Software Engineering Project Evaluation Rubric – Implementation phase

Total Marks covered by this document: 45 (Project Execution)
Project Duration: 4 weeks (2 sprints × 2 weeks each)
Evaluation Date: End of Sprint 2 + Final Demo Presentation

📋 EXECUTIVE SUMMARY - READ THIS FIRST

What's Being Evaluated (45 marks breakdown):

Component	Marks	What We're Looking For
Jira & Sprint Management	12	EPICs, User Stories, 2 Sprints, Burndown, Retrospectives, Traceability
Git & SCRUM Practices	10	Feature Branches, PRs with Reviews, Commit Quality, Team Collaboration
Code Quality	8	Unit + Integration + System Tests, Code Coverage ≥75%
CI/CD Pipeline	15	5 Stages (Build, Test, Coverage, Lint, Security), Deployment Artifact

When You'll Be Evaluated:

- ✅ **NOW (Sprint Planning Phase):** You've submitted SRS, Test Plan, some Architecture docs
- ⏰ **Week 2 (Sprint 1 End):** Checkpoint for feedback only (no marks)
- 🎯 **Week 4 (Sprint 2 End): FULL EVALUATION** during final demo (all 45 marks)

Critical Timeline:

Week 1-2: Sprint 1 Execution

├── Implement at least 4-5 user stories

├── Create feature branches + PRs

├── Write unit + integration tests

├── Set up CI/CD pipeline

└── Sprint 1 retrospective

Week 2 Checkpoint: TA feedback (no grading)

Week 3-4: Sprint 2 Execution

├── Implement remaining user stories

├── Complete system tests

├── Achieve 75% code coverage

├── Fix linting to ≥7.5 score

├── Create deployment artifact

└── Sprint 2 retrospective

Week 4 Final: Demo + Evaluation (45 marks)

⚠️ IMPORTANT: GRADING STARTS NOW

Everything you've done so far (before this document):

- ✅ Jira backlog creation → Will NOT be penalized for format issues
- ✅ Early documents (SRS, Test Plan, Architecture) → Already graded
- ✅ Any initial code/commits → Not held to these standards

Everything from NOW onwards (after this document is published):

- ⚠️ **WILL BE EVALUATED** according to this rubric
- ⚠️ All Git commits, PRs, branches must follow guidelines
- ⚠️ CI/CD pipeline must meet all 5 stage requirements
- ⚠️ Code quality, tests, coverage measured from this point

Translation: You have a fresh start for the next 4 weeks. Focus on following these guidelines going forward.

📌 Quick Start Checklist (What to Do This Week)

Week 1 (Sprint 1 Starts - This Week! Or already started):

Day 1-2: Sprint Planning

- Review your Jira backlog (already created)
- Select at least 4-5 user stories for Sprint 1
- Assign stories to team members
- Create GitHub feature branches for each story

Day 3-7: Development

- Each member works on their feature branch
- Write code + tests simultaneously
- Create PRs with detailed descriptions
- Review teammates' PRs (give meaningful feedback)
- Merge completed stories to main

Week 1 Goal: at least 2-3 user stories merged + basic CI/CD running

Week 2 (Sprint 1 Ends):

Day 8-12: Complete Sprint 1

- Finish remaining Sprint 1 stories
- Ensure all PRs reviewed and merged
- Get CI/CD pipeline to at least 3 stages working
- Write code coverage tests to reach 60%+

Day 13-14: Sprint 1 Closure

- Move all completed stories to "Done"
- Conduct Sprint 1 retrospective (document it!)
- Meet with TA for checkpoint feedback
- Plan Sprint 2 (select remaining stories)

Week 3-4: Sprint 2 (Same process, higher quality)

Complete remaining user stories
Add system tests
Improve coverage to 75%
Complete all 5 CI/CD stages
Prepare final demo

🚨 Common Mistakes to Avoid (Learn from Previous Teams)

Jira Mistakes:

❌ Creating all stories on same day and backdating them

❌ User stories too vague: "Build login" instead of "User can login with email/password"

❌ Forgetting sprint retrospectives (easy marks!)

❌ Not moving stories through workflow (To Do → In Progress → Done)

Git Mistakes:

❌ Committing everything to main (no feature branches)

❌ Commit messages like "update", "changes", "fixed" (be specific!)

❌ Creating PRs but not getting reviews (defeats the purpose)

❌ Branch names like "abhishek-final-code" (use feature/xyz)

Code Quality Mistakes:

❌ Writing tests after failing evaluation (write tests while coding!)

❌ Tests that always pass (test actual functionality)

❌ Only unit tests, forgetting integration/system tests

❌ 30% code coverage (need 75%+)

CI/CD Mistakes:

❌ Pipeline that never runs successfully

❌ Copying template but not understanding it

❌ Missing deployment artifact (easy marks!)

❌ No README explaining how pipeline works

Purpose of This Document

For Students:

This rubric defines **exactly what will be evaluated** at the end of Sprint 2 during your final demo. Everything you do from **now onwards** (after this document is published) must follow these guidelines. Previous work will not be penalized.

For Teaching Assistants / Instructors:

Use this for final evaluation after Sprint 2 completion and demo presentation. Sprint 1 checkpoint is for guidance/feedback only (no marks assigned).

Tips for Success (Updated for Current Stage)

Where You Are Now:

✅ Jira backlog created

✅ SRS, Test Plan, Architecture documents submitted

✅ Project setup phase complete

✅ **About to start Sprint 1 development**

What Changes Starting NOW:

⚠️ All code, commits, PRs from now will be evaluated

⚠️ Follow Git workflow guidelines strictly

⚠️ Write tests as you code (not after!)

⚠️ Document retrospectives for each sprint

Week 1 (Sprint 1 Execution - Starting Now):

Timeline mentioned is representative. No. of user stories mentioned is a min requirement – not max.

Monday-Tuesday: Sprint 1 Kickoff

Review your existing Jira backlog

Sprint Planning Meeting: Select 4-5 user stories for Sprint 1

- Choose stories that can be completed in 2 weeks
- Distribute work among team members (check everyone has ~equal work)

Each member creates their feature branch:

- Format: `feature/user-story-name` (e.g., `feature/user-login`)

Set up basic CI/CD pipeline (can start with just Build + Test stages)

Wednesday-Sunday: Development Sprint 1

Daily standup (5 mins): What did I do? What will I do? Any blockers?

Each member works on their assigned feature branch:

- Write code for the feature
- Write unit tests simultaneously (NOT after code is done!)
- Commit regularly (2-3 commits per day when working)
- Use descriptive commit messages

Create Pull Requests when feature is ready:

- Write detailed PR description (what, why, how)
- Mention Jira ticket ID (e.g., "Implements US-001")
- Request review from at least 1 teammate

Review teammates' PRs:

- Check code logic
- Verify tests are present
- Run code locally if possible
- Give meaningful comments (not just "LGTM")

Goal by Week 1 end: 2-3 user stories completed and merged

Week 1 CI/CD Focus:

- Get Build stage working (dependencies install)
- Get Test stage working (run pytest/jest/junit)
- Start Coverage stage (even if coverage is low initially)

Week 2 (Sprint 1 Completion):

Monday-Thursday: Sprint 1 Final Push

Complete remaining Sprint 1 stories (aim for 4-5 total)
Write integration tests:

- Test how components interact
- Test API endpoints (if applicable)
- Test database operations (if applicable)

Improve code coverage:

- Current target: ≥60% by Sprint 1 end
- Final target: ≥75% by Sprint 2 end

Fix any PR review comments
Ensure all completed stories moved to "Done" in Jira

Friday (Sprint 1 Retrospective - IMPORTANT!):

Conduct Sprint 1 Retrospective Meeting (30-45 mins)
Discuss and document:

- ✅ What went well? (e.g., "Good collaboration on PRs")
- ❌ What didn't go well? (e.g., "CI pipeline kept failing")
- 🔧 Action items for Sprint 2 (e.g., "Test locally before pushing")

Document retrospective in:

- Jira (as attachment or comment) OR
- Google Doc shared with team OR
- Submit as separate file

TA Checkpoint: Show your progress, get feedback (no marks yet!)

Weekend (Sprint 2 Planning):

Review Jira backlog
Select remaining 4-5 user stories for Sprint 2
Based on Sprint 1 learnings, adjust story points if needed
Assign Sprint 2 stories to team members

Week 2 CI/CD Focus:

- Add Lint stage (pylint, ESLint, etc.)
- Fix linting errors to reach 6.0+ score (aim for 7.5 by Sprint 2 end)
- Start Security scan stage (bandit, npm audit, etc.)

Week 3 (Sprint 2 Execution):

Monday-Sunday: Sprint 2 Development

Same workflow as Sprint 1 (feature branches → PRs → reviews → merge)

Key difference: Higher quality expected

- Better commit messages
- More detailed PR descriptions
- Faster PR review turnaround

Add system tests (end-to-end scenarios):

- Complete user workflows
- Full application testing

Improve code coverage from 60% → 75%+:

- Add tests for uncovered code
- Focus on critical business logic

Complete all 5 CI/CD stages:

- Build ✅
- Test ✅
- Coverage ✅ (with 75% threshold)
- Lint ✅ (score ≥7.5)
- Security ✅ (scan running, reports generated)

Week 3 CI/CD Focus:

- Get all 5 stages passing consistently
- Add deployment stage (creates .zip artifact)
- Verify artifact contains: code + reports + README

Week 4 (Sprint 2 Completion + Final Demo):

Monday-Wednesday: Final Polish

Complete remaining Sprint 2 stories
Verify all tests passing

Check code coverage report: ≥75%? ✅

Check lint score: ≥7.5? ✅

Review security scan: No critical vulnerabilities? ✅

Update README with CI/CD documentation:

- What each stage does
- How to run locally

- Quality thresholds

Verify deployment artifact:

- Download from GitHub Actions
- Extract and check contents
- Source code present? ✅
- All reports present? ✅

Thursday (Sprint 2 Retrospective):

Conduct Sprint 2 Retrospective
Focus on improvements from Sprint 1:

- Did we implement action items?
- What improved?
- What would we do differently in future?

Document retrospective

Friday (Final Demo Preparation):

Review Jira burndown charts (both sprints)
Prepare demo presentation:

- Show working application (5 mins)
- Walk through CI/CD pipeline (5 mins)
- Show code quality metrics (3 mins)
- Discuss sprint retrospectives (2 mins)

Practice demo with team (don't wing it!)
Ensure everyone can speak about their contributions

Final Demo Day: 🎯 EVALUATION (45 marks)

Demonstrate working features
Show CI/CD pipeline in action
Answer TA / Instructor questions confidently
Explain your SCRUM process

Golden Rules for Next 4 Weeks:

1. **Commit Often, Commit Smart**

   - Commit 2-3 times per day when working
   - Each commit = one logical change
   - Write clear messages: "Add email validation to login form"

2. **Test While You Code (Not After!)**

    - Write unit test for each function as you code it
    - Run tests locally before pushing
    - Don't wait until Sprint 2 to add tests

3. **Review PRs Within 24 Hours**

    - Don't let teammates wait 3 days for review
    - Give meaningful feedback, not just "looks good"
    - If you're blocked, communicate in Jira/Slack

4. **Update Jira Daily**

    - Move stories through workflow as you work
    - Don't update Jira once at sprint end (bad practice!)
    - Burndown chart should show gradual progress

5. **Run CI/CD Locally Before Pushing**

    - Test: `pytest tests/` or `npm test`
    - Coverage: `pytest --cov=src`
    - Lint: `pylint src/` or `eslint src/`
    - Don't rely on CI to catch basic errors

6. **Communicate Blockers Early**

    - Stuck for 2 hours? Ask for help
    - Don't suffer in silence until deadline
    - Use Jira comments, Slack, WhatsApp—whatever works

7. **Weekend Work ≠ Quality Work**

    - Don't do everything Sunday night
    - Spread work across 2 weeks
    - Your burndown chart will show if you cramming

8. **Document Retrospectives (Easy Marks!)**

    - 30 minutes of discussion = 2-3 marks
    - Don't skip this—it's free marks
    - Retrospectives show you understand SCRUM

Pro Tips from Previous Successful Teams:

✅ **Set up CI/CD in Week 1:** "We spent 3 hours Week 1 setting up pipeline. Saved us 20 hours in Weeks 3-4 because we caught errors early."

✅ **Write tests first (TDD approach):** "We wrote tests before code. Seemed slow at first, but we had 80% coverage by Sprint 1 end with no effort in Sprint 2."

✅ **Daily 5-minute standups:** "We did 5-min video call every evening. Kept everyone aligned, caught blockers early."

✅ **Use PR templates:** "We created a PR template in GitHub. Everyone knew what to write, reviews were faster and better."

✅ **Celebrate small wins:** "Every time we merged a PR, we celebrated in group chat. Kept morale high during tough weeks."

❌ **What NOT to do:** "We wrote all code in Sprint 1, tried to add tests in Sprint 2. Ran out of time, got 40% coverage. Lost 5 marks."

❌ **What NOT to do:** "We had 1 person do 80% of work because they're strongest coder. In evaluation, other members couldn't answer questions. Lost collaboration marks."

❌ **What NOT to do:** "We didn't document retrospectives. Lost 2 marks for literally 10 minutes of work. Dumb mistake."

Evaluation Timeline

Sprint 1 End (Week 4) - Checkpoint Only (No Marks)

Purpose: Ensure students are on track, provide feedback

TA Checks:

- ✅ Jira backlog created with EPICs and user stories
- ✅ Sprint 1 planned and in progress
- ✅ GitHub repository set up with some initial commits

- ✅ Basic CI/CD pipeline attempted (even if failing)
- ✅ At least 1-2 feature branches created

Feedback Given:

- Issues with user story structure
- Git workflow problems
- CI/CD configuration errors
- Recommendations for Sprint 2

Time Required: 10-15 minutes per team

Sprint 2 End + Final Demo (Week 4) - FULL EVALUATION (45 Marks)

Purpose: Comprehensive evaluation of entire project

What's Evaluated:

- Complete Jira backlog with 2 completed sprints
- All Git practices across both sprints
- Final code quality metrics
- Working CI/CD pipeline
- Live demo and presentation

Time Required: 45-50 minutes per team

Mark Distribution Overview

Component	Marks	Key Focus
1. Jira & Sprint Management	12	Backlog, Sprints, User Stories, Retrospectives
2. Git & SCRUM Practices	10	Branching, PRs, Commits, Team Collaboration
3. Code Quality	8	Testing, Coverage

Component	Marks	Key Focus
4. CI/CD Pipeline	15	Automation, Quality Gates, Deployment, Static Analysis
TOTAL	**45**	

---

1. Jira & Sprint Management (12 Marks)

Evaluated At: Sprint 2 End + Final Demo

1.1 Epic & User Story Structure (5 marks)

Full Marks (5/5) ✅

- **Minimum 2 EPICs** defined in Jira backlog
- Each EPIC has **3-4+ user stories** (total 8-10 user stories for project)
- User stories follow **INVEST principles**:
  - **I**ndependent (can be developed separately)
  - **N**egotiable (flexible, not overly detailed)
  - **V**aluable (delivers clear value)
  - **E**stimable (team can estimate effort)
  - **S**mall (completable within sprint)
  - **T**estable (has acceptance criteria)
- Dedicated user stories for **test case development** (unit, integration, system)
- Each user story has:
  - Clear acceptance criteria (what defines "done")
  - Story points assigned (effort estimation)
  - Priority level set (High/Medium/Low)

TA Verification Steps:

1. Open Jira Backlog view
2. Count EPICs (must be ≥2)
3. Randomly select 5 user stories and check:
   - Do they have acceptance criteria? ✅/❌
   - Are they small enough (1-3 day tasks)? ✅/❌
   - Are they testable? ✅/❌
4. Look for test-related stories (e.g., "Write unit tests for login module")

Example Good User Story:

Title: [US-001] User Authentication with JWT

Description:

As a registered user, I want to log in with email/password

so that I can access my dashboard.

Acceptance Criteria:

- Email validation (valid format)

- Password minimum 8 characters

- On success: redirect to dashboard + JWT token

- On failure: show "Invalid credentials"

- Session expires after 24 hours

Story Points: 5

Priority: High

Partial Marks (3-4/5) ⚠️

- Only 1 EPIC OR EPICs have <3 user stories each (6-7 total stories)
- User stories lack some INVEST criteria (missing acceptance criteria or story points)
- Missing dedicated test case user stories
- Some user stories too large (>3 days of work)

Minimal Marks (1-2/5) ⚠️

- Minimal Jira setup with poorly defined user stories
- No clear INVEST adherence
- User stories are tasks, not value-driven features (e.g., "Update database" instead of "User can reset password")

No Marks (0/5) ❌

- No Jira backlog OR incomplete setup
- User stories don't follow any structured format

1.2 Sprint Execution & Burndown (5 marks)

Full Marks (5/5) ✅

Two Completed Sprints:

- **Sprint 1** (Weeks 1-3): Initial features + setup
- **Sprint 2** (Weeks 4-6): Remaining features + polish

Each Sprint Has:

- Defined **sprint goal** (what will be delivered)
- Selected user stories from backlog (realistic commitment)
- Sprint duration clearly marked (Start date → End date)
- User stories moved through workflow states:
    - To Do → In Progress → In Review → Done

Sprint Retrospective:

- Conducted after **each sprint** (2 total)
- Documented (in Jira, Google Doc, or submitted separately)
- Includes:
    - What went well ✅
    - What didn't go well ❌
    - Action items for improvement 🔧

Burndown Chart:

- Shows realistic progress tracking for **both sprints**
- Work remaining decreases gradually over sprint duration
- No "hockey stick" pattern (all work done in last 2 days)
- Chart reflects actual team velocity

TA Verification Steps:

1. Jira → Sprint view → Verify 2 sprints marked "Completed"
2. Open each sprint:
    - Sprint goal defined? ✅/❌
    - Selected stories visible? ✅/❌
3. Check for retrospective documents (Jira attachments or submission folder)
4. Open burndown chart for both sprints:

- Sprint 1: Does it show gradual progress?
- Sprint 2: Does it show improvement from Sprint 1 learnings?

5. Spot-check 3-4 user stories → Verify workflow transitions (not just To Do → Done)

Partial Marks (3-4/5) ⚠️

- Both sprints completed but planning is weak (no clear sprint goals)
- Sprint retrospective missing for one sprint OR incomplete (just 1-2 sentences)
- Burndown chart shows poor planning (e.g., flat line then sudden drop)
- User stories not properly transitioned through workflow

Minimal Marks (1-2/5) ⚠️

- Only 1 sprint completed OR sprints exist but not executed properly
- No sprint retrospectives
- Burndown chart missing or shows unrealistic progress
- Evidence of backdating (all stories created same day, moved to Done immediately)

No Marks (0/5) ❌

- No sprints created OR sprints are just labels (not actually used)
- No evidence of sprint methodology being followed

1.3 Traceability & Requirements Validation (2 marks)

Full Marks (2/2) ✅

- **Requirements traceability matrix** maintained (can be in SRS or separate document)
- User stories link back to requirements/use cases
- Test cases link to user stories (verifiable in test plan or Jira)
- Clear mapping: Requirement → User Story → Implementation → Test Case

Example Traceability:

REQ-001: User Authentication

├── US-001: Implement JWT login

│ ├── Feature Branch: feature/user-auth

│ ├── PR #12: Add login endpoint

│ └── Tests: test_authentication.py (unit + integration)

└── US-002: Implement password reset

    ├── Feature Branch: feature/password-reset

    └── Tests: test_password_reset.py

TA Verification Steps:

1. Check SRS or separate traceability document
2. Randomly pick 2-3 requirements → Trace to user stories
3. Pick 2-3 user stories → Verify they mention Jira ticket IDs in PRs/commits
4. Check test files → Do test names reference user stories? (e.g., `test_US001_login_success`)

Partial Marks (1/2) ⚠️

- Traceability exists but incomplete (not all requirements mapped)
- Some user stories link to requirements, others don't
- Test cases exist but not clearly linked to user stories

No Marks (0/2) ❌

- No traceability matrix
- No connection between requirements and implementation
- Cannot verify what was implemented vs. what was required

---

2. Git & SCRUM Practices (10 Marks)

Evaluated At: Sprint 2 End (analyzing entire Git history)

2.1 Branching Strategy & Team Collaboration (5 marks)

Full Marks (5/5) ✅

Branch Protection:

- **Main branch protected** (verified via GitHub org-level settings)
- No direct commits to main (all changes via PRs)

Feature Branches:

- Proper naming convention: `feature/<feature-name>`
    - ✅ Good: `feature/user-auth`, `feature/payment-gateway`, `feature/dashboard-ui`
    - ❌ Bad: `abhishek-branch`, `final-code`, `temp`
- **One feature branch per user story**
- Branches created by assigned team member
- Branch names match Jira user story titles

Team Collaboration:

- **All team members** (3-5 members) actively contributed
- Visible in:
    - GitHub Insights → Contributors (all members show commits)
    - Jira (all members have assigned + completed stories)
    - PRs (different members as authors and reviewers)
- Work distribution is reasonable:
    - Not perfectly equal, but no single member did 80% of work
    - No team member with zero contributions
    - Adjusted for team size (3-member team vs 5-member team)

TA Verification Steps:

1. GitHub → Settings → Branches → Verify main protection enabled
2. Check commit history on main → Should only see "Merge pull request" entries
3. Branches view → Check all merged branch names
    - Count feature branches (should be 8-10+)
    - Verify naming follows convention
4. Cross-check 3-4 branch names with Jira user stories
5. GitHub → Insights → Contributors:
    - All team members visible with non-zero commits?
    - Contribution distribution reasonable?
6. Jira → Filter by Assignee → All members have completed work?

Collaboration Red Flags:

- Only 1 person did all work (visible in GitHub/Jira)
- Team members listed but no evidence of contribution
- Unequal distribution (one person has 90% commits)

Partial Marks (3-4/5) ⚠️

- Main branch protected BUT inconsistent branch naming
- Some direct commits to main (should be none)
- Branch names don't relate to features
- One team member has minimal contribution (but some work done)

Minimal Marks (1-2/5) ⚠️

- Poor branching strategy (only 2-3 feature branches for entire project)
- No clear naming convention
- Main branch not protected OR heavily violated
- Only 1-2 team members contributed

No Marks (0/5) ❌

- No branching strategy (everyone committing to main)
- Repository structure is chaotic
- Only 1 team member worked (others have zero contribution)

2.2 Pull Request Quality (3 marks)

Full Marks (3/3) ✅

All features merged via Pull Requests (no direct merges)

Each PR Includes:

- **Descriptive title** (e.g., "Implement user authentication with JWT")
- **Detailed description** with:
  - What was implemented
  - Which Jira ticket it addresses (e.g., "Implements US-001")
  - What files were changed
  - How to test the changes
- **At least 1 peer review** with meaningful comments (not just "LGTM")
- **Approval from teammate** before merge
- **All CI/CD checks passed** before merge (green checkmarks visible)
- PR conversation shows actual discussion

TA Verification Steps:

1. GitHub → Pull Requests → Closed
2. Randomly select 5-6 PRs:

- Title descriptive? (not "Update code" or "fixes")
- Description includes what/why/how?
- Links to Jira ticket? (mentions US-XXX)
- Has reviewers assigned?
- Review comments are meaningful? (not just "approved")
- CI checks passed before merge?

Example Good PR:

Title: Implement JWT-based user authentication

Description:

Implements US-001: User Authentication with JWT

Changes:

- Added login endpoint (POST /api/auth/login)

- Implemented JWT token generation

- Created authentication middleware

- Added password hashing with bcrypt

Files Changed:

- src/auth/login.js

- src/middleware/authMiddleware.js

- tests/auth/test_login.js

Testing:

1. POST to /api/auth/login with valid credentials

2. Verify JWT token returned

3. Access protected route with token

4. All unit tests pass (see CI)

Jira: US-001

Partial Marks (2/3) ⚠️

- PRs exist but descriptions are minimal
- Some PRs lack peer review OR reviews are not meaningful ("LGTM" only)
- Some PRs merged without CI/CD checks passing

Minimal Marks (1/3) ⚠️

- Very few PRs OR PRs are extremely poor quality
- No evidence of code review process
- PRs merged immediately without review

No Marks (0/3) ❌

- No PRs used (direct merges/commits to main)
- Cannot verify team collaboration through PRs

2.3 Commit Message Quality (2 marks)

Full Marks (2/2) ✅

Descriptive commit messages explaining what and why

Good Examples:

- ✅ "Add JWT token validation middleware for authentication"
- ✅ "Fix login bug where session expires immediately"
- ✅ "Implement password reset with email verification"
- ✅ "Add unit tests for authentication module"

Bad Examples:

- ❌ "Update code"
- ❌ "Fixed bug"
- ❌ "Changes made"
- ❌ "Final commit"
- ❌ "."

Commit Granularity:

- Multiple small commits showing logical progression
- Not one huge commit with 50+ files changed
- Not 200 tiny commits for minor changes

TA Verification Steps:

1. GitHub → Commits (or check any branch/PR commit history)
2. Randomly check 10-15 commit messages across both sprints
3. Categorize:
    - Good: Descriptive and specific
    - Acceptable: Brief but clear
    - Poor: Vague or meaningless
4. Check commit sizes (files changed per commit)

Expected Range:

- 3-member team: 40-80 commits
- 5-member team: 60-120 commits
- Project duration: 6-7 weeks

Partial Marks (1/2) ⚠️

- Commit messages present but often vague
- Mix of good and poor messages
- Some commits too large (should be broken down)

No Marks (0/2) ❌

- Commit messages consistently poor
- Impossible to understand project history
- One massive commit with everything

---

3. Code Quality (8 Marks)

Evaluated At: Sprint 2 End (using final CI/CD reports)

3.1 Test Case Development (5 marks)

Full Marks (5/5) ✅

All Three Test Types Implemented:

1. **Unit Tests** (Testing individual functions/methods)

    - Test each function in isolation
    - Mock external dependencies
    - Fast execution (<1 second per test)

2. **Integration Tests** (Testing component interactions)

    - Test API endpoints
    - Test database operations
    - Test module interactions

3. **System Tests** (End-to-end workflows)

    - Complete user scenarios
    - Test full application flow
    - May include UI testing (if applicable)

Test Organization:

- Tests properly organized in project structure
- Separate folders for test types (e.g., `tests/unit/`, `tests/integration/`, `tests/system/`)
- Test files follow naming convention (e.g., `test_*.py`, `*.test.js`)

Test Quality:

- Tests are meaningful (actually validate functionality)
- Not placeholder tests (always pass regardless of code)
- Tests have descriptive names (e.g., `test_login_with_invalid_password_returns_401`)
- Tests are automated and run via CI/CD

Example Good Structure:

tests/

├── unit/

│ ├── test_authentication.py

│ ├── test_user_model.py

│ └── test_validators.py

├── integration/

│ ├── test_api_endpoints.py

│ ├── test_database.py

│ └── test_user_flow.py

└── system/

├── test_complete_user_journey.py

└── test_end_to_end_workflows.py

TA Verification Steps:

1. Navigate to repository → Find test folder
2. Check folder structure → Are tests organized by type?
3. Count test files:
    - Unit tests? ✅/❌
    - Integration tests? ✅/❌
    - System tests? ✅/❌
4. Open 2-3 test files → Verify meaningful test cases
5. Run tests locally OR check CI/CD logs → Tests execute and pass
6. Check test reports in CI/CD artifacts

Minimum Expected:

- 15+ unit tests
- 5+ integration tests
- 2+ system tests (Scale down for smaller projects, up for larger ones)

Partial Marks (3-4/5) ⚠️

- Only 2 out of 3 test types (commonly missing: system tests)
- Tests exist but minimal coverage
- Some tests are placeholders or trivial
- Tests not well-organized

Minimal Marks (1-2/5) ⚠️

- Only 1 test type (usually just unit tests)
- Very few test cases (<10 tests total)
- Tests don't provide meaningful validation

No Marks (0/5) ❌

- No tests implemented
- Test files exist but empty/commented out
- Tests don't run/execute

3.2 Code Coverage (3 marks)

Full Marks (3/3) ✅

- **Code coverage ≥ 75-80%** (minimum threshold)
- Coverage report generated and available:
  - HTML report in CI/CD artifacts OR
  - Submitted documentation folder
- Coverage measured for **source code only** (not test files)
- Branch coverage (if applicable) also measured

Language-Specific Tools:

Language	Tool	Command Example
Python	`pytest-cov`	`pytest --cov=src --cov-report=html`
JavaScript	`Jest`	`jest --coverage`
TypeScript	`Jest` + `ts-jest`	`jest --coverage`
Java	`JaCoCo`	Maven/Gradle plugin

TA Verification Steps:

1. Check CI/CD pipeline → Look for coverage stage
2. Download coverage artifact (HTML report) OR check submitted docs
3. Open `index.html` in coverage report
4. Verify:

- Overall percentage ≥75%?
- Coverage is for source code (e.g., `src/` folder)
- Not measuring test code coverage

5. Spot-check critical modules → Should have >60% coverage individually

Acceptable Range:

- 75-80%: Full marks (3/3)
- 65-74%: Partial marks (2/3)
- 50-64%: Minimal marks (1/3)
- <50%: No marks (0/3)

Note: Use discretion for edge cases (e.g., 74% can get 2.5/3)

Partial Marks (2/3) ⚠️

- Coverage 65-74%
- Coverage report exists but not integrated into CI/CD
- Some critical modules have <50% coverage

Minimal Marks (1/3) ⚠️

- Coverage 50-64%
- Coverage tool configured but report incomplete

No Marks (0/3) ❌

- Coverage <50%
- No coverage report generated
- Coverage tool not configured

---

4. CI/CD Pipeline (15 Marks)

Evaluated At: Sprint 2 End + Live Demo

4.1 CI Pipeline with Static Analysis (8 marks)

Full Marks (8/8) ✅

All Required Stages Implemented and Working:

1. Build Stage (Install dependencies, compile)

- Dependencies install successfully
- Build completes without errors
- Environment setup correct

2. Test Stage (Run all test suites)

- Unit tests execute and pass
- Integration tests execute and pass
- System tests execute and pass
- Test results logged and visible

3. Coverage Stage (Code coverage measurement)

- Coverage report generated
- Meets 75-80% threshold
- HTML report saved as artifact
- Coverage metrics visible in logs

4. Lint Stage (Static code analysis)

- **Linting tool configured** for project language
- **Lint score ≥ 7.5/10** (or equivalent)
- Lint report saved as artifact
- Quality gates enforced (pipeline fails if below threshold)

5. Security Scan Stage (Vulnerability detection)

- Security scanner runs successfully
- No critical vulnerabilities OR documented exceptions
- Security report saved as artifact
- Scans both code and dependencies

Language-Specific Linting Tools:

Language	Linting Tool	Threshold	Command
Python	`pylint`	Score ≥ 7.5/10	`pylint src/`
JavaScript	`ESLint`	0 errors, <10 warnings	`eslint src/`
TypeScript	`ESLint` + TS parser	0 errors, <10 warnings	`eslint src/`

Language	Linting Tool	Threshold	Command
Java	`Checkstyle` or `PMD`	0 critical, <15 minor	`checkstyle` / `pmd`
HTML/CSS	`HTMLHint`, `Stylelint`	0 errors, <5 warnings	`htmlhint` / `stylelint`

Security Scanning Tools:

Language	Tool	What It Checks
Python	`bandit`, `safety`	Hardcoded secrets, SQL injection, dependency CVEs
JavaScript/TS	`npm audit`, `Snyk`	Dependency vulnerabilities
Java	`OWASP Dependency-Check`	Known CVEs in dependencies
General	`git-secrets`, `truffleHog`	Exposed secrets in commits

Pipeline Triggers:

- Runs on **every push** to any branch
- Runs on **every Pull Request**
- Prevents merge if checks fail

TA Verification Steps:

1. GitHub → Actions tab
2. Check recent workflow runs (should see 10+ runs)
3. Open latest successful run → Verify all 5 stages:
    - ✅ Build
    - ✅ Test
    - ✅ Coverage
    - ✅ Lint
    - ✅ Security
4. Click each stage → Check logs (verify it ran, not skipped)
5. Download artifacts:

- Coverage HTML report
- Lint report (text or JSON)
- Security report (JSON)

6. Verify lint score meets threshold:
   - **Python:** "Your code has been rated at X/10" → X ≥ 7.5
   - **JavaScript:** Error count = 0
   - **Java:** No critical violations
7. Check a Pull Request → Verify pipeline ran automatically

Example Pipeline Structure (GitHub Actions):

```
name: CI Pipeline

on: [push, pull_request]

jobs:

  build:

    runs-on: ubuntu-latest

    steps:

      - uses: actions/checkout@v3

      - name: Setup

        run: npm install



  test:

    needs: build

    runs-on: ubuntu-latest

    steps:

      - name: Run tests

        run: npm test
```

```
coverage:
  needs: test
  runs-on: ubuntu-latest
  steps:
    - name: Generate coverage
      run: npm run coverage
    - name: Check threshold (75%)
      run: npm run coverage -- --coverageThreshold=75
    - uses: actions/upload-artifact@v3
      with:
        name: coverage-report
        path: coverage/

lint:
  needs: coverage
  runs-on: ubuntu-latest
  steps:
    - name: Run ESLint
      run: npm run lint
    - name: Upload lint report
      uses: actions/upload-artifact@v3
```

```
      with:
        name: lint-report
        path: lint-report.txt

  security:
    needs: lint
    runs-on: ubuntu-latest
    steps:
      - name: Run npm audit
        run: npm audit --audit-level=high
      - name: Upload security report
        uses: actions/upload-artifact@v3
        with:
          name: security-report
          path: security-report.json
```

Partial Marks (6-7/8) ⚠️

- 4 out of 5 stages implemented (commonly missing: security scan)
- All stages present but some don't enforce failure (tests fail but pipeline passes)
- Lint score 6.5-7.4 (below threshold but close)
- Pipeline runs but not automatically on push/PR

Partial Marks (4-5/8) ⚠️

- Only 3 stages (typically: build, test, lint)
- Pipeline runs but has configuration errors
- Quality gates not enforced
- Lint score <6.5

Minimal Marks (1-3/8) ⚠️

- Basic pipeline with 1-2 stages (build and test only)
- Pipeline file exists but frequently fails
- Copy-pasted template with no customization

No Marks (0/8) ❌

- No CI pipeline configured
- Pipeline completely broken (never ran successfully)
- Pipeline disabled or always skipped

4.2 CD Pipeline - Deployment Artifact (5 marks)

Full Marks (5/5) ✅

Automated Deployment Stage:

- Runs **only after all CI stages pass** (build → test → coverage → lint → security)
- Creates **zipped deployment artifact** (.zip file)

Artifact Contains:

1. **Project source code** (compiled/built if applicable)
2. **All CI/CD reports:**
   - Coverage report (HTML folder)
   - Lint report (text/JSON format)
   - Security scan report (JSON format)
   - Test results summary
3. **Essential project files:**
   - README.md with setup instructions
   - Dependency manifest (requirements.txt, package.json, pom.xml)
   - Configuration files (if needed)
4. **(Optional but recommended):**
   - Architecture/Design documentation
   - API documentation (if applicable)

Artifact Naming:

- Includes version or date (e.g., `project-v1.0-2024-01-15.zip`)
- Uploaded and available for download from CI/CD run

TA Verification Steps:

1. GitHub Actions → Open latest successful workflow run
2. Scroll to "Artifacts" section (bottom of page)
3. Verify deployment artifact exists:
    - Name format: `deployment-package.zip` or similar
    - File size reasonable (not empty, not excessive)
4. Download artifact and extract
5. Verify contents checklist:
    Source code present (`src/` folder)
    `coverage-html/` folder with `index.html`
    `lint_report.txt` or `lint-report.json`
    `security_report.json`
    `README.md` with instructions
    Dependency manifest (requirements.txt, package.json, etc.)
2. Check deployment stage in pipeline:
    - Runs after all CI stages? ✅/❌
    - Fails if CI stages fail? ✅/❌

Example Deployment Stage (GitHub Actions - Python):

```
deploy:

  needs: [build, test, coverage, lint, security]

  runs-on: ubuntu-latest

  steps:

    - uses: actions/checkout@v3



    - name: Download all artifacts

      uses: actions/download-artifact@v3

      with:

        path: artifacts/
```

```
- name: Create deployment package
  run: |
    mkdir -p deployment/
    cp -r src/ deployment/
    cp -r artifacts/ deployment/reports/
    cp README.md requirements.txt deployment/
    cd deployment/
    zip -r ../deployment-package-$(date +%Y%m%d).zip .

- name: Upload deployment artifact
  uses: actions/upload-artifact@v3
  with:
    name: deployment-package
    path: deployment-package-*.zip
    retention-days: 30
```

Partial Marks (3-4/5) ⚠️

- Deployment artifact created but missing some reports:
  - Has source code + coverage, missing lint or security report
- Artifact exists but not properly zipped (just folder upload)
- Deployment stage doesn't wait for all CI stages to pass

Minimal Marks (1-2/5) ⚠️

- Basic artifact with only source code (no reports)
- Manual artifact creation (not automated via pipeline)
- Artifact upload unreliable or incomplete

No Marks (0/5) ❌

- No deployment stage in pipeline

- No artifact generated
- Deployment stage exists but never succeeded

4.3 Pipeline Documentation & Reliability (2 marks)

Full Marks (2/2) ✅

CI/CD Documentation in README:

- **Clear section** explaining pipeline (e.g., "## CI/CD Pipeline" or "## Development Workflow")
- Explains:
  - What each stage does (Build, Test, Coverage, Lint, Security, Deploy)
  - What tools are used (pytest, pylint, bandit, etc.)
  - Quality gates/thresholds (75% coverage, 7.5 lint score)
  - How to run pipeline locally (optional but helpful)
- **(Optional):** Pipeline status badge showing current build status

Pipeline Configuration Quality:

- Pipeline file (`.github/workflows/*.yml`) has comments
- Comments explain each stage and key steps
- Easy to understand for someone new to project

Pipeline Reliability:

- Pipeline has run **multiple times** (≥10 runs visible)
- Recent runs mostly successful (≥60% success rate)
- Failures are legitimate (actual test failures, not config errors)

TA Verification Steps:

1. Open project README.md
2. Look for CI/CD section:
   - Explains what pipeline does? ✅/❌
   - Lists tools and thresholds? ✅/❌
   - Provides local run instructions? ✅/❌ (optional)
3. Open `.github/workflows/*.yml`
4. Check for comments explaining stages

5. GitHub Actions → Check run history:
    - Count total runs (should be 10+)
    - Calculate success rate from last 15 runs
    - Check if failures are legitimate (test failures) or config issues

Example Good README Section:

CI/CD Pipeline

Our project uses GitHub Actions for continuous integration and deployment.

Pipeline Stages:

1. **Build**: Installs dependencies from requirements.txt
2. **Test**: Runs pytest for unit, integration, and system tests
3. **Coverage**: Ensures ≥75% code coverage using pytest-cov
4. **Lint**: Code quality check with pylint (minimum: 7.5/10)
5. **Security**: Vulnerability scan using bandit
6. **Deploy**: Creates deployment artifact with all reports

Quality Gates:

- Code coverage must be ≥75%
- Pylint score must be ≥7.5/10
- No critical security vulnerabilities
- All tests must pass

Running Locally:

```
# Install dependencies

pip install -r requirements.txt

# Run tests

pytest tests/ -v

# Check coverage

pytest --cov=src --cov-report=html

# Check code quality
```

pylint src/

Security scan

bandit -r src/

Partial Marks (1/2) ⚠️

- Minimal CI/CD documentation (just mentions it exists)
- Pipeline runs but low success rate (<50%) due to config issues
- Configuration file lacks comments or clarity

No Marks (0/2) ❌

- No CI/CD documentation in README
- Pipeline never ran successfully or <5 total runs
- Configuration file is incomprehensible

Summary: Quick Reference for TAs

Final Evaluation Checklist (45 Marks)

1. Jira & Sprint Management (12 marks)

5 marks: EPICs (≥2) + User Stories (8-10) with INVEST principles

5 marks: Two completed sprints + retrospectives + burndown chart

2 marks: Requirements traceability matrix

2. Git & SCRUM Practices (10 marks)

5 marks: Protected main + feature branches + team collaboration

3 marks: Quality PRs with reviews and CI checks

2 marks: Descriptive commit messages

3. Code Quality (8 marks)

5 marks: Unit + Integration + System tests implemented

3 marks: Code coverage ≥75%

4. CI/CD Pipeline (15 marks)

8 marks: All 5 stages (Build, Test, Coverage, Lint, Security)

5 marks: Deployment artifact with all reports
2 marks: Pipeline documentation + reliability

Total: 45 marks

Evaluation Workflow for TAs

Efficient Evaluation Process (40-45 minutes per team)

Phase 1: Pre-Demo Review (15 minutes)

Do this before the team's scheduled demo

Jira Check (5 min):

1. Open team's Jira workspace
2. Count EPICs and User Stories
3. Check Sprint 1 and Sprint 2 completion
4. Open burndown chart for both sprints
5. Look for retrospective documents

GitHub Quick Scan (10 min):

1. Actions tab → Check recent runs (10+ runs?)
2. Open latest successful run → All 5 stages present?
3. Download deployment artifact → Extract and verify contents
4. Branches view → Count feature branches, check naming
5. Pull Requests → Spot-check 3-4 PRs
6. Insights → Contributors → All team members visible?

Fill Pre-Demo Scorecard:

- Jira: Preliminary score (may adjust during demo)
- Git Practices: Preliminary score
- CI/CD: Check if pipeline works

Phase 2: Live Demo & Presentation (20 minutes)

Team presents to evaluation panel

Demo Components (15 min):

1. **Application Demo** (5 min)

    - Team shows working application
    - Demonstrates key features
    - Shows system tests running (optional)

2. **CI/CD Pipeline Walkthrough** (5 min)

    - Show GitHub Actions runs
    - Explain each stage
    - Show deployment artifact contents
    - Demo how to download and deploy

3. **Code Quality Metrics** (3 min)

    - Show coverage report (HTML)
    - Show lint score
    - Show security scan results

4. **Sprint Retrospective Discussion** (2 min)

    - What went well in sprints?
    - What challenges faced?
    - How did team improve from Sprint 1 to Sprint 2?

Q&A Session (5 min):

- Ask clarifying questions about implementation
- Verify understanding of SCRUM/Agile principles
- Check individual contributions

Phase 3: Final Scoring (5 minutes)

After demo, finalize scores

1. Adjust Jira scores based on demo discussion
2. Verify code quality metrics match submitted reports
3. Check test execution during demo (if shown)
4. Complete evaluation scorecard
5. Document any red flags or concerns

Total Time: 40 minutes

Sample Evaluation Scorecard

Team Name: _____
Team Members: _____
Date: _____
Evaluator: _____

---

1. Jira & Sprint Management (/12)

1.1 Epic & User Story Structure (/5)

- EPICs count: _____ (need ≥2)
- User Stories count: _____ (need 8-10)
- INVEST compliance: ☐ Excellent ☐ Good ☐ Acceptable ☐ Poor
- Test-related stories: ☐ Yes ☐ No
- **Score: _____/5**
- **Notes:** _____

1.2 Sprint Execution & Burndown (/5)

- Sprint 1 completed: ☐ Yes ☐ No
- Sprint 2 completed: ☐ Yes ☐ No
- Retrospectives documented: ☐ Both ☐ One ☐ None
- Burndown chart quality: ☐ Excellent ☐ Good ☐ Poor
- **Score: _____/5**
- **Notes:** _____

1.3 Traceability (/2)

- Traceability matrix: ☐ Complete ☐ Partial ☐ Missing
- User stories linked to requirements: ☐ Yes ☐ Partial ☐ No
- **Score: _____/2**
- **Notes:** _____

Subtotal: _____/12

---

2. Git & SCRUM Practices (/10)

2.1 Branching & Collaboration (/5)

- Main branch protected: ☐ Yes ☐ No
- Feature branch naming: ☐ Consistent ☐ Inconsistent
- Feature branches count: _____ (expect 8-10+)
- All members contributed: ☐ Yes ☐ Mostly ☐ No
- Work distribution: ☐ Balanced ☐ Uneven ☐ One person did everything
- **Score: _____/5**
- **Notes:** _____

2.2 Pull Request Quality (/3)

- PRs reviewed (sample 5-6): / met criteria
- PRs have descriptions: ☐ All ☐ Most ☐ Few
- PRs have reviews: ☐ All ☐ Most ☐ Few
- CI checks before merge: ☐ Always ☐ Usually ☐ Rarely
- **Score: _____/3**
- **Notes:** _____

2.3 Commit Messages (/2)

- Commit messages quality (sample 10-15): ☐ Good ☐ Acceptable ☐ Poor
- Commit count: _____ (expect 40-120 depending on team size)
- **Score: _____/2**
- **Notes:** _____

Subtotal: _____/10

---

3. Code Quality (/8)

3.1 Test Case Development (/5)

- Unit tests: ☐ Yes ☐ No | Count: _____
- Integration tests: ☐ Yes ☐ No | Count: _____
- System tests: ☐ Yes ☐ No | Count: _____
- Tests organized: ☐ Yes ☐ No
- Tests meaningful: ☐ Yes ☐ Partially ☐ No
- **Score: _____/5**
- **Notes:** _____

3.2 Code Coverage (/3)

- Coverage percentage: _____%
- Coverage report available: ☐ Yes ☐ No
- Measures source code (not tests): ☐ Yes ☐ No
- **Score: _____/3**
- **Notes:** _____

Subtotal: _____/8

---

4. CI/CD Pipeline (/15)

4.1 CI Pipeline with Static Analysis (/8)

- Build stage: ☐ Working ☐ Failing ☐ Missing
- Test stage: ☐ Working ☐ Failing ☐ Missing
- Coverage stage: ☐ Working ☐ Failing ☐ Missing
- Lint stage: ☐ Working ☐ Failing ☐ Missing
    - Lint score: _____ (need ≥7.5 for Python)
- Security stage: ☐ Working ☐ Failing ☐ Missing
- Pipeline triggers on push/PR: ☐ Yes ☐ No
- Quality gates enforced: ☐ Yes ☐ Partially ☐ No
- **Score: _____/8**
- **Notes:** _____

4.2 Deployment Artifact (/5)

- Deployment artifact exists: ☐ Yes ☐ No
- Artifact contents checklist:
    - ☐ Source code
    - ☐ Coverage report (HTML)
    - ☐ Lint report
    - ☐ Security report
    - ☐ README + dependencies
- Runs after CI stages pass: ☐ Yes ☐ No
- **Score: _____/5**
- **Notes:** _____

4.3 Pipeline Documentation (/2)

- README has CI/CD section: ☐ Yes ☐ Minimal ☐ No
- Pipeline config has comments: ☐ Yes ☐ No
- Pipeline runs count: _____ (expect 10+)
- Success rate: ____% (expect ≥60%)
- **Score: _____/2**
- **Notes: _____**

Subtotal: _____/15

FINAL SCORE: _____/45

Overall Assessment

Strengths:

Weaknesses:

Demo Performance: ☐ Excellent (All features working, confident presentation) ☐ Good (Most features working, clear explanations) ☐ Acceptable (Basic functionality shown) ☐ Poor (Many issues, unclear presentation)

Team Collaboration Evidence: ☐ Strong (All members contributed equally, good communication) ☐ Adequate (Most members contributed) ☐ Weak (Uneven distribution, limited collaboration)

Recommended Grade Band:

- ☐ 40-45: Excellent (A grade equivalent)
- ☐ 35-39: Very Good (B+ grade equivalent)
- ☐ 30-34: Good (B grade equivalent)

- ☐ 25-29: Satisfactory (C+ grade equivalent)
- ☐ 20-24: Acceptable (C grade equivalent)
- ☐ <20: Needs Improvement

Action Items (if score <25): ☐ Flag for instructor review ☐ Schedule follow-up meeting with team ☐ Recommend improvements for future projects

---

Common Pitfalls to Avoid (For Students)

Jira Mistakes:

❌ User stories too large ("Build entire application")

❌ No acceptance criteria

❌ Forgetting test case stories

❌ All work done in last day (bad burndown)

❌ Backdating sprints/stories

Git Mistakes:

❌ Committing directly to main

❌ Vague commit messages ("Fixed stuff", "Update")

❌ PRs without descriptions

❌ No code reviews

❌ Branch names like "abhishek-branch"

Code Quality Mistakes:

❌ Placeholder tests that always pass

❌ Only unit tests, no integration/system tests

❌ Ignoring code coverage

❌ Disabling linter to "fix" warnings

CI/CD Mistakes:

❌ Copy-paste pipeline without understanding

❌ Pipeline never runs successfully

❌ No deployment artifact

❌ Skipping security scans

❌ No documentation of how pipeline works

FAQ for Students

Q1: When will we be evaluated?

A: Main evaluation happens at **end of Sprint 2** during your final demo presentation. However, we do a checkpoint after Sprint 1 to provide feedback (no marks assigned at Sprint 1 checkpoint).

Q2: Our team has 3 members. Same expectations as 5-member teams?

A: No. Expectations scale with team size:

- **3-member team:** 8-10 user stories, 40-60 commits
- **4-member team:** 10-12 user stories, 60-80 commits
- **5-member team:** 12-15 user stories, 80-100 commits

Q3: Can we use GitLab instead of GitHub?

A: No. For this course, **GitHub is required** due to organization-level branch protection rules. CI/CD principles are the same across platforms.

Q4: What if our pipeline failed once due to GitHub outage?

A: Legitimate external failures don't affect your score. We look at **overall reliability** (≥60% success rate). One failure due to infrastructure issues is fine.

Q5: Do ALL files need 75% coverage?

A: No. **Overall project coverage** must be ≥75%. Individual files can vary. However, critical business logic should have >60% coverage.

Q6: Can we use Python for backend and JavaScript for frontend?

A: Yes! Multi-language projects are fine. You need CI/CD for each:

- Python: pytest, pylint, bandit
- JavaScript: Jest, ESLint, npm audit

Q7: Our lint score is 7.3. Will we get marks?

A: Partial marks (likely 6-7/8 for CI Pipeline section). Threshold is 7.5, but we use discretion for close scores. Try to improve to 7.5+ for full marks.

Q8: We forgot to document Sprint 1 retrospective. Lost all marks?

A: No. If Sprint 2 retrospective is documented and sprints are properly executed, you'll get partial marks (3-4/5 instead of 5/5).

Q9: Can one teammate focus only on testing?

A: Specialization is okay, but **all members must contribute code**. Someone can focus on testing but should also implement 1-2 features. We verify via GitHub contributors.

Q10: Our security scan found vulnerabilities. Will we fail?

A: Not automatically. Document them:

- **Low/Medium severity:** Justify why acceptable
- **High/Critical:** Update dependencies or explain mitigation
- **False positives:** Document why they're false

Key: Show you **reviewed** the report.

Q11: Do we need to deploy to AWS/Heroku?

A: No. "Deployment" means creating a **zipped artifact** with code + reports. Live cloud deployment is optional bonus but not required.

Q12: Team member stopped contributing. What do we do?

A: Document immediately:

1. Show attempted communication (Jira comments, messages)
2. Redistribute their work in Jira
3. Notify instructor/TA
4. We evaluate individual contributions separately

Q13: Can we resubmit if we score low?

A: Check with instructor. Generally:

- Missing critical deliverables: May get 1 resubmission chance with penalty
- Poor quality: Focus on learning for next project
- Incomplete work: No resubmission

Q14: How many commits is "enough"?

A: No fixed number. Guidelines:

- **Too few:** <20 commits for entire project = Poor practice
- **Good range:** 40-80 commits (3-member team), 80-120 (5-member team)
- **Too many:** 200+ tiny commits = Also poor practice Focus on **meaningful commits** representing logical work units.

Q15: What if we can't finish all user stories?

A: Better to complete 6-8 stories **well** (with tests, documentation) than 15 stories poorly. Quality over quantity. Incomplete stories can stay in backlog—document why in retrospective.

Tips for Success

Timeline is representative only.

Week 1-2: Project Setup

✅ Set up Jira workspace

✅ Create 2 EPICs with 8-10 user stories

✅ Set up GitHub repo with branch protection

✅ Create basic CI/CD pipeline (even if failing)

✅ Plan Sprint 1 (select 4-5 stories)

✅ Submit SRS document

Week 3-4: Sprint 1 Execution

✅ Each member takes feature branch

✅ Implement features + write tests

✅ Create PRs with detailed descriptions

✅ Review teammates' PRs

✅ Merge completed stories

✅ **Sprint 1 checkpoint with TA (feedback only)**

✅ Conduct Sprint 1 retrospective

✅ Submit Test Plan + Architecture docs

Week 5-6: Sprint 2 Execution

✅ Plan Sprint 2 (remaining stories)

✅ Continue development + testing

✅ Improve coverage to ≥75%

✅ Fix linting to reach ≥7.5

✅ Add security scanning

✅ Complete all user stories

✅ Conduct Sprint 2 retrospective

✅ Submit Design document

Week 7: Final Polishing + Demo

✅ Verify all 5 CI/CD stages working

✅ Generate deployment artifact

✅ Update README with CI/CD docs

✅ Review Jira burndown charts

✅ Prepare demo presentation

✅ **Final demo and evaluation**

✅ Submit all remaining deliverables

Golden Rules:

1. **Commit daily** when working
2. **Write clear messages** (what + why)
3. **Review PRs within 24 hours** (don't block team)
4. **Run tests before committing**
5. **Update Jira daily** (move stories through workflow)

6. **Communicate blockers early**
7. **Ask questions—don't wait until deadline**

Red Flags for TAs (Plagiarism Indicators)

⚠️ **Report to instructor immediately if you observe:**

GitHub Anomalies:

- All commits in single day (entire project on deadline day)
- Commits dated before project start date
- 100+ commits pushed simultaneously
- Repository forked from another team
- Commit history from different project/course

Quality Inconsistencies:

- Professional-level README but beginner code
- Complex CI/CD but team can't explain it
- Uses tools/techniques not taught in course
- Pipeline identical to another team (beyond template)

Jira Manipulation:

- All stories created same day then backdated
- Impossible burndown velocity
- Sprint dates don't match GitHub activity
- Stories marked "Done" before branches created

Team Collaboration Red Flags:

- All PRs created and approved by same person
- Zero review comments or discussion
- One member has 95%+ of commits
- GitHub shows 1 contributor, Jira shows 5 members

Tool Reference Guide

Python

- **Testing:** pytest, unittest

- **Coverage:** pytest-cov (`pytest --cov=src --cov-report=html`)
- **Linting:** pylint (`pylint src/`), flake8
- **Security:** bandit (`bandit -r src/`), safety
- **CI/CD:** GitHub Actions

JavaScript/Node.js

- **Testing:** Jest, Mocha + Chai
- **Coverage:** Jest (`jest --coverage`)
- **Linting:** ESLint (`eslint src/`)
- **Security:** npm audit, Snyk
- **CI/CD:** GitHub Actions

TypeScript

- **Testing:** Jest + ts-jest
- **Coverage:** Jest (`jest --coverage`)
- **Linting:** ESLint with @typescript-eslint
- **Security:** npm audit, Snyk
- **CI/CD:** GitHub Actions

Java

- **Testing:** JUnit 5, TestNG
- **Coverage:** JaCoCo (Maven/Gradle plugin)
- **Linting:** Checkstyle, PMD, SpotBugs
- **Security:** OWASP Dependency-Check
- **CI/CD:** GitHub Actions, Jenkins

HTML/CSS

- **Testing:** Selenium, Cypress (functional)
- **Linting:** HTMLHint, Stylelint, Prettier
- **Security:** Focus on XSS prevention in JS
- **CI/CD:** GitHub Actions

Closing Notes

This evaluation measures your ability to apply **industry-standard software development practices** in an academic setting. The goal is demonstrating understanding of:

✅ **SCRUM/Agile** through Jira sprints and retrospectives

✅ **Collaborative development** through Git workflows

✅ **Code quality** through comprehensive testing

✅ **Automation** through CI/CD pipelines

Remember: Process matters as much as the product. A simple calculator with excellent development practices scores higher than a complex app with poor practices.

Most evaluation happens at project end, allowing you to iterate and improve throughout both sprints. Use the Sprint 1 checkpoint as a learning opportunity—feedback given then will help you succeed in final evaluation.

Good luck! 🚀

Document Version: 2.0
Last Updated: January 2024
Total Marks: 45 (Project Execution) + 55 (Documentation) = **100 marks**
Course: Software Engineering Tools & Practices
Contact: [TA Email/Office Hours]