

End-to-End Development Standards

Title	Standards Document
Document Version	1.0
Author	@Elvin Taghizade
Date	29.07.2025
Status	ACTIVE
Purpose	End-to-end Development Standards for Digital Transformation

System Design Standards

1. Architecture Standards

a. Microservices & API Design

- Use **RESTful APIs** for external communication and for internal service-to-service communication.
- Follow **domain-driven design (DDD)** principles.
- Use **OpenAPI (Swagger)** for API documentation.
- Ensure APIs are **backward compatible** to avoid breaking changes.

2. Authentication & Authorization

- Use **OAuth 2.0 / OpenID Connect** for authentication (Keycloak).
- Implement **RBAC (Role-Based Access Control)** and **ABAC (Attribute-Based Access Control)**.
- Use **JWT** or **opaque tokens** with introspection for secure API calls.

3. Service Communication

- Prefer asynchronous communication (RabbitMQ, Kafka, or EventBridge) over synchronous calls where possible.
- Use circuit breaker (Resilience4j, Hystrix) for fault tolerance.

c. Implement retry logic for transient failures.

4. Coding Standards

a. General Coding Guidelines

- i. Follow **SOLID principles** and **Clean Code** practices.
- ii. Use **meaningful variable and function names**.
- iii. Follow **consistent indentation, spacing, and line breaks** (use **Checkstyle**).
- iv. Avoid **code duplication**, use reusable components and functions.

b. Backend Development

- i. Use **Java/Spring Boot** for backend development in core.
- ii. Implement **DTOs (Data Transfer Objects)** for structured input/output.
- iii. Use **repositories & services** for business logic separation.
- iv. Log requests/responses using **centralized logging** (ELK stack).
- v. Adhere to language-specific style guides (e.g., PEP 8 for Python, Google Style Guide for Java, PHP_CodeSniffer, PHPStan for PHP).
- vi. Avoid magic numbers; use constants or enumerations instead.

c. Frontend Development

- Use **Vue.js (Nuxt.js)** with component-based architecture.
- Implement **state management** with Redux, Vuex, or Zustand.
- Enforce **lazy loading** and **code splitting** to improve performance.
- Use **TailwindCSS or Material UI** for consistent UI design.

d. Database Standards

- Use **PostgreSQL** for relational databases, **MongoDB** for NoSQL, **Redis** for caching.
- Follow **proper indexing** strategies.

- Use **Liquibase** for schema migrations.
- Apply **soft deletes** instead of hard deletes where necessary.

5. Security Standards

a. **API Security**

- i. Validate input **using schemas (Zod, Joi, Yup)** to prevent injection attacks.
- ii. Use **rate limiting (API Gateway, Redis)** to prevent abuse.
- iii. Implement **CORS policies** and restrict allowed origins.

b. **Data Security**

- i. Store sensitive data **encrypted (AES-256, bcrypt for passwords)**.
- ii. Use **vaults (k8s secrets, GitLab variables, HashiCorp Vault)** to manage secrets.
- iii. Ensure **data masking & anonymization** for logs and non-authorized users.

c. **Secure DevOps (DevSecOps)**

- i. Implement **SAST (Static Analysis Security Testing)** in CI/CD (SonarQube).
- ii. Regularly scan for **vulnerabilities (OWASP Dependency Check, Snyk)**.

6. Testing Standards

a. **Unit Testing**

- i. Follow **TDD (Test-Driven Development)** where applicable.
- ii. Cover **at least 80% of business logic** with unit tests (JUnit, Jest).

b. **Integration Testing**

- i. Use **Postman, RestAssured** for API integration tests.
- ii. Mock dependencies with **MockServer or WireMock**.

c. **End-to-End Testing**

- i. Automate UI testing using **Cypress, Selenium**.
- ii. Automate API testing using **Newman (Postman CLI)**.

d. **Performance & Load Testing**

- i. Use **JMeter or k6** for performance tests.
- ii. Ensure APIs handle **high concurrency & stress loads**.

7. CI/CD & Deployment Standards

a. **Continuous Integration (CI)**

- i. Every **pull request must pass linting & tests** before merging.
- ii. Enforce **branch protection** and **review approvals**.
- iii. Run tests in **GitLab CI**.
- iv. Stages of CI must be:

```
1 build:
2   - clean
3   - build
4   - checkstyle
5 unit-testing:
6   - unit_test
7   - jacoco_test_report
8 dependency-check:
9   - dependency_check
10  - report
11 sonarqube:
12  - static_code_analysis
```

b. **Continuous Deployment (CD)**

- i. Use **Docker for containerization** and **Kubernetes for orchestration**.
- ii. Store container images in **DockerHub** or **local image registry**.
- iii. Deploy via **Helm**.

c. **Environment Configuration**

- i. Use **.env files only for local development**.
- ii. Manage configurations centrally with **Consul, Kubernetes ConfigMaps**.

8. Observability & Monitoring Standards

a. **Logging**

- i. Use **structured logging (JSON format)**.
- ii. Store logs in **ELK Stack**.

b. **Monitoring**

- i. Implement **Prometheus + Grafana** for system health metrics.
- ii. Use **Jaeger or OpenTelemetry** for distributed tracing.

c. **Alerting**


- i. Set up alerts using **Prometheus AlertManager**.
- ii. Define **SLOs (Service Level Objectives) & SLIs (Service Level Indicators)**.

9. Documentation & Code Review

a. **Code Review Process**

- i. Follow a **4-eye principle** (minimum 2 reviewers per MR).
- ii. Ensure every PR has:
 1. **Clear description** of changes.
 2. **Linked Jira issue**.
 3. **Passing CI/CD checks**.

b. **Documentation Standards**

- i. Maintain **API documentation in OpenAPI (Swagger)**.
- ii. Use **Markdown (.md)** for internal documentation  **ReadMe**).
- iii. Keep an updated **README with setup & usage instructions**.

Naming Standards for Jira, Git, Databases, and REST APIs

10. Jira Naming Standards

a. **Ticket Types & Structure**

- i. Each ticket should follow a **consistent format**:

[PROJECT-KEY]-[Type]: [Short description]

Example:

- AGR-102: BUG Fix authentication token expiration issue
- AGR-203: FEAT Implement order creation API

b. Ticket Types & Prefixes

- FEAT – New feature (AGR-101: FEAT - Add order tracking)
- BUG – Bug fix (AGR-102: BUG - Fix checkout failure)
- TASK – General task (AGR-103: TASK - Optimize SQL queries)
- DOC – Documentation (AGR-104: DOC - Update API specs)
- REF – Refactoring (AGR-105: REF - Restructure service layer)
- HOTFIX – Hot fix (AGR-102: HOTFIX - Fix payment gateway downtime issue)

c. Ticket Workflow

- Backlog → To Do → In Progress → Code Review → QA → Done**

- ii. **Move tickets through statuses regularly** to avoid stagnation.
- iii. **Tag Jira tickets in Git commits & PRs** (see Git standards below).

11. Git Naming Standards

a. **Branch Naming**

Branches should follow this format:

```
[type]/[JIRA-ID]-[short-description]
```

Examples:

- i. `feature/AGR-101-add-order-tracking`
- ii. `bugfix/AGR-102-fix-checkout-failure`
- iii. `hotfix/AGR-500-critical-db-issue`
- iv. `refactor/AGR-201-optimize-sql-queries`

b. **Commit Message Format**


Each commit should be clear and reference the related Jira ticket:

```
[JIRA-ID] [Change type]: [Short description]
```

Examples:

- i. `AGR-101 FEAT: Add order tracking functionality`
- ii. `AGR-102 BUG: Fix checkout failure due to token expiry`
- iii. `AGR-103 TASK: Update logging format for better debugging`
- iv. `AGR-500 HOTFIX: Fix payment gateway downtime issue`

c. **Git Tagging for Releases**

i. **Semantic versioning:** `v[MAJOR].[MINOR].[PATCH]` ( [Semantic Versioning 2.0.0](#))

- `v1.0.0` → First stable release
- `v1.1.0` → Minor feature updates
- `v1.1.1` → Patch (small bug fixes)

ii. **For hotfixes:** `hotfix-[version]`

`hotfix-v1.1.1-db-lock`

d. Branching Strategy: **Trunk-Based Development**

- Single Long-Lived Branch:** The **main branch** (or **trunk**) is the only long-lived branch in the repository. All development takes place here.
- Developers create **short-lived feature branches** for each task or user story. Branches should not stay open for more than **1-2 days**.
- Frequent Commits:** Developers should commit their work often (multiple times a day), before code review they have to squash their commits to achieve **1MR - 1commit** approach.
- For each new feature/task a custom branch should be taken from main.

12. Database Naming Standards

a. **General Guidelines**

- Use snake_case** for all relational database entities (`order_items`, `user_roles`).
- Avoid abbreviations** unless commonly understood (`customer` NOT `cust`).

iii. **No plurals in table names** (use `account` NOT `accounts`).

b. **Column Naming**

Format: `[entity]_id` , `[attribute]`

Examples:

- i. `user_id` (foreign key to `user` table)
- ii. `created_at` , `updated_at` (timestamps)
- iii. `is_active` (boolean flags use `is_` prefix)

c. **Index Naming**

Format: `idx_[table]_[columns]`

Example: `idx_order_transaction_user_id`

d. **Foreign Key Naming**

Format: `fk_[table]_[referenced_table]`

Example: `fk_order_transaction`

13. REST APIs Naming Standards

a. **General Guidelines**

- i. **Use nouns** for resources (entities).
- ii. **Use plural** nouns for collections (e.g., `/users` , `/orders`).
- iii. **Avoid verbs** in endpoint paths (e.g., `/getUsers` is incorrect).
- iv. **Use hyphens (-)** to separate words for readability (e.g., `/user-roles`).
- v. **Use camelCase** for query parameters (e.g., `userId` , `orderDate`).
- vi. **Use camelCase** for request/response bodies

vii. Use **HTTP methods** to define actions:

- **GET**: Fetch resource(s)
- **POST**: Create a resource
- **PUT/PATCH**: Update an existing resource
- **DELETE**: Delete a resource

viii. **Examples:**

- **GET** `/api/v1/users` → Fetch all users
- **POST** `/api/v1/users` → Create a new user
- **GET** `/api/v1/users/{userId}` → Get a specific user
- **PUT** `/api/v1/users/{userId}` → Update a specific user
- **PATCH** `/api/v1/users/{userId}` → Update a specific user partially (`activate`, `deactivate`, etc.)
- **DELETE** `/api/v1/users/{userId}` → Delete a specific user

b. **Nested Resources**

i. For relationships between entities, use **nested paths**:

ii. **Example:**

- **GET** `/users/{userId}/orders` → Get all orders of a specific user

- **GET** `/orders/{orderId}/items` → Get items in a specific order
- **POST** `/users/{userId}/orders` → Create a new order for a specific user

c. **Filtering, Sorting, and Pagination**

i. **Query parameters** should be used for filtering, sorting, and pagination.

ii. Example

- **pagination:** `/users?page=1&size=20`
- **filtering:** `/products?category=electronics&price=100-500`
- **sorting:** `/products?sort=price,asc`

14. HTTP Status Codes Naming

a. **Success Codes**

- 200 OK:** The request was successful (typically for `GET` requests).
- 201 Created:** The resource was created successfully (typically for `POST` requests).
- 204 No Content:** The request was successful, but there is no content to return (typically for `DELETE` or `PUT` requests).

b. **Error Codes**

- 400 Bad Request:** The request is malformed, or missing required data.

- ii. **401 Unauthorized:** Authentication is required but missing or invalid.
- iii. **403 Forbidden:** The user does not have permission to access the resource.
- iv. **404 Not Found:** The resource does not exist.
- v. **409 Conflict:** There is a conflict with the current state (e.g., duplicate data).
- vi. **500 Internal Server Error:** General server error.
- vii. **503 Service Unavailable:** The server is currently unable to handle the request due to temporary overloading or maintenance.

15. Versioning Standards

a. **API Versioning**

- i. Version your API to ensure backward compatibility.
- ii. **URL versioning** is the most commonly used approach:
 - `/api/v1/users` for version 1 of the API
 - `/api/v2/users` for version 2 of the API
- iii. *Alternatively*, you can use **request header versioning** (e.g., `Accept: application/vnd.myapi.v1+json`), but URL versioning is clearer and easier to debug.
- iv. **Example:**
 - `GET /api/v1/users`
 - `GET /api/v2/users`

16. HTTP Methods Naming

a. **GET**

- i. Used to **retrieve** data from the server.

ii. **Example:**

- `GET /users` → Fetch all users
- `GET /users/{userId}` → Fetch a specific user by ID

b. **POST**

- i. Used to **create** a new resource on the server.

ii. **Example:**

- `POST /users` → Create a new user

c. **PUT**

- i. Used to **update** an existing resource. It generally replaces the entire resource.

ii. **Example:**

- `PUT /users/{userId}` → Update a user by ID

d. **PATCH**

- i. Used to **partially update** an existing resource.

ii. **Example:**

- `PATCH /users/{userId}` → Update specific attributes of a user

e. **DELETE**

- i. Used to **remove** a resource from the server.

ii. **Example:**

- `DELETE /users/{userId}` → Delete a specific user by ID

17. Error Response Naming

a. **Error Format**

i. Return consistent error messages with relevant information, including the **status code** and **description**.

ii. **Example Response:**

```
1 {
2   "errorCode": "employee_not_found",
3   "errorMessage": "Employee not found with given userId: {userId}.",
4   "requestId": "62012553-74f3-4a8e-9a57-43ea91350c47",
5   "timestamp": "2024-12-23T12:34:56Z"
6 }
```

18. API Documentation and Documentation Standards

a. **Use OpenAPI (Swagger)**

- i. Document all endpoints using **OpenAPI Specification** (Swagger).
- ii. Include detailed information about request/response format, HTTP methods, status codes, and possible errors.

19. File and Media Handling Naming

a. **File Uploads and Downloads**

- i. `POST /uploads/images` → Upload an image.
- ii. `GET /uploads/images/{imageId}` → Download an image by ID.

b. Keep file paths clean, short, and readable.

RabbitMQ Standards

20. Exchange Types & Usages

Type Key	Type	Use Case Example	Description
dx	direct	Routing to a specific queue	Message goes to a queue with exact key

fx	fanout	Broadcasting notifications to multiple services	All bound queues get the message
tx	topic	Pattern-based routing	Flexible routing using wildcards
hx	headers	Advanced routing based on headers	Rarely used — discouraged unless needed
dlx	dead-letter-exchange	Helps isolate problematic messages.	Special exchange where messages are routed when they cannot be processed.

21. Exchange Naming

a. Format:

```
1 <microservice-short-name>.<operation-entity>.<exchange-type>
```

b. Examples:

```
1 up.assign-roles.dx
2 up.assign-roles.fx
3 up.assign-roles.tx
4 up.assign-roles.hx
5 up.assign-roles.dlx
```

22. Routing Key Naming

a. Format:

```
1 <microservice-short-name>.<operation>.<entity>.rk
```

b. Examples:

```
1 up.assign.roles.rk
```

23. Queue Naming

a. Format:

```
1 <microservice-short-name>.<operation>-<entity>.<queue-type>
```

b. Examples:

```
1 up.assign-roles.q
2 up.assign-roles.dlq
```

24. Message Structure

- a. All messages published to RabbitMQ should follow a unified format based on the `BaseEvent<T>` structure as follows:

```
1 public class BaseEvent<T extends Serializable> implements Serializable {
2     private String eventId; // Unique ID for this event (UUID)
3     private ResponseQueueInfo responseQueueInfo; // Optional info for response routing
4     private Map<String, String> headers; // Metadata (e.g. request-id,
timestamp)
5     private T payload; // Actual business payload
6 }
```

- b. Every message should follow a **standard JSON format**:

```
1 {
2     "eventId": "fd3b62e1-1bcf-4f8b-9db4-812e1fc7a8ce",
3     "responseQueueInfo": {
4         "exchangeName": "exchange-name",
5         "routingKey": "routing-key"
6     },
7     "headers": {
8         "x-request-id": "req-123", // Traceable request ID (correlation-
friendly)
9         "x-timestamp": "2025-04-15T12:45:00Z", // ISO timestamp for observability
10        "x-event-version": "1.0.0", // Payload schema versioning
11        "x-origin-service": "ms-user", // Service that produced this event
12        "x-target-service": "ms-user-permission", // Intended consumer, useful in debugging
13        "x-retry-count": "0", // Retry tracking across dead-letter queues
14        "x-content-type": "application/json" // Optional, but helpful for multi-format
support
15    },
16    "payload": {
17        "userId": 101,
18        "username": "jdoe",
19        "email": "jdoe@example.com"
20    }
21 }
```

25. Message Field Descriptions (Headers)

Header Key	Required	Description
<code>x-request-id</code>	✓	Traceable correlation ID
<code>x-timestamp</code>	✓	ISO-8601 event timestamp
<code>x-event-version</code>	✓	Version of the payload format
<code>x-origin-service</code>	✓	Service that generated the message


x-target-service	✗	Intended consumer (for debugging/monitoring)
x-retry-count	✗	Number of retry attempts (DLQ related)
x-content-type	✗	Message content format (e.g., application/json)

26. Messaging Responsibilities Matrix

Resource	Owner	Description
Exchange	Producer	Created by the publisher service that emits events.
Routing Key	Producer	Defined when sending messages, indicates event type.
Queue	Consumer	Declared by the service that processes messages.
Binding	Consumer	Created based on which routing keys the consumer wants to handle.

27. Technology Choice: Spring Cloud Stream with RabbitMQ

- a. We are standardizing message-based communication between microservices using [Spring Cloud Stream](#) with the **RabbitMQ binder**:

 Official docs: [Spring Cloud Stream RabbitMQ Binder Reference Guide](#)

- b. Why Spring Cloud Stream?

- Abstraction** over messaging middleware (RabbitMQ) — lets us write business logic without worrying about low-level queue/exchange setup.
- Automatic resource provisioning** — creates exchanges, queues, and bindings based on configuration (`application.yml` &

`application-local.yml`).

- iii. **Built-in support** for routing keys, consumer groups, partitioning, retries, and DLQs.
- iv. **Flexible naming and convention control** via simple YAML properties.
- c. All microservices communicating via RabbitMQ will use **Spring Cloud Stream** for publishing and consuming messages. This enforces a standard, declarative configuration model across the ecosystem, reducing boilerplate and ensuring consistency.
- d. Spring Cloud Stream Configuration Example

```
1 spring:
2   cloud:
3     stream:
4       bindings:
5         userCreated-out-0:
6           destination: user.created.dx
7           contentType: application/json
8         userCreated-in-0:
9           destination: user.created.dx
10          group: user-service
```

28. RabbitMQ Security Guidelines

#todo: to be implemented

29. Built-in RabbitMQ Management Plugin

#todo: to be implemented

30. Metrics Integration with Prometheus + Grafana

#todo: to be implemented

Functional Requirements & Non-Functional Requirements Standards

31. Why It's Important?

- a. **Functional Requirements (FRs)** describe **what the system should do** (features, use cases, business logic, etc.) in terms of features, behaviors, and interactions from a business perspective.
- b. **Non-Functional Requirements (NFRs)** describe **how the system should perform** (performance, scalability, security, etc.).

Non-functional requirements define the **system qualities, constraints, and performance benchmarks** that must be adhered to across the development lifecycle. These requirements are critical for system reliability, user experience, security, and maintainability.

32. Functional Requirements (FRs)

- a. FRs are primarily owned and defined by Business Analysts (BAs) in collaboration with Product Owners (POs) and business stakeholders. The development team is responsible for ensuring that these requirements are correctly interpreted, designed, implemented, and tested. The development team is responsible for ensuring that these requirements are correctly interpreted, designed, implemented, and tested.
- b. All FRs must be documented in a **centralized system** (e.g., Jira, Confluence, or BRD).
- c. Each user story must include **Acceptance Criteria**.
- d. Requirements must be **version-controlled** to track changes.
- e. Functional changes must be **reviewed and signed off** by BAs before development begins.

33. Non-Functional Requirements (NFRs)

- a. **Performance**: Response times < 200ms for critical APIs
- b. **Scalability**: Support for 10K concurrent users
- c. **Availability**: 99.9% uptime in production
- d. **Security**: JWT-based auth, OWASP top 10 compliance
- e. **Maintainability**: Code quality rules, modular design
- f. **Observability**: Logging, tracing, alerting standards via Grafana/Prometheus and ELK stack
- g. **Deployment**: CI/CD with rollback, multi-env compatibility

Key Categories of NFRs		
Category	Description	Standards & Benchmarks

Performance	System response times, throughput, latency	API response time < 200ms for 95th percentile
Scalability	Ability to handle increased load without affecting performance	Horizontal scaling to support 10,000 concurrent users
Availability	Uptime guarantees and fault tolerance	≥ 99.9% availability in production
Security	Protection of data and services against unauthorized access and attacks	JWT auth, OAuth2, encryption at rest and in transit
Maintainability	Ease of updating and extending the system with minimal risk	Code coverage > 80%, modular design, CI/CD pipelines
Reliability	System stability and error recovery mechanisms	Graceful fallbacks, retry logic on critical services
Observability	Ability to monitor, log, and trace system behavior	Centralized logging (ELK/Grafana), trace IDs in logs
Compliance	Conformance to legal, regulatory, and internal standards	GDPR, ISO 27001, internal audit logging
Portability	System's ability to operate across different environments or platforms	Containerization via Docker, environment variable configs
Usability	Ease of use and learning curve for end users	UX guidelines followed, onboarding within 10 mins
Disaster Recovery	Measures for backup, failover, and recovery	RTO < 1 hour, RPO < 15 minutes

34. FR & NFR Responsibilities Matrix in the SDLC

Functional Requirements in the SDLC

Stage	Responsibility	Activities
Requirements Gathering	BA/PO	Define and document functional requirements in BRD, user stories, or epics
Requirement Validation	BA, SA, TL, QA	Confirm technical feasibility and testability
Design	SA, TL	Design solutions aligned with functional expectations
Implementation	Developers	Develop features according to acceptance criteria
Testing	QA Engineers	Validate implementation against functional requirements (UAT, system tests)
Release & Sign-off	PO/BA, QA, Stakeholders	Verify business objectives are met before release

NFR Responsibility Matrix		
NFR Category	Responsible Role(s)	Activities / Deliverables
Performance	TL, Developers	Load testing, API benchmarking, DB optimization
Scalability	SA, DevOps Engineers	Horizontal scaling setup, stateless service architecture
Availability	SA, DevOps Engineers, SRE	HA setup, monitoring alerts, redundancy configuration
Security	Security Lead, Developers	Threat modeling, token validation, secure code reviews
Maintainability	Developers, QA Leads	Modular architecture, clean code, automated tests

Reliability	Developers, QA, SRE	Circuit breakers, retries, chaos testing
Observability	DevOps, Developers	Structured logging, dashboards, alerts
Compliance	Legal/Compliance Officer, Security Engineer	Compliance reports, data handling policies
Portability	SA, DevOps Engineers	Containerization, infra-as-code, multi-env testing
Usability	UI/UX Designers, Frontend Developers	Accessibility testing, UI consistency reviews
Disaster Recovery	DevOps, IT Operations	Backup scripts, DR drills, recovery runbooks

Development Workflow & Lifecycle: Agile Process Standards

35. This section defines the full lifecycle of a development task, from planning to release, and outlines the standards for what makes a task ready for development and what qualifies it as done. It ensures alignment across engineering, QA, and product teams. We follow an agile workflow across all projects, where a feature or task flows through the following stages:
- Backlog** – Ideas and requests are captured here.
 - Ready for Development** – Meets the Definition of Ready.
 - In Progress** – Actively being implemented.
 - Code Review** – Undergoing peer review.
 - Ready for QA** – Feature is implemented and awaits validation.
 - In QA** – Being tested manually and/or automatically.
 - Ready for Release** – QA passed, accepted by product owner.
 - Done/Released** – Deployed to production or scheduled for release.
36. **Definition of Ready (DoR)**. DoR applies at the start of the development process (Backlog → Ready). A task is considered **Ready** when it satisfies all the following:
- Business value and context are clearly described.

- b. Acceptance criteria are clearly written and testable.
- c. Task is estimated and appropriately sized for a sprint.
- d. UI/UX designs are attached if applicable.
- e. Dependencies are identified and unblocked.
- f. API contracts (if needed) are defined.
- g. Required environments, data, and test scenarios are prepared.
- h. No open questions or blocking issues remain.
- i. The Product Owner/Business Analyst has reviewed and confirmed readiness.
- j. QA has visibility into how to validate the feature.

37. **Definition of Done (DoD).** DoD applies at the end of the process (Ready → Done). A task is considered **Done** only when all of the following are complete:

- a. All acceptance criteria are implemented and verified.
- b. Code is committed, pushed, and peer-reviewed.
- c. CI pipeline passes without errors.
- d. Functional and non-functional tests (unit/integration) are written and passed.
- e. Code follows team's formatting and linting rules.
- f. Feature is tested in staging/test environment.
- g. No high/critical severity bugs remain.
- h. Feature demoed and/or accepted by the Product Owner.
- i. Documentation is written or updated (code comments, user docs, API docs).
- j. Monitoring, alerting, and logging hooks are in place if applicable.
- k. Feature is deployable or already deployed per release process.

38. **Acceptance Criteria (AC).** AC define the **conditions that must be met** for a feature, user story, or task to be accepted as complete by the product owner and stakeholders. Clear and consistent AC help eliminate ambiguity, improve estimation, and streamline testing. AC must be:

- a. **Clear** – No ambiguity or assumptions.
- b. **Testable** – Can be verified with automated/manual tests.
- c. **Independent** – Avoid overlap with other tasks unless linked.

d. **User-centered** – Framed from the user's perspective (e.g., "As a user, I want...")

39. Structure of Good Acceptance Criteria. We follow a **Given-When-Then** (Gherkin-style) format where possible:

```
1 Given: [initial context],
2 When: [an action is taken],
3 Then: [an expected result should occur]
```

Example:

```
1 Title: As a user, I want to reset my password so that I can regain access to my account if I
  forget it.
2 Description: Users should be able to request a password reset via email. When they provide a
  registered email address, the system should send them a secure link to reset their password. The
  link should expire after a predefined time 1 hour.
3 Acceptance Criteria:
4   Scenario: Navigating to password reset
5     Given: the user is on the login page
6     When: they click "Forgot Password"
7     Then: they are redirected to the "Reset Password" page
```

40. Jira story example

```
1 Id: SEC-201
2 Title: As a user, I want to enable two-factor authentication (2FA) so that my account is more
  secure.
3 Type: Story
4 Epic: User Account Security
5 Priority: High
6 Story points: 8
7 Assignee: developer_name
8 Sprint: Sprint 26
9
10 Description:
11   Implement Two-Factor Authentication (2FA) using TOTP (e.g., Google Authenticator).
12   Users should be able to enable or disable 2FA in their account settings.
13   Once enabled, they will need to enter a TOTP code on each login in addition to their password.
14
15 Acceptance Criteria (AC):
16   - User sees an option to enable 2FA in the Security Settings page.
17   - Clicking "Enable 2FA" displays a QR code for an authenticator app.
18   - Entering the correct TOTP code activates 2FA.
19   - 2FA is required at login after activation.
20   - Invalid TOTP codes result in an error and denied access.
21   - Disabling 2FA requires password confirmation.
22   - TOTP secrets are stored securely and expire as required.
23
24 Definition of Ready (DoR):
25   - Business context and value are defined.
26   - Acceptance criteria are clear and testable.
27   - Designs and flows are available.
28   - API contracts are agreed upon.
29   - No blocking dependencies.
30   - Story is estimated and approved by product owner.
```



```
31
32 Definition of Done (DoD):
33   - Code is implemented, peer-reviewed, and merged.
34   - All acceptance criteria are met.
35   - Unit and integration tests are complete and passing.
36   - QA has verified the feature.
37   - Security validations are complete.
38   - Feature demoed and accepted by product owner.
39   - User documentation is updated.
```

41. to be continued...