1. what test cases which are not automated?

Answer:

Test cases that are not automated are those that require manual intervention, human judgment, or subjective evaluation. Here are some examples:

1. **Usability Testing**: Assessing how user-friendly a software application is, including user interface (UI) navigation, intuitiveness, and overall user experience. This often involves real users interacting with the system and providing feedback.
2. **Exploratory Testing**: This involves ad-hoc testing where testers explore the software application without predefined test cases. Testers rely on their domain knowledge, intuition, and creativity to uncover defects and issues.
3. **User Acceptance Testing (UAT)**: UAT involves end-users verifying whether the software meets their requirements and business needs. Since it's conducted by the end-users themselves, it cannot be fully automated.
4. **Compatibility Testing**: Testing how well the software performs across different platforms, devices, browsers, and operating systems. While some aspects of compatibility testing can be automated, human observation and validation are often necessary to ensure the software functions correctly in diverse environments.
5. **Localization Testing**: Verifying that the software application is adapted for different languages, cultures, and regions. This involves checking for language accuracy, cultural appropriateness, and proper formatting, which are best assessed by native speakers and cultural experts.
6. **Security Testing**: Assessing the security aspects of the software application, including vulnerability scanning, penetration testing, and security audits. While automated tools can assist in identifying known vulnerabilities, manual testing is often required to uncover complex security flaws and assess overall security posture.
7. **Performance Testing**: While performance testing can be partially automated using tools to simulate load and measure response times, manual intervention is often necessary to analyze performance metrics, identify performance bottlenecks, and optimize system performance.
8. **Ad Hoc Testing**: Testers perform random checks or test scenarios that are not covered by predefined test cases to identify any unforeseen issues
9. what type of test automation we have seen:

Answer:

Here are some common types of test automation:

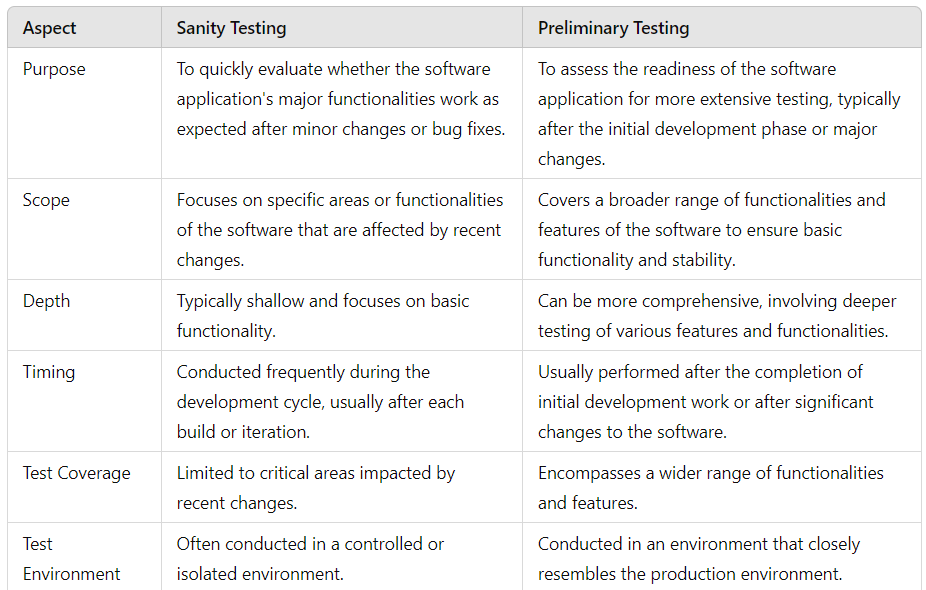
1. **Unit Testing**: Automated tests written by developers to verify the behavior of individual units or components of the software, typically at the code level. Unit testing frameworks like JUnit (for Java) or NUnit (for .NET) are commonly used for this purpose.
2. **Integration Testing**: Tests that verify the interactions and communication between different modules or components within a system. Integration testing frameworks like TestNG or Mockito (for Java) or PyTest (for Python) are often used to automate these tests.
3. **Functional Testing**: Automated tests that verify the functional requirements of the software application, such as verifying that specific features work as expected. Tools like Selenium WebDriver, Cypress, or TestCafe are popular for automating functional tests, especially for web applications.
4. **Regression Testing**: Tests that ensure that new changes or updates to the software do not negatively impact existing functionality. Test automation frameworks like Selenium, Robot Framework, or TestComplete are commonly used for regression testing.
5. **GUI Testing**: Automated tests that validate the graphical user interface (GUI) of the software application, ensuring that UI elements are displayed correctly and function as intended. Tools like Selenium WebDriver, TestComplete, or Appium (for mobile apps) can be used for GUI testing.
6. **API Testing**: Tests that validate the functionality, reliability, performance, and security of application programming interfaces (APIs). Tools like Postman, SoapUI, or RestAssured (for Java) are commonly used for automating API tests.
7. **Performance Testing**: Automated tests designed to assess the performance characteristics of the software application, such as response times, throughput, and scalability. Tools like Apache JMeter, Gatling, or LoadRunner are often used for performance testing automation.
8. **Security Testing**: Automated tests that assess the security vulnerabilities and weaknesses of the software application, such as penetration testing or vulnerability scanning. Tools like OWASP ZAP, Burp Suite, or Nessus can be used for automating security testing.
9. **Mobile Testing**: Automated tests designed specifically for mobile applications, covering aspects such as UI testing, functionality testing, and compatibility testing across different devices and platforms. Tools like Appium, Espresso, or XCUITest are commonly used for mobile test automation.
10. **Cross-Browser Testing**: Automated tests that verify the compatibility and functionality of web applications across different web browsers and browser versions. Tools like Selenium WebDriver, TestComplete, or BrowserStack are often used for cross-browser testing automation.
11. what is error handling in SDM?

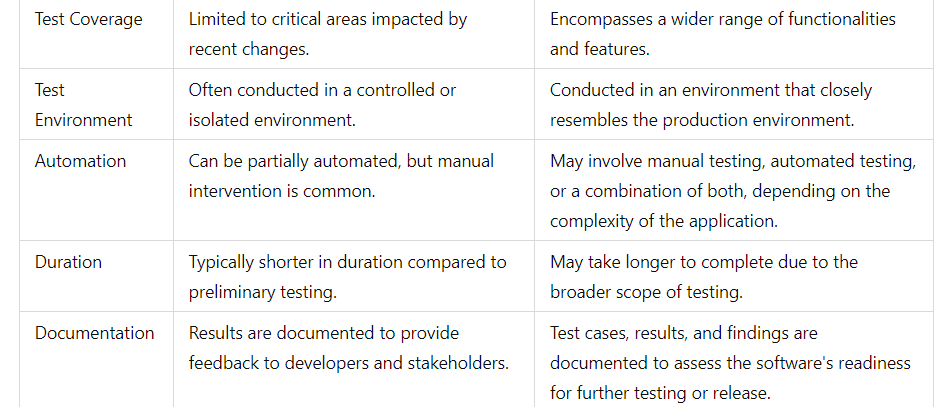
Answer:

Here's how error handling typically works within the context of SDM:

1. **Identification of Potential Errors**: During the development phase, software developers identify potential errors that may occur during the execution of the software application. These errors could be caused by invalid inputs, unexpected system behaviors, or external factors.
2. **Error Reporting**: Developers implement mechanisms to report errors and exceptions that occur during runtime. This may involve logging error messages, capturing stack traces, or generating error reports that provide detailed information about the nature of the error, its location, and the context in which it occurred.
3. **Exception Handling**: Exception handling is a programming construct used to manage errors and exceptions gracefully. Developers write code to handle specific types of exceptions that may arise during the execution of the software. This typically involves using try-catch blocks or similar constructs to catch and handle exceptions in a controlled manner.
4. **Error Recovery**: In some cases, software applications may be designed to recover from errors automatically or to provide users with options for recovering from errors manually. Error recovery mechanisms may involve retrying failed operations, rolling back transactions, or prompting users to take corrective actions.
5. **User Feedback and Error Messages**: When errors occur, it's important to provide users with meaningful feedback and error messages that help them understand what went wrong and how to resolve the issue. Error messages should be clear, concise, and actionable, guiding users toward the appropriate course of action to address the error.
6. **Testing Error Handling**: As part of the testing process, software testers evaluate the effectiveness of error handling mechanisms by deliberately introducing errors and exceptions into the system and observing how the software responds. This helps identify any weaknesses or deficiencies in the error handling logic and ensures that the software can gracefully handle unexpected situations.
7. **Continuous Improvement**: Error handling is an ongoing process, and software teams continuously monitor and refine error handling mechanisms based on feedback from users, automated error reports, and real-world usage data. Continuous improvement helps ensure that the software remains robust and resilient in the face of evolving requirements and environmental conditions.
8. difference between sanity,preliminary testing?

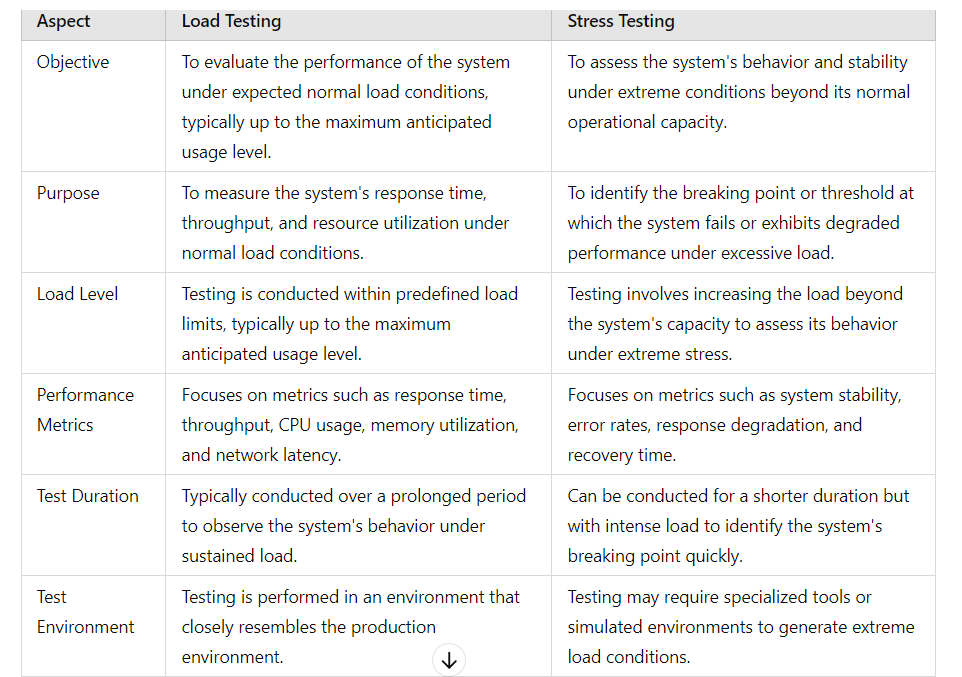
Answer:

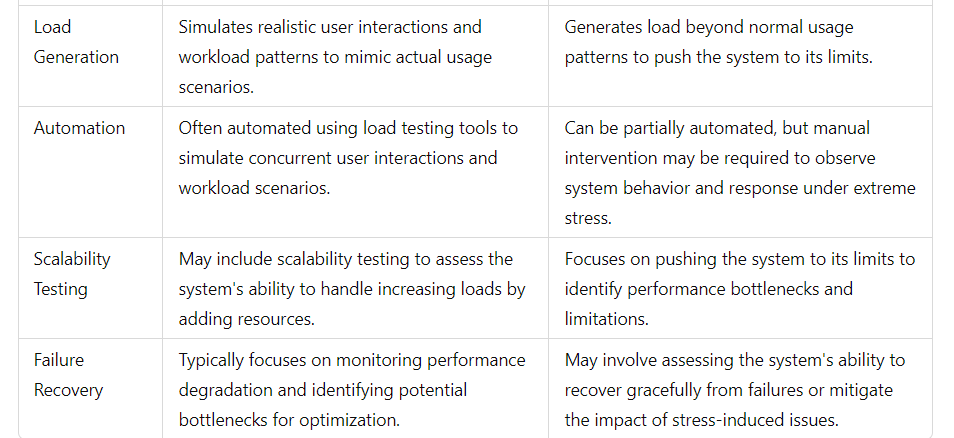




1. difference between load and stress testing?

Answer:





1. why automation testing?

Answer:

Here are some key reasons why automation testing is widely adopted:

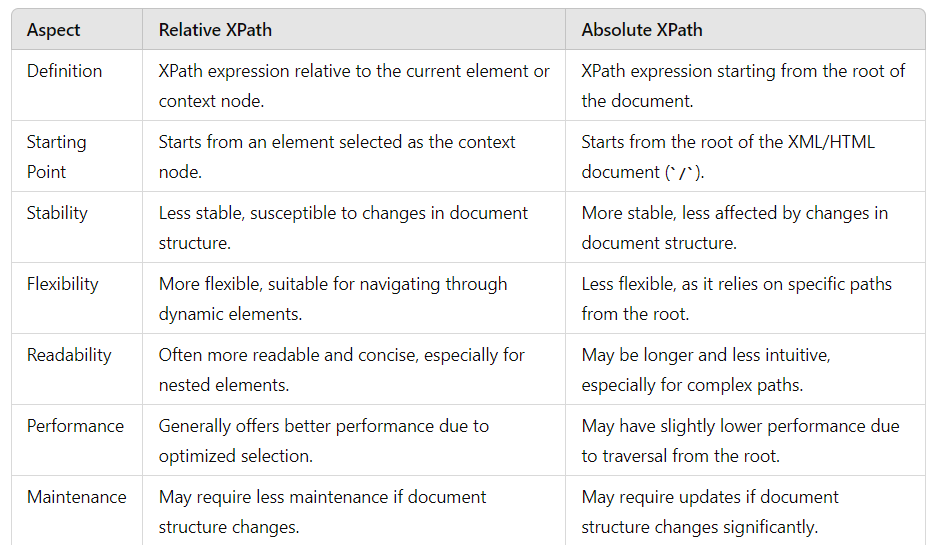
1. **Efficiency**: Automation testing can execute tests much faster than manual testing. It allows for the simultaneous execution of multiple test cases across different environments and configurations, reducing testing time significantly.
2. **Reusability**: Automated test scripts can be reused across multiple test cycles, releases, and projects. Once created, they can serve as a regression suite to ensure that existing functionality remains intact with each new release.
3. **Accuracy**: Automated tests eliminate the risk of human error associated with manual testing, ensuring consistent and reliable test results. They precisely follow predefined test steps and accurately compare expected outcomes with actual results.
4. **Cost-Effectiveness**: While there is an initial investment in developing automated test scripts, automation ultimately reduces the overall cost of testing. It decreases the need for manual effort, accelerates time-to-market, and helps identify defects early in the development cycle when they are less expensive to fix.
5. **Increased Test Coverage**: Automation testing enables broader test coverage by executing a large number of test cases across various configurations, platforms, and environments. It can cover scenarios that may be impractical or time-consuming to test manually.
6. **Frequent Execution**: Automated tests can be executed more frequently, allowing for continuous integration and continuous delivery (CI/CD) practices. They provide rapid feedback on the quality of code changes, facilitating faster iterations and deployments.
7. **Parallel Execution**: Automation testing supports parallel execution of test cases, enabling faster feedback and scalability. It allows teams to distribute tests across multiple machines or virtual environments, reducing overall execution time.
8. **Regression Testing**: Automated tests serve as an effective means of regression testing, ensuring that new changes do not introduce unintended side effects or break existing functionality. They help maintain software quality and stability over time.
9. **Support for Agile and DevOps**: Automation testing aligns well with Agile and DevOps methodologies, which emphasize iterative development, frequent releases, and automated workflows. It enables teams to deliver high-quality software at a rapid pace while maintaining agility and flexibility.
10. **Improved Product Quality**: By identifying defects early in the development cycle and ensuring consistent testing across releases, automation testing contributes to overall product quality. It helps mitigate risks, enhances customer satisfaction, and fosters a culture of continuous improvement.
11. Important things for web based application?

Answer:

For web-based applications, several important aspects need to be considered to ensure their functionality, usability, security, and performance. Here are some key considerations:

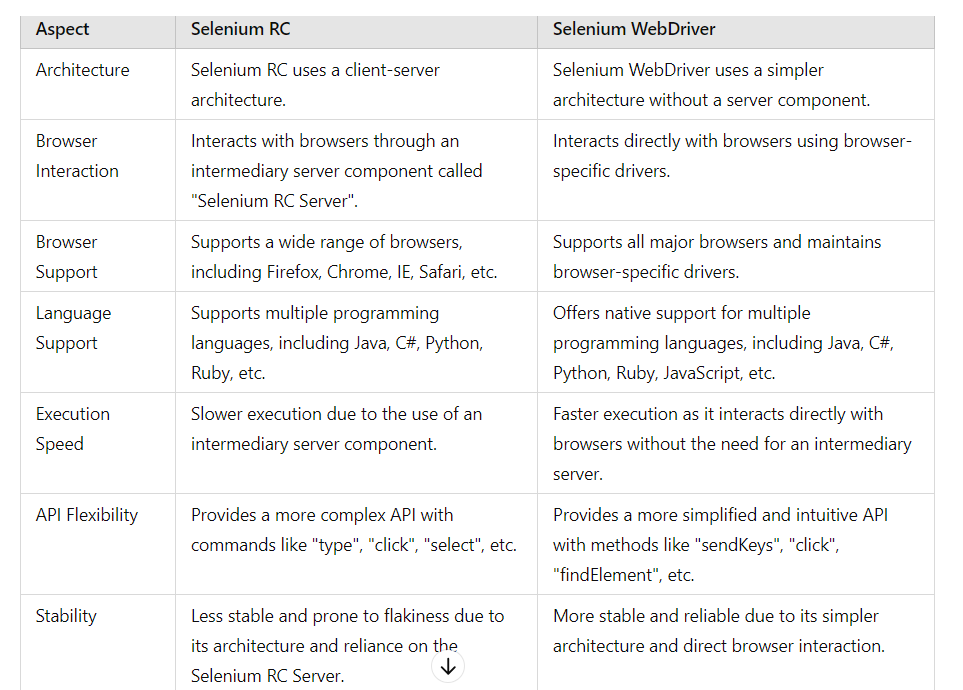
1. **User Interface (UI) Design**: The user interface should be intuitive, responsive, and visually appealing. It should be designed to provide a seamless and enjoyable user experience across different devices and screen sizes.
2. **Cross-Browser Compatibility**: The application should be compatible with various web browsers (e.g., Chrome, Firefox, Safari, Edge) to ensure consistent performance and functionality for all users.
3. **Responsive Design**: The application should be responsive, meaning it adapts to different screen sizes and resolutions, including desktops, laptops, tablets, and mobile devices.
4. **Accessibility**: Ensure that the application is accessible to users with disabilities by adhering to web accessibility standards (e.g., WCAG). This includes providing alternative text for images, keyboard navigation support, and other accessibility features.
5. **Performance Optimization**: Optimize the application's performance to ensure fast loading times and smooth user interactions. This involves minimizing HTTP requests, leveraging caching mechanisms, and optimizing code and asset delivery.
6. **Security Measures**: Implement robust security measures to protect against common web vulnerabilities such as cross-site scripting (XSS), SQL injection, and cross-site request forgery (CSRF). Use encryption (e.g., HTTPS) to secure data transmission and implement proper authentication and authorization mechanisms.
7. **Scalability**: Design the application architecture to be scalable, allowing it to handle increasing loads and accommodate growing user bases without sacrificing performance or reliability. This may involve using scalable cloud infrastructure and distributed systems.
8. **Data Backup and Recovery**: Implement regular data backups and disaster recovery procedures to safeguard against data loss and ensure business continuity in the event of system failures or disasters.
9. **Testing and Quality Assurance**: Conduct thorough testing, including functional testing, usability testing, performance testing, security testing, and compatibility testing, to identify and address any issues before deployment.
10. **Monitoring and Analytics**: Set up monitoring tools and analytics to track the application's performance, user behavior, and usage patterns. This data can be used to identify areas for improvement and make informed decisions about future enhancements.
11. **Compliance and Regulations**: Ensure that the application complies with relevant laws, regulations, and industry standards, such as GDPR (General Data Protection Regulation) for data privacy or PCI DSS (Payment Card Industry Data Security Standard) for handling payment information.
12. **Documentation and Support**: Provide comprehensive documentation and user support resources to help users understand how to use the application effectively and troubleshoot common issues.
13. Difference between relative xpath, absolute xpath?

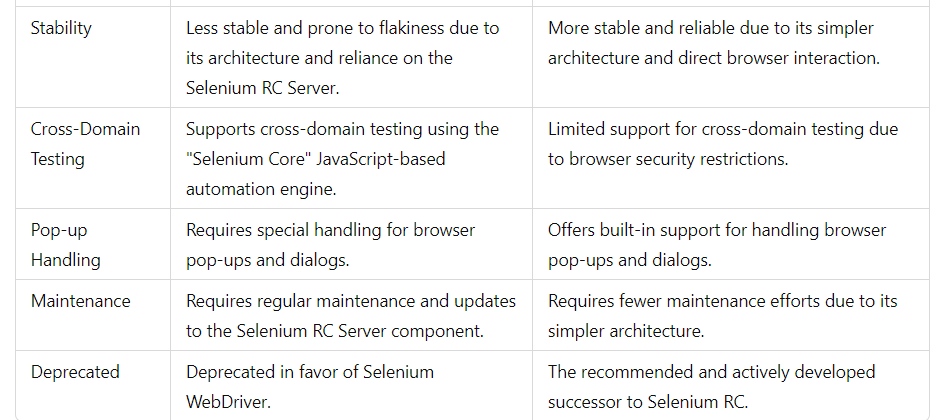
Answer:



1. Difference between RC and web driver?

Answer:





1. Devops test phases?

Answer:

The testing phases in DevOps typically include the following:

1. **Unit Testing**: Developers write unit tests to verify the functionality of individual units or components of the software application. Unit tests are typically automated and run frequently during the development process.
2. **Integration Testing**: Integration testing verifies the interactions between different modules or components of the software application. It ensures that the integrated system functions correctly as a whole. Integration tests may be automated and run as part of the CI/CD pipeline.
3. **System Testing**: System testing validates the entire software system against the specified requirements. It involves testing the system as a whole to ensure that it meets functional, performance, and security criteria. System tests may include functional testing, usability testing, performance testing, and security testing.
4. **Acceptance Testing**: Acceptance testing, also known as user acceptance testing (UAT), involves validating the software application against business requirements and user expectations. It ensures that the software meets the needs of end-users and stakeholders. UAT may be performed manually by end-users or automated using testing frameworks.
5. **Regression Testing**: Regression testing verifies that new changes or updates to the software application do not negatively impact existing functionality. It involves retesting previously tested features to ensure that they still work as expected. Regression tests are typically automated and run as part of the CI/CD pipeline to detect regressions early.
6. **Performance Testing**: Performance testing evaluates the performance characteristics of the software application, including response times, throughput, and scalability. It ensures that the software can handle expected levels of user traffic and data volume. Performance tests may include load testing, stress testing, and scalability testing.
7. **Security Testing**: Security testing assesses the security posture of the software application, identifying and mitigating potential security vulnerabilities and weaknesses. It includes testing for common security threats such as injection attacks, cross-site scripting (XSS), and authentication vulnerabilities.
8. **Smoke Testing**: Smoke testing, also known as sanity testing, involves running a subset of tests to quickly verify that critical functionalities of the software application work as expected. It ensures that the software is stable enough for further testing or deployment.
9. Use of git commit(), git init() and git clone()?

Answer:

**git commit()**:

* Purpose: git commit() is used to record changes to the repository. It creates a snapshot of the current state of the working directory and stages the changes to be included in the next commit.
* Usage: After making changes to files in the working directory, you use git add to stage the changes, and then git commit to save the staged changes to the repository along with a commit message describing the changes.

**git init()**:

* Purpose: git init() is used to initialize a new Git repository in the current directory or in a specified directory. It creates a new .git subdirectory containing the necessary Git repository files and sets up the repository.
* Usage: You typically use git init when starting a new project or when converting an existing project into a Git repository.

**git clone()**:

* Purpose: git clone() is used to create a copy of an existing Git repository, including all its branches and commit history. It allows you to clone a repository from a remote location (e.g., GitHub, GitLab) to your local machine.
* Usage: You use git clone followed by the URL of the repository you want to clone. This command creates a new directory with the same name as the cloned repository and initializes it as a Git repository with the remote repository set as the origin.

1. kubernate advantage and disadvantages?

Answer:

kubernate advantage and disadvantages

ChatGPT

ChatGPT

Kubernetes (K8s) is a powerful container orchestration platform widely used in modern cloud-native application development. Here are some advantages and disadvantages of Kubernetes:

**Advantages:**

1. **Scalability**: Kubernetes provides built-in support for scaling applications horizontally and vertically. It can automatically scale up or down based on resource usage metrics, ensuring optimal performance and resource utilization.
2. **High Availability**: Kubernetes ensures high availability of applications by automatically restarting failed containers, rescheduling them on healthy nodes, and distributing traffic among healthy instances. It supports rolling updates and zero-downtime deployments.
3. **Resource Efficiency**: Kubernetes optimizes resource allocation by packing multiple containers onto a single node and dynamically adjusting resource allocations based on application requirements. It helps maximize resource utilization and reduce infrastructure costs.
4. **Portability**: Kubernetes offers a consistent deployment and management experience across various environments, including on-premises data centers, public clouds (e.g., AWS, Azure, Google Cloud), and hybrid cloud environments. It enables seamless workload migration and hybrid cloud deployments.
5. **Container Orchestration**: Kubernetes automates container deployment, scaling, and management tasks, such as scheduling, load balancing, health checks, and self-healing. It simplifies complex deployment workflows and reduces operational overhead.
6. **Extensibility**: Kubernetes provides a rich ecosystem of plugins, extensions, and APIs that allow for seamless integration with third-party tools, services, and platforms. It supports custom resource definitions (CRDs) and operators for extending its capabilities.
7. **Declarative Configuration**: Kubernetes uses declarative configuration files (YAML or JSON) to describe desired application state and infrastructure requirements. This approach enables infrastructure as code (IaC) practices and facilitates version control and reproducibility.

**Disadvantages:**

1. **Complexity**: Kubernetes has a steep learning curve and can be complex to set up, configure, and manage, especially for novice users. It requires a deep understanding of containerization, networking, storage, and distributed systems concepts.
2. **Resource Overhead**: Kubernetes imposes additional resource overhead for managing the control plane components (e.g., etcd, API server, scheduler) and networking infrastructure (e.g., overlay networks, service meshes). This overhead may impact the overall performance and cost of the infrastructure.
3. **Operational Complexity**: Operating Kubernetes clusters at scale requires expertise in cluster lifecycle management, monitoring, troubleshooting, security, and compliance. Managing upgrades, patches, and backups can be challenging, particularly in production environments.
4. **Networking Complexity**: Kubernetes networking can be complex, especially in multi-cluster or hybrid cloud deployments. Configuring networking policies, service discovery, and load balancing requires careful planning and may introduce network latency and overhead.
5. **Storage Orchestration**: Kubernetes storage orchestration is still evolving, and managing stateful workloads (e.g., databases, message queues) can be challenging. Although Kubernetes offers various storage options (e.g., PersistentVolumes, StatefulSets), setting up and managing storage solutions requires careful consideration.
6. **Community Support**: While Kubernetes has a vibrant open-source community and ecosystem, obtaining timely support and documentation for specific use cases or edge cases may be challenging. Enterprises may need to rely on commercial support offerings or consulting services for complex deployments.
7. **Vendor Lock-In**: Adopting managed Kubernetes services (e.g., GKE, EKS, AKS) from cloud providers may lead to vendor lock-in, limiting portability and interoperability with other platforms. It's essential to consider the long-term implications and exit strategies when choosing a Kubernetes deployment model.
8. what is agile and how many deployment methodology in agile?

Answer:

Agile is a software development methodology that prioritizes flexibility, collaboration, and iterative development. It emphasizes delivering working software in small, incremental releases and adapting to changing requirements and feedback throughout the development process. Agile methodologies prioritize individuals and interactions over processes and tools, working software over comprehensive documentation, customer collaboration over contract negotiation, and responding to change over following a plan.

There are several deployment methodologies within the Agile framework. Some of the commonly used ones include:

1. **Scrum**: Scrum is one of the most popular Agile methodologies. It involves breaking down the development process into short iterations called "sprints," typically lasting 2-4 weeks. Each sprint begins with a planning meeting where the team selects a set of user stories to work on. Daily stand-up meetings are held to track progress and address any issues, and at the end of each sprint, a review and retrospective are conducted to assess the work done and identify areas for improvement.
2. **Kanban**: Kanban is a visual Agile methodology that focuses on continuous delivery and flow. Work items are represented as cards on a Kanban board, which consists of columns representing different stages of the workflow (e.g., to do, in progress, done). Teams pull work items from the backlog into the workflow as capacity allows, with the goal of minimizing work in progress (WIP) and maximizing throughput.
3. **Extreme Programming (XP)**: Extreme Programming is an Agile methodology that emphasizes technical excellence, continuous feedback, and rapid iterations. It includes practices such as test-driven development (TDD), pair programming, continuous integration, and frequent releases. XP aims to deliver high-quality software quickly by prioritizing simplicity, communication, and customer involvement.
4. **Feature-Driven Development (FDD)**: Feature-Driven Development is an Agile methodology that focuses on building software around specific features or functionalities. It involves breaking down the project into smaller, more manageable feature sets, with each feature set being developed and delivered incrementally. FDD emphasizes domain modeling, design inspections, and regular progress reporting.
5. **Lean Software Development**: Lean Software Development is an Agile methodology that draws inspiration from lean manufacturing principles. It focuses on minimizing waste, maximizing customer value, and optimizing the flow of work. Lean practices include value stream mapping, continuous improvement, and just-in-time delivery.
6. **Dynamic Systems Development Method (DSDM)**: DSDM is an Agile methodology that provides a framework for delivering projects quickly and efficiently while ensuring alignment with business goals. It emphasizes active user involvement, iterative development, and incremental delivery. DSDM incorporates principles such as frequent delivery, collaborative and empowered teams, and focus on business needs.
7. what is scrum ceremony explain it?

Answer:

Scrum ceremonies are key events or meetings held within the Scrum framework to facilitate collaboration, communication, and transparency among team members. These ceremonies provide opportunities for the Scrum Team to plan, inspect, adapt, and review their work throughout the development process. Here are the main Scrum ceremonies and their purposes:

1. **Sprint Planning**: Sprint Planning is a collaborative meeting at the start of each sprint where the Scrum Team plans the work to be done during the sprint. It involves discussing the sprint goal, selecting user stories or tasks from the product backlog, and creating a sprint backlog with the tasks needed to achieve the sprint goal.
2. **Daily Stand-up (Daily Scrum)**: The Daily Stand-up is a short, daily meeting where the Scrum Team synchronizes their work and discusses progress, challenges, and plans for the day. Each team member answers three questions: What did I do yesterday? What will I do today? Are there any impediments blocking my progress?
3. **Sprint Review**: The Sprint Review is held at the end of each sprint to inspect the increment of work completed and gather feedback from stakeholders. The Scrum Team demonstrates the features or functionality implemented during the sprint, and stakeholders provide feedback, which informs future development.
4. **Sprint Retrospective**: The Sprint Retrospective is a meeting held at the end of each sprint to reflect on the team's processes and identify opportunities for improvement. The Scrum Team discusses what went well, what could be improved, and action items to address identified issues in future sprints.
5. **Backlog Refinement (Grooming)**: Backlog Refinement is an ongoing activity where the Scrum Team collaboratively refines and prioritizes items in the product backlog. It involves breaking down user stories, estimating effort, updating acceptance criteria, and ensuring the backlog is ready for future sprints.
6. Sprint Defination its time period?

Answer:

In Agile and Scrum methodologies, a sprint is a time-boxed iteration during which a development team works to deliver a potentially shippable increment of product functionality.

**Definition of a Sprint**: A sprint is a fixed time period, typically ranging from one to four weeks, during which the Scrum Team collaborates to complete a set of user stories or tasks from the product backlog. It represents a focused effort to achieve a specific goal or deliverable within a defined timeframe.

**Key Characteristics of a Sprint**:

1. **Time-Boxed**: Sprints have a fixed duration, meaning that they start on a specific date and end on a predetermined date, regardless of whether all planned work is completed.
2. **Focused**: Each sprint has a clear goal or objective, known as the sprint goal, which guides the team's work and defines what will be delivered by the end of the sprint.
3. **Iterative**: Sprints follow an iterative development approach, with the team delivering incremental value at the end of each sprint. This allows for frequent feedback, adaptation, and continuous improvement.
4. **Cross-Functional**: The Scrum Team, which includes developers, testers, designers, and other relevant roles, collaborates closely during the sprint to deliver a cohesive and integrated increment of work.
5. **Self-Organizing**: The Scrum Team is self-organizing and empowered to determine how to accomplish the sprint goal. They plan, execute, and manage their work autonomously, with guidance from the Product Owner and Scrum Master.

**Duration of a Sprint**: The duration of a sprint is determined by the Scrum Team and may vary depending on factors such as the complexity of the work, the team's capacity, and the organization's preferences. Common sprint durations include:

* 1-week sprint
* 2-week sprint (most common)
* 3-week sprint
* 4-week sprint

The choice of sprint duration should balance the need for frequent feedback and adaptability with the overhead of sprint planning and coordination. Shorter sprints allow for faster feedback and more frequent releases but may require more frequent planning and coordination. Longer sprints provide more time for development but may delay feedback and increase the risk of scope creep.

Overall, the sprint duration should be chosen based on the team's capacity, the nature of the work, and the organization's goals and constraints. It should allow the team to deliver valuable increments of work consistently and sustainably over time.

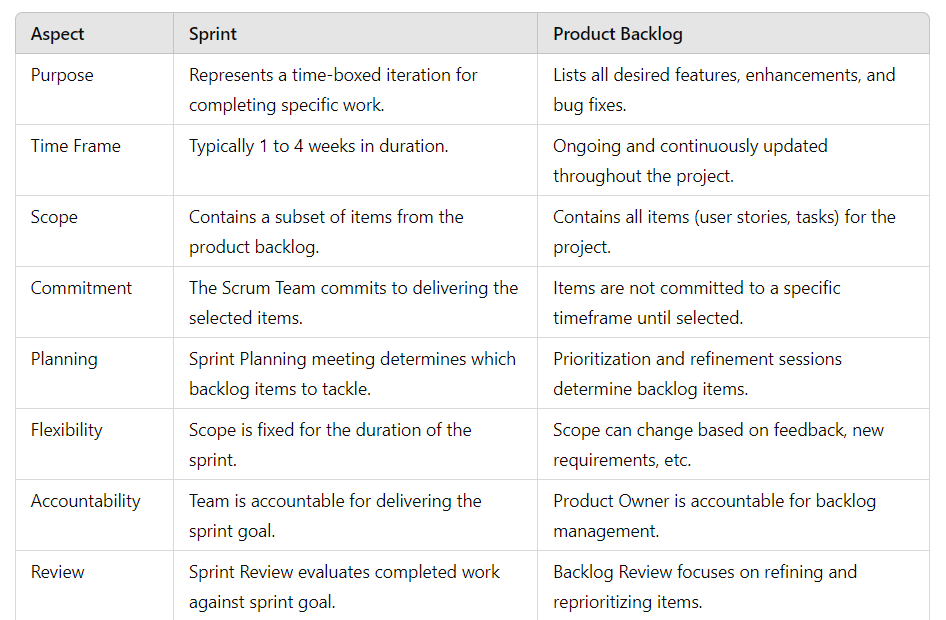
1. A time required to perform particular task in sprint?

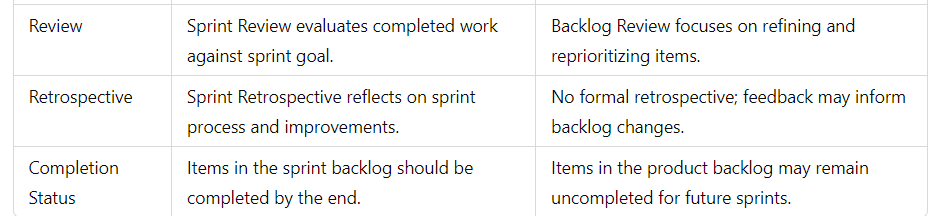
Answer:

The time required to perform a particular task within a sprint can vary depending on factors such as the complexity of the task, the skills and experience of the team members, and any dependencies or blockers that may arise. However, in Scrum, tasks are typically estimated in hours rather than days to facilitate more accurate planning and tracking. Here are some guidelines for estimating the time required for tasks within a sprint:

1. **Use Relative Sizing**: Instead of estimating tasks in absolute time (e.g., hours or days), use relative sizing techniques such as story points or t-shirt sizes (e.g., small, medium, large). This allows the team to compare the effort required for different tasks without getting bogged down in precise time estimates.
2. **Break Down Tasks**: Break down larger user stories or features into smaller, more manageable tasks that can be completed within a single sprint. This makes it easier to estimate and track progress and reduces the risk of incomplete work at the end of the sprint.
3. **Involve the Team**: Task estimation is a collaborative effort that involves the entire Scrum Team, including developers, testers, designers, and other relevant roles. Encourage open communication and discussion to ensure that everyone understands the task and agrees on the estimated effort required.
4. **Use Historical Data**: Leverage historical data from previous sprints to inform task estimates. Review past performance and velocity to identify patterns and trends that can help inform future estimations.
5. **Consider Dependencies**: Take into account any dependencies or blockers that may impact the time required to complete a task. If a task is dependent on another task or external factors, factor this into the estimation process.
6. **Use Time-Boxing**: Limit the time allocated for task estimation to prevent over-analysis and ensure that the team focuses on delivering value. Consider using time-boxing techniques such as Planning Poker or Fist of Five to facilitate quick and efficient estimation.
7. **Refine Estimates as Needed**: Task estimates are not set in stone and may need to be adjusted as the sprint progresses and more information becomes available. Encourage the team to revisit and refine estimates as needed to ensure that they remain accurate and relevant.
8. Chart difference between sprint and product backlog?

Answer:





20. Estimations of Agile?

Answer:

In Agile methodologies, estimations play a crucial role in planning and managing the delivery of work. Estimations help teams understand the effort required to complete tasks, prioritize work items effectively, and make informed decisions about resource allocation and project timelines. Here are some common approaches to estimations in Agile:

1. **Story Points**: Story points are a relative measure of the effort required to complete a user story or task compared to other stories or tasks. Instead of estimating in hours or days, teams assign story points based on factors such as complexity, effort, and uncertainty. Story points are often determined through collaborative estimation techniques like Planning Poker, where team members discuss and vote on the relative size of each story. Story points allow teams to estimate work quickly and account for uncertainty and complexity more effectively than absolute time estimates.
2. **Ideal Days**: Some Agile teams prefer to estimate work in ideal days, which represent the amount of effort required to complete a task assuming ideal conditions and no interruptions. Ideal days are a more concrete measure of time compared to story points and can be useful for teams that need to provide more accurate estimates or track progress against specific timeframes. However, ideal days may not account for variability and uncertainty as effectively as story points.
3. **T-Shirt Sizing**: T-shirt sizing is a simple estimation technique where tasks or user stories are categorized into size buckets like small, medium, large, or extra-large based on their complexity, effort, or scope. T-shirt sizing provides a quick and easy way to estimate work without getting bogged down in precise time estimates. It can be particularly useful for high-level planning and prioritization.
4. **Relative Sizing**: Relative sizing involves comparing tasks or user stories to each other and estimating their size relative to one another. This approach helps teams quickly prioritize work and identify dependencies without getting caught up in precise estimates. Relative sizing can be done using techniques like ordering tasks from smallest to largest or grouping similar tasks together.
5. **Cycle Time**: Cycle time is the amount of time it takes for a task or user story to move from start to finish, including both active and idle time. Agile teams may use cycle time as a measure of performance and productivity, allowing them to identify bottlenecks, optimize workflows, and make data-driven decisions about resource allocation and process improvement.
6. **Velocity**: Velocity is a measure of the amount of work a team can complete in a sprint or iteration. It represents the average number of story points, ideal days, or other units of work completed by the team in each sprint. Velocity helps teams predict how much work they can realistically commit to in future sprints and adjust their plans accordingly.

21. Techniques of agile?

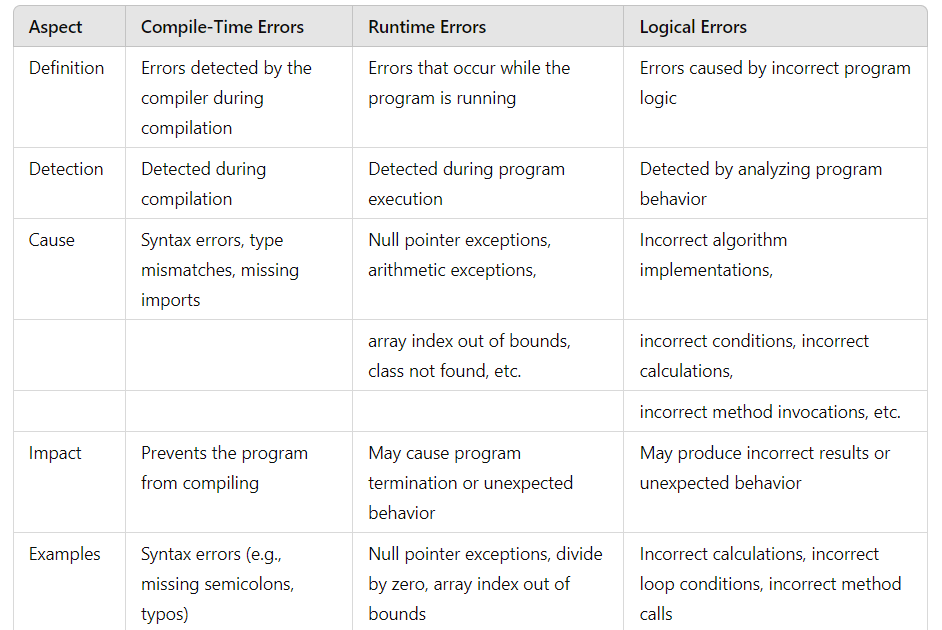
Answer:

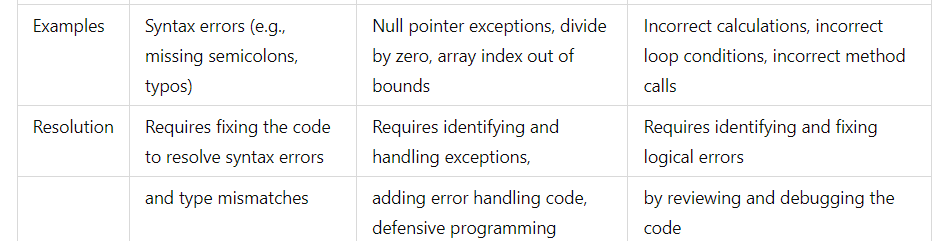
Agile is an iterative and flexible approach to software development that emphasizes collaboration, adaptability, and delivering value to customers. There are various techniques and practices associated with Agile methodologies that help teams effectively plan, execute, and deliver work. Here are some common Agile techniques:

1. **Scrum**: Scrum is one of the most widely used Agile frameworks, which provides a structured approach to managing iterative development. Key practices in Scrum include:
   * Sprint Planning: Planning sessions at the beginning of each sprint to determine the work to be completed.
   * Daily Stand-up: Short daily meetings for the team to synchronize, discuss progress, and identify any impediments.
   * Sprint Review: Meetings at the end of each sprint to review and demonstrate the completed work to stakeholders.
   * Sprint Retrospective: Meetings at the end of each sprint to reflect on the team's performance and identify areas for improvement.
2. **Kanban**: Kanban is an Agile methodology focused on visualizing and managing workflow. Key practices in Kanban include:
   * Kanban Board: Visual representation of work items and their status, typically using columns and cards.
   * Work In Progress (WIP) Limits: Constraints placed on the number of work items allowed in each stage of the workflow to optimize flow and prevent bottlenecks.
   * Continuous Improvement: Encouraging teams to continuously identify and implement improvements to their process.
3. **Extreme Programming (XP)**: Extreme Programming is an Agile methodology focused on technical excellence and rapid feedback. Key practices in XP include:
   * Test-Driven Development (TDD): Writing automated tests before writing the code to ensure test coverage and drive design.
   * Pair Programming: Collaborative programming technique where two developers work together at the same computer.
   * Continuous Integration: Integrating code changes frequently and automatically, typically multiple times per day.
4. **Lean Software Development**: Lean principles emphasize delivering value to customers and minimizing waste. Key practices in Lean Software Development include:
   * Value Stream Mapping: Identifying and optimizing the value stream to eliminate waste and improve efficiency.
   * Just-In-Time (JIT) Delivery: Delivering work items just in time to meet customer demand and minimize inventory.
   * Kaizen: Continuous improvement through small, incremental changes driven by the team.
5. **Feature-Driven Development (FDD)**: Feature-Driven Development is an Agile methodology focused on delivering features incrementally. Key practices in FDD include:
   * Feature List: Creating a list of features to be developed, prioritized based on business value and customer need.
   * Feature Teams: Organizing development teams around specific features or components to facilitate focus and ownership.
   * Regular Builds: Delivering working software at regular intervals, typically every few weeks.

22. Chart difference between types of errors in java?

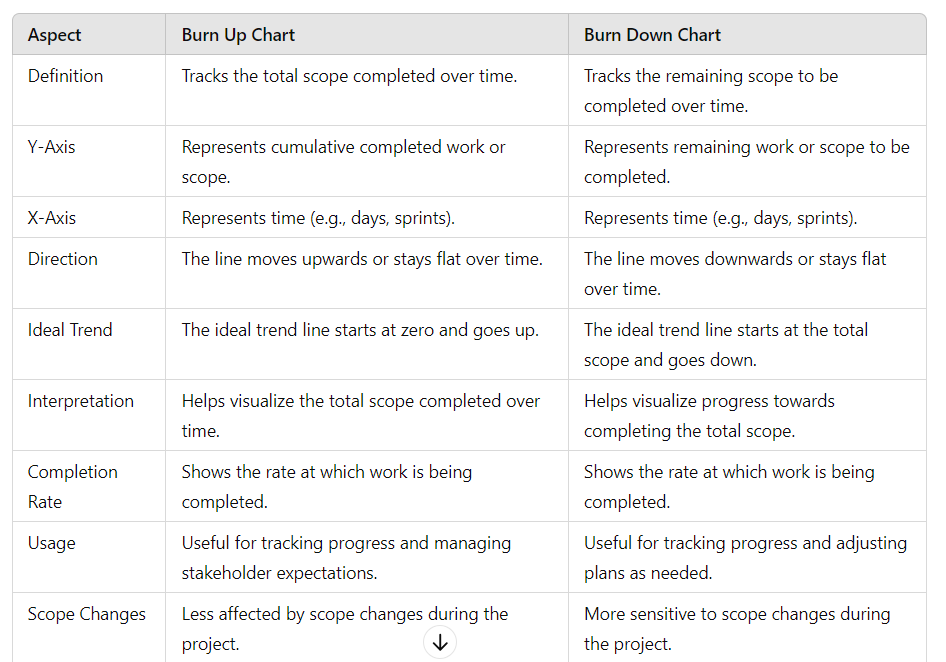
Answer:





23. Burn up and burn down difference?

Answer:



24. Velocity, remaining and completed task in Jira?

Answer:

In Jira, Velocity, Remaining, and Completed tasks are metrics used to track the progress of Agile teams and their work. Here's a brief explanation of each:

1. **Velocity**:
   * Velocity is a measure of the amount of work a team completes in a sprint.
   * It represents the total number of story points, tasks, or other units of work completed by the team in each sprint.
   * Velocity is calculated by summing up the estimates of all completed user stories, tasks, or other work items at the end of each sprint.
   * Velocity helps teams predict how much work they can realistically commit to in future sprints and adjust their plans accordingly.
2. **Remaining**:
   * Remaining refers to the amount of work that still needs to be completed in a sprint or project.
   * It represents the sum of estimates for all unfinished user stories, tasks, or other work items.
   * Remaining work is often tracked throughout the sprint or project to monitor progress, identify potential bottlenecks, and make adjustments as needed.
3. **Completed**:
   * Completed refers to the amount of work that has been finished and delivered by the team.
   * It represents the sum of estimates for all completed user stories, tasks, or other work items.
   * Completed work is typically reviewed and accepted by stakeholders before being considered done.

In Jira, these metrics are often visualized using various reports and dashboards to provide visibility into team performance, progress, and capacity. For example, a Burndown Chart may show remaining work over time, while a Velocity Chart may show completed work across multiple sprints. These metrics help Agile teams make data-driven decisions, optimize their workflow, and deliver value to customers more effectively.

Top of Form

Bottom of Form