भारतीय राष्ट्रीय भुगतान निगम
NATIONAL PAYMENTS CORPORATION OF INDIA

NPCI DEEP TECH

The way to Go

Document History

Date	**Changes**	**Version**	**Author**	**Reviewed**	**Approver**
29 Jul 2023	First version of EA & Governance Principles and Practices	1.0	Arul Kumar	Nabar	Nabar
24 Nov 2025	Re-Reviewed the documentation	2.0	Thillaikkarasan Murali	Jemima Joy Thangaraj	Nitin Mishra

© 2025 NPCI, all rights reserved

Contents

© 2025 NPCI, all rights reserved

1. Purpose

The objective is to create NPCI a standard governance principles and practices document for Architecture, design and code quality with principles, tools & frameworks, and skillsets. This is to help the organization to achieve build platforms and products to achieve 4D Principles to facilitate growth, higher productivity, and competitive advantage.

This document details the principles and governance rules of following key areas and its subsections as required which each application, systems need to be followed.

- Architecture
- Design
- Tech Stack
- Code & Quality

2. Enterprise Architecture

Enterprise Architecture (EA) aligns an organization's business processes, technology infrastructure, information flow, and human resources with its strategic vision. It serves as a blueprint, offering a holistic view of the organization's current state and guiding its future state to achieve efficient business operations.

Here are major areas covered in the document.

2.1 Business Architecture: Defines business strategies, processes, and functions, aligning them with overall goals.

2.2 Information Architecture: Manages data and information assets, ensuring consistency, accuracy, and security.

2.3 Application Architecture: Designs and manages software applications and systems supporting business processes.

2.4 Technology Architecture: Addresses infrastructure and technology components to support organizational operations.

2.5 Integration Architecture: Ensures seamless communication and data flow among systems and applications.

2.6 Security Architecture: Safeguards assets, data, and systems against potential threats and vulnerabilities.

© 2025 NPCI, all rights reserved

2.7 Governance and Compliance: Aligns EA efforts with governance and compliance requirements.

2.8 Human Capital Architecture: Addresses the human resources and skills required for successful EA implementation.

Enterprise Architecture enables organizations to understand their current state, define a desired future state, and chart a strategic roadmap to achieve business objectives while adapting to changes effectively.

3. Architecture Principles

The principles are designed to optimize performance, enhance scalability, ensure data consistency, and maintain reliability. Here is a description of principles should be followed.

3.1 Complete In-Memory Processing: Optimize critical path operations to be performed entirely in memory, no dependency on database.

3.2 Minimize Network Hops: Reduce the number of network calls by processing transactions as close as possible to the respective site, minimizing latency.

3.3 Efficient Replication Strategy: Utilize Kafka as the primary mechanism for replication, minimizing the reliance on native replication methods.

3.4 Asynchronous Communication: Implement asynchronous communication patterns to decouple components and enable parallel processing, enhancing overall system performance.

3.5 Event-Driven Message Architecture: Implement an event-driven message architecture to avoid back pressure and ensure efficient handling of messages.

3.6 Technology Upgrades: Stay up to date with the latest technology trends and regularly upgrade system components to leverage new features and improvements.

3.7 Effective Sharding and Partition Management: Utilize efficient sharding and partitioning techniques to ensure data consistency and optimize performance.

3.8 High Availability: Active – Active online transaction processing across sites, Applications should run in single site in worst-case scenarios with minimal or near-zero data loss to ensure continuous availability.

3.9 Minimize Disk Writes: Limit disk writes to ensure data safety without compromising system performance.

3.10 Site Affinity: Consistently redirect to the same site for a personalized experience, and switch to alternative sites in the event of a failure to maintain service continuity.

© 2025 NPCI, all rights reserved

3.11 Message Compression: Selective compression / decompression algorithm based on payload and performance and decision based to end to end to needs.

3.12 Software as service: Application should be built on top of pluggable architecture which should allow plugins to run on top of an app container which should provide services such as security, connectivity, rendering and business services. This possibly allows open-source communities to build plugins to drive innovation.

3.13 Self-Serviceable: Anonymized data interface / service available to understand trends to predict future and innovate new products for up selling or cross selling.

3.14 Shared-Nothing Parallelism: Allow large and complex tasks to be performed quickly by operating on many data store shards/instances records concurrently.

3.15 Uniform interface: Design a uniform way of interacting with a given service regardless of device or application type.

3.16 Stateless: No client context shall be stored on the server between requests. The client is responsible for managing the state of the application.

3.17 Caching: Well-managed caching partially or eliminates some client-server interactions, further improving scalability and performance.

3.18 Layered: Complete Isolation of business and technical layers, should allow to change the technical frameworks without any impact to business services

4. Architecture Artefacts

Below are artefacts required to be submitted for a new / existing for any change impacts overall architecture or design.

4.1 Use Case Diagrams: Use case diagrams depict the interaction between users and the system, showcasing the various actions and outcomes.

4.2 System Architecture Diagram: This high-level diagram provides an overview of the application's structure, showing major components and their interactions.

4.3 Component Diagrams: Component diagrams illustrate the physical and logical components of the system, including libraries, modules, and services.

4.4 Class Diagrams: Class diagrams present the static structure of the application, depicting classes, their attributes, and the relationships between classes.

4.5 Sequence Diagrams: Sequence diagrams show the interactions and order of messages between objects or components, highlighting the flow of control in the application.

4.6 Deployment Diagrams: Deployment diagrams display the physical arrangement of components on hardware, such as servers and devices.

© 2025 NPCI, all rights reserved

4.7 Data Flow Diagrams (DFD): DFDs represent the flow of data within the system, showing how information moves between processes, data stores, and external entities.

4.8 State Diagrams: State diagrams model the behavior of objects or components over time, illustrating different states and state transitions.

4.9 Entity-Relationship Diagram (ERD): ERD is used to model and visualize the database schema, representing entities, attributes, and relationships.

5. Design

Software application design principles are guidelines and best practices that help in creating well-structured, maintainable, and scalable systems / applications. Here are the essential principles of software application design:

5.1 Modularity: Divide the application into independent modules that can be developed, tested, and maintained separately. Promote code reusability, easier to understand and modify specific parts of the application without affecting others.

5.2 Separation of Concerns (SoC): Organize the application into distinct layers or components, each responsible for specific functionality. For example, separate the presentation layer (user interface) from the business logic and data access layers.

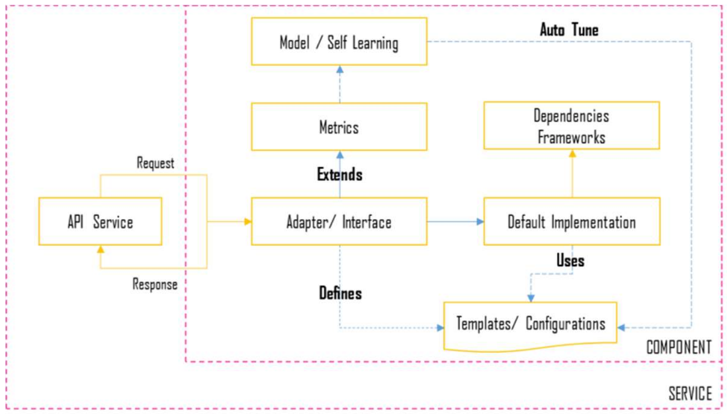
5.3 Single Responsibility Principle (SRP): Each module, class, or function should have a single responsibility and should focus on doing one thing well.

5.4 Don't Repeat Yourself (DRY): Avoid duplicating code by extracting common functionality into reusable functions, classes, or libraries. DRY principle reduces code maintenance effort and minimizes bugs.

5.5 Open/Closed Principle (OCP): Software entities (classes, modules, etc.) should be open for extension but closed for modification. This means new functionality should developed by extending the existing code rather than changing core of it.

5.6 Liskov Substitution Principle (LSP): Objects of a superclass should be replaceable with objects of its subclasses without affecting the correctness of the program.

Below figure shows how an independent component architecture & design should be



- Independent maven / gradle project
- Consumers are allowed access the behaviors through Interface only
- Default Implementation to cover most of the functionalities

- Component will be placed inside the NCPCI Artifactry Repository
- Versioned, Customization will be staged at product level
- Change in Dependency framework should not impact product behaviors

5.7 Dependency Inversion Principle (DIP): Depend on abstractions rather than concrete implementations.

5.8 Interface Segregation Principle (ISP): Clients should not be forced to depend on interfaces they do not use.

5.9 Composition over Inheritance: Favor composition (building complex objects by combining simpler ones) over inheritance (creating complex objects as subclasses of existing ones). Composition provides greater flexibility and reduces the tight coupling between classes.

5.10 Encapsulation: Encapsulate the internal state and implementation details of objects, providing access only through well-defined interfaces. This shields the internal complexity from outside interference.

5.11 Keep It Simple and Stupid (KISS): Strive for simplicity in design and implementation.

5.12 Don't Optimize Prematurely (YAGNI): Only add features or optimizations when they are needed.

5.13 Testing and Maintainability (TDD): Design the application with testing in mind. Make it easy to write unit tests and ensure that the application is maintainable over time.

5.14 Consistency: Maintain consistency in code style, naming conventions, and overall design throughout the application.

6. Architecture & Design Review

Architecture review is an essential process to assess the design and structure of a software system to ensure it meets the required quality standards and aligns with the organization's goals. Principles and best practices outlined in the document will be used to review the architecture.

6.1 Architecture Review Submission to EA Team: To ensure compliance with architecture, and design for any new product or modification to existing must be submitted to the EA Team.

The EA Team will review the design contracts and static contracts to guarantee alignment with the established architecture.

6.2 Mandatory Architecture Process: Before any implementation/coding/ development, it is mandatory to conduct a complete architecture review of the proposed quality process. The objective is to ensure that the architecture meets our high standards before it goes live.

6.3 Security and CMC teams to ensure the review compliance during change management approval before any actual implementation in UAT.

Below are the documents should be submitted for any new or modification of existing architecture for review.

6.4 Software Requirements Specification (SRS): Outlines the functional and non-functional requirements of the software system. It provides a foundation for evaluating how well the architecture addresses these requirements.

6.5 High-Level Design (HLD) Document: Describes the overall architecture, major components, and their interactions.

6.6 Low-Level Design (LLD) Document: Provides detailed design specifications for each component, including algorithms, data structures, and interfaces to proper implementation of the system's functionalities.

6.7 Architectural Diagrams: Architectural diagrams, such as block diagrams, flowcharts, sequence diagrams, and UML (Unified Markup Language) diagrams, visually represent the software's structure and behaviour.

6.8 Data Flow Diagrams (DFDs): DFDs show how data flows through the system. They help identify data dependencies, processing logic, and potential bottlenecks in the software.

6.9 Interface Definitions: Detailed specifications of APIs, protocols, and communication mechanisms used by the system should be provided. Reviewers evaluate these interfaces for consistency and compatibility.

6.10 Security Documentation: Describes the details of sensitive data or critical operations which includes security architecture, threat models, and measures taken to protect against vulnerabilities.

6.11 Performance Requirements: Provides performance requirements, such as response times, throughput, and scalability expectations, is crucial for evaluating whether the architecture meets these criteria.

6.12 Testing and Validation Documentation: Information about the testing strategy, test plans, and validation results provides insights into the software's reliability and robustness.

© 2025 NPCI, all rights reserved

6.13 API Repository: Ensure standardization and consistency across APIs for all products.

6.13.1 Implement a clear and consistent versioning strategy for APIs,

6.13.2 Provide comprehensive and easily accessible documentation for each API

6.13.3 Define clear access control policies to determine who can access specific APIs and what actions they can perform.

6.13.4 Standardized status codes and provide meaningful error messages to help API consumers understand and address issues.

6.13.5 Clearly communicate and follow a deprecation strategy for retiring older API versions.

6.14 Detailed Secure coding and best practices documented in NPCI Coding Principles & Practices – Secure, Quality, Performance v1.0.docx

7. Tech Stack

Below table shows accepted technical tools and frameworks could be used to build the systems and application. Any new framework or tool filled review form could be shared with EA team to find suitability of requirement.

Type	**Tool / Framework**
Architecture & Design	Planetuml, Drawnet, draw.io
App Languages	Java, Go, Python, Xml, Xsd, Html, Javascript, typescript, rust
Mobile Languages	Objective C, Android Java, React Native, kotlin
Infra Languages	Ansible, Terraform
Java Frameworks	Spring, Vertx, Undertow, Lettuce, J-React, Netty
Go Frameworks	gin, fasthttp, gokit
Front end Languages	Angular, React
Data Ingestion Languages	Scala, Java
Python Api Frameworks	Fastapi,
Python Ml Frameworks	Scikit-Learn, Tensor Flow, Pytorch
Python Frond End	React, Django
Ml-Model-Tools	Feast, Seldon Core
Proxy	Nginx, Ha-Proxy
Ci-Cd	Jenkins, gitlab runners, argo
Orchestration	Kubernetes (Okd / Rancher) (Istio, Linkerd), Hashicorp Nomad
Container	Docker, containerd
Software Configuration	Git (Gitlab Community Version)
Software Repository	Nexus Community Version

Build Tools	Maven, Gradle, Groovy
Code Analyzer	Sonarqube, Pmd, Findbugs, Checkmarx, Uc-Detector
Testing	Jmeter, Selenium
Blockchain	Hyperledger, substrate, hyperledger firefly
Ide	Ecplise, Pycharm, Intelji, Visual Studio Code, Jupyter, goland
Data Visualization	Grafana, Kibana, Superset, Tableau
In-Memory Store	Redis, Keydb, dragonfly
Embedded Database	HSQL, Badger, LevelDB, RocksDB
Middleware	Kafka
No Sql Store	Cassandra, Elastic Search, Mongo, Clickhouse, Promethus, Hadoop
Sql Store	Postgres, Maria, Mysql
MLOps	Mlflow, Kubeflow, Airflow
API Gateway	Apache APISIX, Kong
Data Processing	Flink, Druid, Spark, Trino, Dagster
Data Catalogues	Data Hub
Realtime Datastore Format	Apache Hudi
Vault Service	Hashcorp-Vault
Data Object Store	Minio, Ceph
Pocket Analyzer	Moloch, Nmap, Wireshark
Container - Security	Anchore, Clair, Bane, Trivy
Vulnerability - Assessment	Openvas, Openscap
Security Smoke Testing	Zap
Identity Management	Arcos, Open-Pim
File Integrity	Carbon Black, Trend Micro
Anti Advanced Persistent Threats	Fire-Eye
Database Activity Monitoring	Imperva
Internet Proxy	McAfee
Security Incident Event Monitoring	HP Arcsight
Vulnerability Management	Nessus, Nexpose

Static & Dynamic Security Testing	Checkmarx And HP Fortify

8. Coding Standards

Below are best practices should be followed during coding / development.

8.1 General/Secure Coding Best Practices

8.1.1 Add clear comments on each code commit to git version control to track code changes and facilitate collaboration.

8.1.2 Deploy only tested and approved managed code rather than creating new unmanaged code.

8.1.3 Utilize only specific built-in APIs to conduct operating system tasks. Do not allow the application to issue commands directly to the Operating System, especially through the use of application-initiated command shells.

8.1.4 Do not pass user supplied data to any dynamic execution function.

8.1.5 Use checksums or hashes to verify the integrity of interpreted code, libraries, executables, and configuration files.

8.1.6 Utilize locking to prevent multiple simultaneous requests or use a synchronization mechanism to prevent race conditions.

8.1.7 Review all secondary applications, third party code and libraries to determine business necessity and validate safe functionality, as these can introduce new vulnerabilities.

8.2 Request Parameters / Input Validation

8.2.1 Define boundary (data length, data type, data range) for every request parameter and validation failure of any parameter should lead to request rejection.

8.2.2 Do not allow common hazardous characters such as < > " ' % ( ) & + \ \' \" ../ ..\

8.2.3 Allow only request and response headers which are required, defined application and then by protocol.

8.3 Authentication and Password Management

8.3.1 By default, no access to any resources of application or system, allow only whitelisted master data for processing.

8.3.2 Centralized multi factor authentication and authorization through clearly defined access control list, controls should fail safely.

8.3.3 Source code should not have any credentials (Example: database passwords), should be stored in location where it could attached to production server dynamically.

8.3.4 Enforce complex password policy, password should be always hashed and stored.

8.3.5 Force to use HTTP Post for all data updates.

8.4 Session Management

8.4.1 Applicable only for websites, portals, session should not be established for any api based services.

8.4.2 Generate a new session identifier on any re-authentication.

8.4.3 Do not allow concurrent logins with same user ID

8.4.4 Logout should fully terminate the associated session or connection, logout option should be available in pages protected by authorization.

8.5 Access Control

8.5.1 Classify the services internal and external, each category has to be guarded with appropriate access control.

8.5.2 Access control should be approved with maker/checker whenever service wants to be enabled for external or internal systems.

8.5.3 By default, each variable should be explicitly declared as private and only declare a functional required to be accessed externally.

8.5.4 Access to services exposed, always using dynamic time-based token derived from account and secret key which is given by producer of service.

8.5.5 The time to live of a token needs to be configurable and should have an option to invalidate token any point of time.

8.6 Error Handling & Logging

8.6.1 Do not disclose sensitive information (personal, systems) in error responses, including system details, session identifiers or account information

8.6.2 Error handlers should not display debugging or stack trace information

8.6.3 Implement generic error messages with application specific custom error pages

8.6.4 Do not store data in context and if needed clean-up once the transaction processing over with defined time limit

8.6.5 Log for both success and failure of specified security events

8.6.6 Log data as non-executable as code or simply as log plan text.

8.6.7 Log all input validation failures, authentication attempts, access control failures, TLS connection failures, cryptographic module failures

8.6.8 Audit the logs consistency for anomalies using different dynamic ruleset (incoming IP, data size, no of hits per second, comparisons)

© 2025 NPCI, all rights reserved

8.7 Application/System Configuration

8.7.1 Ensure servers, frameworks and system components are running the latest approved version with approved security patches.

8.7.2 Turn off directory listings in web and app servers to prevent the disclosure of directory structure. Remove unnecessary information from HTTP response headers related to the OS, web-server version and application frameworks

8.7.3 Remove all unnecessary functionality and files

8.7.4 Remove test code or any functionality not intended for production, prior to deployment.

8.7.5 Allow only HTTP Methods (POST, GET) and application should not use versions which are obsolete or deprecated.

8.7.6 Isolate development environments from the production network avenue for exploitation

8.7.7 Ensure to have CI/CD to manage and record changes to the code both in development and production.

8.7.8 Use secure credentials for database access, Connection strings should not be hard coded within the application. Connection strings should be stored in a separate configuration file on a trusted system and they should be encrypted

9. Secure Source Code Practices

9.1 Implement the Principle of Least Privilege: Ensure that users, applications, and systems are granted only the minimum permissions necessary to perform their designated tasks. This practice minimizes the potential impact of a security breach and limits unauthorized access to sensitive data.

9.2 Secure Temporary Data Storage: Protect all cached or temporary copies of sensitive data stored on the server (e.g., Application Memory, Redis, Kafka) from unauthorized access. It is essential to define a time to live for cached data and promptly purge it when no longer required to reduce the exposure of sensitive information.

9.3 Encryption of Highly Sensitive Stored Information: Implement strong encryption (Example: >= AES 256) for highly sensitive data marked as PII. This ensures that even if unauthorized access occurs, the data remains unreadable and secure.

9.4 Protection of Master Secrets: Safeguard master secrets from unauthorized access by encrypting keys using latest encryption algorithms and store in Hardware Security Modules (HSM) and persisting them securely in the database.

9.5 Avoid Storing Sensitive Information in Clear Text: Refrain from storing sensitive information such as passwords, connection strings, or other critical data in clear text.

Use strong encryption techniques to protect such information.

9.6 Minimize Application and System Documentation: Limit the amount of sensitive information exposed in application and system documentation, as it can be valuable to attackers. Avoid revealing specifics about encryption methods or other security measures.

9.7 Avoid Sensitive Information in HTTP GET Requests: Do not include sensitive data in HTTP GET request parameters, as this information can be easily visible and exposed in various logs or network traces.

9.8 Disable Client-Side Caching for Sensitive Information: Prevent client-side caching of data containing sensitive information by setting appropriate HTTP headers like "Cache-Control: no-store" and "Pragma: no-cache."

9.9 Support Configuration-Based Sensitive Data Removal: Ensure that the application allows sensitive data to be removed or obfuscated through configuration settings when it is deemed sensitive.

9.10 Encryption for Data in Motion: Implement encryption for the transmission of all sensitive information over networks. This should include using TLS to protect connections, and additional encryption can be applied for sensitive files or non-HTTPbased connections.

9.11 Utilize TLS and Specific Character Encodings: Enforce the use of TLS for all connections to secure data transmission. Additionally, ensure that connections use specific character encodings, like UTF-8, to avoid data corruption or security vulnerabilities.

9.12 Secure Parameter Removal in Requests: Remove parameters containing sensitive information while forwarding or transforming requests (e.g., PSPs) to avoid unintentional exposure.

10. Source Code Tool

10.1 Internal git(https://git.npci.org.in) should be used for all source code Management, use two-factor authentication (2FA) to access the git account.

10.2 Vendors and Partners are required to have NPCI Owned Laptop and valid NPCI domain id to access the code repo and product owners are responsible to ensure all agents are in place along with IS Team.

10.3 Each employee should access their git account only, the changes should committed to git using own domain id and password only.

© 2025 NPCI, all rights reserved

11. Source Code Roles

Four main roles are configured to secure the repository management and code quality

11.1 Enterprise architecture team owns the administrative control of git and new repo creation should be formally requested with the filled template from development / production support teams.

11.2 EA team assigns maintainer(s) to ensure every code gets reviewed before any commit to the branch.

11.3 Maintainer will help to create the feature branches and ensure every branch sync, merge constantly with master, any deviation is a critical violation of code quality.

11.4 Developers push the code to feature branch, after successful review of code by maintainer(s).

11.5 Review the license, IP and copyright information before using / copying code from internet source code repos in order to protect organization from indemnity, compliance, copyright patent and infringement issues.

12. Source Code – Review

12.1 Source Code review will be performed against all best practices outlined in the document (but not limited to), Time to time this will be changed to adopt industry best practices.

12.2 Source Code Review Submission to EA Team: To ensure compliance with architecture, all source code reviews for any new release must be submitted to the EA Team. The EA Team will review the design contracts and static contracts to guarantee alignment with the established architecture.

12.3 Traceability of Deployed Code: Every binary/object code deployed in production must have complete traceability against the code review comments. This ensures that any changes or updates made during the code review process are reflected accurately in the production environment.

12.4 Mandatory Code Quality Process: Before any production deployment, it is mandatory to conduct a code quality process. This step aims to streamline and improve code quality by identifying and addressing potential issues early on. The objective is to ensure that the code meets our high standards before it goes live.

12.5 Code Review Completion and Risk Acceptance: In situations where the application owner cannot complete the code review comments, they must request to fill and submit a code review risk acceptance form. This form will document the identified risks and indicate that the code is being deployed despite outstanding review comments.

12.6 CMC Approval for Production Deployment: The EA Team approval is must before any production deployment takes place. This ensures that all necessary code reviews have been completed and any potential risks have been appropriately acknowledged and accepted.

13. Source Code - Branching Strategy

13.1 Only one master will be allowed for a project in a repo which will be always in sync with production deployed code.

13.2 Use always branch(s) for development. Master is not a branch.

13.3 Any development activities are not allowed in master.

13.4 Access restricted and only maintainer can accept the merge requests which will all the development changes to master from branch(s).

13.5 Maintainer responsible for clean master which is all times working.

13.6 Merge the changes from a branch only when there is a release, then create a tag, use for deployment.

13.7 Each product is allowed to have one or more feature branches; a release branch(tagging) can be used before proceeding to production rollout.

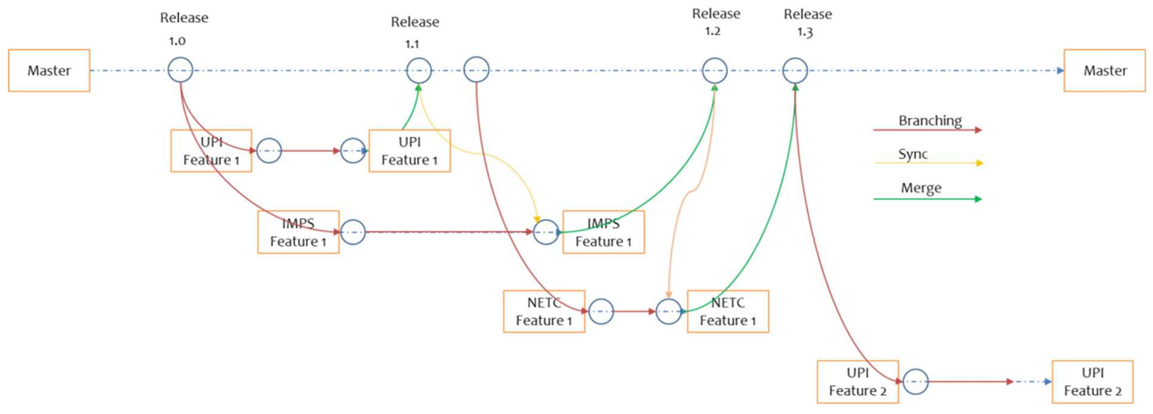
13.8 Short/Hotfix branches can be created for any urgent fixes

13.9 In-charge/ Lead developer of the product will be maintainer for the given repo/project.

13.10 After successful release and merge to master, the feature branches should be deleted and no further development.

13.11 Maintainer is responsible to maintain healthy no of branches, in case not EA administrator will free the repo/project until it corrected.

Below figures shows the branching strategy

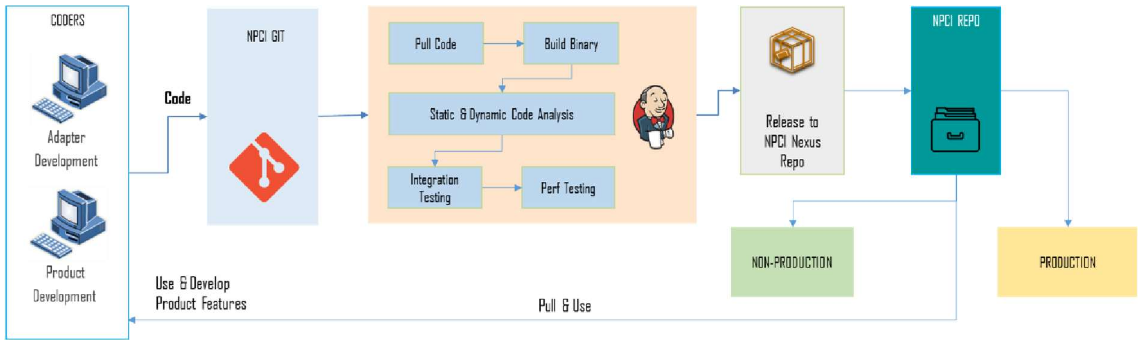


13.12 In case two branches have the same release date, merge both branches to avoid reworks and then it could be deployed.

13.13 Branch sync from master should be done when there is a merge to master from another release, this should be critical and failing will not be allowed to go for a release.

13.14 Merge to master should be done in co-ordination with In-charge / Lead developer of branches.

14. DevOps



14.1 Every application should have independent pipelines to build, test and deploy to production and non-production environments, no manual deployment allowed and ensured by run division of application team also by security team (checker) and compliance/governance to ensure this will be done by enterprise architecture team 14.2 Each successful application binary for release code should be pushed to nexus repo.

14.3 Deployment of application binary/object code should be only from nexus repo.

14.4 Performance testing results for each application release should be shared (minimum single instance statistics) for a review.

15. Conclusion

Enterprise Architecture and Governance Policies and its reviews are critical for developing robust and future-ready technology platforms aligned with industry best practices. These reviews play a pivotal role in ensuring that the organization's IT landscape is optimized for efficiency, security, and scalability.

Here are key points:

© 2025 NPCI, all rights reserved

15.1 Alignment with Technology Trends: Enterprise architecture team reviews ensure that technology platforms and solutions align with current industry trends and standards. Regular assessments help identify and incorporate emerging technologies that can provide a competitive advantage.

15.2 Compliance with Best Practices: By adhering to established governance policies, the organization can maintain consistent practices across IT projects. These best practices encompass aspects such as system design, data management, security protocols, and scalability, fostering a cohesive and standardized IT environment.

15.3 Integration with 4D Principles: The 4D principles encompass Zero Downtime, Zero Delay, Zero Danger, Zero Defects. Enterprise architecture and governance reviews will ensure that these principles are implemented effectively throughout the software development life cycle. This leads to better-controlled projects and reduces the likelihood of costly errors or security vulnerabilities.

15.4 Collaboration with Security and Audit Functions: Incorporating security and audit functions into the review process enhances the overall effectiveness of IT governance. Security professionals can assess the architecture's resilience against potential threats, ensuring that adequate security measures are in place to safeguard data and systems. Audit functions can provide an independent evaluation of adherence to governance policies, thus enhancing transparency and accountability.

15.5 Adherence to Compliance Regulations: Enterprise architecture reviews ensure that the organization's IT systems and applications comply with relevant regulatory requirements. By proactively addressing compliance issues, the organization reduces the risk of legal repercussions and reputational damage.

15.6 Identification of Opportunities for Optimization: Regular reviews enable the identification of areas where optimization can be achieved. This might involve streamlining processes, consolidating systems, or eliminating redundant technologies, leading to cost savings and operational efficiencies.

15.7 Continuous Improvement and Agility: The iterative nature of enterprise architecture reviews allows the organization to adapt quickly to evolving business needs and market dynamics. By continuously improving technology platforms, the organization remains agile and responsive in an ever-changing landscape.

© 2025 NPCI, all rights reserved