Java Delivery Playbook Document

Table of Contents

PURPOSE OF THIS DOCUMENT **4**

SOFTWARE DELIVERY JOURNEY **5**

PLANNING: 5
DELIVERY METHODOLOGY - SELECTION OF PROJECT LIFE CYCLE: 5
Linear Development Lifecycle (LDM) *5*
Agile Delivery Lifecycle – Cap Gemini Agile Framework (CAF) *8*
DEVELOPMENT: 8
Estimation and Sizing: *8*
Continuous Integration *10*
Software development *10*
Build process – Used for building Java projects. *10*
Recommended Build tools *11*
Maven (POM): *11*
Gradle: *11*
Coding Conventions: *12*
Use Correct Naming Conventions *12*
Logging *12*
Defensive Coding and resiliency *12*
Naming Conventions: *13*
Java Application Logging Framework *13*
Junit Unit Testing Framework *13*
Exception Handling *13*
BACK-END JAVA 14
Best Practices: *14*
Validate Inputs and handle errors: *14*
Handle error using: *14*
Unit Testing: *14*
Source Code: *14*
Package Naming Conventions: *14*
Test Case Naming Conventions: *15*
Expected Vs Actual: *15*
Appropriate Annotations: *15*
Specific Unit Tests: *15*
TESTING AND VALIDATION 15
Functional Testing *15*
Non-Functional Testing *16*
Other Aspects of Testing Strategy *16*
Monitoring: *17*
METRICS LANDSCAPE AND OPERATIONAL DEFINITIONS 18
Terminologies: *19*
CHECK LISTS, GUIDELINES AND TEMPLATES 19
Linear Development Method of SDLC: *19*
Agile Delivery Lifecycle – Cap Gemini Agile Framework (CAF) *19*
GEN AI TOOLS: 20
Phase wise tools overview: *20*
Tools Samples: *20*
CASE STUDIES: 21

© 2024 Capgemini - All rights reserved

Printed copies are current on date of printing only - 06/05/2024. Always refer to the electronic version for the current release.

Reducing Technical Debts with AI: *21*
Harnessing the power of AI Coding Assistant: *21*
APPENDIX A: 22
Java-Microservices *22*
Dev Ops *22*
Java *22*
Use API (Application Program Interface) Gateway *23*
Secure your communication! *23*

© 2024 Capgemini - All rights reserved

Printed copies are current on date of printing only - 06/05/2024. Always refer to the electronic version for the current release.

Purpose of this document

Provides:

Describe the approach to overall delivery of a typical Java project for a subset of the organization(wave) that is embarking upon larger transformation program.

Provide guidelines/guardrails on activities to be performed as part of overall Delivery.

- Possible approach to various phase of implementation
- Guidelines and roadmaps to be followed for a selected SDLC cycle.
- Phase planning, tool sets to be used.
- Methodology and checklists

Not provide:

- Does not explore the benefits and rationale for adopting a particular way of delivery cycle.
- Does not provide a holistic compendium/how to guide on the Delivery process.

This playbook is a living document that will evolve as the team continues to embark of new projects and learn through its implementation.

© 2024 Capgemini - All rights reserved

Printed copies are current on date of printing only - 06/05/2024. Always refer to the electronic version for the current release.

Software Delivery Journey

This documentation provides helpful guides and documentation for engineers to work on the tasks involved in software delivery. The information is split into following categories:

- Technical guides
- Reference documentation for tooling
- Technical How Tos
- Playbooks to explain end-to-end.

Planning:

- The development journey starts with requirements from business, which is then translated to workable technical work by Product Owners and Architects
- The tooling used for planning, tracking and Managing is JIRA, Rally ..etc.

Getting Started:

1. Download Spring or Quarkus from the official websites
    a. https://start.spring.io/
    b. https://code.quarkus.io/

2. Establish the coding conventions in the team
3. Align the architecture of the new service with an architecture pattern

Delivery Methodology - Selection of Project Life Cycle:

Linear Development Lifecycle (LDM)

Overview:

The Linear Development Method is the Capgemini recommended lifecycle method for development engagements executed in waterfall/v-model. LDM is designed to support application development where there are clearly stated business requirements.

Linear Development Method is structured as a progressive, deliverables-driven life cycle that is divided into successive phases. In this method, both development team and test teams work in parallel and final product is produced at the end.

Lifecycle selection criteria:

- Requirements are well defined, clearly documented and fixed.
- Product definition is stable.
- Technology is not dynamic and is well understood by the project team.
- There are no ambiguous or undefined requirements.
- The project is short.

© 2024 Capgemini - All rights reserved

Printed copies are current on date of printing only - 06/05/2024. Always refer to the electronic version for the current release.

LDM Lifecycle phases:

DELIVERY LIFECYCLE WISE Deliverables – Guidelines, Templates, & Checklists

Phases	Entry Criteria	Tasks	Verification	Exit Criteria
Business Requirements	Proposal MSA, SOW or LOI	• Capture And Analyze Business Requirements • Provide High level estimates for budgeting. • Develop Test Strategy • Create Acceptance Test Plan and Test Cases	Review and sign off of Business Requirements, test strategy	• Baselined Business Requirement Document • Baselined Test Strategy • Traceability Matrix updated with Business Requirements and User Acceptance Testcases • User Acceptance Test Plan and Test Cases
Functional/Non-Functional Requirements	Business Requirements NFRs Traceability Matrix Test Strategy	• Capture And Analyze Functional Requirements • Create System Test Plan • Create Performance Test Plan • Create System Test Cases • Create Test Data	Review and sign off of Functional/Non-Functional Requirements, test plan and test cases	• Baselined Requirement Specification Document • Baseline NFR specs with defined Acceptance criteria • Baselined System Test Plan and Test Cases • Traceability Matrix updated with Functional requirements and System Testcases
Solution Architecture	• Requirement Documents • Traceability Matrix	• Understand Architecture Requirements • Create Architecture document	Review and sign off of Architecture document	• Baselined Architecture Document • Traceability Matrix updated with Architecture details
High Level Design	• Requirement Documents • Architecture Document • Test strategy, system test plan and testcases	• Create High Level Design Document • Create Integration Test Plan • Create Integration Test Cases	Review and sign off of HLD document, IT test plan and test cases	• Baselined High Level Design Document • Baselined Integration Test Plan and Test Cases • Traceability Matrix updated with High Level Design and Integration Testcases

© 2024 Capgemini - All rights reserved

Printed copies are current on date of printing only - 06/05/2024. Always refer to the electronic version for the current release.

Low Level Design	• High Level Design Document • Updated Traceability Matrix • Architecture Document • Requirement Specification Document	• Create Low Level Design • Create Unit Test Cases	Review and sign off of LLD document and UT cases	• Baselined LLD Document • Baselined Unit Test Cases • Traceability Matrix updated with Low Level Design details and the Unit Testcases
Development/Unit Testing	• High Level Design Document • Low Level Design Document • Architecture Document • Requirement Specification Document	• Perform Coding by following test driven development and coding standards • Establish CI/CD pipeline • Run code quality tools in the IDE • Fix the code quality issues, Check-in the code daily into the config repository • Perform Code Review by following pull approach and fix code review comments • Perform Unit Testing and fix the defects	• Code quality checks using tools • Manual Code reviews and signoff • Code coverage checks using tools	• Baselined Source Code • Traceability Matrix updated with Source Code details • Code quality targets passed • Manual Code reviews and signoff • Code coverage targets passed
Integration Testing	• Integration Testcases • Integrated Source Code	• Perform Integration Testing and fix the defects	Quality gate checks using tools integrated into CI/CD pipeline	• Quality gates passed
System Testing	• System Testcases • Requirement Specifications • Build • System Test Environment	• Set Up System Test Environment • Perform System Testing and fix the defects	• Test coverage • Test execution results • Release/product delivery audits	• System Test Results Passed • User documentation • Release notes
User Acceptance Testing	• Acceptance Test Plan/Cases • Business Requirement Document • UAT Test Environment	• Provide Support for Acceptance test by customer and fix the defects	UAT coverage Test execution results	• Acceptance Test Results Passed • UAT Signoff from Customer and Go-live approval
Implementation	• Release Package • Implementation Plan • Release Notes • UAT Signoff from Customer	• Prepare For Implementation • Deploy The code in production • Conduct User Training • Cut over and go live	Review and Sign off on successful deployment	• Successful code deployment in Production
Warranty and Handover	• Training Materials • All deliverables from above steps	• Provide Warranty Support • Handover To Support	Review and sign off of Handover document	Handover to support team successfully completed

Refer -
https://capgemini.sharepoint.com/sites/QAIMS/SEPG/QMS_Wrapper/SitePages/LDM%20Wrapper.aspx

© 2024 Capgemini - All rights reserved

Printed copies are current on date of printing only - 06/05/2024. Always refer to the electronic version for the current release.

Agile Delivery Lifecycle – Cap Gemini Agile Framework (CAF)

Agile planning covers three different levels—the release, the iteration, and the current day—each plan can be made with a different level of precision thus enabling a wholistic planning view

Refer - https://capgemini.sharepoint.com/sites/SSO10/SitePages/Agile-CAF-Lifecycle.aspx

Development:

Along with industry standard tooling, there are some recommended platforms that support the development and CICD journey for engineers.

Estimation and Sizing:

Estimation is used as an input to :

- Pricing
- Planning/Scheduling
- Staffing

Estimation includes 3rd parties, Integration, UAT and other types of project work that may not be delivered in an Agile way

This section recommends standard Estimation models which are recommended to be used by FS SBU projects

Life Cyle Method:

1. **LDM or Application Maintenance**

Sizing:

1. Sizing an application is a crucial step for any estimation & measurement activity in the industrialization process:
   - Estimating effort for building or maintaining applications
   - Project progress tracking
   - Productivity measurement
2. Types of sizing
   - Measure/estimate the physical size of the solution (e.g. counting/estimating # of lines of code)
   - Determine the size of the requirements (functional sizing)
3. **Lines of Code (LOC):** counts the total number of lines of source code in a project. The units of LOC are

© 2024 Capgemini - All rights reserved

Printed copies are current on date of printing only - 06/05/2024. Always refer to the electronic version for the current release.

- KLOC- Thousand lines of code
- NLOC- Non-comment lines of code
- KDSI- Thousands of delivered source instruction

The size is estimated by comparing it with the existing systems of the same kind. The experts use it to predict the required size of various components of software and then add them to get the total size.

Advantages:

- Universally accepted and is used in many models like COCOMO
- Estimation is closer to the developer's perspective
- Simple to use.

Disadvantages:

- Different programming languages contain a different number of lines
- No proper industry standard exists for this technique
- When platforms and languages are different, LOC cannot be used to normalize.

4. **Functional Sizing:** In this method, the number and type of functions supported by the software are utilized to find FPC(function point count). The steps in function point analysis are:
   - Count the number of functions of each proposed type: Find the number of functions belonging to the following types:
     i. **External Inputs**: Functions related to data entering the system.
     ii. **External outputs**: Functions related to data exiting the system.
     iii. **External Inquiries**: They lead to data retrieval from the system but don't change the system.
     iv. **Internal Files**: Logical files maintained within the system. Log files are not included here.
     v. **External interface Files**: These are logical files for other applications which are used by our system.
   - Compute the Unadjusted Function Points(UFP): Categorize each of the five function types like simple, average, or complex based on their complexity. Multiply the count of each function type with its weighting factor and find the weighted sum
   - Find the Total Degree of Influence(TDI):
     i. Use the '14 general characteristics' of a system to find the degree of influence of each of them. The sum of all 14 degrees of influence will give the TDI. The range of TDI is 0 to 70.
        - The 14 general characteristics are: Data Communications, Distributed Data Processing, Performance, Heavily Used Configuration, Transaction Rate, On-Line Data Entry, End-user Efficiency, Online Update, Complex Processing Reusability, Installation Ease, Operational Ease, Multiple Sites and Facilitate Change
        - Each of the above characteristics is evaluated on a scale of 0-5.
   - Compute Value Adjustment Factor(VAF): Use the following formula to calculate VAF i.e. VAF = (TDI * 0.01) + 0.65

© 2024 Capgemini - All rights reserved

Printed copies are current on date of printing only - 06/05/2024. Always refer to the electronic version for the current release.

- Find the Function Point Count(FPC): Use the following formula to calculate FPC i.e. FPC = UFP * VAF

2. **Agile:**
   - CapGemini e-GREAT estimating models
   - User Story points, T'shirt sizing & Other Relative Sizing-based approaches

Relative Sizing Technique:

i. Estimation of Size can be measured using a neutral unit called Story points
ii. User Stories(or Epics, Features or PBI(product backlog items) gives a brief description of a piece of functionality which add value and should be able to be completed in a single iteration
iii. The Fibonacci series and T'shirt sizing are mostly used
iv. Using Fibonacci sequence (1,2,3,5,8, 13) or T'Shirt sizes – XS, S, M, L, XL, XXL
v. Story points are sized by team who be carrying out the actual work

Advantages of Relative Sizing Technique:

- More realistic estimates
- Reduce planning time
- Team is committed to the estimates
- Takes into account:
  - Effort
  - Complexity
  - Knowledge
  - Risk & Uncertainty

Continuous Integration

Software development

- Code development:
  - Code development would be done via IDE IntelliJ, Eclipse.
  - It can be test driven development to resolve some coding bugs.

Build process – Used for building Java projects.

- Gradle :
  - https://spring.io.guides/gs/gradle/
  - IntelliJ set up for Gradle https://www.jetbrains/help/idea/gradle.html
  - Sample gradle configuration
- Maven:
  - https://sping.io.guide/gs/maven/
  - IntelliJ set up for Maven https://www.jetbrains.com/help/idea/maven.html
  - Sample pom configuration

© 2024 Capgemini - All rights reserved

Printed copies are current on date of printing only - 06/05/2024. Always refer to the electronic version for the current release.

https://maven.apache.org/guides/introduction/introduction-to-the-pom.html

- Git and GitHub- These guides cover new repo creation, branch permissions and creation and other features.
- Jenkins
  - Jenkins Pipeline – Jenkins pipeline is a suite of plugins that supports implementing and integrating continuous delivery pipelines into Jenkins.

Recommended Build tools

Maven (POM):

- A Project Object Model or POM is the fundamental unit of work in Maven. It is an XML file that contains information about the project and configuration details used by Maven to build the project
- Minimal POM: The minimum requirement for a POM are the following:
  - project root
  - modelVersion - should be set to 4.0.0
  - groupId - the id of the project's group.
  - artifactId - the id of the artifact (project)
  - version - the version of the artifact under the specified group
- Sample pom

```
<project>
<modelVersion>4.0.0</modelVersion>

<groupId>com.mycompany.app</groupId>
<artifactId>my-app</artifactId>
<version>1</version>
</project>
```

Refer - https://maven.apache.org/guides/introduction/introduction-to-the-pom.html

Gradle:

- Like we have pom.xml with Maven, we have build.gradle with Gradle.
- A Gradle project will look something like:

```
project
├── gradle
│   ├── libs.versions.toml
│   └── wrapper
│       ├── gradle-wrapper.jar
│       └── gradle-wrapper.properties
├── gradlew
├── gradlew.bat
├── settings.gradle(.kts)
├── subproject-a
│   ├── build.gradle(.kts)
│   └── src
└── subproject-b
```


Printed copies are current on date of printing only - 06/05/2024. Always refer to the electronic version for the current release.

```
├── build.gradle(.kts)
└── src
```

Refer - https://docs.gradle.org/current/userguide/getting_started_eng.html

Additional Development Tooling:

Include tools that contribute in static analysis and code security scanning aspects:

- Nexus – Distribute packaged and containerized apps like Docker, Helm, Yum and APT
- SonarQube – Continuous inspection of code code quality to detect bugs and code smells.
- NexusIQ – Scan software libraries(3rd party, open source and custom) to create a detailed inventory of the components that comprise your applications. It constantly monitors inventoried components for new risks

Coding Conventions:

Coding best practices in Java are often ignored by developers due to lack of knowledge or time to implement them. The best practices in this article are intended for both junior and senior Java developers. These best practices will help improve performance, reduce security risks, converse heap memory and simplify troubleshooting

Use Correct Naming Conventions

A well formed name not only aids in reading the code but also conveys a great deal about the code's intent.
Here is an example

```
public class User {
        private String UserName;
        public void SetUserName( String userName) {
                this.userName = username;
        }
}
```

As you can see from the example:-

- Class or interface names should be nouns because they represent real-world objects
- Method names should be verbs because they are always a part of the class and represent an action in general
- Variable names should be short and meaningful and follow Camel case format

Logging

- Avoid excessive logging
- Log the info that may be useful in troubleshooting
- Select the log levels carefully, as we may need to enable log levels in production for troubleshooting

Defensive Coding and resiliency

Always check for NULL value.

- Failure to check for null value will cause the system to throw NullPointerException

Catch Exception and Log:

- If you are expecting exception thrown by the method call, always put try-catch block especially when the call occurred within a loop
- Exception handling should be specific and error message should be meaningful

Printed copies are current on date of printing only - 06/05/2024. Always refer to the electronic version for the current release.

In an effort to continuously push developers toward "a standard way to develop", we recommend to refer –
https://google.github.io/styleguide/javaguide.html
https://www.oracle.com/technetwork/java/codeconventions-150003.pdf
https://devonfw.com/docs/java/current/cross_cutting/coding_conventions.html

In this the below topics are covered in the coding standards

- File Names
- File Organization
- Indentation
- Comments
- Declarations
- Statements
- White Space
- Naming Conventions for class, method, variables used by the Java program

Naming Conventions**:**

We recommend using the following naming conventions for services, object, functions, programs, tables, files etc as they are means by which we identify things
Refer- https://www.oracle.com/java/technologies/javase/codeconventions-namingconventions.html
The naming standards are focussed on following areas:

- XML namespaces
- Java packages
- EJB Object Names
- JNDI names
- CORBA Object Names
- Constants

Java Application Logging Framework

Logging equips the developer with detailed context for application failures. To align with the global /
Refer: https://logging.apache.org/log4j/2.x/manual/index.html

Junit Unit Testing Framework

Unit tests should be done to ensure the functionality and behaviour of individual software component prior to any system level testing. Junit, an open source, light-weighted framework for Java application unit testing. Refer – https://junit.org

- It allows developer to write test code faster while increasing quality
- It is elegantly simple and inexpensive.
- It checks their own results and provides immediate feedback
- It can be composed into a hierarchy of tests suites

Exception Handling

To preserve the readability, maintainability and extensibility of our code, it's a good idea to create our own subtree of application-specific exceptions that extend the base exception class. Hence it is always recommended to have your own Application Error hierarchy
Refer:

- https://www.javaguides.net/2018/06/guide-to-java-exception-handling-best-practices.html

© 2024 Capgemini - All rights reserved

Printed copies are current on date of printing only - 06/05/2024. Always refer to the electronic version for the current release.

Back-End Java

Best Practices:

Validate Inputs and handle errors:

You can validate backend inputs at two levels:

1. **API gateway** — the validation of inputs at the API gateway level via policies; mainly generic validation, i.e. schema, format.
2. **Microservice** — the microservice level validations involve the checking of the existence of entities, etc. There are libraries, e.g. Joi validator, that you can leverage on for ease of input validation depending on your development stack.

Handle error using:

You can consider using two methods to handle errors to mitigate the downstream impact

1. **Circuit breaker —** prevents repeated invocation of services that are likely to fail. There are available libraries and examples for NodeJS, Springcloud, etc. which you can implement the pattern quickly.
2. **Handle exceptions with error codes** — returns a response without crashing the service with a business or HTTP error code to facilitate troubleshooting.

Unit Testing:

Source Code:

It's a good idea to keep the test classes separate from the main source code. So, they are developed, executed and maintained separately from the production code.
Also, it avoids any possibility of running test code in the production environment.
We can follow the steps of the build tools such as Maven and Gradle that look for src/main/test directory for test implementations.

Package Naming Conventions:

We should create a similar package structure in the src/main/test directory for test classes, this way improving the readability

© 2024 Capgemini - All rights reserved

Printed copies are current on date of printing only - 06/05/2024. Always refer to the electronic version for the current release.

and maintainability of the test code**.**

Test Case Naming Conventions:

The test names should be insightful, and users should understand the behaviour and expectation of the test by just glancing at the name itself.
It's often helpful to name the test cases in given_when_then to elaborate on the purpose of a unit test
We should also describe code blocks in the *Given*, *When* and *Then* format. In addition, it helps to differentiate the test into three parts: input, action and output

Expected Vs Actual:

A test case should have an assertion between expected and actual values.
To corroborate the idea of the expected vs actual values, we can look at the definition of the *assertEquals* method of JUnit's *Assert* class:
public static void assertEquals(Object expected, Object actual)
It's suggested to prefix the variable names with the actual and expected keyword to improve the readability of the test code.

Appropriate Annotations:

Always use proper assertions to verify the expected vs. actual results. We should use various methods available in the *Assert* class of JUnit or similar frameworks such as AssertJ.
For eg: Assert.assertEquals, assertNotEquals, assertNotNull, assertTrue

Specific Unit Tests:

Instead of adding multiple assertions to the same unit test, we should create separate test cases. Always write a unit test to test a single specific scenario

Testing and Validation

Functional Testing

System Testing

Functional testing aims to figure out whether given functionality works as specified. System testing aims to figure out whether the whole system fulfills the requirements given to it.

System Integration Testing

System integration testing helps to ensure that all the components of a system work together by ensuring that interactions between different systems work as expected.

Regression Testing

Software testing technique that re-runs functional and non-functional tests to ensure that a software application works as intended after any code changes, updates, revisions, improvements, or optimizations.

© 2024 Capgemini - All rights reserved

Printed copies are current on date of printing only - 06/05/2024. Always refer to the electronic version for the current release.

User Acceptance Testing

User acceptance testing (UAT), also called application testing or end-user testing, is a phase of software development in which the software is tested in the real world by its intended audience. It is typically completed after unit testing, quality assurance, system testing and integration testing.

Non-Functional Testing

Performance Testing

Performance Testing is a type of software testing that ensures software applications perform properly under their expected workload.

Security Testing

Security Testing uncovers vulnerabilities in the system and determines that the data and resources of the system protected against unauthorized access, data breaches, and other security-related issues.

Other Aspects of Testing Strategy

Test Production Scenarios:

Write tests considering real scenarios in mind, it helps to make unit tests more relatable

Mock External Services:

We should mock the external services and merely test the logic and execution of our code for varying scenarios. We can use various frameworks such as Mockito, EasyMock and JMockit for mocking external services

Avoid Code Redundancy:

Create more and more helper functions to generate the commonly used objects and mock the data or external services for similar unit tests. This enhances the readability and maintainability of the test code

Annotations:

We should leverage annotations to prepare the system for tests by creating data, arranging objects and dropping all of it after every test to keep test cases isolated from each other. Often, testing frameworks provide annotations for various purposes, for example, performing setup, executing code before and tearing down after running a test. Various annotations such as JUnit's @Before, @BeforeClass and @After and from other test frameworks such as TestNG are at our disposal.

80% Test Coverage:

As a rule of thumb, we should try to cover 80% of the code by unit tests. Additionally, we can use tools such as JaCoCo and Cobertura along with Maven or Gradle to generate code coverage reports

Coverage Tools:

© 2024 Capgemini - All rights reserved

Printed copies are current on date of printing only - 06/05/2024. Always refer to the electronic version for the current release.

1. **Jacoco**: JaCoCo stands for Java Code Coverage. It is a free code coverage library for Java, which has been created by the EclEmma team. It creates code coverage reports and integrates well with IDEs like IntelliJ IDEA, Eclipse IDE, etc. JaCoCo also integrates with CI/CD tools like Jenkins, Circle CI, etc., and project management tools like SonarQube, etc. https://www.geeksforgeeks.org/how-to-generate-code-coverage-report-with-jacoco-in-java-application/

2. **Sonar:**

   SonarQube is the leading tool for continuously inspecting the Code Quality and Security of your codebases, and guiding development teams during Code Reviews
   https://docs.sonarsource.com/sonarqube/latest/try-out-sonarqube/

Contract driven testing:

Contract testing is about making sure your consumer team and provider team have a shared understanding of what the requests and responses will be in each possible scenarios

- Which consumer-side tests specifications should I write?
  - For a given service, for a given end point, write a contract test for all methods/responses status combinations, unless your service explicitly returns 401 and 403, you don't need to test this
  - Headers when they are specific to the service
  - In return response, validate only that the information your consumer cares about was delivered in the format you are expecting

Monitoring:

This term is also referred as APM i.e. Application performance monitoring. Using application performance monitoring (APM) solutions, businesses can monitor whether their IT environment meets performance standards, identify bugs and potential issues, and provide flawless user experiences via close monitoring of IT resources. In short, end-to-end application performance monitoring works by:

- Observing whether your apps are behaving normally.
- If not, alerting to and collecting data on the source of the problem (be it the app, app dependencies, or supporting infrastructure)
- Analysing the data in the context of the impact on the business
- Adapting your application environment to fix similar problems before they impact the business.

Few of the most critical application performance monitoring metrics:

- CPU usage: At the server level, APM looks at CPU usage, memory demands, and disk read/write speeds to make sure usage doesn't affect app performance.
- Error rates: At the software level, APM tracks how often app performance degrades or fails. For example, when web requests end in an error or during memory-intensive processes like searching a database.
- Response times: Average Response Time is the metric that shows whether speed is affecting app performance.

© 2024 Capgemini - All rights reserved

Printed copies are current on date of printing only - 06/05/2024. Always refer to the electronic version for the current release.

- Number of instances: For elastic, cloud-based applications, you need to know how many server or app instances you have running at any one time. APM solutions that support autoscaling can then cost-effectively scale your app to meet user demand.
- Request rates: This metric measures how much traffic your application receives — any spikes, inactivity, or numbers of concurrent users.
- Application availability/uptime: This metric, which monitors whether your app is online and available, is the one most enterprises use to check compliance with SLAs.
- Garbage collection (GC): If your app is written Java or another programming language that uses GC, you'll be all too familiar with the problems that arise from its heavy use of memory. This is an often-hidden performance problem that's worth paying attention to.
- Commonly used APM tools: Splunk, New Relic, Elastic Search

Metrics Landscape and Operational Definitions

Category	LDM Lifecycle		Agile Delivery Lifecycle		Testing
	Development & Enhancements	**Maintenance**	**Development /Enhancement**	**Devops**	**Manual & Automation**
Quality	Code Coverage%		Code Coverage %	% Availability	Testcase Execution Coverage% (SIT)
	Testcase Execution Coverage% (SIT)	Testcase Execution Coverage% (SIT)	Testcase Execution Coverage% (SIT)	% Deployment Success Rate	Testcase Design Coverage% (SIT)
	Defect Density SIT	Defect Density SIT	Production Defects (Sev1 &Sev2)		Defect Rejection %
	Production Defects (Sev1 &Sev2)	Production Defects (Sev1 &Sev2)	Defect leakage % (UAT) (Sev1 &Sev2)		Defect Density SIT
	Code Smell	Defect leakage % (UAT) (Sev1 &Sev2)	Code Smell		Defect leakage % (UAT) (Sev1 &Sev2)
	Security **Vulnerability**		**S**ecurity **Vulnerability**		Defect leakage% (Production) (Sev1 &Sev2)
	Technical Debt	Technical Debt			
Schedule	Schedule Variance %	Response Time SLA Compliance %	Release Burndown	Deployment Time	Schedule Variance%
	SLA/KPI Met%	SLA/KPI Met%	Story Point Delivery % (Predictability of velocity)		SLA/KPI Met%
		Resolution Time SLA Compliance %	SLA/KPI Met%	SLA/KPI Met%	Requirement Coverage%
Efficiency	Productivity	Productivity (size: Lines of code)	Product Backlog Readiness%	Deployment Frequency	Testcase Authoring Productivity
	Test Case Automation % (SIT)	Regression Testing Automation %	Velocity		Testcase Execution Productivity
		Monthly Throughput	Productivity		Test Script Creation Productivity
			Test Case Automation % (SIT)		Test Script Execution Productivity

© 2024 Capgemini - All rights reserved

Printed copies are current on date of printing only - 06/05/2024. Always refer to the electronic version for the current release.

					Regression Testing Automation %
			Regression Testing Automation %		Automation Efficiency%
Cost	Effort Variance	Effort Variance	Sprint Stability %	**Time to Market**	% Capacity Utilization (At Project Level)
	% Capacity Utilization		Sprint Spillover %		Effort Variance
			Sprint Non-productive time		

Terminologies:

- **Defect Density SIT/QA:** Number of defects detected in System Integration Testing OR QA Environment / Size of work tested. [Size can be measured in Test case points OR Test cases OR Story points OR Function points OR LOC OR Volume of data migrated in MB for files OR # of rows for database etc]
- **Code Review Coverage%:** Size of code reviewed /Size of code checked-in*100
  [Size can be measured in Story points OR Function points OR LOC etc]
- **Code Coverage %:** Amount of code covered during unit testing by Static Analyzers
- **Review Effectiveness % :** Total review defects/ (Review defects +Testing Defects)*100
- **Testcase Execution Coverage**%: (Number of Testcases executed in test environment/Total no. of testcases estimated to be executed) *100
- **Code Smell:** No of Code Smells (maintainability issues in code) from Static Analyzers like SONAR, CAST etc

Check Lists, Guidelines and Templates

Linear Development Method of SDLC:

The link below will provide you with

- Method Overview
- SDLC Phases
- Artifacts
- Roles
- Guidelines
- Templates

https://capgemini.sharepoint.com/sites/QAIMS/SEPG/QMS_Wrapper/SitePages/LDM%20Wrapper.aspx

Agile Delivery Lifecycle – Cap Gemini Agile Framework (CAF)

Capgemini Agile Framework (CAF) provides guidance and good practices for efficient execution of our agile engagements. It is lightweight, based on Scrum, extended with eXtreme Programming (XP) practices and complemented by Capgemini's best practices. It can be tailored based on the client needs. It covers activities to be performed for delivering an Agile engagement. The Link below will provide you with:

- Agile terminologies

Printed copies are current on date of printing only - 06/05/2024. Always refer to the electronic version for the current release.

- Agile Ceremonies
- Artifacts
- Agile Roles
- Guidelines

https://deliver2.capgemini.com/components/CAF-V4.1/#Capgemini%20Agile%20Framework/guidances/supportingmaterials/CAF%20Overview_B368AD2D.html

Gen AI tools:

Phase wise tools overview:

Tools Samples:

Project management	Code Development	Code Quality	Code Build/Packaging	Unit Testing	Functional Testing	Performance testing	Deployment/Monitoring
GitFluence	GITHUB COPILOT	TABNINE GITHUB	SNYK	TABNINE DIFFBLUE	TESTIM	NEOTYS	SYNK
Sidekick	Amazon CodeWhisperer	COPILOT MINTLIFY	GITHUB COPILOT	INTELLITEST	Functionize		KubeFlow
Kite	Codeium	AskCodi	Deep Code	GITHUB COPILOT	DiffBlue		DataDog
		CodiumAI	Codiga				

** Most of the tools are available at a cost. They either have a fixed or subscription costs associated with them

© 2024 Capgemini - All rights reserved

Printed copies are current on date of printing only - 06/05/2024. Always refer to the electronic version for the current release.

Case Studies:

Reducing Technical Debts with AI:

What is a Technical Debt - Technical debt refers to the costs incurred from taking shortcuts in the development process to get a product or service out to market quickly. Suboptimal code and system design can slow down development, increase the occurrence of bugs, and raise maintenance costs.

Cause of Technical Debt:

- Making suboptimal modifications to the system without a full understanding of its architecture
- Skipping tests, leading to undetected bugs and defects that are harder to fix later
- Copying and pasting code instead of creating reusable modules or libraries
- Taking shortcuts causing introduction of vulnerabilities and sub-optimal security

Role of AI in reducing technical debt:

AI offers speed, **consistency, scalability, and** continuous monitoring advantages in software development. By harnessing the power of AI, you can automate various aspects of the development cycle. Here are a few examples of how AI can help minimize technical debt:

1. **Automated Code reviews**: AI can automate code reviews to help identify and fix technical debt early on. It can also monitor and identify problem areas in your codebase, such as duplicate code**.**
2. **Refactoring codebase:** When you refactor code, you're essentially improving your code without changing its functionality. AI can help you make small changes to your code's structure, readability, and maintainability.
3. **Prioritized Workflows:** AI can predict which areas of the codebase are most likely to develop technical debt, which can help you prioritize development stages effectively.
4. **Automated Testing:** AI can automate testing to help ensure new code does not introduce new technical debt.
5. **Task Prioritization**: AI can prioritize tasks related to technical debt, which helps developers address critical issues quickly
6. **Test case generation:** AI-enabled test case generation automates SQL output checks against expected results in your test case generation. Based on the results, you can either modify the test case or keep it as-is, which can help lower technical debt.

Harnessing the power of AI Coding Assistant:

AI Coding Assistant, IntelliJ plugin that harnesses the power of cutting-edge Language Model APIs to revolutionize the way you code!

Key Features:

- **Extensive API Support**: AI Coding Assistant seamlessly integrates with a wide range of leading LLM APIs, including OpenAI, AWS Bedrock, Anthropic, Google, Groq, Perplexity AI, and ModelsLab. 🔑 Get your API keys and unlock a world of possibilities!

© 2024 Capgemini - All rights reserved

Printed copies are current on date of printing only - 06/05/2024. Always refer to the electronic version for the current release.

- **No Membership Fees**: While API access is typically paid, AI Coding Assistant charges no base fee. You only pay for what you use, giving you complete control over your expenses
- **Free and Open Source**: AI Coding Assistant is a free, open-source tool designed with a focus on utility. No hype, no gimmicks – just a powerful, user-friendly plugin that enhances your coding experience
- **Intuitive Toolbar UI**: Easily configure temperature/model settings and monitor your current token count with AI Coding Assistant's sleek, intuitive toolbar UI
- **API Request Management**: Intercept, edit, and log API requests with ease, giving you granular control over your plugin's behavior.

Appendix A:

Java-Microservices

Architecture-Context

- https://microservices.io/patterns/microservices.html

Design Patterns

- https://microservices.io/patterns/index.html

Microservices security patterns

- https://tsh.io/blog/microservices-security-patterns/

Dev Ops

Git Hub Commands

- https://git-scm.com/docs/gittutorial

CI/CD Pipeline Set up for Java Apps

- https://buddy.works/docs/quickstart/java

Jenkins CI/CD pipeline setup

https://www.jenkins.io/doc/tutorials/#pipeline

Build a Java app with Maven

- https://www.jenkins.io/doc/tutorials/build-a-java-app-with-maven/

Java

Java 8 Features

- https://www.digitalocean.com/community/tutorials/java-8-features-with-examples

Java Design Solid Principles

- https://www.jrebel.com/blog/solid-principles-in-java

© 2024 Capgemini - All rights reserved

Printed copies are current on date of printing only - 06/05/2024. Always refer to the electronic version for the current release.

Java OOPS Design Patterns

- https://www.oodesign.com/

Code cleans up Tools for Java Apps

- https://snyk.io/learn/code-review/java-tools/

Mockito, Junit Test cases for Spring boot

- https://howtodoinjava.com/spring-boot2/testing/spring-boot-mockito-junit-example/

Use API (Application Program Interface) Gateway

Usually, microservices applications contain many services that are accessible by different clients and systems. It exposes them to many security risks since it's hard to monitor every single service.

Microservices architecture simplified

That's why you can deploy API Gateways that handle, monitor and scan all incoming traffic before it is passed to the target service. It will act as a point of entry for all requests

Secure your communication!

Secure communication in microservices is a very important topic. Various testing methods are used such as dynamic application security testing, when the tester tries to enter the app through the frontend. Still, there are some more basic

Printed copies are current on date of printing only - 06/05/2024. Always refer to the electronic version for the current release.

things you can do to implement security in your communication.
By default, you should enforce the use of HTTPS (hypertext transfer protocol secure) in your entire application. When sending sensitive information, such as client credentials, passwords, keys or secrets, encrypt it as soon as possible, and decrypt it as late as possible. Never send this kind of information in plain text.
There are several ways how you can secure communication between services. However, these two are the most common: on: Mutual Transport Layer Security (mTLS) and Json Web Token (JWT).

- mTLS – each microservice has a public/private key pair issued by a trusted certificate authority. The client then uses the key-pair to perform basic authentication through the mTLS.

mTLS communication

JWT: each microservice has its own unique JWT token. Clients must know this token to access it (access tokens).

Printed copies are current on date of printing only - 06/05/2024. Always refer to the electronic version for the current release.

JWT Microservices security pattern for secure communication

Printed copies are current on date of printing only - 06/05/2024. Always refer to the electronic version for the current release.