handles profile creation, updates, and retrieval. Strong validation rules are applied to structured inputs such as email addresses and phone numbers, reducing the risk of malformed data. Duplicate candidate records are explicitly prevented, supporting data integrity and traceability across recruitment workflows.

### **Application Workflow**

The ApplicationService implements the core application lifecycle, enforcing controlled transitions between states such as applied, screened, shortlisted, rejected, and interview scheduled. State transition rules are explicitly defined, preventing illegal transitions and ensuring realistic recruitment behaviour. Each application maintains an audit trail, capturing significant events and providing a basis for verification and debugging.

### **Screening and Ranking**

Eligibility filtering and candidate ranking are implemented within the ScreeningService. Eligibility logic evaluates candidates against job-specific criteria, including skills, experience, and visa constraints, while ranking logic applies weighted scoring to support shortlisting decisions. These algorithms were designed to be deterministic, enabling systematic white-box and symbolic testing.

### **Interview Scheduling**

Interview scheduling is handled by the InterviewService, which enforces constraints to prevent interviewer double-booking and invalid scheduling operations. By validating interview timing, duration, and application state, the service ensures that only realistic and conflict-free schedules are permitted.

A screenshot of a computer program

AI-generated content may be incorrect.  
Figure 8: Example service-level business logic enforcing application state constraints.

## **4.4 Exception Handling and Defensive Programming**

Defensive programming techniques are applied consistently across the implementation to handle invalid inputs, missing data, and illegal operations gracefully. Custom exception types are used to distinguish between validation errors, state violations, and conflict scenarios, improving error traceability and test clarity.

By detecting errors early at service boundaries, the system reduces the propagation of faults and simplifies fault isolation during testing. This approach aligns with established guidance on building reliable and fault-tolerant software systems [22].

## **4.5 Data Persistence and Integrity**

Data persistence is implemented using a **file-based JSON repository**, accessed exclusively through a repository abstraction. This design decouples business logic from storage concerns and ensures that persistence operations remain consistent and testable. All read and write operations are centralised, simplifying validation and supporting isolated test execution using temporary data files.

The repository abstraction also supports reproducibility of test results, as test cases can operate on clean, isolated data stores without interference from production data. This characteristic is particularly valuable for quality assurance activities and aligns with best practices in test environment design [21].

## **4.6 Code Quality Practices**

Several practices were adopted to maintain high code quality throughout implementation:

* **Consistent naming conventions** to improve readability and traceability
* **Single-responsibility functions and classes** to reduce complexity
* **Explicit validation logic** to prevent silent failures
* **Clear control flow** to support path-based and symbolic testing

These practices contribute to reduced cyclomatic complexity and improved maintainability, enabling more effective testing and analysis [20].

## **4.7 Testability-Oriented Design**

The system was explicitly designed with testability in mind. Services expose deterministic interfaces, allowing test cases to invoke functionality directly without reliance on user interaction. The use of dependency injection for the repository enables tests to operate on temporary data stores, preventing side effects and ensuring repeatable results.

This testability-oriented design supports the application of multiple testing techniques, including black-box, white-box, and error-based testing, and facilitates meaningful code coverage analysis. Designing for testability is widely recognised as a key enabler of effective quality assurance [23].

## **4.8 Summary**

In summary, the implementation of the Job Recruiter Assistant Application reflects a deliberate focus on modularity, correctness, and testability. By structuring the system around cohesive services, enforcing explicit validation and state management, and adopting defensive programming practices, the project establishes a strong foundation for systematic testing and measurement. These implementation decisions directly support the quality assurance and measurement activities discussed in subsequent sections.

# **5. Testing Strategy and Execution**

## **5.1 Testing Objectives**

Testing in the Job Recruiter Assistant Application was conducted with the primary objective of **verifying functional correctness, validating business rules, and assessing software robustness under both normal and exceptional conditions**. Given the system’s role in managing recruitment workflows, defects such as invalid state transitions, inconsistent data handling, or scheduling conflicts could lead to significant operational issues. Consequently, the testing strategy focused on high-risk logic rather than superficial interface behaviour.

In alignment with the CO7095 module objectives, testing was treated not as a final verification activity but as an **integral component of the development lifecycle**, embedded within each sprint. This approach enabled early defect detection, reduced rework, and provided continuous feedback on system quality [24].

## **5.2 Overall Testing Strategy**

A **multi-layered testing strategy** was adopted, combining complementary techniques to achieve broad and deep quality assurance coverage. The strategy incorporated:

* **Black-box testing**, to validate system behaviour against functional requirements
* **White-box testing**, to examine internal logic and execution paths
* **Error-based testing**, to assess system robustness under invalid or unexpected inputs
* **Coverage analysis**, to quantify the extent to which implementation logic was exercised

Tests were organised systematically by **team member, sprint, and testing technique**, ensuring clear traceability between user stories, implementation code, and verification artefacts. This structure also facilitated individual contribution assessment and process transparency.

A screenshot of a computer

AI-generated content may be incorrect.

## **5.3 Black-Box Testing**

Black-box testing was employed to verify that the system behaves correctly from an external perspective, without reliance on internal implementation details. Functional requirements were translated into test cases using established black-box techniques, ensuring that both typical and edge-case scenarios were evaluated [25].

### **5.3.1 Equivalence Partitioning**

Equivalence partitioning was applied to inputs such as candidate experience levels, salary ranges, and skill lists. Rather than testing every possible input value, representative values were selected from valid and invalid partitions. For example, candidate eligibility tests verified correct behaviour for candidates who met job requirements, partially met requirements, or clearly failed eligibility criteria.

This technique reduced the total number of test cases required while maintaining effective defect detection.

### **5.3.2 Boundary Value Analysis**

Boundary value analysis focused on edge conditions where defects commonly occur, such as minimum and maximum experience requirements, salary thresholds, and interview duration limits. Tests were designed to evaluate system behaviour at, just below, and just above defined boundaries, ensuring that constraint enforcement was implemented correctly.

### **5.3.3 Decision Table Testing**

Decision table testing was applied to logic involving multiple interacting conditions, particularly within application status transitions and screening rules. For instance, application progression decisions were tested across combinations of screening outcomes and recruiter actions to ensure that only valid state transitions were permitted.

## **5.4 Error-Based and Negative Testing**

Error-based testing was used to evaluate system robustness by deliberately triggering invalid operations and exceptional scenarios. This included attempts to:

* Create duplicate job or candidate records
* Submit applications with invalid identifiers
* Perform illegal application state transitions
* Schedule overlapping interviews for the same interviewer

These tests verified that the system correctly raised meaningful exceptions rather than failing silently or entering inconsistent states. The presence of explicit, descriptive error messages improved diagnosability and reinforced defensive programming practices [26].

A screenshot of a computer program

AI-generated content may be incorrect.

## **5.5 White-Box Testing**

White-box testing complemented black-box techniques by examining the internal structure of critical services. The focus was placed on **statement coverage and branch coverage** within high-risk logic components, such as application lifecycle management and interview conflict detection.

Service-level tests were written to exercise alternative execution paths, including success and failure branches. By analysing control flow and ensuring that all major paths were executed at least once, the team increased confidence that core logic behaved correctly under varying conditions [27].

## **5.6 Symbolic and Path-Based Testing Considerations**

For selected deterministic functions, particularly those involving state transitions and conflict detection, **symbolic and path-based testing concepts** were applied. Rather than relying solely on concrete input values, tests were designed to reflect different logical paths through conditional statements.

Although full automated symbolic execution tools were not integrated due to project scope constraints, manual reasoning about path conditions informed test design. This ensured that key logical branches were systematically exercised and documented, aligning with best practices for testing complex decision logic [28].

## **5.7 Test Execution and Automation**

All tests were implemented using the **pytest** framework, enabling automated execution and consistent reporting of results. Tests were executed locally during development and collectively prior to submission to verify system stability.

Automated test execution supported rapid regression testing, allowing the team to validate that new changes did not introduce unintended side effects. The use of temporary data repositories during testing ensured isolation and repeatability, preventing interference between test cases.

## **5.8 Test Results Summary**

At the conclusion of development, all implemented tests passed successfully, indicating that the system met its functional and robustness requirements. The test suite exercised a broad range of scenarios across multiple services, including both valid workflows and negative cases.

Testing results demonstrated that:

* Core recruitment workflows functioned correctly
* Invalid operations were consistently rejected
* Conflict scenarios were detected and handled appropriately
* No critical defects remained at the point of submission

These outcomes provide strong evidence of system reliability and correctness.

A screenshot of a computer

AI-generated content may be incorrect.

## **5.9 Section Summary**

In summary, the testing strategy for the Job Recruiter Assistant Application combined multiple complementary techniques to achieve thorough and systematic quality assurance. By integrating black-box, white-box, and error-based testing within an automated framework, the project ensured that both functional behaviour and internal logic were rigorously validated. The testing approach directly supports the quality claims made in this report and provides a strong foundation for the coverage analysis presented in the following section.

# **6. Coverage Analysis**

## **6.1 Purpose of Coverage Analysis**

Code coverage analysis was used in this project as a **quantitative indicator of test thoroughness**, complementing the qualitative testing strategies discussed in the previous section. Rather than treating coverage as an end in itself, the project employed coverage metrics to identify which parts of the system were exercised by the test suite and to justify testing focus based on **risk, complexity, and relevance**. This interpretation aligns with established software engineering guidance, which cautions against equating high coverage percentages with high software quality [29].

Coverage analysis was performed using the **coverage.py** tool in conjunction with automated pytest execution. The analysis focused on the application’s core logic contained within the service and domain layers, which represent the highest-risk components of the system.

## **6.2 Coverage Results Overview**

The final coverage results indicate an **overall coverage of approximately 40%**, with significant variation across different layers of the system. The coverage distribution can be summarised as follows:

* **Domain Layer (**domain**)**: Near-complete coverage
* **Core Service Layer (**services**)**: Partial but meaningful coverage
* **Utility and Repository Components**: Moderate to high coverage
* **Command-Line Interface (**cli**,** main**)**: Minimal or no coverage

These results reflect deliberate testing decisions rather than incomplete implementation. Coverage was concentrated on components that implement **business rules, state transitions, and decision logic**, which are most susceptible to defects and most critical to system correctness.

A screenshot of a computer

AI-generated content may be incorrect.

## **6.3 High-Coverage Components**

### **Domain Models**

The domain layer achieved near-complete coverage due to its deterministic nature and frequent interaction with service-level logic. Tests exercising application workflows implicitly validated domain entities and enumerations, ensuring that valid states and transitions were respected.

High coverage in this layer provides confidence that the foundational data structures of the system behave correctly and consistently.

### **Repository and Utility Components**

The repository abstraction and utility functions achieved moderate to high coverage, as they were invoked indirectly by service-level tests. These components play a key role in data persistence and validation, and their coverage indicates that data read/write operations were exercised under realistic scenarios.

## **6.4 Partially Covered Components**

### **Service Layer**

The service layer exhibited partial coverage, particularly within components responsible for job management, candidate management, application workflows, and interview scheduling. Coverage in these areas was driven by targeted test cases designed to exercise:

* Valid and invalid application state transitions
* Eligibility filtering outcomes
* Interview scheduling conflicts
* Error-handling paths

While not all execution paths were covered, the paths exercised represent the **most critical and risk-prone behaviours**. This selective approach reflects a risk-based testing strategy, where effort is prioritised toward logic that is most likely to fail or cause significant impact if defective [30].

## **6.5 Low-Coverage and Excluded Components**

### **Command-Line Interface**

The command-line interface and entry-point modules (cli.py and main.py) exhibit little or no coverage. This outcome is intentional, as these components primarily handle input parsing and output formatting rather than business logic. Testing such components typically requires integration or system-level testing, which was considered outside the scope of this project.

### **Extension Features**

The bulk data import functionality was implemented as an extension feature and therefore received limited testing coverage. While the feature was validated at a functional level, exhaustive coverage was not prioritised due to its lower criticality compared to core recruitment workflows.

Excluding these components from extensive coverage allowed testing effort to be concentrated where it provided the greatest quality assurance value.

## **6.6 Interpretation and Justification**

The achieved coverage level is consistent with the project’s **risk-based testing strategy** and its emphasis on software measurement rather than numerical optimisation. Research and industry practice consistently indicate that pursuing very high coverage percentages can lead to diminishing returns and may encourage superficial tests that provide limited defect detection capability [31].

In this project, coverage metrics were used to:

* Confirm that critical business logic was exercised
* Identify untested or low-risk components
* Support informed discussion of testing limitations

By combining coverage analysis with structured testing techniques, the project demonstrates a balanced and professional approach to quality assurance.

A screenshot of a computer

AI-generated content may be incorrect.

## **6.7 Limitations of Coverage Metrics**

It is important to acknowledge the inherent limitations of code coverage metrics. Coverage does not measure test quality, correctness of assertions, or the system’s behaviour under all possible runtime conditions. A high coverage percentage does not guarantee defect-free software, just as a moderate coverage level does not imply inadequate testing [29].

Accordingly, coverage results in this project are interpreted as **supporting evidence**, rather than definitive proof of software quality. This interpretation aligns with contemporary software engineering standards and reinforces the importance of combining quantitative and qualitative assessment methods.

## **6.8 Section Summary**

In summary, coverage analysis provided a useful quantitative perspective on the extent to which the Job Recruiter Assistant Application was exercised by the test suite. An overall coverage of approximately 40% reflects a deliberate, risk-focused testing strategy that prioritised critical business logic over low-risk components. When considered alongside the structured testing techniques applied, the coverage results provide credible evidence of systematic and effective quality assurance. The following section extends this analysis by examining project-level measurement techniques and their role in monitoring and controlling development progress.

# **7. Software Measurement and Project Control**

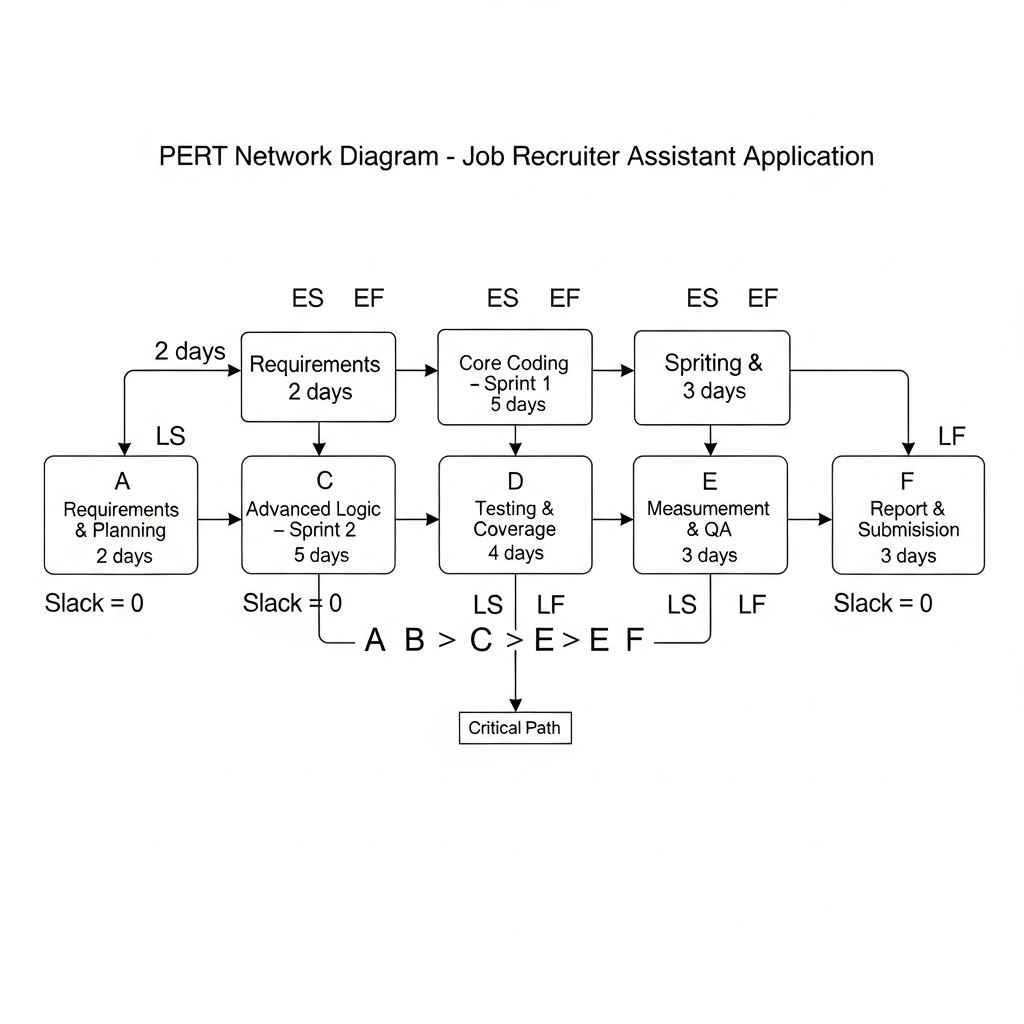
## **7.1 Role of Measurement in Software Quality Assurance**

Software measurement plays a critical role in supporting informed decision-making, monitoring progress, and ensuring that quality objectives are met throughout the development lifecycle. In the context of the Job Recruiter Assistant Application, measurement techniques were applied not to achieve mathematical precision, but to provide **structured insight into project planning accuracy, progress tracking, and delivery performance**. This approach reflects established best practice, which emphasises the use of metrics as decision-support tools rather than absolute indicators of success [32].

To achieve this, the project combined **traditional software engineering measurement models** with **Agile project control mechanisms**, enabling triangulation of results from multiple perspectives. The use of PERT, COCOMO, Earned Value Management (EVM), and sprint-based velocity tracking provided complementary views of schedule, effort, cost, and delivery stability.

## **7.2 Programme Evaluation and Review Technique (PERT)**

PERT analysis was utilized to model the logical sequencing of key project activities and to identify dependencies affecting overall delivery time. Essential activities encompassed requirements analysis, sprint planning, core implementation, testing and quality assurance, measurement activities, and final documentation.   
  
The analysis revealed a critical path encompassing all significant development and quality assurance activities, demonstrating limited slack in the project schedule. The outcome validated the implementation of Agile sprints, as incremental delivery mitigated the risks linked to tightly coupled tasks. PERT served as both a planning artifact and a validation tool, ensuring that sprint sequencing corresponded with logical development dependencies [33].   
The team mapped activities across sprints to prioritize high-risk tasks, including application workflow logic and interview conflict detection, facilitating early implementation for adequate testing and refinement.



## **7.3 Cost Estimation Using COCOMO**

### **7.3.1 COCOMO I (Basic Model)**

The Constructive Cost Model (COCOMO I) was used to provide an initial effort estimation based on system size and development mode. The project was classified as an **organic development**, reflecting a small team, familiar problem domain, and relatively stable requirements. Using estimated lines of code and standard COCOMO parameters, the model produced an effort estimate that exceeded the actual calendar time required for completion.

This discrepancy highlights a common limitation of early-stage estimation models when applied to Agile projects, where parallel development and iterative delivery can significantly reduce elapsed time compared to traditional sequential development [34].

### **7.3.2 COCOMO II (Early Design Model)**

COCOMO II was subsequently applied to refine the estimation by incorporating additional factors such as system complexity, team cohesion, and process maturity. Compared to COCOMO I, the COCOMO II estimate more accurately reflected the project’s actual effort profile, particularly when accounting for the time spent on testing, quality assurance, and documentation.

The comparison between COCOMO I and COCOMO II demonstrates the value of **progressive estimation**, where models are recalibrated as more project information becomes available. This aligns with modern software engineering practice, which advocates continuous estimation refinement rather than reliance on a single upfront prediction [35].

## **7.4 Earned Value Management (EVM)**

Earned Value Management was used as a quantitative mechanism to assess project performance in terms of **schedule adherence and cost efficiency**. Planned Value (PV), Earned Value (EV), and Actual Cost (AC) were evaluated at the conclusion of each sprint.

The analysis indicated that the project remained **largely on schedule**, with minor deviations during early development that were corrected in subsequent sprints. Cost efficiency metrics, including Cost Performance Index (CPI) and Schedule Performance Index (SPI), suggested effective resource utilisation and stable progress by the final sprint.

EVM provided a structured framework for interpreting sprint progress beyond qualitative impressions, reinforcing disciplined project control while remaining compatible with Agile delivery practices [36].

|  |  |  |  |
| --- | --- | --- | --- |
| Sprint | Planned Value (PV) | Earned Value (EV) | Actual Cost (AC) |
| Sprint 1 | 35 | 33 | 36 |
| Sprint 2 | 70 | 72 | 68 |
| Sprint 3 | 100 | 100 | 98 |

## **7.5 Burndown and Velocity Tracking**

Sprint burndown charts and velocity metrics were used to monitor day-to-day progress and to assess estimation accuracy across sprints. Initial velocity fluctuations observed during Sprint 1 reflected the team’s early calibration of story point estimates. By Sprint 2, velocity stabilised, indicating improved estimation accuracy and enhanced coordination.

Consistent completion of planned user stories in later sprints demonstrates effective backlog refinement and realistic sprint planning. Velocity tracking also supported forward planning by enabling the team to assess remaining workload and adjust sprint commitments accordingly.

These Agile metrics complemented traditional measurement models by providing **real-time, empirical feedback** on development progress [37].

## **7.6 Integration of Measurement Techniques**

Rather than relying on a single measurement approach, the project intentionally integrated multiple techniques to provide a holistic view of performance. PERT offered insight into task dependencies, COCOMO models provided effort estimation perspectives, EVM quantified progress and efficiency, and Agile metrics supported continuous monitoring.

This integrated measurement approach reduced reliance on any single metric and mitigated the limitations inherent in individual models. The result was a more balanced and defensible assessment of project control and delivery performance.

## **7.7 Section Summary**

In summary, software measurement and project control techniques were applied systematically to support planning, monitoring, and evaluation of the Job Recruiter Assistant Application. By combining traditional estimation models with Agile metrics, the project achieved effective control over schedule, effort, and quality objectives. These measurement activities not only supported successful delivery but also provided evidence of disciplined engineering practice. The next section evaluates how these practices align with recognised process maturity and quality assurance frameworks.

# **8. Process Maturity and Quality Assurance**

## **8.1 Importance of Process Maturity in Software Quality**

Process maturity refers to the degree to which an organisation’s software development processes are **defined, managed, measured, and continuously improved**. Mature processes contribute directly to improved software quality by reducing variability, enhancing predictability, and enabling systematic quality assurance activities [38]. In academic and industrial contexts alike, process maturity frameworks provide a structured lens through which development practices can be evaluated and justified.

In this project, process maturity was not pursued as a formal certification objective but as a **conceptual framework** to guide disciplined development. The practices adopted during the development of the Job Recruiter Assistant Application align closely with **Capability Maturity Model Integration (CMMI) Level 2**, which focuses on project management fundamentals such as requirements management, planning, monitoring, and measurement [39].

## **8.2 Alignment with CMMI Level 2 Practices**

### **8.2.1 Requirements Management**

Requirements management was achieved through the use of **user stories** maintained within a structured product backlog. Each user story captured a functional requirement from the perspective of system stakeholders and was refined during sprint planning sessions. Changes to requirements were controlled through backlog prioritisation rather than ad hoc modifications, ensuring traceability between requirements, implementation, and testing artefacts.

This approach aligns with CMMI Level 2 requirements management practices by ensuring that requirements were clearly defined, reviewed, and managed throughout the project lifecycle [39].

|  |  |  |
| --- | --- | --- |
| CMMI Level 2 Practice Area | Project Practice | Evidence |
| Requirements Management | User stories captured and refined in sprint backlog | Kanban board screenshots |
| Project Planning | Sprint planning with story point estimation | Sprint plans and milestones |
| Project Monitoring & Control | Sprint reviews and progress tracking | Kanban dashboard |
| Configuration Management | Version control with commit tracking | GitHub commit history |
| Measurement & Analysis | Use of PERT, COCOMO, EVM, velocity | Measurement section in report |

### **8.2.2 Project Planning**

Project planning activities were conducted at both the **release level** and **sprint level**. High-level planning defined sprint timelines and major deliverables, while sprint planning sessions decomposed user stories into implementable tasks. Story point estimation provided a relative measure of effort, supporting realistic sprint commitments.

The use of structured sprint plans, defined responsibilities, and time-boxed iterations demonstrates alignment with CMMI Level 2 project planning objectives, which emphasise the establishment and maintenance of workable plans [40].

### **8.2.3 Project Monitoring and Control**

Progress was monitored continuously using the **Kanban dashboard**, sprint reviews, and automated test execution results. Deviations from planned progress, such as estimation inaccuracies identified in early sprints, were addressed through retrospective discussions and adjustment of future estimates.

Earned Value Management metrics and sprint velocity data further supported objective monitoring of progress and performance. This combination of qualitative and quantitative monitoring reflects CMMI Level 2 practices related to project tracking and control [40].

### **8.2.4 Configuration Management**

Configuration management was supported through the use of a **version control system**, enabling controlled evolution of the codebase. Code changes were associated with specific user stories and tasks, providing traceability and accountability. The separation of production code, test code, and documentation artefacts further contributed to controlled configuration management.

Although lightweight in nature, these practices satisfy the core intent of CMMI Level 2 configuration management by ensuring that work products were identified, versioned, and protected against unintended changes [39].

### **8.2.5 Measurement and Analysis**

Measurement and analysis activities were integrated throughout the project rather than confined to a final evaluation stage. Metrics such as story points, sprint velocity, coverage percentages, and EVM indicators were used to assess progress, quality, and efficiency. These measurements supported evidence-based decision-making and informed retrospective improvements.

This systematic use of measurement aligns with CMMI Level 2 expectations for quantitative monitoring and analysis [41].

## **8.3 Quality Assurance Practices**

Quality assurance in the project extended beyond testing to encompass **process-level assurance**. Quality objectives were defined early, focusing on correctness, robustness, and maintainability. Verification and validation activities were embedded within each sprint, ensuring that quality considerations influenced development decisions continuously.

The separation of verification (checking that the system was built correctly) and validation (checking that the correct system was built) provided conceptual clarity and strengthened the overall quality framework [42].

## **8.4 Continuous Improvement and Lessons Learned**

Retrospective discussions at the end of each sprint provided a structured mechanism for process improvement. Early challenges, such as initial estimation inaccuracies and test data conflicts, were addressed through refined planning and improved test isolation techniques. These iterative refinements demonstrate a commitment to continuous improvement, a key characteristic of mature software processes [38].

The ability to adapt practices based on empirical feedback reflects both Agile principles and process maturity concepts, illustrating that flexibility and discipline are not mutually exclusive.

## **8.5 Section Summary**

In summary, the development of the Job Recruiter Assistant Application demonstrates a clear alignment with **CMMI Level 2 process maturity practices**, despite operating within a lightweight Agile framework. Through structured requirements management, disciplined planning, continuous monitoring, controlled configuration, and systematic measurement, the project achieved a mature and defensible quality assurance process. These practices underpin the reliability and correctness of the delivered system and provide a strong foundation for reflective evaluation. The following section examines team collaboration, challenges encountered, and lessons learned during the project.

# **9. Team Collaboration and Reflection**

## **9.1 Team Structure and Collaboration Approach**

The Job Recruiter Assistant Application was created by a team of four members collaborating within an Agile Scrum framework. Team collaboration was organized through well-defined roles, collective accountability for quality, and open communication. While individual ownership of user stories was implemented to promote accountability, the team embraced a collective ownership approach for the entire system, fostering peer review, discussion, and collaborative problem-solving.   
  
Collaboration was enhanced via consistent sprint planning sessions, updates to the Kanban board, and informal technical discussions. These practices facilitated alignment on priorities, enabled early risk identification, and ensured consistent progress throughout sprints. The implementation of a shared version control repository facilitates coordinated development and regulated integration of individual contributions, which is crucial in collaborative software engineering projects [43].

## **9.2 Allocation of Responsibilities**

Workload distribution was carefully planned to ensure equitable contribution from all team members. Each member was responsible for nine user stories, distributed evenly across the three sprints. Responsibilities were aligned with functional areas such as job management, candidate handling, application workflows, interview scheduling, testing, and measurement activities.

This structured allocation enabled clear traceability between individual contributions and system features, while still allowing flexibility for cross-support when challenges arose. The explicit mapping of user stories to sprint goals and implementation artefacts strengthened accountability and simplified progress monitoring.

A screen shot of a computer

AI-generated content may be incorrect.

## **9.3 Communication and Coordination**

Effective communication was critical to maintaining project momentum and quality. Sprint planning and review meetings provided structured opportunities to synchronise understanding of requirements, clarify technical decisions, and review completed work. Between formal meetings, coordination was maintained through frequent Kanban updates and repository activity.

This combination of formal and informal communication channels helped prevent misalignment and reduced the likelihood of duplicated effort or conflicting changes. The approach reflects recognised best practices in Agile team collaboration, where continuous communication supports both productivity and quality outcomes [44].

## **9.4 Challenges Encountered**

Multiple challenges arose throughout the project's duration. During the initial development phase, the team encountered inaccuracies in estimation, especially regarding the effort needed for the implementation of state-dependent logic and thorough testing. The challenges were mitigated by recalibrating story point estimates in later sprints.   
  
Another challenge was the management of test data conflicts and the assurance of isolation among test cases. The issue was addressed through the implementation of temporary repositories and the adoption of more rigorous testing setup procedures. The final sprint necessitated careful prioritization to balance functional completeness with quality assurance and documentation tasks due to time constraints.   
These challenges did not hinder delivery; rather, they enhanced planning discipline and testing practices.

## **9.5 Learning Outcomes and Professional Development**

The project provided valuable learning opportunities in both technical and professional domains. From a technical perspective, team members gained hands-on experience in designing testable systems, implementing defensive programming techniques, and applying multiple testing strategies. Exposure to software measurement models such as COCOMO and EVM enhanced understanding of project control beyond purely functional concerns.

From a professional standpoint, the project reinforced the importance of clear communication, realistic planning, and shared responsibility for quality. Working within an Agile framework highlighted the value of adaptability, reflection, and evidence-based decision-making, all of which are essential skills for professional software engineers [45].

## **9.6 Reflection on Team Effectiveness**

Overall, the team functioned effectively, demonstrating strong coordination, balanced contribution, and a shared commitment to quality. The ability to identify issues early, adapt processes, and maintain consistent progress across sprints reflects a mature approach to collaborative software development.

The experience also underscored the interdependence of technical and process-oriented practices. High-quality outcomes were achieved not solely through good coding, but through disciplined planning, systematic testing, and continuous reflection. These insights will inform future team-based projects and professional practice.

## **9.7 Section Summary**

In summary, team collaboration played a crucial role in the successful delivery of the Job Recruiter Assistant Application. Through structured workload allocation, effective communication, and reflective practice, the team was able to manage challenges and deliver a robust, well-tested system. The lessons learned from this collaborative experience reinforce the value of Agile principles and process discipline in achieving high-quality software outcomes. The final section concludes the report by summarising key achievements and outlining potential future enhancements.

# **10. Conclusion**

This project set out to design, implement, and evaluate a **Job Recruiter Assistant Application** with a strong emphasis on **software measurement and quality assurance**, in alignment with the objectives of the CO7095 module. Rather than focusing solely on feature delivery, the project prioritised disciplined engineering practices, systematic testing, and evidence-based project control. The resulting system demonstrates that high-quality software outcomes can be achieved through a balanced integration of sound implementation, structured testing, and meaningful measurement.

From a technical perspective, the application successfully supports the core stages of a recruitment lifecycle, including job management, candidate handling, application workflows, screening logic, and interview scheduling. The adoption of a modular, service-oriented architecture enabled clear separation of concerns, improved maintainability, and facilitated effective testing. Explicit validation rules, controlled state transitions, and defensive programming practices contributed to system robustness and reduced the likelihood of inconsistent or invalid behaviour.

Quality assurance was embedded throughout the development lifecycle rather than treated as a post-implementation activity. A combination of black-box, white-box, and error-based testing techniques was applied to validate both functional behaviour and internal logic. Although overall code coverage was moderate, it reflected a deliberate, risk-based testing strategy that prioritised critical business logic over low-risk components. This approach aligns with established guidance that emphasises test effectiveness and relevance over numerical coverage targets [46].

In addition to testing, the project incorporated multiple **software measurement and project control techniques**, including PERT analysis, COCOMO cost estimation, Earned Value Management, and Agile metrics such as burndown and velocity. These techniques provided complementary insights into schedule feasibility, effort estimation, progress tracking, and delivery stability. The integration of traditional measurement models with Agile practices demonstrated how quantitative control can coexist with iterative and flexible development approaches [47].

Process maturity considerations further strengthened the project’s quality framework. By aligning development practices with **CMMI Level 2 principles**, the team demonstrated structured requirements management, disciplined planning, continuous monitoring, controlled configuration, and systematic measurement. These practices contributed to predictable delivery outcomes and supported reflective improvement across sprints.

Equally important were the collaborative and professional skills developed during the project. Clear allocation of responsibilities, effective communication, and reflective sprint retrospectives enabled the team to manage challenges, adapt processes, and maintain consistent progress. The experience reinforced the interdependence of technical competence and process discipline in delivering reliable software systems.

In conclusion, the Job Recruiter Assistant Application provides a credible demonstration of how **software quality assurance and measurement techniques can be applied in a realistic Agile project context**. The project not only achieved its functional objectives but also delivered substantial learning outcomes in testing, measurement, process maturity, and team collaboration. Future work could extend the system through enhanced analytics, database-backed persistence, or user-facing interfaces. However, within its defined scope, the project successfully meets its academic and engineering objectives, providing strong evidence of disciplined and professional software development practice.

References

[1] Breaugh, J.A. (2017) *Talent Acquisition: A Guide to Understanding and Managing the Recruitment Process*. SHRM Foundation.

[2] Chapman, D.S. and Gödöllei, A.F. (2017) ‘E-recruiting: Using technology to attract job applicants’, *International Journal of Selection and Assessment*, 25(3), pp. 211–223.

[3] Sommerville, I. (2016) *Software Engineering*. 10th edn. Boston: Pearson.

[4] Schwaber, K. and Sutherland, J. (2020) *The Scrum Guide*. Scrum.org.

[5] Beck, K. et al. (2001) ‘Manifesto for Agile Software Development’, *Agile Alliance*.

[6] Pressman, R.S. and Maxim, B.R. (2020) *Software Engineering: A Practitioner’s Approach*. 9th edn. New York: McGraw-Hill.

[7] Jones, C. (2014) *Software Quality: Analysis and Guidelines for Success*. Boston: Cengage Learning.

[8] CMMI Institute (2018) *CMMI for Development, Version 2.0*. Pittsburgh: CMMI Institute.

[9] Breaugh, J.A. (2017) Talent Acquisition: A Guide to Understanding and Managing the Recruitment Process. SHRM Foundation.

[10] Sommerville, I. (2016) Software Engineering. 10th edn. Boston: Pearson.

[11] Pressman, R.S. and Maxim, B.R. (2020) Software Engineering: A Practitioner’s Approach. 9th edn. New York: McGraw-Hill.

[12] Fowler, M. (2002) Patterns of Enterprise Application Architecture. Boston: Addison-Wesley.

[13] Myers, G.J., Sandler, C. and Badgett, T. (2011) The Art of Software Testing. 3rd edn. Hoboken: Wiley.

[14] ISO/IEC 25010 (2011) Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE). Geneva: ISO.

[15] Schwaber, K. and Sutherland, J. (2020) The Scrum Guide. Scrum.org.

[16] Beck, K. et al. (2001) ‘Manifesto for Agile Software Development’, Agile Alliance.

[17] Cohn, M. (2005) Agile Estimating and Planning. Upper Saddle River, NJ: Prentice Hall.

[18] Pressman, R.S. and Maxim, B.R. (2020) Software Engineering: A Practitioner’s Approach. 9th edn. New York: McGraw-Hill.

[19] Sommerville, I. (2016) Software Engineering. 10th edn. Boston: Pearson.

[20] Sommerville, I. (2016) Software Engineering. 10th edn. Boston: Pearson.

[21] Pressman, R.S. and Maxim, B.R. (2020) Software Engineering: A Practitioner’s Approach. 9th edn. New York: McGraw-Hill.

[22] McConnell, S. (2004) Code Complete. 2nd edn. Redmond, WA: Microsoft Press.

[23] Myers, G.J., Sandler, C. and Badgett, T. (2011) The Art of Software Testing. 3rd edn. Hoboken: Wiley.

[24] Sommerville, I. (2016) Software Engineering. 10th edn. Boston: Pearson.

[25] Myers, G.J., Sandler, C. and Badgett, T. (2011) The Art of Software Testing. 3rd edn. Hoboken: Wiley.

[26] McConnell, S. (2004) Code Complete. 2nd edn. Redmond, WA: Microsoft Press.

[27] Pressman, R.S. and Maxim, B.R. (2020) Software Engineering: A Practitioner’s Approach. 9th edn. New York: McGraw-Hill.

[28] Anand, S. et al. (2013) ‘Automated concolic testing’, ACM Computing Surveys, 45(3), pp. 1–44.

[29] Sommerville, I. (2016) Software Engineering. 10th edn. Boston: Pearson.

[30] Pressman, R.S. and Maxim, B.R. (2020) Software Engineering: A Practitioner’s Approach. 9th edn. New York: McGraw-Hill.

[31] Inozemtseva, L. and Holmes, R. (2014) ‘Coverage is not strongly correlated with test suite effectiveness’, Proceedings of the 36th International Conference on Software Engineering, pp. 435–445.

[32] Fenton, N.E. and Pfleeger, S.L. (1998) Software Metrics: A Rigorous and Practical Approach. 2nd edn. Boston: PWS Publishing.

[33] Kerzner, H. (2017) Project Management: A Systems Approach to Planning, Scheduling, and Controlling. 12th edn. Hoboken: Wiley.

[34] Boehm, B.W. (1981) Software Engineering Economics. Englewood Cliffs, NJ: Prentice Hall.

[35] Boehm, B. et al. (2000) ‘Software cost estimation with COCOMO II’, Prentice Hall.

[36] Fleming, Q.W. and Koppelman, J.M. (2016) Earned Value Project Management. 4th edn. Newtown Square, PA: PMI.

[37] Schwaber, K. (2004) Agile Project Management with Scrum. Redmond, WA: Microsoft Press.

[38] Fenton, N.E. and Pfleeger, S.L. (1998) Software Metrics: A Rigorous and Practical Approach. 2nd edn. Boston: PWS Publishing.

[39] CMMI Institute (2018) CMMI for Development, Version 2.0. Pittsburgh: CMMI Institute.

[40] Kerzner, H. (2017) Project Management: A Systems Approach to Planning, Scheduling, and Controlling. 12th edn. Hoboken: Wiley.

[41] Florac, W.A. and Carleton, A.D. (1999) Measuring the Software Process: Statistical Process Control for Software Process Improvement. Reading, MA: Addison-Wesley.

[42] ISO/IEC 25010 (2011) Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE). Geneva: ISO.

[43] Sommerville, I. (2016) Software Engineering. 10th edn. Boston: Pearson.

[44] Schwaber, K. and Sutherland, J. (2020) The Scrum Guide. Scrum.org.

[45] Pressman, R.S. and Maxim, B.R. (2020) Software Engineering: A Practitioner’s Approach. 9th edn. New York: McGraw-Hill.

[46] Inozemtseva, L. and Holmes, R. (2014) ‘Coverage is not strongly correlated with test suite effectiveness’, Proceedings of the 36th International Conference on Software Engineering, pp. 435–445.

[47] Fenton, N.E. and Pfleeger, S.L. (1998) Software Metrics: A Rigorous and Practical Approach. 2nd edn. Boston: PWS Publishing.