CSA10 - Software Engineering

LAB MANUAL

1. Create a Kanban Board to Visualize the Tasks.

- **Create Columns for To Do, In-Progress and Done.**
- **Add At least 5 Sample Tasks**
- **Move the Tasks across the Columns to Simulate the Workflow**

AIM: To create a Kanban board to visualize tasks and simulate the workflow, follow these step-by-step instructions:

DESCRIPTION / ALGORITHM:

Step 1: Set Up the Kanban Board

Start by setting up three columns that represent the stages of your workflow:

1. **To Do** – This column will contain tasks that need to be done.
2. **In-Progress** – This column will contain tasks that are currently being worked on.
3. **Done** – This column will contain tasks that have been completed.

You can use a physical whiteboard with sticky notes, or a digital Kanban board tool like Trello, Jira, or Asana.

Step 2: Add Sample Tasks to the "To Do" Column

Next, create at least five sample tasks to add to the **To Do** column. These can represent any kind of work. Here are five sample tasks:

1. **Task 1**: Research computer vision algorithms
2. **Task 2**: Implement image processing pipeline
3. **Task 3**: Write unit tests for model accuracy
4. **Task 4**: Prepare course presentation slides
5. **Task 5**: Review student project submissions

Now, place these tasks as sticky notes (physical) or cards (digital) in the **To Do** column.

Step 3: Move Tasks to the "In-Progress" Column

Next, pick one or more tasks from the **To Do** column and move them to the **In-Progress** column to indicate they are being worked on. For example:

- **Task 1**: Research computer vision algorithms
- **Task 2**: Implement image processing pipeline

Step 4: Update the Tasks in the "In-Progress" Column

As work progresses, continue to update the tasks in the **In-Progress** column. You can add details like the current stage of the task or any challenges you're facing. For example:

- **Task 1**: Research computer vision algorithms (50% complete)
- **Task 2**: Implement image processing pipeline (Waiting for dataset)

Step 5: Move Tasks to the "Done" Column

Once a task is completed, move it from the **In-Progress** column to the **Done** column. For example, after completing **Task 1** and **Task 2**, you would move them to **Done**.

- **Task 1**: Research computer vision algorithms (Completed)
- **Task 2**: Implement image processing pipeline (Completed)

Step 6: Keep Track and Repeat the Process

As you finish tasks and new ones come in, continue to move tasks across the columns from **To Do** to **In-Progress** to **Done**. This visualizes your workflow and helps track progress.

Example Workflow (Simulated):

- **To Do**:
    - Task 3: Write unit tests for model accuracy
    - Task 4: Prepare course presentation slides
    - Task 5: Review student project submissions
- **In-Progress**:
    - Task 1: Research computer vision algorithms (50% complete)
    - Task 2: Implement image processing pipeline (Waiting for dataset)
- **Done**:
    - Task 1: Research computer vision algorithms (Completed)
    - Task 2: Implement image processing pipeline (Completed)

Final Step: Monitor and Optimize

Continue to monitor the board regularly. If tasks are stuck in the "In-Progress" column for too long, consider investigating the bottleneck. Use the Kanban board to improve workflow and ensure tasks are moving efficiently towards completion.

---

2. Sketch a Simple Prototype of a Bus Ticket Booking System using Figma Tool

AIM:

To create a Prototype for a Bus Ticket Booking System using Figma

DESCRIPTION / ALGORITHM

Step 1: Set Up a Figma Account and Project

1. **Sign in to Figma**: If you haven't already, go to Figma and create an account or log in.
2. **Create a New File**: On the dashboard, click the **New File** button to create a new project for your bus ticket booking system.

Step 2: Define the Basic Structure of the App

You'll start by defining the core layout and essential screens for the system.

1. **Create Frames for Screens**:
    - On the left sidebar, select the **Frame Tool (F)**.
    - Create frames for the different screens of your app. For a basic prototype, you can have the following frames:
        - **Home Screen** (starting screen)
        - **Search Page** (where users enter details for their journey)
        - **Booking Details Page** (where users can review and confirm their booking)
        - **Payment Page** (for payment options)

Step 3: Design the Home Screen

1. **Add a Background**:
    - Select the **Frame Tool (F)** and draw a rectangle that will serve as your screen size. Use standard mobile dimensions, like 375x812 px (iPhone X).
    - Add a simple background color or image by selecting the frame and adjusting the **Fill** color in the right properties panel.
2. **Header**:

- Use the **Text Tool (T)** to add the header, like “Bus Ticket Booking”.
- Style the text to make it bold and larger for visibility.

3. **Navigation Elements**:
   - Use the **Rectangle Tool (R)** to create buttons like “Search Buses”, “My Bookings”, etc. These can be placed below the header.
   - Add text on each button (e.g., “Search Buses”).
   - Style the buttons with rounded corners for a modern look.
4. **Search Box**:
   - Below the buttons, design a simple search bar for users to enter their journey details (e.g., from, to, date of travel).
   - Use the **Rectangle Tool (R)** to create a search bar with a text input area. You can add icons (e.g., a calendar icon for selecting the date).
5. **Add Icons and Visual Elements**:
   - You can use Figma’s built-in **Icon** library or import icons from resources like **Material Icons**.
   - Add simple icons like a bus for transportation or a calendar for selecting travel dates.

Step 4: Design the Search Page

1. **Journey Details Form**:
   - Create input fields using **Rectangle Tool (R)** for entering departure location, destination, and date of travel.
   - Label each input field with appropriate text such as “From”, “To”, and “Date”.
2. **Search Button**:
   - Add a **Search** button at the bottom of the page to submit the journey details.
   - Style it with rounded corners and use the **Text Tool (T)** to label it “Find Buses”.
3. **Bus Options**:
   - Below the search bar, design a list of bus options that will appear after the user enters details.
   - Use **Rectangle Tool (R)** for card-like elements showing bus details (e.g., bus name, price, time).

- You can create multiple options as cards with different colors to distinguish them.

Step 5: Design the Booking Details Page

1. **Booking Summary**:
   - Use **Rectangle Tool (R)** to create a section showing the summary of the journey: Departure, Arrival, Date, and Time.
2. **Passenger Details**:
   - Create text fields for user to enter personal information (name, age, contact).
   - Use input fields and label them appropriately.
3. **Booking Confirmation Button**:
   - Add a "Confirm Booking" button below the details section.

Step 6: Design the Payment Page

1. **Payment Options**:
   - Add a section for payment methods (e.g., credit card, PayPal).
   - Use checkboxes or radio buttons for the user to select the payment method.
   - Add a simple **Text Input** for credit card number or payment information.
2. **Total Amount**:
   - At the bottom, display the total ticket cost using the **Text Tool (T)**.
3. **Submit Button**:
   - Add a **Submit Payment** button to confirm the payment and finalize the booking.

Step 7: Create Interactions Between Screens

1. **Prototype Mode**:
   - Once you've designed the screens, switch to **Prototype Mode** by clicking the **Prototype** tab in the top-right corner of the Figma interface.
2. **Link Screens**:
   - Click on elements like buttons (e.g., "Search Buses") and drag the interaction arrow to the target screen (e.g., "Search Page").

- Set transitions like **On Click**, **Navigate To**, and animation effects (e.g., **Smart Animate** for smooth transitions).

3. **Flow Simulation**:
   - Click **Present** in the top-right corner to test the interactive flow of your prototype. Check the transitions between screens and ensure everything is linked correctly.

Step 8: Polish and Refine

1. **Refine UI Elements**:
   - Adjust spacing, align text properly, and make sure the visual hierarchy is clear.
2. **Add Visual Design**:
   - Apply consistent colors, fonts, and branding.
   - Use Figma's **Styles** feature to maintain consistency for fonts, colors, and effects across your design.

Step 9: Share the Prototype

1. **Sharing the Prototype**:
   - Once you are satisfied with the design and interactions, click on the **Share** button in the top-right corner of Figma.
   - You can share the prototype by generating a link or inviting others to view/edit your design.

Ex.No: 3

Create a Scrum Project in Jira.

- **Add a backlog with at least 5 items (e.g., "Create user registration page", "Develop API for login").**
- **Prioritize the backlog and create a 1-week sprint.**
- **Move backlog items into the sprint and start the sprint.**
- **Finally show the Screenshot of the sprint board at the start and end of the sprint.**

AIM: Demonstrate the Backlogs using Scrum on Jira also show the sprint board to show the start and end process.

Description / Algorithm:

To create a Scrum project in **Jira**, manage your backlog, and simulate a 1-week sprint, follow these detailed steps. Please note that these instructions assume you are using Jira Software with Scrum project management enabled.

Step 1: Create a Scrum Project in Jira

1. **Log into Jira**: Go to your Jira instance and log in.
2. **Create a New Project**:
    - From the **Jira Dashboard**, click on **Create Project**.
    - Choose **Scrum Software Development** from the list of templates.
    - Select the project name (e.g., "Bus Ticket Booking System") and project key.
    - Choose **Create** to set up the project.

Step 2: Set Up the Backlog with 5 Items

1. **Navigate to the Backlog**:
    - Once your Scrum project is created, go to the project's **Backlog** view.
    - On the left sidebar, you'll see a section labeled **Backlog**.
2. **Create Issues (Backlog Items)**:
    - Click on the **Create Issue** button to add tasks to your backlog.
    - Add at least five items. Each item should represent a task that can be completed within the sprint. Examples include:
        - **Create user registration page**
        - **Develop API for login**
        - **Design homepage layout**
        - **Implement bus search functionality**
        - **Create payment integration page**
    - For each item, select **Task** as the issue type, and provide a brief description of what needs to be done.
    - Click **Create** after adding each issue.

Step 3: Prioritize the Backlog

1. **Drag and Drop to Prioritize**:
    - In the **Backlog** view, you'll see the list of backlog items.
    - To prioritize them, click and drag the items to reorder them based on importance or priority. For example, the task **"Create user registration page"** might be more important and should be placed at the top of the list.
    - You can also click on each item to edit its details (e.g., adding labels or assigning it to a user).

Step 4: Create a Sprint for 1 Week

1. **Add a Sprint**:
    - In the **Backlog** view, find the section labeled **Sprints**. If it's not visible, click on the **Backlog** menu.
    - Click **Create Sprint** to create a new sprint.
    - Give the sprint a name, such as "Sprint 1 - Week 1".
    - Set the **Start Date** and **End Date**. For a 1-week sprint, choose dates like **Start Date: Today**, **End Date: 7 days later**.
2. **Move Items into the Sprint**:
    - Drag and drop the tasks from the backlog into the **Sprint 1** section to add them to the sprint.
    - Ensure the most critical tasks are included in the sprint.

Step 5: Start the Sprint

1. **Start the Sprint**:
    - Once you've added all the items to the sprint, click on the **Start Sprint** button in the **Sprint 1** section.
    - A pop-up window will appear asking you to confirm the start and end dates, and sprint goals.
    - Click **Start** to begin the sprint.

Step 6: Show the Sprint Board (Before and After the Sprint)

Before the Sprint Starts:

1. **Navigate to the Scrum Board**:
    - Click on **Active Sprints** in the left sidebar of the project to view the **Sprint Board**.

- This board shows columns such as **To Do**, **In Progress**, and **Done**.
- The tasks you added to the sprint will appear in the **To Do** column.
- You can see the tasks listed under the **Sprint 1** header.
- **Screenshot Before Sprint**: Take a screenshot of this view showing the tasks in the **To Do** column.

During the Sprint:

1. **Work on Tasks**:
   - As team members work on tasks, they can move them from **To Do** to **In Progress**, and eventually to **Done**.
   - The Scrum board will update in real-time as tasks progress through the workflow.

After the Sprint (End of Sprint):

1. **Complete the Sprint**:
   - Once all tasks are completed, or the sprint ends, click **Complete Sprint** at the top of the Scrum board.
   - Jira will prompt you to review the completed issues and move any remaining incomplete issues to the next sprint (if needed).
2. **Screenshot After Sprint**:
   - Take a screenshot of the sprint board after the sprint ends. You should see tasks moved to the **Done** column, and the overall progress of the sprint.

4. Link Jira tasks with Confluence to streamline task tracking and progress monitoring for the Library Management System development.

- **Create a new page in Confluence titled "Library Management System Project Overview."**
- **Embed at least 5 Jira issues related to the development of the Library Management System (e.g., tasks from the sprint like "Develop book search functionality," "Create user login page," etc.).**
- **Use the Jira macro to display issues with status (e.g., "To Do," "In Progress," "Done").**
- **Add a progress bar in the Confluence page to visually track the completion of each embedded Jira task (e.g., percentage of tasks completed in the sprint).**

- **Submit a screenshot of the Confluence page showing the embedded Jira tasks and the progress bar.**

AIM: Demonstrate Library Management System to connect Jira tasks with Confluence to streamline task tracking and progress monitoring

Description / Algorithm:

Step 1: Create a New Page in Confluence

1. **Log in to Confluence**: Go to your Confluence instance and log in with your credentials.
2. **Navigate to the Space**: Navigate to the space where you want to create the page (e.g., your project space for the Library Management System).
3. **Create a New Page**:
    - On the Confluence dashboard, click on the **Create** button at the top.
    - A new blank page will open.
    - Title the page as **"Library Management System Project Overview"**.
4. **Write an Introduction**: Provide a brief description of the page to explain its purpose. For example:
    - "This page provides an overview of the development progress of the Library Management System. It links directly to Jira tasks to track and monitor the project's progress."

Step 2: Embed Jira Issues into Confluence

1. **Edit the Page**: Click on the "Edit" button on the page you just created to start editing the content.
2. **Insert the Jira Macro**:
    - In the page editor, click on the **"+" (Insert More Content)** button in the toolbar.
    - Select **"Jira"** from the drop-down menu.
3. **Search for Jira Issues**:
    - A pop-up window will appear where you can search for Jira issues.
    - Enter relevant keywords like "Library Management System" or the specific tasks such as "Develop book search functionality", "Create user login page", etc.
    - You can also paste Jira issue keys (e.g., LMS-101, LMS-102) directly into the search bar.
4. **Select and Embed Issues**:

- Select at least **5 Jira issues** that are relevant to the development of the Library Management System.
- Click **Insert** to add the selected issues to the Confluence page.

5. **Display the Issues with Status**:
   - The embedded Jira issues will display with details such as the task name, issue key, assignee, and status (e.g., "To Do", "In Progress", "Done").
   - Ensure that the Jira macro is set to display **status** as part of the issue display.

Step 3: Add a Progress Bar to Track Completion

1. **Determine Completion Percentage**:
   - Estimate the percentage of tasks completed in the current sprint based on the embedded Jira issues.
   - For example, if you have 5 tasks and 3 are "Done" and 2 are "In Progress," you can calculate the completion percentage as follows:

- For example, if you have 5 tasks and 3 are "Done" and 2 are "In Progress," you can calculate the completion percentage as follows:

$$\text{Completion Percentage} = \frac{\text{Number of "Done" tasks}}{\text{Total number of tasks}} \times 100$$

In this case:

$$\frac{3}{5} \times 100 = 60\%$$

2. **Insert Progress Bar**:
   - Click the **"+"** (Insert More Content) button again in the Confluence editor.
   - Select **"Progress Bar"** from the options.
   - A dialog box will appear where you can input the progress percentage (e.g., 60%).
   - Enter **60** as the percentage and choose the desired color for the progress bar.
   - Click **Insert** to add the progress bar to the page.

Step 4: Review and Finalize the Page

1. **Arrange Content**:
   - Ensure that the embedded Jira issues are displayed clearly, and the progress bar is placed below or above them for visual clarity.
   - You can add headings like “Tasks Overview” or “Sprint Progress” to better organize the page.
2. **Preview the Page**:
   - Click **Preview** to check how the page looks.

- Ensure that the Jira tasks are properly embedded, displaying their status, and the progress bar correctly reflects the completion percentage.

3. **Publish the Page**:
   - Once you're satisfied with the content, click the **Publish** button to make the page live.

Step 5: Take a Screenshot of the Page

1. **View the Published Page**: After publishing, view the Confluence page you just created.
2. **Take a Screenshot**:
   - Capture a screenshot of the Confluence page showing the embedded Jira tasks with their statuses and the progress bar.
   - Ensure that the screenshot clearly displays the key elements: the Jira issues and the progress bar indicating the sprint's progress.

Step 6: Submit the Screenshot

1. **Save the Screenshot**: Save the screenshot as an image file (e.g., PNG, JPG).
2. **Submit the Screenshot**:
   - If required, upload the screenshot to your submission platform (e.g., Google Classroom, Learning Management System, etc.).
   - Alternatively, you can share it directly with your supervisor or team.

Ex.No: 5

Demonstrate how to work collaboratively in Git/GitHub on a project using the fork-and-pull request workflow.

Tasks:

1. **Fork an existing public GitHub repository (e.g., a sample JavaScript or Python project).**
2. **Clone the forked repository to your local machine using Git.**
3. **Create a new branch for the feature or change you want to work on.**
4. **Make modifications or add new features (e.g., add a function, fix a bug, or update the README).**
5. **Commit your changes and push the branch to GitHub.**
6. **Go to the GitHub repository and create a pull request to merge your feature branch into the main branch.**

7. **Review the pull request and provide feedback on the changes.**
8. **Respond to feedback by making additional commits to the feature branch if necessary.**
9. **Once the pull request is approved, merge it into the main branch.**
10. **Finally submit the link to the pull request along with a summary of the changes you made and how you collaborated.**

AIM: Demonstrate how to work collaboratively in Git/GitHub on a project using the fork-and-pull request workflow.

Description / Algorithm:

Step 1: Fork an Existing Public GitHub Repository

1. **Log in to GitHub**: Go to GitHub and log in with your credentials.
2. **Find the Repository**: Search for a public GitHub repository you want to contribute to (e.g., a sample JavaScript or Python project).
    - For example, you can search for a repository like "freeCodeCamp" or a small utility project.
3. **Fork the Repository**:
    - Navigate to the repository's main page.
    - Click the **"Fork"** button in the upper-right corner of the page.
    - GitHub will create a copy of the repository under your GitHub account.

Step 2: Clone the Forked Repository to Your Local Machine

1. **Go to Your Forked Repository**: After forking, go to your GitHub profile and find the forked repository.
2. **Copy the Repository URL**:
    - On the main page of your forked repository, click the **"Code"** button.
    - Copy the URL (either HTTPS or SSH) of the repository.
3. **Clone the Repository Locally**:
    - Open a terminal on your local machine.
    - Run the following command to clone the repository:

```
bash
Copy code
git clone https://github.com/your-username/repository-name.git
```

- Replace `your-username` and `repository-name` with your actual GitHub username and the repository name.

4. **Navigate to the Repository Folder**:
   - Change into the repository directory:

```
bash
Copy code
cd repository-name
```

Rest of the command as discussed in previous class

Ex.No: 6

You are managing a product backlog for an e-commerce application. The product team must prioritize features to develop based on limited resources using MoSCoW prioritization.

AIM: Demonstrate the product backlog for an e-commerce application using MosCow prioritization.

Description / Algorithm:

Step 1: Define the Product Backlog Items (PBIs)

Before using MoSCoW prioritization, you need to gather and define the key features and user stories for the e-commerce application. This can include user stories related to:

- User registration and login
- Product search and filtering
- Shopping cart functionality
- Payment gateway integration
- Order confirmation and tracking
- Admin panel for product management
- Customer support system

Step 2: Set Up JIRA Project for Your E-commerce Application

- **Create a JIRA Project**: Navigate to the JIRA dashboard and create a new project.
    - Choose **Kanban** or **Scrum** as your project template (depending on your team's preference).
    - Set up the project name (e.g., "E-Commerce Platform Development") and relevant details.
- **Create Product Backlog**: Under your JIRA project, create an initial backlog with all identified user stories and features.

Step 3: Create User Stories in JIRA

Now, you'll start adding user stories to the backlog. For example:

- **As a user**, I want to register an account so that I can log in and shop.
- **As a shopper**, I want to search for products by category so that I can find items I am interested in.
- **As an admin**, I want to manage product inventory so that I can add, update, and remove products.

For each of these, you'll create **issues** (user stories) in JIRA.

1. Go to the **Backlog** section in JIRA.
2. Click on the **Create Issue** button.
3. Select the issue type as **Story**.
4. Provide the title and description for each user story.
5. Optionally, assign the user story to a team member.

Repeat this for all features of the e-commerce application.

Step 4: Categorize User Stories Using MoSCoW Prioritization

In JIRA, you can categorize user stories using labels, custom fields, or versions (or sprints). Here's how you can do it using **labels** to implement MoSCoW:

Label Creation

1. Go to the **Backlog** page of your JIRA project.
2. Click on **Create Issue** and add a new user story.
3. In the issue creation screen, scroll down to the **Labels** field.
4. Add labels to categorize each user story as one of the MoSCoW categories.

   The MoSCoW labels would be:

    - **Must Have**: Essential features that must be delivered in the current release or sprint.
    - **Should Have**: Important features that should be included but not critical.
    - **Could Have**: Nice-to-have features, which can be deferred if needed.

    - **Won’t Have**: Features that are not part of the scope for the current release.

For example:

- **Label "Must Have"** for features like "user registration," "payment integration," or "order confirmation."
- **Label "Should Have"** for features like "product filtering" or "customer support chat."
- **Label "Could Have"** for features like "user reviews" or "wishlists."
- **Label "Won’t Have"** for features like "social media integration" or "augmented reality shopping."

Apply MoSCoW Labels to Issues

1. Once you’ve created your user stories, go back to the **Backlog**.
2. Open each issue (user story) and in the **Labels** section, add one of the following labels: **Must Have**, **Should Have**, **Could Have**, or **Won’t Have**.

Example:

- **Must Have**:
    - User Registration and Login
    - Shopping Cart
    - Payment Integration
- **Should Have**:
    - Search Functionality
    - Order History
- **Could Have**:
    - Product Reviews
    - Wish List
- **Won’t Have**:
    - Augmented Reality Shopping

Step 5: Prioritize the Backlog

Once you've applied the MoSCoW labels, it’s time to prioritize your backlog. Here's how to do it in JIRA:

1. **Drag and Drop**: In the **Backlog** view, you can drag and drop the issues (user stories) to reorder them. Move the user stories with the **Must Have** label to the top of the list, followed by **Should Have**, **Could Have**, and finally **Won’t Have**.
2. **Use Sprints** (if Scrum is being used): Group the user stories in sprints, prioritizing **Must Have** items first in Sprint 1, then **Should Have** in Sprint 2, and so on.

Step 6: Create Epics and Breakdown

If your e-commerce application has larger features, break them down into **Epics**:

1. Create a new **Epic** for each major feature (e.g., "User Registration" or "Product Search").
2. Associate the user stories with their respective Epics.
3. You can now track the progress of Epics and related stories more easily in JIRA.

Step 7: Start the Sprint (If Using Scrum)

If you're following a Scrum methodology:

1. Once the backlog is prioritized, you can plan the upcoming sprint.
2. Move the **Must Have** and **Should Have** stories to the **Sprint**.
3. Start the sprint, track progress, and ensure the **Must Have** items are completed first.

Step 8: Track Progress and Update Regularly

As work progresses, continuously update the backlog:

- Update labels and story points (if using estimation).
- Mark items as **Done** when completed.
- Adjust the prioritization as needed based on customer feedback or changes in business requirements.

Ex.no: 7

Two key stakeholders want to implement conflicting requirements in a new system. You are tasked with resolving these conflicts while maintaining the integrity of the project scope.

AIM:

Description / Algorithm:

Step 1: Document Conflicting Requirements in JIRA

1. **Create a JIRA Issue for Each Requirement**:
    - **Navigate to the Backlog**:
        - Go to **JIRA Dashboard** → Select the relevant **project**.
        - Click on the **Backlog** option in the left-hand sidebar.
    - **Create a new Issue**:
        - Click on the **Create** button in the top navigation bar.
        - In the **Create Issue** dialog, select **Issue Type** as **Story**.

        - Enter a title like “Stakeholder 1: Requirement A” and “Stakeholder 2: Requirement B”.
        - In the **Description** field, write a detailed explanation of each conflicting requirement.
        - Under **Labels**, use a label like “Conflicting Requirement”.
        - Click **Create**.
2. **Document Stakeholder Requirements**:
    - For each newly created story, click on the issue to open it.
    - In the **Description** section, add details about each stakeholder's requirement, including their reasoning and project impact.

Step 2: Analyze the Conflict and Create an Impact Assessment

1. **Assess Dependencies**:
    - **Navigate to the Backlog**:
        - Open the **Backlog** section from the left sidebar.
        - Look at the user stories and see if there are any dependencies between the conflicting requirements.
    - **Link Issues**:
        - Open the first issue, then click on **More** (three dots) → Select **Link**.
        - Choose a link type like **Relates to** or **Blocks** and link it to the other conflicting issue. This helps track how the two requirements are interrelated.
2. **Estimate the Impact**:
    - **Navigate to the Issue Details**:
        - Open each issue and go to the **Story Points** field (or **Time Estimate** field).
        - Add an estimation of effort or time for each conflicting requirement.
    - **Create an Epic if Necessary**:
        - If the conflicting requirements are large, click on the **Epic Link** field to create an Epic.
        - Enter a name for the Epic (e.g., “Conflicting Stakeholder Requirements”) and associate both stories with it.
3. **Risk Assessment**:
    - Open each issue and in the **Comments** section, add any risk-related insights regarding the conflicting requirements.

Step 3: Facilitate a Discussion and Reach a Consensus

1. **Schedule a Meeting with Stakeholders**:
    - **Use JIRA’s Calendar Integration**:

        - Schedule a meeting outside JIRA (via Outlook or Google Calendar), but you can also link any **Confluence page** or documents in the **Issue Comments** if relevant, for reference.
    - **Document Discussion Points**:
        - After the meeting, add key points, decisions, or compromises in the **Comments** section of the JIRA issues.
2. **Propose Potential Solutions**:
    - In the **Comments** section, suggest potential solutions such as phased implementations, compromises, or splitting the requirements.
    - Use **@mentions** to notify stakeholders and relevant team members to engage in the discussion.

Step 4: Update the JIRA Backlog with Resolution

1. **Document the Agreed Solution**:
    - Open each JIRA issue and update the **Description** section to reflect the agreed-upon resolution.
    - For example, state that the requirements will be phased, split into multiple user stories, or implemented in future releases.
2. **Create New User Stories if Necessary**:
    - If the solution involves splitting the requirements into smaller tasks or phases:
        - **Navigate to the Backlog** and click on **Create**.
        - Create new user stories representing different phases or parts of the conflicting requirements.
        - Link these new stories back to the original conflicting issues using the **Issue Link** feature.
        - For example, you can link a new story to "Stakeholder 1: Requirement A" with the link type "is blocked by."
3. **Reprioritize the Backlog**:
    - In the **Backlog** view, **drag and drop** the updated issues to reflect the new priority order.
    - Ensure that the most critical requirements (based on stakeholder priorities) are moved to the top of the list.

Step 5: Communicate the Resolution to Stakeholders

1. **Notify Stakeholders via JIRA Notifications**:
    - Use **@mentions** in the **Comments** section to notify stakeholders and team members of the resolution.
    - **Example**: `@Stakeholder1, @Stakeholder2, the conflict has been resolved by phasing the implementation of both requirements. Please review the changes in the Backlog.`

2. **Share the Updated Issues**:
    - Once updates are made, stakeholders can access the issues in the **Backlog** or **Active Sprint** (if relevant).
    - Share the JIRA board or **filter results** using the **Share** button or by copying the URL of the Backlog to allow easy access for stakeholders.
3. **Provide Access to Release Notes**:
    - **Navigate to Project Settings** → Go to **Releases** → Create or update **Release Notes** to include the resolution of the conflicting requirements.
    - You can link the Release Notes to the relevant issues and ensure stakeholders can easily access them.

Step 6: Track Progress and Adjust as Necessary

1. **Monitor the Implementation**:
    - **Navigate to the Active Sprint (or Kanban Board)**:
        - Open the **Active Sprint** or **Kanban Board** from the left sidebar.
        - Ensure that the resolved user stories are actively being worked on according to the updated scope.
    - Use **JIRA Filters** to create a custom filter for tracking the progress of the resolution.
        - Example: Save a filter like "Conflicting Requirement Resolution" to track the relevant stories.
2. **Adjust if New Conflicts Arise**:
    - If new conflicts emerge, return to the **Backlog** and create new issues for them, following the same process to resolve them.
    - **Link new issues** to the previous conflicts for tracking.

Step 7: Final Documentation and Close the Issues

1. **Close the Issues**:
    - Once both conflicting requirements are resolved and implemented, go to each JIRA issue and click **More** → **Close Issue**.
    - Add a final **Comment** confirming the completion and resolution of the conflict.
2. **Update the Project Documentation**:
    - If the resolution affects the overall project, update any **Confluence pages** or **project documentation** linked to the JIRA project.
3. **Provide Final Communication**:
    - Use the **Release Notes** or JIRA **Dashboard** to send a final summary to stakeholders, explaining how the conflict was resolved and how the solution will be implemented moving forward.

Summary of JIRA Navigation Steps:

1. **Document conflicting requirements** using **Create Issue** in **Backlog**.
2. **Analyze dependencies and risks** using **Issue Linking**, **Story Points**, and **Comments**.
3. **Facilitate discussion** using **@mentions** and **Comments** to capture decisions.
4. **Update JIRA Backlog** with resolutions by **Reprioritizing**, **Linking New Issues**, and adjusting **Epics**.
5. **Notify stakeholders** via **@mentions**, **Release Notes**, and **Issue Comments**.
6. **Track progress** via **Active Sprint**, **Kanban Board**, or **JIRA Filters**.
7. **Close issues** and update final project documentation.

Ex.No: 8

Automate the build and testing process to provide fast feedback to the development team after each code change, ensuring faster release cycles and maintaining high code quality.

AIM:

Automate the build and testing process to provide fast feedback to the development team and ensure faster release cycles while maintaining high code quality.

Description / Algorithm:

Step 1: Set Up Jenkins Environment

1. **Install Jenkins**:
   - If Jenkins is not already installed, download and install it from the official Jenkins website.
   - After installation, navigate to Jenkins via your browser (typically at `http://localhost:8080`).
2. **Install Necessary Plugins**:
   - Navigate to **Manage Jenkins → Manage Plugins**.
   - In the **Available** tab, search for and install the following plugins (if not already installed):
     - **Git Plugin** (for integrating Git repositories)

        - **Maven Plugin** (for building Java applications using Maven)
        - **JUnit Plugin** (for test reporting)
        - **Pipeline Plugin** (for defining Jenkins Pipelines using code)
    - Restart Jenkins if necessary.

Step 2: Create a New Jenkins Job (Pipeline)

1. **Navigate to Jenkins Dashboard**:
    - On the **Jenkins Dashboard**, click **New Item**.
2. **Configure Job**:
    - Enter a name for your job, e.g., **Build-and-Test-Pipeline**.
    - Select **Pipeline** and click **OK**.
3. **Set Up Pipeline Configuration**:
    - Under **Pipeline** section, scroll to the **Definition** field and select **Pipeline script**. This allows you to write a custom pipeline script.

Step 3: Define Pipeline Script for Build and Test

In the **Pipeline** section, write the following Jenkins pipeline script (example in **Declarative Pipeline** syntax) to automate the build and testing process:

```
groovy
Copy code
pipeline {
    agent any

    environment {
        // Set the path to the Maven home if required
        MAVEN_HOME = '/opt/maven'  // Adjust path based on your environment
    }

    stages {
        stage('Checkout') {
            steps {
                // Checkout code from Git repository
                git 'https://github.com/your-repository.git'
            }
        }

        stage('Build') {
            steps {
                // Use Maven to build the project
                script {
                    sh "${MAVEN_HOME}/bin/mvn clean install"
                }
            }
        }
```

```
        stage('Test') {
            steps {
                // Run unit tests using Maven and report results
                script {
                    sh "${MAVEN_HOME}/bin/mvn test"
                }
            }
        }

        stage('Archive Test Results') {
            steps {
                // Archive test results in Jenkins for easy access
                junit '**/target/test-*.xml'  // Specify the location of test
reports
            }
        }

        stage('Notify') {
            steps {
                // Send notifications (optional)
                echo "Build and testing complete. Notifications sent."
                // Add additional notification steps, e.g., Slack, email
            }
        }
    }

    post {
        success {
            echo 'Build and tests were successful!'
        }
        failure {
            echo 'Build or tests failed. Please check the logs.'
        }
    }
}
```

Explanation of Each Stage:

- **Checkout**: This stage fetches the latest code from the Git repository (make sure you replace the Git URL with your repository's URL).
- **Build**: Runs the Maven `clean install` command to compile and package the project.
- **Test**: Runs unit tests with Maven using `mvn test`.
- **Archive Test Results**: Archives test results so you can view them in Jenkins.
- **Notify**: Optionally, notify stakeholders about the status of the build and tests.

Step 4: Configure Build Triggers

1. **Navigate to Build Triggers**:

- Under the **Build Triggers** section in the job configuration page, select the option **GitHub hook trigger for GITScm polling** (if you're using GitHub) or choose **Poll SCM** (for other Git repositories).
- For **Poll SCM**, set the schedule to **H/5 * * * *** to check for changes every 5 minutes (you can adjust this based on your needs).

2. **Save Job**:
   - After configuration, click **Save** to save your pipeline job.

Step 5: Test the Pipeline

1. **Manually Trigger a Build**:
   - On the **Job Dashboard** page, click **Build Now** to manually trigger a pipeline run. Jenkins will check out the code, build it, run the tests, and archive the results.
   - You will see the progress and logs in the **Build History** section.
2. **Check the Console Output**:
   - After the build completes, click on the **Build Number** (e.g., **#1**) under **Build History** to view detailed logs, including build and test outputs.
   - If any tests fail, Jenkins will display the relevant test failures and logs in the console output.

Step 6: View Test Results

1. **Navigate to Test Reports**:
   - On the **Job Dashboard**, after the pipeline runs, you will see a **Test Result** section.
   - Jenkins will show test results based on the **JUnit plugin** configuration (e.g., number of passed/failed tests).
   - Click on the test result link to view detailed information about test failures or successes.
2. **Set Up Email/Slack Notifications** (Optional):
   - To notify stakeholders about build and test status, install the **Slack Notification Plugin** or **Email Extension Plugin** in Jenkins.
   - Configure the plugin under **Manage Jenkins → Configure System**, providing the necessary credentials and settings for your notification service.

Step 7: Monitor and Iterate

1. **Monitor Jenkins Dashboard**:
   - Regularly monitor the **Jenkins Dashboard** for feedback on builds and test runs.

    - Use **Jenkins Monitoring Plugins** to track the health and performance of your Jenkins instance.
2. **Iterate on Pipeline**:
    - Over time, improve your pipeline by adding stages like **Static Code Analysis** (e.g., using SonarQube), **Integration Testing**, or **Deployment**.
    - Consider integrating with **JIRA** to automatically create tickets for failed builds or tests.

Summary of Jenkins Navigation and Actions:

1. **Manage Jenkins** → **Manage Plugins** → Install necessary plugins (Git, Maven, JUnit, Pipeline).
2. **Jenkins Dashboard** → **New Item** → Create a **Pipeline** job.
3. Configure **Pipeline Script** with the build, test, and notification steps.
4. Set **Build Triggers** to automatically start builds on code changes.
5. **Build Now** on the **Job Dashboard** to trigger the pipeline manually.
6. View test results in the **Test Results** section of the job.
7. Set up **Notifications** for build and test status using Slack or email plugins.

Ex.No: 9

Automate the release process, including versioning, building, testing, and deploying to ensure smooth and consistent releases in an Agile process.

Aim:

Automate the release process, including versioning, building, testing, and deploying to ensure smooth and consistent releases in an Agile process.

Description / Algorithm:

To implement **Automated Release Pipeline with Versioning** in Jenkins, you will automate the release process, including versioning, building, testing, and deploying your application. This will ensure smooth and consistent releases in an Agile environment, reducing manual effort and providing continuous delivery.

Step-by-Step Jenkins Navigation for the Automated Release Pipeline with Versioning

---

Step 1: Set Up Jenkins Environment

1. **Install Jenkins** (if not already done):
   - Download Jenkins from Jenkins Website and install it.
   - Open Jenkins in your browser (`http://localhost:8080`).
2. **Install Required Plugins**:
   - Navigate to **Manage Jenkins → Manage Plugins**.
   - Install the following plugins:
     - **Pipeline Plugin**: To define the pipeline in Jenkins.
     - **Git Plugin**: To fetch code from Git repositories.
     - **Maven or Gradle Plugin**: Depending on your build tool.
     - **GitHub Plugin**: If using GitHub repositories.
     - **Build Timeout Plugin** (optional): To control the build time and prevent hanging.
     - **Slack Notification Plugin or Email Extension Plugin**: To notify stakeholders about build and deployment status.
   - After installing, restart Jenkins if necessary.

Step 2: Create a New Jenkins Pipeline Job

1. **Navigate to Jenkins Dashboard**:
   - On the Jenkins home page, click **New Item**.
2. **Create Pipeline Job**:
   - Enter a name for your job (e.g., **Release-Pipeline**).
   - Select **Pipeline** as the job type and click **OK**.

Step 3: Configure the Pipeline Script

1. **Set Up the Pipeline Definition**:
   - In the **Job Configuration** page, scroll down to the **Pipeline** section.
   - Under the **Definition** field, select **Pipeline script**.
2. **Write the Pipeline Script**:
   - In the **Pipeline Script** box, write the following code (using **Declarative Pipeline** syntax). This script automates versioning, building, testing, and deployment:

```
groovy
Copy code
```

```
pipeline {
    agent any

    environment {
        // Define variables such as project version, deploy server details
        VERSION = "1.0.${BUILD_NUMBER}"  // Dynamic versioning using Jenkins
build number
        DEPLOY_SERVER = 'your-deploy-server.com'
        DEPLOY_USER = 'username'
        DEPLOY_PATH = '/path/to/deploy'
    }

    stages {
        stage('Checkout Code') {
            steps {
                // Checkout the latest code from Git repository
                git 'https://github.com/your-repository.git'
            }
        }

        stage('Set Version') {
            steps {
                // Set the version using Jenkins build number
                script {
                    sh 'echo "Version: ${VERSION}" > version.txt'
                }
            }
        }

        stage('Build') {
            steps {
                // Run the build process (e.g., using Maven or Gradle)
                script {
                    sh 'mvn clean package -Dversion=${VERSION}'  // Example
with Maven
                }
            }
        }

        stage('Test') {
            steps {
                // Run unit tests and produce test reports
                script {
                    sh 'mvn test'  // Example with Maven
                }
            }
        }

        stage('Deploy to Staging') {
            steps {
                // Deploy to staging environment (SSH example)
                script {
                    sh "scp target/myapp-${VERSION}.jar
${DEPLOY_USER}@${DEPLOY_SERVER}:${DEPLOY_PATH}/"
                    sh "ssh ${DEPLOY_USER}@${DEPLOY_SERVER} 'java -jar
${DEPLOY_PATH}/myapp-${VERSION}.jar'"
                }
```

```
            }
        }

        stage('Integration Testing') {
            steps {
                // Run integration tests in the staging environment
                script {
                    sh './run_integration_tests.sh'  // Custom script for
integration tests
                }
            }
        }

        stage('Deploy to Production') {
            when {
                branch 'main'  // Only deploy to production from the main
branch
            }
            steps {
                // Deploy to the production environment
                script {
                    sh "scp target/myapp-${VERSION}.jar
${DEPLOY_USER}@${DEPLOY_SERVER}:${DEPLOY_PATH}/"
                    sh "ssh ${DEPLOY_USER}@${DEPLOY_SERVER} 'java -jar
${DEPLOY_PATH}/myapp-${VERSION}.jar'"
                }
            }
        }

        stage('Notify') {
            steps {
                // Send notifications (via Slack or Email)
                slackSend(channel: '#deployments', message: "Release
${VERSION} deployed successfully: ${env.BUILD_URL}")
            }
        }
    }

    post {
        success {
            echo "Release ${VERSION} was successful!"
        }
        failure {
            echo "Release failed. Please check the logs."
        }
    }
}
```

Explanation of Key Stages:

- **Checkout Code**: Pulls the latest code from the Git repository.
- **Set Version**: Sets the version dynamically using the Jenkins build number. This ensures that each build gets a unique version (e.g., `1.0.1`, `1.0.2`).
- **Build**: Compiles the application and creates a deployable artifact (e.g., `.jar`, `.war`).
- **Test**: Runs unit tests to verify the code.

- **Deploy to Staging**: Deploys the artifact to a staging environment for further testing.
- **Integration Testing**: Runs integration tests in the staging environment to validate the deployment.
- **Deploy to Production**: Deploys to the production environment only if the `main` branch is deployed.
- **Notify**: Sends notifications (Slack or Email) about the deployment status.

Step 4: Configure Build Triggers

1. **Navigate to Build Triggers**:
   - In the **Job Configuration** page, scroll to the **Build Triggers** section.
2. **Set Up Automatic Triggers**:
   - Select **GitHub hook trigger for GITScm polling** (if you're using GitHub).
   - Alternatively, select **Poll SCM** and set the schedule to `H/5 * * * *` (to poll every 5 minutes) to check for code changes automatically.

Step 5: Save the Jenkins Job

- After configuring the pipeline script and build triggers, click **Save**.

Step 6: Test the Pipeline

1. **Trigger the Build Manually**:
   - Go to the **Job Dashboard**.
   - Click on **Build Now** to trigger the pipeline manually for the first time.
2. **Monitor the Progress**:
   - After the build is triggered, click on the **Build Number** (e.g., **#1**) under **Build History** to monitor the build progress.
   - View the **Console Output** to check the logs and ensure that the build, test, and deployment stages are working correctly.

Step 7: View Deployment and Notifications

1. **Check Console Output**:
   - After the build completes, click **Console Output** to view the detailed logs from each stage, including any build failures, test failures, or deployment issues.
2. **Notifications**:

- After the release is successful, Jenkins will send a Slack message (or an email, depending on your configuration) to notify the stakeholders about the successful deployment.

Step 8: Verify Deployment

1. **Verify Staging Deployment**:
   - Ensure that the application is deployed correctly in the **staging environment** and that all integration tests pass.
2. **Verify Production Deployment**:
   - After successful tests in staging, verify that the deployment to the **production environment** is successful and the application is functioning correctly.

Ex.No: 10

Create a Static Website and Containerize, Build & Serve it using Docker.

Tasks:

1. **Create a Simple Static Website (index.html file) with basic HTML content.**
2. **Write/create a Dockerfile to serve the website using Nginx.**
3. **Build the Docker Image**
4. **Run the container:**
5. **Access the Website using a Browser**

Aim: Create a static website to demonstrate the containerization with Docker for creating a simple static website.

Description /Algorithm:

Step 1: Create a Simple Static Website (index.html)

1. **Create a New Directory for the Project**:
   - Open your terminal and create a new directory to store your project files.

```
bash
Copy code
mkdir my-static-website
cd my-static-website
```

2. **Create the `index.html` File**:
   - In the `my-static-website` directory, create a file named `index.html`.

```
bash
Copy code
touch index.html
```

3. **Add Basic HTML Content**:
   - Open the `index.html` file in your favorite text editor (e.g., VSCode, Sublime Text, or simply nano/vim).
   - Add the following HTML content to create a simple webpage.

```
html
Copy code
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>My Static Website</title>
</head>
<body>
    <h1>Welcome to My Static Website</h1>
    <p>This is a simple website served using Docker and Nginx.</p>
</body>
</html>
```

4. **Save and Close the File**:
   - Save the file and close the editor.

Step 2: Write/Create a Dockerfile to Serve the Website Using Nginx

1. **Create a Dockerfile**:
   - In the `my-static-website` directory, create a file named `Dockerfile`.

```
bash
Copy code
```

```
touch Dockerfile
```

2. **Write the Dockerfile**:
   - Open the `Dockerfile` in your editor and add the following content:

```
Dockerfile
Copy code
# Use the official Nginx image from Docker Hub
FROM nginx:alpine

# Copy the index.html file into the Nginx default public directory
COPY index.html /usr/share/nginx/html/

# Expose port 80 to allow external access to the website
EXPOSE 80

# Run Nginx in the foreground (default command for the Nginx container)
CMD ["nginx", "-g", "daemon off;"]
```

   Explanation:

   - `FROM nginx:alpine`: This uses the official Nginx image based on Alpine Linux, which is lightweight and ideal for serving static websites.
   - `COPY index.html /usr/share/nginx/html/`: This copies the `index.html` file into the directory where Nginx looks for static files.
   - `EXPOSE 80`: This exposes port 80, which is the default HTTP port, so the website can be accessed from outside the container.
   - `CMD ["nginx", "-g", "daemon off;"]`: This tells Docker to start Nginx in the foreground when the container runs.
3. **Save the Dockerfile**:
   - Save the file and close the editor.

Step 3: Build the Docker Image

1. **Open Terminal**:
   - Navigate to the directory where your `Dockerfile` and `index.html` file are located (the `my-static-website` directory).
2. **Build the Docker Image**:
   - Run the following command to build the Docker image. You can name your image `static-website`:

```
bashdoc
Copy code
docker build -t static-website .
```

Explanation:

- `docker build -t static-website .`: This tells Docker to build an image from the current directory (`.`) and tag it as `static-website`.

3. **Wait for the Build to Complete**:
   - Docker will pull the Nginx image (if not already cached), copy the files, and create the new image. This may take a minute or two.

Step 4: Run the Container

1. **Run the Docker Container**:
   - After the image has been built, run it as a container using the following command:

```
bash
Copy code
docker run -d -p 8080:80 static-website
```

Explanation:

- `docker run`: Starts a new container.
- `-d`: Runs the container in detached mode (in the background).
- `-p 8080:80`: Maps port 80 in the container to port 8080 on your host machine (so you can access it via `localhost:8080`).
- `static-website`: The name of the image you just built.

2. **Check the Running Container**:
   - To verify that the container is running, use the following command:

```
bash
Copy code
docker ps
```

   - This will list all running containers, and you should see the `static-website` container in the list with port `8080` exposed.

Step 5: Access the Website Using a Browser

1. **Open Your Browser**:
   - Open your browser and type the following URL in the address bar:

```
bash
Copy code
http://localhost:8080
```

2. **View the Static Website**:
    - You should see the simple static website that you created in `index.html`. The page should display the message: **"Welcome to My Static Website"** and the paragraph content.

Step 6: Optional - Clean Up Docker Resources

If you're done with the website and want to stop and remove the container, follow these steps:

1. **Stop the Running Container**:
    - Use the following command to stop the container:

```
bash
Copy code
docker stop <container_id>
```

    - You can find the `container_id` by running:

```
bash
Copy code
docker ps
```

2. **Remove the Stopped Container**:
    - Once the container is stopped, you can remove it with:

```
bash
Copy code
docker rm <container_id>
```

3. **Remove the Docker Image (optional)**:
    - If you no longer need the Docker image, you can remove it with:

```
bash
Copy code
docker rmi static-website
```

Summary:

1. You created a simple **static website** with HTML.
2. You wrote a **Dockerfile** to containerize the website using Nginx.
3. You **built the Docker image** and **ran the container**.
4. You accessed the website in your browser at **`http://localhost:8080`**.