

Abyssinia Software

Technology

PLC

developer Training Manual

Prepared By Marshet.A

Table of Contents

1. Coding Standards & Best Practices	6
A. Code Formatting & Style Guide	6
B. Clean Code Principles (SOLID, DRY, KISS)	7
C. Error Handling & Logging	7
D. Security Best Practices	7
2. Project Structure & Code Organization	7
A. Folder & File Naming Conventions	7
B. Modular Code Structure	8
C. API Design & Documentation	8
3. Version Control & Git Workflow	8
A. Git Branching Strategy	8
B. Commit Message Guidelines	9
C. Code Reviews & Pull Requests	9
4. Development Workflow & Team Rules	9
A. Agile & Scrum Practices	9
B. Testing & Quality Assurance	9
Unit Testing	9
2 Integration Testing	10
3 End-to-End (E2E) Testing	10
4 Code Coverage & Best Practices	11
C. Continuous Integration & Deployment (CI/CD)	11
5. Documentation & Knowledge Sharing	11
A. Writing Documentation	11
B. Code Comments & Inline Documentation	12

6. Soft Skills & Team Culture	12
A. Communication & Collaboration	12
B. Ownership & Responsibility	12
C. Continuous Learning	12
Criteria to Accept Completed Development Tasks.....	12
1. Functionality & Requirements	13
2. Code Quality & Standards	13
3. Testing & Stability	13
4. Code Review & Peer Feedback.....	13
5. Documentation & Comments	13
6. Performance & Optimization	14
7. Security & Best Practices.....	14
8. Deployment Readiness.....	14
9. Business & User Acceptance Testing (UAT).....	14
Final Checklist Before Accepting a Task:	14
Common Problems Existing	15
1. Not Writing Clean & Readable Code	15
🚩 Problem:.....	15
✅ Solution:.....	15
2. Not Handling Errors Properly	15
🚩 Problem:.....	15
✅ Solution:.....	16
3. Hardcoding Values Instead of Using Constants	16
🚩 Problem:.....	16
✅ Solution:.....	16
4. Poor Git Practices	16
🚩 Problem:.....	16
✅ Solution:.....	16
5. Not Understanding Asynchronous Code	17
🚩 Problem:.....	17
✅ Solution:.....	17
6. Poor Database Querying Practices	17

📄 Problem:.....	17
✓ Solution:.....	17
7. Ignoring Security Best Practices	17
📄 Problem:.....	17
✓ Solution:.....	18
8. Not Writing Tests	18
📄 Problem:.....	18
✓ Solution:.....	18
9. Overcomplicating the Code (Not Following KISS & DRY Principles)	18
📄 Problem:.....	18
✓ Solution:.....	18
10. Not Documenting Code.....	19
📄 Problem:.....	19
✓ Solution:.....	19
1. Problems in Team Collaboration.....	19
📄 Common Mistakes:	19
✓ Solutions:	19
2. Problems Affecting the Product	19
📄 Common Mistakes:	19
✓ Solutions:	20
3. Problems in Communication	20
📄 Common Mistakes:	20
4. Problems in Presenting Their Work	20
📄 Common Mistakes:	20
✓ Solutions:	21
5. Problems in Project Delivery	21
📄 Common Mistakes:	21
✓ Solutions:	21
6. Problems in Using Project Management Tools	21
📄 Common Mistakes:	21
✓ Solutions:	21

7. Problems in Office Activities & Workplace.....	22
📋 Common Mistakes:	22
✓ Solutions:	22
8. Problems in Understanding Business Logic.....	22
📋 Common Mistakes:	22
✓ Solutions:	22
9. Problems in Workplace Growth & Learning.....	23
📋 Common Mistakes:	23
✓ Solutions:	23
Agile Software Development Training Guide	23
1. Understanding Agile.....	23
✓ Core Principles (from the Agile Manifesto)	23
🔥 Key Agile Values	23
2. Agile Frameworks (Scrum & Kanban).....	24
🏆 Scrum Basics.....	24
3. Agile Development Best Practices.....	24
📋 Common Agile Mistakes & How to Fix Them	25
1. Poor Backlog Management	25
2. Ineffective Daily Stand-ups	25
3. Resistance to Change	26
4. Not Delivering Working Software	26
5. No Proper Retrospectives	27
6. Team Overloading & Burnout	27
7. Lack of Customer Collaboration	27
8. Poor Agile Tool Usage	28
Effective Communication & Team Collaboration.....	28
3 Understanding Business Logic & User Requirements	29
4 Product Development Lifecycle & Delivery	29
5 Using Project Management Tools Effectively	30
6 Presentation & Documentation Skills.....	30
7 Office Activities & Workplace Etiquette.....	31
Developer Do's (What They Should Do).....	31

1 Writing High-Quality Code	31
2 Effective Team Collaboration	32
3 Product & Business Understanding	32
4 Efficient Work Habits	32
1 Software Development Standards & Best Practices	35
2 Agile & Scrum Principles	35
3 Effective Use of Project Management Tools	35
4 Git & Version Control Best Practices	35
5 Business Logic & Product Thinking	36
6 Writing & Maintaining Technical Documentation	36
7 Testing & Quality Assurance	36
8 Soft Skills & Professional Behavior	36
9 CI/CD & Deployment Best Practices	37
Acceptable Reasons for Delay	37
✗ Unacceptable Reasons for Delay	38
Handling Delays Professionally	39
Backend Development Tools	39
6 Frontend Development Tools	40
7 Testing & Quality Assurance	40
8 DevOps & CI/CD	40
9 Productivity & Team Collaboration	41
◆ Bonus: AI & Code Assistance	41

1. Coding Standards & Best Practices

A. Code Formatting & Style Guide

- Follow a consistent style guide (e.g., Google JavaScript Style Guide, PEP 8 for Python).
- Use tools like **Prettier** (for JavaScript), **ESLint**, or **Black** (for Python) to enforce formatting.
- Keep **function names, variables, and class names meaningful**.

- Stick to **camelCase** (JavaScript, Java) or **snake_case** (Python) consistently.

B. Clean Code Principles (SOLID, DRY, KISS)

- **S**: Single Responsibility Principle (each function/class should have one purpose).
- **O**: Open/Closed Principle (extendable but not modifiable).
- **L**: Liskov Substitution Principle (subclasses should replace parent classes without breaking the app).
- **I**: Interface Segregation Principle (avoid large interfaces, split into smaller ones).
- **D**: Dependency Inversion Principle (use interfaces instead of concrete implementations).
- **DRY**: Don't Repeat Yourself – avoid duplicating code.
- **KISS**: Keep It Simple, Stupid – avoid over-engineering.

C. Error Handling & Logging

- Always handle errors gracefully with try-catch blocks.
- Use proper logging mechanisms (e.g., Winston for Node.js).
- Never expose stack traces in production.

D. Security Best Practices

- Never store passwords in plain text (use **bcrypt** or **Argon2**).
- Sanitize user inputs to prevent **SQL Injection/XSS attacks**.
- Use environment variables for secrets (.env files, never hardcode credentials).

2. Project Structure & Code Organization

A. Folder & File Naming Conventions

- Follow a clear folder structure based on the type of project:

```

|— src
|  |— controllers
|  |— models

```

```
| |— routes
| |— middleware
| |— services
| |— utils
| |— config
| |— tests
|— public (for frontend assets)
|— .gitignore
|— package.json
|— README.md
|— .env
```

- Use **index.js** or **app.js** as the main entry point.

B. Modular Code Structure

- Break down code into **small reusable modules** instead of large files.
- Use **MVC (Model-View-Controller)** for backend projects.
- Keep business logic in **services**, separate from controllers.

C. API Design & Documentation

- Follow **RESTful API** design principles.
- Use **Postman** for API documentation.
- Use **proper HTTP status codes** (200 OK, 400 Bad Request, 404 Not Found, 500 Internal Server Error).
- Save test data on post man

3. Version Control & Git Workflow

A. Git Branching Strategy

- Follow **Git Flow**:

main → develop → feature branches → bugfix branches

work on ur branch and integrate merge at main

- Use meaningful branch names:
 - user-authentication
 - bugfix/fix-login-error

B. Commit Message Guidelines

- Use structured commit messages:

feat: Added user authentication

fix: Resolved login bug

refactor: Improved database queries

- Keep commits **small and frequent** instead of large, messy commits.

C. Code Reviews & Pull Requests

- Every code change must go through **pull requests (PRs)**.
- Always add a **PR description** (what was changed and why).
- Follow a **peer review process** before merging.

4. Development Workflow & Team Rules

A. Agile & Scrum Practices

- Use **daily stand-up meetings** to discuss progress and blockers.
- Work in **sprints** (1-2 weeks) with clear deliverables.
- Use tools like **cickup** for task tracking.

B. Testing & Quality Assurance

Unit Testing

⚡ What is it?

- Testing **individual functions or components** in isolation.
- Ensures each **unit of code** works as expected.

⚡ Tools:

- Jest (JavaScript, React, Node.js)
- PyTest (Python)
- JUnit (Java)
- PHPUnit (PHP)

✓ Why is it important?

- Detects **small issues early** before they become big problems.
- Helps with **code refactoring** without breaking functionality.

2 Integration Testing

⚡ What is it?

- Tests how **different modules or services** work together.
- Ensures that **APIs, databases, and external services** interact properly.

⚡ Tools:

- Jest + Supertest (Node.js APIs)
- PyTest + Requests (Python APIs)

⚡ Example (Node.js API - Jest & Supertest):

✓ Why is it important?

- Catches **issues in API calls, database interactions, and third-party integrations.**

3 End-to-End (E2E) Testing

◆ What is it?

- Tests the **entire application flow** as a real user would.
- Ensures **UI, backend, and database** work correctly together.

◆ Tools:

- **Cypress** (Web UI Testing)
- **Selenium** (Browser Automation)

◆ Example (Cypress - UI Testing):

✓ Why is it important?

- Detects **UI and functionality issues before deployment**.
- Ensures **real user interactions** work correctly.

4 Code Coverage & Best Practices

- ✓ **Ensure at least 90% test coverage** before deployment.
- ✓ **Use CI/CD pipelines** to automate testing (GitHub Actions).
- ✓ **Fix failing tests immediately** before adding new features.

C. Continuous Integration & Deployment (CI/CD)

- Automate deployments using **GitHub Actions, Jenkins, or GitLab CI/CD**.
- Use **Docker** for containerization.
- Deploy to **staging** before production.

5. Documentation & Knowledge Sharing

A. Writing Documentation

- Maintain a **README.md** for every project with:
 - How to set up and run the project.
 - API endpoints and expected responses.
 - Contribution guidelines.

B. Code Comments & Inline Documentation

- Comment **why the code is written**, not just what it does.
- Use **JSDoc/PyDoc** for function documentation.

6. Soft Skills & Team Culture

A. Communication & Collaboration

- Be responsive on **telegram click up**.
- Use **clear and concise messages** when discussing code issues.
- Give and receive **constructive feedback** during code reviews.

B. Ownership & Responsibility

- Take responsibility for assigned tasks.
- Always test code before submitting.
- Never push untested or broken code to production.

C. Continuous Learning

- stay updated with **new frameworks and best practices**.
- Conduct **weekly knowledge-sharing sessions** within the team.

Criteria to Accept Completed Development Tasks

Before marking a software development task as "Completed" or "Accepted," ensure it meets the following criteria:

1 Functionality & Requirements

- ✓ The feature or bug fix works as expected according to the task description.
- ✓ The implementation meets all functional and non-functional requirements.
- ✓ The output aligns with the business logic and user expectations.

● **If any requirement is missing:** Send it back for revision.

2 Code Quality & Standards

- ✓ Code follows best practices (SOLID, DRY, KISS principles).
- ✓ Proper naming conventions, indentation, and formatting are maintained.
- ✓ No hardcoded values or unnecessary complexity.
- ✓ Code is modular and reusable.

● **If code is messy or violates standards:** Request refactoring and Warning.

3 Testing & Stability

- ✓ Unit tests cover at least 90% of the code.
- ✓ End-to-end (E2E) testing is performed where applicable.
- ✓ No major bugs or unexpected crashes.
- ✓ Edge cases are handled properly.

● **If tests are missing or failing:** Task is not accepted.

4 Code Review & Peer Feedback

- ✓ Code has been reviewed and approved by at least one senior developer.
- ✓ PR (Pull Request) comments are addressed.
- ✓ No unresolved merge conflicts.

● **If not reviewed or conflicts exist:** Send for review before accepting.

5 Documentation & Comments

- ✓ Important functions and logic are well-documented.
- ✓ API documentation is updated if applicable.
- ✓ README files or setup instructions are included if needed.

● **If documentation is unclear:** Request improvements.

6 Performance & Optimization

- ✓ Code does not introduce unnecessary slowdowns.
- ✓ Database queries are optimized (avoid N+1 problems).
- ✓ Memory usage and API response times are efficient.

● **If performance issues exist:** Request optimization.

7 Security & Best Practices

- ✓ No sensitive information (API keys, passwords) is exposed.
- ✓ Input validation and security checks are implemented.
- ✓ Code is protected against common vulnerabilities (SQL injection, XSS, etc.).

● **If security issues exist:** Task is rejected until fixed.

8 Deployment Readiness

- ✓ The feature is integrated into the main branch without breaking existing functionality.
- ✓ It works in different environments (staging, production).
- ✓ The CI/CD pipeline runs successfully.

● **If deployment fails:** Task is not accepted.

9 Business & User Acceptance Testing (UAT)

- ✓ Stakeholders (PM, QA, or end-users) confirm that the feature meets business needs.
- ✓ The UI/UX design aligns with expectations.
- ✓ No unexpected behavior in real-world scenarios.

● **If users report issues:** Task is not marked as completed.

10 deliverability the task to be completed

- ✓ Clickup status completed
- ✓ Git repo updated
- ✓ Deployed and working platform

Final Checklist Before Accepting a Task:

- Does the feature work as described?
- Is the code clean, maintainable, and reviewed?
- Are tests written, and do they pass?
- Is the documentation complete?
- Is security handled properly?
- Is the code optimized and performant?
- Is the feature ready for deployment?
- Have stakeholders validated it?

If all are ✔ **YES**, the task is **ACCEPTED!** 🎉

Would you like me to format this into a structured training manual section?

Common Problems Existing

1. Not Writing Clean & Readable Code

🚫 Problem:

- Messy, unstructured code with poor indentation.
- Poor variable and function naming (a, b, foo, bar).
- Long functions that try to do everything.

✔ Solution:

- Follow a consistent **coding style guide** (Prettier, ESLint, Black, PEP 8).
- Use meaningful variable names (userEmail instead of u or data).
- Keep functions **short and focused** (one function = one responsibility).

2. Not Handling Errors Properly

🚫 Problem:

- Ignoring error handling (try/catch blocks missing).
- Displaying vague error messages like "Something went wrong."

- Failing to log errors for debugging.

✓ Solution:

- Always use try/catch and return **descriptive error messages**.
- Log errors properly (console.error(), Winston, Log4j).
- Avoid exposing technical details in error messages (Internal Server Error instead of Cannot read property 'name' of undefined).

3. Hardcoding Values Instead of Using Constants

🔍 Problem:

- Using **magic numbers** and hardcoded values in the code.

✓ Solution:

- Use **constants** instead of hardcoding values.

4. Poor Git Practices

🔍 Problem:

- Committing directly to main without PR reviews.
- Messy commit messages (fix bug, changed code).
- Forgetting to pull the latest code before pushing.

✓ Solution:

- Follow **Git Flow** (create feature branches, use pull requests).
- Write **clear commit messages**:
- Always **pull before pushing** (git pull origin main).

5. Not Understanding Asynchronous Code

Problem:

- Calling `await` inside a function but not making it `async`.
- Using `.then()` and `await` together incorrectly.
- Not handling Promise rejections.

Solution:

- Always mark functions with `async` if using `await`.
- Use **`try/catch`** to handle `async` errors.

6. Poor Database Querying Practices

Problem:

- Fetching too much data (e.g., `SELECT * FROM users` instead of selecting needed columns).
- Not using indexes on frequently queried fields.
- Allowing SQL injection by concatenating user input into queries.

Solution:

- Select only the required columns:
- Use **parameterized queries** to prevent SQL injection:
- Index frequently searched fields (`CREATE INDEX idx_users_email ON users(email);`).

7. Ignoring Security Best Practices

Problem:

- Storing passwords in plain text.
- Hardcoding API keys and credentials.

- Allowing Cross-Site Scripting (XSS) and SQL Injection vulnerabilities.

✓ **Solution:**

- **Hash passwords** with bcrypt:
- Always **sanitize user inputs**:

8. Not Writing Tests

🔍 **Problem:**

- Deploying without testing
- Testing APi has no saved test data.
- Not writing unit or integration tests.
- Manually testing everything.

✓ **Solution:**

- Use **Jest/Mocha** for unit tests.
- Write tests before pushing code.

9. Overcomplicating the Code (Not Following KISS & DRY Principles)

🔍 **Problem:**

- Writing overly complex logic when a simple solution exists.
- Duplicating code instead of reusing functions.

✓ **Solution:**

- Follow **KISS (Keep It Simple, Stupid)**:
- Follow **DRY (Don't Repeat Yourself)**:

10. Not Documenting Code

Problem:

- Writing code without comments.
- No README file explaining the project setup.

✓ Solution:

- Use **JSDoc/PyDoc** for documentation:
- Always **write a README.md** file with setup instructions.

1. Problems in Team Collaboration

Common Mistakes:

- Working in isolation without updating the team.
- Not asking for help when stuck.
- Ignoring team coding conventions.
- Not presenting as team saying we did this instead of I did
- Not reviewing or giving feedback on pull requests.

✓ Solutions:

- Use **daily stand-up meetings** to update progress.
- Follow the team's **coding guidelines & Git branching strategy**.
- Encourage **code reviews & pair programming** to share knowledge.
- Always **communicate blockers** early instead of struggling alone.

2. Problems Affecting the Product

Common Mistakes:

- Writing **code that only works on their local machine** (hardcoded file paths, OS dependencies).
- Not thinking about **scalability & performance**.
- Ignoring user experience (UX) and accessibility.
- Not responsive design
- Don't considering previous feedbacks
- Don't learning from experience
- Poor API design leading to difficult integrations.

✓ Solutions:

- Use **environment variables** instead of hardcoded values.
- Follow **best practices for performance** (lazy loading, caching).
- **Test on multiple devices & browsers** before deployment.
- Write **clear API documentation** for smooth integration.

3. Problems in Communication

🔊 Common Mistakes:

- Using unclear or overly technical explanations in discussions.
- Avoiding communication out of fear of looking incompetent.
- Not keeping stakeholders informed about progress and blockers.
- Provide **regular updates** using project management tools (telegram click up github).
- Encourage **open communication** without fear of judgment.

4. Problems in Presenting Their Work

🔊 Common Mistakes:

- Using too much technical jargon without considering the audience.
- Showing code instead of explaining the solution and its impact.
- Failing to prepare before a demo.

✓ Solutions:

- Tailor presentations based on the audience (developers vs managers).
- Use **diagrams & real-world analogies** to explain concepts.
- **Rehearse** before important presentations and demos.
- Highlight **value & benefits** instead of just code details.

5. Problems in Project Delivery

🚫 Common Mistakes:

- Underestimating time required to complete tasks.
- Failing to test code thoroughly before delivery.
- Rushing last-minute fixes, leading to more bugs.
- Not deliver on time
- Trying to deliver without knowing what u did

✓ Solutions:

- Break tasks into **smaller, manageable parts**.
- Use **test-driven development (TDD)** to reduce bugs.
- Follow **agile methodologies** to adapt and iterate.

6. Problems in Using Project Management Tools

🚫 Common Mistakes:

- **Not updating task status on tools on clicikup**
- Ignoring comments and feedback in project discussions.
- Not updating github
- Not logging work hours properly in time-tracking tools.

✓ Solutions:

- **Make project updates a daily habit.**
- **Use notifications wisely** to stay informed on changes.
- Regularly check **assigned tasks & deadlines.**
- Check email and clickup notifications

7. Problems in Office Activities & Workplace

Common Mistakes:

- Ignoring meeting schedules and coming unprepared.
- Not respecting deadlines and team expectations.
- Using informal language or unprofessional behavior.
- Not implementing all guideline of the company

Solutions:

- **Be punctual for meetings & deadlines.**
- Maintain a **professional yet friendly attitude.**
- Understand and **respect company policies & culture.**

8. Problems in Understanding Business Logic

Common Mistakes:

- Focusing only on coding without understanding why they are building the feature.
- Not considering how users will interact with the product.
- Misinterpreting business requirements due to lack of clarification.
- Copy past AI code without knowing and considering its Impact on users

Solutions:

- **Ask questions** before starting a task to clarify business needs.
- Understand how the software **solves a real-world problem.**
- Work closely with **product managers & designers** to align with business goals.

9. Problems in Workplace Growth & Learning

Common Mistakes:

- Sticking to only what they know and not learning new technologies.
- **Not taking feedback seriously.**
- Not keeping up with industry trends.

Solutions:

- Set aside time for **learning & improving skills**.
- Seek **constructive feedback** and work on improvement areas.
- Follow **tech blogs, attend webinars, and contribute to open-source**.

Agile Software Development Training Guide

1. Understanding Agile

Core Principles (from the Agile Manifesto)

- **Individuals & interactions** over processes & tools.
- **Working software** over comprehensive documentation.
- **Customer collaboration** over contract negotiation.
- **Responding to change** over following a plan.

Key Agile Values

- **Iterative & incremental development** (deliver small pieces quickly).
- **Continuous feedback** (adapt based on user needs).
- **Collaboration & communication** (between teams & customers).
- **Embracing change** (requirements evolve).

2. Agile Frameworks (Scrum & Kanban)

🏆 Scrum Basics

✓ Roles:

- **Product Owner:** Defines priorities & works with customers.
- **Scrum Master:** Facilitates the process & removes roadblocks.
- **Development Team:** Builds and delivers the software.

✓ Process:

1. **Sprint Planning** → Define work for the sprint (1 weeks).
2. **Daily Stand-ups** → Quick progress updates.
3. **Sprint Execution** → Team works on the sprint goals.
4. **Sprint Review** → Demonstrate completed work.
5. **Sprint Retrospective** → Analyze & improve the process.

3. Agile Development Best Practices

✓ User Stories & Acceptance Criteria

- Write **clear, concise user stories:**
- Define **acceptance criteria** before coding starts.

✓ Frequent Releases & Iterations

- Break down features into **small, deliverable increments**.
- Deploy updates **frequently & iteratively**.

✓ Continuous Integration & Testing

- Use **CI/CD pipelines** to automate testing & deployment.
- Follow **test-driven development (TDD)** where possible.

✓ Agile Metrics to Track Progress

- **Velocity** → How much work is completed in a sprint.
- **Burndown Chart** → Shows remaining work vs time.
- **Lead Time & Cycle Time** → Measure efficiency.

📌 Common Agile Mistakes & How to Fix Them

1. Poor Backlog Management

✗ Mistakes:

- Too many unprioritized tasks.
- **Lack of clear user stories.understanding**
- Large, unmanageable tasks.

✓ Solutions:

- Keep a **prioritized & groomed backlog**.
- Break large features into **smaller, testable increments**.
- Define **clear acceptance criteria** before development.

2. Ineffective Daily Stand-ups

✗ Mistakes:

- Meetings that take too long.
- Team members not engaging.
- Focusing on problem-solving instead of updates.

✓ Solutions:

- Keep stand-ups **short (max 10 minutes)**.
- Follow the **3 key questions**:
 1. What did you do yesterday?

2. What will you do today? Or next
 3. Any blockers?
- Handle detailed discussions **after the stand-up**.

3. Resistance to Change

✗ Mistakes:

- Sticking to the old way of doing things.
- Ignoring feedback from retrospectives.
- Avoiding collaboration with non-technical teams.

✓ Solutions:

- Foster a **culture of adaptability**.
- Act on **feedback from retrospectives**.
- Educate the team on the **benefits of Agile**.

4. Not Delivering Working Software

✗ Mistakes:

- Focusing too much on documentation.
- Delaying releases for “perfection”.
- Building features without testing.

✓ Solutions:

- Deliver **small, working features frequently**.
- Follow **test-driven development (TDD)**.
- Use **continuous integration & automated testing**.

5. No Proper Retrospectives

✗ Mistakes:

- Ignoring retrospective meetings.
- No follow-up on improvement points.
- Blaming individuals instead of fixing processes.

✓ Solutions:

- Conduct retrospectives **at the end of every sprint**.
- Focus on **what went well, what didn't, and improvements**.
- Implement at least **one improvement per sprint**.

6. Team Overloading & Burnout

✗ Mistakes:

- Taking on too many tasks at once.
- Ignoring work-life balance.
- Unrealistic sprint goals.

✓ Solutions:

- Use **work-in-progress (WIP) limits** to control task load.
- Adjust **velocity based on past sprint performance**.
- Ensure **realistic sprint planning** with achievable goals.

7. Lack of Customer Collaboration

✗ Mistakes:

- Developing in isolation without user feedback.
- Ignoring changing requirements.
- Assuming customer needs instead of asking.

✓ **Solutions:**

- Get **frequent feedback** from stakeholders & customers.
- Use **MVP (Minimum Viable Product) approach**.
- Adapt to **changing requirements** based on real-world use.

8. Poor Agile Tool Usage

✗ **Mistakes:**

- Not updating task boards (clickup, etc.).
- Ignoring sprint reports & Agile metrics.
- Using Agile tools just for the sake of it.

✓ **Solutions:**

- Keep clickup **updated daily**.
- Use Agile reports (**velocity, burndown charts**) to improve.
- Choose a tool that **fits the team's needs**.

Effective Communication & Team Collaboration

✦ **Agile Team Roles & Responsibilities**

- Define **Scrum roles (Product Owner, Scrum Master, Developers, Testers)** clearly.
- Ensure **cross-functional collaboration** between frontend, backend, and testing teams.

✦ **Proper Use of Communication Channels**

- Use telegram **quick discussions**.
- Keep **important project updates** in project management tools (ClickUp).
- Always **document important decisions** in clickup github postman.

✦ **Handling Team Conflicts**

- Encourage **constructive feedback**.
- Focus on **solutions, not blame**.
- Involve Scrum Master or Manager for **escalation if needed**.

3 Understanding Business Logic & User Requirements

✦ Why Business Logic Matters?

- Developers should **understand the problem before coding**.
- Avoid coding **features without knowing how they fit into the business**.
- Always clarify **requirements with stakeholders** before development.

✦ User-Centric Development

- Write **User Stories** based on real-world **user needs**.
- Conduct **user research & feedback sessions**.
- Avoid **building features that customers don't need**.

4 Product Development Lifecycle & Delivery

✦ Agile Software Development Lifecycle (SDLC)

1. **Requirement Gathering** – Understand project goals.
2. **Planning & Prioritization** – Sprint backlog & story estimation.
3. **Development** – Follow Agile coding practices.
4. **Testing & Quality Assurance** – Automated & manual testing.
5. **Deployment** – CI/CD pipeline, rolling releases.
6. **Monitoring & Feedback** – Gather post-deployment feedback.

✦ CI/CD Pipeline & Deployment Standards

- Use **Continuous Integration (CI)** for automated testing.
- Implement **Continuous Deployment (CD)** for rapid releases.

- Define **rollback strategies** for failed deployments.

✦ Release Management & Versioning

- Follow **Semantic Versioning** (e.g., v1.2.3).
- Maintain **release notes & changelogs**.
- Use **staging & production environments** properly.

5 Using Project Management Tools Effectively

✦ ClickUp Guidelines

- **Keep tasks updated daily.**
- Use proper **labels, priorities, and due dates**.
- Add **detailed descriptions & links** to tasks.
- Follow **Sprint Board structure**:
 - ✦ **Backlog** → Ideas & upcoming tasks.
 - ⚡ **In Progress** → Actively worked tasks.
 - ✓ **Review / Testing** → Pending QA approval.
 - 🏁 **Done** → Completed & deployed.

✦ Sprint Planning & Task Estimation

- Use **Story Points or T-Shirt sizing** (S, M, L, XL).
- Developers should **not overcommit** to too many tasks per sprint.
- Estimate based on **effort & complexity, not just time**.

6 Presentation & Documentation Skills

✦ Technical Documentation

- Document **API endpoints** (Postman).
- Maintain **README files** with setup instructions.

- Keep **Confluence pages updated** with architecture & workflows.

✦ **Effective Demo Presentations**

- Prepare **well-structured sprint demos** for stakeholders.
- Show **live working software, not just code**.
- Explain **problems solved & user benefits** during demos.

7 Office Activities & Workplace Etiquette

✦ **Professional Work Ethics**

- Be **on time** for meetings & daily stand-ups.
- Respect **colleagues' focus time** (avoid unnecessary interruptions).
- Provide **timely status updates** on tasks.

✦ **Handling Feedback & Continuous Learning**

- Accept **constructive criticism professionally**.
- Continuously **learn & adapt to new technologies**.
- Share knowledge through **team workshops & knowledge-sharing sessions**.

Developer Do's (What They Should Do)

1 Writing High-Quality Code

- ✓ Write clean, readable, and maintainable code
- ✓ Follow SOLID principles & DRY (Don't Repeat Yourself) coding
- ✓ Use meaningful variable & function names
- ✓ **Break big tasks into smaller, manageable pieces**
- ✓ Write unit tests to ensure code reliability
- ✓ Follow **version control (Git) best practices**

2 Effective Team Collaboration

- ✓ **Communicate clearly**—ask questions when stuck
- ✓ Participate actively in **stand-up meetings**
- ✓ **Help teammates** when they need assistance
- ✓ keep **clcikup tasks updated**
- ✓ Always **review code before merging**

3 Product & Business Understanding

- ✓ **Understand the project's business logic**—know why you are building a feature
- ✓ Prioritize **user needs & experience**
- ✓ Read and follow **technical documentation**
- ✓ **Validate assumptions** before implementing a feature

4 Efficient Work Habits

- ✓ Focus on **one task at a time**—avoid multitasking
- ✓ **Plan the day** before starting work
- ✓ Take **short breaks** to stay productive
- ✓ **Write proper documentation** for new features

5 Professionalism & Growth

- ✓ Accept **feedback positively**
- ✓ Always **learn and improve skills**
- ✓ **Meet deadlines** and communicate delays early
- ✓ Follow **company policies & work ethics**

✗ Developer Don'ts (What They Should Avoid)

1 Bad Coding Practices

- ⊘ Writing **spaghetti code** (unstructured & unreadable)
- ⊘ Hardcoding values instead of using **configurations**

⊗ **Copy-pasting** code without understanding it from AI and not reviewing and make it human friendly

⊗ **Ignoring code reviews** or merging unreviewed code

2 Poor Teamwork & Communication

⊗ Working **in isolation** without updating the team

⊗ **Ignoring reviews** and feedback from teammates

⊗ Blaming others instead of **solving problems together**

⊗ Making **last-minute changes** without notifying the team

3 Neglecting Business Requirements

⊗ Coding features **without understanding user needs**

⊗ Ignoring **project specifications & documentation**

⊗ **Skipping testing** and pushing untested code

4 Inefficient Work Habits

⊗ Not **tracking progress** on clilup

⊗ **Not delivery task on time**

⊗ **Wasting time on unnecessary optimizations**

⊗ Rushing to complete a task **without proper testing**

5 Unprofessional Behavior

⊗ Ignoring **company guidelines & work ethics**

⊗ Arguing instead of **constructively discussing**

⊗ **Missing meetings** without informing the team

□ **Developer Mindset (How They Should Think Daily)**

1 Be Solution-Oriented

💡 **Focus on solving problems**, not just writing code

💡 If you face a challenge, **first try to solve it before asking for help**

💡 Always **think about scalability & future maintenance**

Continuous Improvement & Learning

 Learn **new technologies & best practices**

 Read **documentation before asking for help**

 Take **constructive feedback positively**

Work as a Team Player

☐ Help others **when they are stuck**

☐ Ask **for help when needed**—don't struggle alone

☐ Respect others' opinions & **collaborate effectively**


Stay Organized & Productive

✓ Plan your tasks **before coding**


✓ Take **notes during meetings**


✓ Keep your workspace **clean & distraction-free**


Daily Activities & Work Routine for Developers


 Morning Routine (Before Work Starts)

☐ **Check emails & messages** for important updates

 **Review today's Clickup tasks**

 **Plan your day & set clear goals**

 **Attend daily stand-up meeting**

 Work Sessions (Main Working Hours)


 **Write code for assigned tasks**

 **Document any important changes**

 **Push changes & commit code regularly**

 **Review PRs & give feedback to teammates**

 **Test the application to ensure quality**

 Short Breaks (To Stay Productive)

⏸️ Take a 5-10 min break every 2 hours

💧 Stay hydrated & move around

📅 End of Day Routine

✓ Update clickup with task progress

✓ Push final commits & open PRs if needed

✓ Write a summary of what was completed

✓ Plan for the next day's work

1 Software Development Standards & Best Practices

- **Code Style Guides & Formatting** (e.g., Prettier, ESLint, PSR-12 for PHP, PEP 8 for Python)
- **Folder Structure & Project Organization**
- **Error Handling & Debugging Techniques**
- **Logging & Monitoring Practices**
- **Security Best Practices** (Avoiding SQL Injection, XSS, Authentication flaws, etc.)

2 Agile & Scrum Principles

- **Understanding Agile Frameworks** (Scrum)
- **Roles in Agile Teams** (Scrum Master, Product Owner, Developers, etc.)
- **Sprint Planning & Story Estimation**
- **How to Conduct Stand-Up Meetings Efficiently**
- **Handling Sprint Retrospectives & Feedback**

3 Effective Use of Project Management Tools

- **How to Track Tasks & Update Status**
- **Writing Clear Task Descriptions & Acceptance Criteria**
- **Handling Bug Reports & Issue Tracking**
- **Properly Assigning & Delegating Tasks**

4 Git & Version Control Best Practices

- **Branching Strategies (Git Flow, Feature Branching, etc.)**
- **Writing Meaningful Commit Messages**
- **How to Handle Merge Conflicts**
- **Pull Request Etiquette & Code Reviews**
- **Using CI/CD Pipelines for Deployment**

5 Business Logic & Product Thinking

- **Understanding the Business Purpose Behind Features**
- **Building Features Based on User Needs, Not Just Code**
- **Reading & Analyzing Product Requirements**
- **How to Communicate With Non-Technical Stakeholders**

6 Writing & Maintaining Technical Documentation

- **How to Write Good API Documentation** (Postman)
- **Commenting Code Effectively**
- **Creating ReadMe Files & Setup Guides**
- **Keeping Documentation Up to Date**

7 Testing & Quality Assurance

- **Unit Testing, Integration Testing, End-to-End Testing**
- **Using Testing Frameworks (Jest, Mocha, PHPUnit, PyTest, etc.)**
- **How to Perform Code Reviews Efficiently**
- **Manual vs. Automated Testing**
- **Bug Reporting & Reproduction Steps**

8 Soft Skills & Professional Behavior

- **How to Give & Receive Feedback Properly**
- **Handling Conflicts in a Professional Manner**
- **Presentation Skills & Technical Demos**

- **Workplace Etiquette & Team Communication**
- **Time Management & Productivity Tips**

9 CI/CD & Deployment Best Practices

- **How to Automate Deployments (Docker, Kubernetes, GitHub Actions with server)**
- **Handling Production Deployments & Rollbacks**
- **Monitoring Performance & Handling Downtime**

Acceptable Reasons for Delay

1. Unclear or Changing Requirements

- The product owner or client modified the requirements after development started.
- The task needs further clarification before implementation.

2. Technical Challenges & Dependencies

- The task depends on another feature, API, or external service that is not ready.
- Unforeseen technical issues (e.g., integration challenges, third-party API failures).

3. Unresolved Bugs or System Instability

- A major bug or system issue prevents further development.
- Fixing a critical bug takes priority over new development.

4. Infrastructure or Tooling Issues

- Development environment problems (server down, database issues, CI/CD failure).
- The required tools, frameworks, or libraries are unavailable or outdated.

5. Blocked by Another Team or Person

- Awaiting review, approval, or inputs from a manager, QA, or another team.
- Dependency on a task assigned to another developer that is incomplete.

6. Sudden Personal or Health Issues

- Emergency health conditions (personal illness, family emergency).
- Unexpected leave due to valid personal reasons (must be communicated).

7. Significant Scope Increase

- The task was originally estimated incorrectly, and additional work is required.
- The task complexity is higher than initially assessed.

8. Security or Compliance Review Delays

- The feature requires legal or security approvals that are taking longer than expected.

9. Unexpected Performance Issues

- The feature affects system performance, requiring optimizations before delivery.

10. Unforeseen External Factors

- Power outage, internet failure, or other unavoidable technical disruptions.
- Natural disasters, political instability, or major disruptions affecting work.

✗ Unacceptable Reasons for Delay

1. Poor Time Management

- Developer did not properly prioritize tasks.
- Work was delayed due to unnecessary distractions (social media, excessive breaks).

2. Lack of Communication

- The developer faced an issue but did not report it early.
- No updates were given to the team or manager about the delay.

3. Failure to Ask for Help

- The developer struggled with a problem for too long without seeking assistance.
- Lack of collaboration with senior developers or team members.

4. Ignoring Deadlines & Commitments

- Developer missed the deadline due to personal negligence.
- Task was deprioritized for no valid reason.

5. Unnecessary Perfectionism

- Over-optimizing code without significant improvements.
- Delaying delivery due to minor, non-essential refinements.

6. Skipping Planned Work Hours

- Regularly missing work without valid justification.
- Starting work late and not making up for lost time.

7. Failure to Use Project Management Tools

- Not updating task status in Jira, Trello, or other tracking systems.
- Ignoring comments or requests for progress updates.

8. Blaming Others Without Justification

- Making excuses instead of proactively solving problems.
- Claiming dependency issues without verifying the actual blockers.

9. Frequent Task Switching

- Jumping between multiple tasks without completing any.
- Working on tasks outside of assigned priorities.

10. Ignoring Code Reviews & Feedback

- Task was delayed due to repeated rejections in code review.
- Developer did not address review comments on time.

Handling Delays Professionally

- ✓ If a delay is **acceptable**, inform the team early and propose solutions.
- ✓ If a delay is **unacceptable**, identify the cause and take corrective actions to improve productivity.
- ✓ Regularly update task status in project management tools and seek help when needed.

Backend Development Tools

- ◆ **Node.js** – For JavaScript backend development
- ◆ **Express.js** – Lightweight Node.js framework
- ◆ **MongoDB / MySQL** / – Popular databases
- ◆ **Docker** – Containerization tool for microservices
- ◆ **Redis** – Caching tool for fast responses

✂ VS Code Extensions:

- **Docker** → Manage containers inside VS Code
- **MongoDB for VS Code** → Connect to MongoDB from VS Code

6 Frontend Development Tools

- ◆ **React / next** – Popular frontend frameworks
- ◆ **Tailwind CSS / Material-UI** – Styling frameworks

✂ VS Code Extensions:

- **ES7+ React/Redux snippets** → Quick React snippets
- **Tailwind CSS IntelliSense** → Autocomplete Tailwind classes
- **Live Server** → Instantly refresh browser on save

7 Testing & Quality Assurance

- ◆ **Jest** – JavaScript unit testing
- ◆ ◆ **ESLint & Prettier** – Code formatting and linting
- ◆ **SonarQube** – Code quality analysis

✂ VS Code Extensions:

- **ESLint** → Automatically lint your code
- **Prettier** → Auto-format code for consistency

8 DevOps & CI/CD

- ◆ **Docker** – Container management
- ◆ **Kubernetes** – Orchestrating containers
- ◆ **Jenkins / GitHub Actions** – Automate build & deployment
- ◆ **Terraform** – Infrastructure as code

✂ VS Code Extensions:

- **YAML** → For Kubernetes & CI/CD pipelines
- **GitHub Actions** → Monitor GitHub workflows

9 Productivity & Team Collaboration

◆ **Microsoft Teams or telegram** – Communication

◆ **ClickUp** – Task management

◆ **Figma** – UI/UX design collaboration

✂ VS Code Extensions:

- **Todo Tree** → Manage to-do lists in code
- **Markdown All in One** → Write better documentation

◆ Bonus: AI & Code Assistance

◆ **GitHub Copilot** → AI-powered code suggestions

◆ **Tabnine** → AI code completion

Chatgpt, claude, deepseek , perplexity

✂ VS Code Extensions:

- **GitHub Copilot** → AI-assisted coding
- **Code Spell Checker** → Avoid typos in variable names