Architectural Understanding

& Framework Application

1.1) Framework Design (Conceptual)

The architecture would be built using a behaviour-driven framework. This makes it easy to automate test cases by putting each scenario in the given, when, then format. Specific key components of the framework include:

1. **Feature files** - allows test cases to be written in plain english before code implementation, this helps to make sure test cases are fully understood which reduces chances of mistakes
2. **Step files** - stored in step folders, this is where the test cases will be implemented using code. Methods can be called according to the steps outlined in the feature file which allows for sequential execution of test cases.
3. **Helper files** - this is where the core code is written. This includes a base class with reusable methods. This is good because it applies principles like inheritance, polymorphism and abstraction.
4. **Utility files** - these are files that use additional code like database connections for external test data.
5. **Logger files** - Customizable logger files that can be useful to log certain responses that can be used as test evidence.
6. **Report functionality** - this is a component that will allow the executed automation code to be presented as detailed reports. These reports are used as test evidence and are more visually appealing than logs.

Flow: Feature files are called by Step files. Helper files are called by the step files. Utility files are called by helper files leveraging concepts like inheritance

1.1.2) How I would handle:
1. Asynchronous API Interactions
API requests and interaction is an integral aspect of a testing framework. Using the framework outlined in section 1.1.1. The first thing I would do is set up a feature file. E.g.

- Given “Ï want to validate a successful GET request
- When “I make a GET request to endpoint X”
- Then “I should receive a 200 status code and a JSON response

Then I would create a helper class for the API endpoint I am testing. In that helper class I would define a method for a GET request. It can be expanded to include POST, PUT, DELETE. In the GET request method I would then define the request, this can be done using several open source libraries. The parameters of the request can be decoupled using variables in case multiple requests to the same endpoint using different parameters is required. The method should include a return statement that returns the valid JSON and status code. Once the method has been completed it can be called in the step file. The request is now automated and can be executed to receive the response. The async aspect can be handled using the helper files with async libraries for non-blocking GET requests, with responses awaited only when needed in steps
2. Dynamic Data Generation and Management
Data generation and management should be another foundational aspect to an automation framework. This can be done in a few ways. The way I am familiar with is to leverage the benefits of the behavior driven framework. In the feature files, test cases can be tagged with a test case id. This test case id will act as a primary key. In another external datasource, like a database or an excel spreadsheet the test data is stored. The rows of test data then have a unique id in the form of the test case id as mentioned above. Once the data has been entered into the data source connectors can be written in utility files and implemented into the specific helper files. This allows test case specific data to be used as test scenarios are executed. This type of data management is highly beneficial because external data sources can be scaled better than in memory data. Data generation can be implemented through utility files which use libraries that can simulate users, emails and names.

3. Cross Browser and Cross platform testing capabilities
Cross testing or compatibility testing is an important aspect when it comes to testing. Fortunately it is easy to include in the test automation framework as outlined above. When executing frontend tests or browser tests a utility file for the specific browser is created.Eg. chrome_utility, edge_utility, firefox_utility. The utility files contain code that activates the specific browser drivers. These drivers are then used to perform frontend automation tasks like clicking and scrolling. The utilities are implemented in the helper file of the4. specific scenario being executed. Because the BDD framework allows sequential step execution it is possible to execute a compatibility test in one test run. Cross platform or device testing like mobile can be implemented using mobile specific automation utilities like appium in the utilities. Scalable test execution and reporting This is one of the benefits of running a BDD automation framework. Because the structure of this kind of framework is based on different automation files being created. A directory of test automation scripts can be created. This results in the solution to scalability. If more tests are needed then more feature files can be created. This does not affect any of the past feature files and the framework has the ability to execute specific files at a time. In terms of reporting an open source reporting tool can be used, this tool can be integrated into the framework using utility files and reports can be generated based on test files executed. Reports generated aggregate the test logs and. A runner utility can also be added to run tests in parallel which brings down execution time.

1.2) State Management & Data Integrity

1.2.1) strategy for ensuring data integrity and verifying state transitions throughout the test
Using the BDD framework following a Given-When-Then structure which defines test conditions at the beginning and user actions in the middle and expected results at the end. The format enables smooth tracking of pages and API state modifications for validation purposes.

To ensure data integrity I would start every test with a fresh/ clean environment by either resetting the test data or implementing API calls to establish necessary states. The test verifies both user interface modifications and backend state through a combination of UI interactions with API validations and database checks.

E.g. Users log in and update their profile. After a profile update,I would not only check confirmation messages on the UI but verify the UI confirmation message while simultaneously checking the API or database to confirm the update persisted. The system achieves accurate results through multiple validation checks.

In multiple-step workflows I would build scenario chains with attention to detail to ensure each step output transforms into the next step’s input. The system validates both data consistency and transitions through strategic placement of assertions at critical points.

In the automation I would use clear/ precise logging and tagging features which enables fast failure isolation and help determine when the state started to diverge. Lastly I would collaborate with developers and BAs to ensure test scenarios accurately reflect business logic. Which will keep the automation reliable and meaningful.

1.2.2) how you would handle scenarios where data dependencies exist between different test cases
My objective is to make sure that each test maintains its independence and dependability while adhering to the necessary flow in situations where there are data dependencies between test cases.
I use a BDD framework to create Given-When-Then scenarios that explicitly outline the necessary prerequisites and anticipated results. I usually handle the data setup for dependent scenarios by using shared hooks or utility functions that use direct database operations or APIs to create the required context. In this manner, I can avoid flakiness by not depending on the order in which tests are executed.

When a scenario must inevitably come after another (for example, completing registration before testing login), I either:

- Consolidate both into a single end-to-end scenario when it makes business sense, or
- Use fixtures or API calls to replicate the precondition state without actually running the prior test.

Additionally, I use naming conventions and test tags to logically group related tests, but execution remains isolated. To ensure traceability and prevent overwriting across tests, any shared data is stored in context-specific variables. In order to prevent adverse effects, I lastly make sure that all test data is cleaned up after execution. Particularly in CI pipelines, this preserves test reliability. I maintain data-dependent scenarios stable without compromising test maintainability or clarity by meticulously controlling setup, teardown, and mocking as needed.

1.2.3) Test data management through test run
I treat test data as an essential component of my given steps using a BDD framework, making sure that everything is set up correctly before starting any test actions.

- Example 1 - User Account Creation: I don't rely on users who already exist for situations like login or profile updates. Instead, during the test setup, I make a new user with particular attributes (like role and permissions) using an API call that runs in the background. This eliminates reliance on static data and guarantees consistency.
- Example 2-Dynamic Test Data: I use helper methods to create dynamic test data during the test run for scenarios that call for unique values (such as emails, IDs, or order numbers). This guarantees that tests can run concurrently and prevents conflicts.
- Example 3: Data Sharing Among Steps I keep important values like the order ID in a shared context or environment object in end-to-end processes like "create order > approve > dispatch." This eliminates the need for hardcoding and enables the data to be reused across steps.
- Data Cleanup: I use DB scripts or APIs to eliminate created records following tests. This keeps the data state consistent for the following run and helps prevent environmental pollution.
- Test Environments: I also manage test data differently depending on the environment, using seeded data in staging, carefully selected data sets in pre-production, and mocking in development.

It ensures stability, adaptability, and confidence in my test results by exercising control over the creation, use, and cleanup of test data.

1.3) Complex UI Interactions

1.3.1) Explain how you would automate testing for these complex UI elements.
In order to keep scenarios readable, reusable, and maintainable, I use the BDD framework in a layered manner for highly interactive UI elements like drag-and-drop, canvas manipulation, or real-time updates.
1. **Drag-and-drop** (for example, reordering items or a Kanban board): In the format of Given-When-Then

- Given: the user is on the task board,
- When: they drag task A to Column B,
- Then: Task A ought to show up in Column B after they drag it there.

I use automation tools that support native drag-and-drop events, such as Playwright or Cypress, or, if necessary, I can simulate them using custom JavaScript. Additionally, I use an API or state check to validate the DOM change and initiate any backend validations.

2. **Canvas Manipulation** (charts, drawing tools, etc.): Direct DOM assertions are insufficient for components that are canvas-based. So, I would use:

- Use JS events to interact with the canvas (e.g., triggering mousedown, mousemove, and mouseup at specific coordinates).
- Verify side-effects such as logs, coordinates, or object counts, or record the canvas state before and after.

E.g, when the user draws a rectangle on an empty canvas, one shape should be added to the object list.
3. Real-time updates (such as sockets and dashboards): When feasible, I use WebSocket triggers or mocked data to mimic real-time updates. Then make sure the user interface updates appropriately within a specified time frame by using polling or wait conditions. E.g. if the dashboard is open and a new transaction is received, the new entry should appear on the dashboard in five seconds. I always use retries, waits, and state validations in addition to visual checks to make sure the test stays stable.

I can consistently test even the most challenging UI elements by combining BDD with smart selectors, API hooks, and browser events.

1.3.2) Describe how you would handle synchronization issues and ensure reliable test execution
I would focus on intelligent waiting strategies, robust selectors, and meaningful assertions—all within the framework of the BDD framework—to address synchronisation issues and maintain dependable test execution, particularly in contemporary web apps with async behaviour.
1. **Instead of using hard waits, use intelligent wait conditions**: staying away from static waits and sleep(). Rather, employ conditional waits such as:

- waitForResponse
- waitForNetworkIdle
- waitForElementToBeVisible

For example:

- Given: A user submits a form,
- When: the system processes the request
- Then: the confirmation message should show up

Instead of assuming a delay in this case, I'll wait for the success message or API response associated with that action.

2. **Await backend actions or API responses**: I use tools like Cypress intercepts or Playwright route listeners to hook into the API call if a UI change requires data loading (such as charts or tables). I don't start DOM validations until I receive a successful response.

3. **DOM stability checks**: I make sure the element is stable and interactable in addition to being present. For instance, waiting for a button to be visible and enabled outside of the DOM.

4. **Retry logic for shaky areas:** I incorporate retry logic in areas like real-time updates or animations at the step-definition level (for example, by using retryUntilSuccess).

5. **Keep BDD scenarios focused:** To minimise complexity and the possibility of race conditions across steps, each BDD scenario is made to test a single behaviour at a time.

By combining these techniques even in dynamic, async-heavy environments, I maintain tests' speed and dependability.

1.3.3) Discuss the challenges of testing real-time data updates and provide possible solutions.
Unpredictable timing, network latency, and dependence on WebSockets or push mechanisms make testing real-time data updates challenging. The following are the primary difficulties I've faced and how I address them using a BDD framework:
1. **Timing unpredictability:** Since real-time updates don't follow a set timetable, tests that anticipate instantaneous changes may become erratic. Solution: To continuously check for the anticipated change within a timeout, I employ polling mechanisms or dynamic waits.

- Given: the dashboard is open,
- When: a new order is placed,
- Then: the order list should update within 10 seconds.

In this case, I claim that the change happens within a window rather than immediately.

2. **Inability to control data flow:** Real-time data is frequently sourced from outside sources or initiated by other systems.
Solution: Whenever feasible, I use test WebSocket messages or backend API calls that mimic incoming data to mimic or simulate the real-time triggers. This maintains the test's consistency and dependability.

3. **Partial updates or UI delays:** Occasionally, the UI misses or lags behind backend data updates. Solution: I verify that the data arrived through an API or socket and that the user interface (UI) reflects it by validating both the frontend and the backend. This two-layer check aids in identifying sync problems.

4. **Instability of the test environment**: In shared environments, real-time systems are more difficult to manage.
Solution: To keep control over data flow and minimise flakiness, I conduct these tests in isolated environments using dev sockets or mock servers.

I can confidently test real-time updates while maintaining clear and understandable BDD tests by combining smart waits, controlled data injection, and backend validation.

Part 2: Coding & Problem-Solving

2.1) Algorithmic Challenge (Focus on Logic)

2.1.1) function that analyses log files:
Created a .py file with the function/class

2.2) API Testing with Dynamic Data & Edge Cases

2.2.1) Test Cases:
Created an excel table containing the test cases

2.3) Debugging & Root Cause Analysis

2.3.1) systematic approach to debugging the issue.
I use a methodical approach when troubleshooting a problem to make sure I find the source quickly and reduce the team back and forth.
1. **Recreate the Problem Consistently:** I start by attempting to replicate the problem in a controlled setting, preferably using identical data and environmental circumstances. I repeat the test several times and look for trends in the failures if it's erratic or intermittent.

2. **Gather Evidence:** I gather comprehensive evidence, such as stack traces, error logs, screenshots, HAR files (if they pertain to UI/API), and test run specifics. I also examine the raw test reports and logs from CI pipelines to record timestamps, response payloads, and browser/network behaviour for automated failures.

3. **Isolate Variables:** In order to determine what is causing the behaviour, I start by adjusting one variable at a time, including environment data, user roles, timing, and test data. For instance, I look for token-related, payload-specific, or backend timing issues if it's an API problem. This makes it easier to determine if the problem is a real bug, a test flaw, or a data problem.

4. **Collaborate and Communicate**: After determining probable causes, I provide the development team with a summary of the results (supported by logs and test run evidence). I take care to specify the precise procedures to replicate the problem, the expected and actual behaviour, and the particular circumstances in which it arises.

This procedure makes sure that problems are not only noted but also comprehended, which facilitates quicker fixes and improved teamwork.

2.3.2) Explain the tools and techniques you would use to gather information and identify the root cause.
I combine frontend, backend, and test-level tools to efficiently determine the underlying cause of a failure. This is my usual toolkit, along with how I use each one:
1. **Test Automation Artefacts (Selenium + BDD Framework):** I examine the outcomes of the test automation, such as screenshots, browser console logs, and detailed traces from the test reports (like Cucumber HTML reports). Screenshots taken at the moment of failure are frequently the most efficient method to visually comprehend what went wrong if the issue is UI-related.

2. **CI/CD Logs (Jenkins/GitLab):** Pipeline logs can be used to determine whether an issue is environment-related (such as timeouts, missing test data, or non-running services). Contextual information like timestamps, environment variables, and any setup or teardown issues are provided by these logs.

3. **Kibana or ELK Stack:** For backend log analysis, I prefer Kibana. I look for error stacks, trace logs, and correlation IDs that happened during the test window. It assists in tracking down the request from the UI/API to the database or microservices layer.

4. **Postman or other API testing tools:** To confirm if the problem is with the test data, authentication, or the endpoint itself, I manually initiate API calls for the same scenarios using Postman. I examine payloads, response times, and status codes in order to compare them to automated outcomes.

5. **Browser Dev Tools (for UI issues):** I launch DevTools to examine network requests, console errors, and DOM changes if the failure involves frontend behaviour. This is particularly useful for troubleshooting JavaScript errors or asynchronous user interface problems.

6. **Database/Query Tools (such as DBeaver):** I check for data consistency and make sure that records are present or updated as expected following a test, if necessary. This makes it easier to determine whether the problem is with the data layer, API layer, or frontend.

I can track problems throughout the entire stack, from test scripts to system behaviour, by combining these tools and approaches, and I can give developers concise, useful insights.

2.3.3) Provide examples of log analysis, network monitoring, and code inspection techniques.
1. **Log Analysis:** To find errors and stack traces, I start by looking over application logs (through Kibana, Splunk, or local server logs). For instance, I look through logs using a correlation ID or timestamp to determine which service threw the exception if an API test fails with a 500 error. I search for validation errors, DB constraint violations, and NullPointerExceptions. Additionally, log levels (INFO, WARN, and ERROR) aid in identifying the location of the breakdown, particularly when logs contain user-specific context or request/response payloads.

2. **Network Monitoring:** During manual runs, I examine the Network tab using Chrome DevTools or Fiddler for frontend/API interactions. This displays HTTP status codes, timings, response payloads, and request headers. For example, I verify whether WebSocket/API calls are being made, whether they are delayed, or whether CORS or authorisation issues are the cause if real-time updates don't appear on the user interface. During UI automation tests, I also keep an eye on XHR calls to make sure backend calls were initiated.

3. **Code inspection techniques:** I examine pertinent git commits to search for recent changes in the affected module if the problem appears to be logic-based or cannot be replicated from the UI/API alone. I track changes line by line using git blame. In the IDE, I also execute debug-mode unit tests, inspecting variable states by setting breakpoints around failing logic. For instance, an unhandled edge case in conditional logic or an off-by-one error in a loop could cause a test to fail.

I can get past superficial errors and explore the underlying cause, whether it's in the client, server, or code logic, by combining these strategies.

2.3.4) Explain how you would create a robust report to give to the development team.
To cut down on time spent on back-and-forth clarification, I try to make bug reports as clear, succinct, and actionable as possible. I format a solid defect report as follows:

1. Title: A concise and unambiguous title that encapsulates the problem, such as "[UI] Submit button unresponsive after selecting file (Chrome only)".

2. Environment Details: I include details about the test environment, including the OS, browser version, environment (QA/UAT/Prod), test data used, and, if relevant, the automation run ID. This aids in accurately reproducing the problem.

3. How to Reproduce: A detailed explanation of how to cause the problem. I include any setup or prerequisites (e.g., logged in as a user with specific permissions) and keep it brief and bullet-pointed.

4. Actual vs. Expected Behaviour: Clearly explain the differences between what actually occurred and what was anticipated. The developer can quickly see what's broken and how it differs from expected functionality thanks to this.

5. Supporting Evidence: To provide visual support for the report, include screenshots, console logs, stack traces, video recordings, or test automation links (from CI pipelines). In order to facilitate log cross-referencing, I also include timestamps.

6. Impact & Severity: I specify the impact on the user or test coverage as well as the severity (blocker, critical, minor, etc.). "This issue blocks regression for checkout flow on all browsers," for example.

7. Extra Notes (Optional): To expedite triage and resolution, I include any workarounds or root cause analyses that I have already completed.

This format guarantees that developers can comprehend, replicate, and fix problems as soon as possible.

Part 3: Critical Thinking & Design

3.1) Test Strategy for Evolving Systems

3.1.1) Describe your approach to maintaining a stable and effective test automation suite in this dynamic environment.
My goal is to create an automation suite that is robust, scalable, and simple to maintain in a system that is constantly changing in terms of both user interface and API. This is how I go about it:
1. Give Test Coverage Priority Strategically, I focus my automation efforts on high-risk domains, essential APIs that are unlikely to change, and core user journeys. I use exploratory or manual testing more in areas that are unstable until the design stabilises.

2. Employ a Modular and Layered Framework: I use a BDD framework (such as Cucumber with Selenium/RestAssured) to organise the automation suite. This framework involves loose coupling between page objects, API clients, and data layers. This enables me to update everything in one location without affecting any of the tests.

3. Implement Smart Locators & APIs: I avoid brittle XPath selectors for UI automation in favour of stable attributes, dynamic waits, and flakiness-handling tools like TestCafe or Playwright. I make data-driven tests for APIs and use schema validation to verify contracts in order to version-proof them.

4. Continuous Integration & Feedback: In order to obtain prompt feedback, I incorporate the test suite into the CI pipeline. Every day, failed tests are categorised. In order to prevent noise in regression results, I also mark unstable tests as "quarantine" and report them separately.

5. Collaboration with Developers: To receive early notice of game-changing changes, I communicate with developers through Slack or stand-ups. In order to predict test impacts, I also go over UI design tickets or API schema beforehand.

6. Frequent Refactoring & Review: I set aside time for each sprint to update selectors or data mappings, deprecate unnecessary code, and clean up outdated tests. In addition, I constantly strive to enhance test stability by reviewing faulty test patterns.

This strategy guarantees that the automation suite will continue to be a dependable safety net rather than a maintenance burden even as the system undergoes rapid change.

3.1.2) Discuss the importance of testability and collaboration with developers.
Delivering dependable and maintainable software depends heavily on testability, which is a shared responsibility. I passionately support creating systems that are test-friendly from the ground up in my capacity as a QA engineer.

The importance of testability We can reduce the amount of time spent debugging unstable or flaky scenarios, write clear and reusable automated tests, and detect bugs early with a highly testable system. It is more difficult to consistently validate behaviours in systems that are not testable, such as those that lack API response validation or missing element identifiers in the user interface. This raises the possibility of regressions going unnoticed while also slowing down testing.

Enabling testability requires cooperation with developers. I collaborate closely with developers during design or grooming sessions at the start of a sprint to bring up issues like:

- Is it possible for this UI element to have a unique ID for reliable automation?
- Will error messages or status codes from the API be consistent?
- Is it possible to make test hooks, toggle flags, or logs visible for simpler verification?

In order to help developers understand failures, I also work in pairs with them during bug triage or test impact analysis. Occasionally, I even help develop ideas for unit or integration tests.

Early and frequent collaboration allows us to have an impact on implementation choices that enhance testability, such as incorporating logging, creating simulated endpoints, or planning APIs with validation in mind. Better software and quicker feedback loops are the results of QA and developers having a common understanding.

3.1.3) How will you handle versioning of test scripts, and test data?
1. Versioning Test Scripts: I use Git to manage test scripts in addition to application code, preferably in the same monorepo or an independent automation repository that is connected. Relevant test updates are included in every feature branch, and I use naming conventions or tags (such as v1.2-api-tests and feature/login-tests) to match test versions with matching application versions.

I keep distinct test suites or directories for every version of the application when it is versioned (for example, v1 and v2 APIs or UI modules). Only the active tests are included in CI execution; deprecated tests are either flagged or archived. This guarantees that current coverage won't be affected by outdated test logic.

2. Managing Test Data Versioning: I use external files or data providers (such as JSON, YAML, Excel, or DB fixtures) to isolate test data from test logic. To guarantee consistency across environments, I version the test data in addition to the test scripts.

I use versioned folders like testdata/v1, testdata/v2, to support backward compatibility when schema changes impact test data (e.g., new fields, removed attributes). I verify inputs against JSON schemas for APIs and update these schemas with version-specific modifications.

3. Environment-Specific Data Management: I use test data injection or environment-based configuration files to support several test environments (QA, UAT, and staging). This eliminates the need for manual data changes and enables the same tests to run in various environments.

Even as the system changes, I guarantee traceability, rollback capability, and seamless CI/CD test execution by versioning both scripts and data in accordance with application releases.

3.2) Performance and Security Integration

3.2.1) Explain how you would integrate performance and security testing into your automated test suite.
I can incorporate performance and security into the automation pipeline as part of shift-left testing to make sure they are not considered afterthoughts. I approach both areas as follows:
Integration of Performance Testing:

- Use of Lightweight Tools in CI: To perform performance checks on important APIs, I incorporate tools like Locust, k6, or JMeter into the CI/CD pipeline. These tests measure error rates, throughput, and response times while simulating multiple users.
- Threshold-Based Alerts: I set up thresholds in the pipeline and specify performance baselines, such as "response time must be < 500ms." The pipeline indicates a performance regression if these are violated.
- Scenario Reuse from Functional Tests: I base performance testing on core BDD scenarios. To test how well they function under load, I, for example, repurpose login or checkout steps in Locust or K6 scripts.
- Execute Pre-Prod load tests: Particularly in staging or pre-prod, where infrastructure reflects production, performance-intensive tests are planned for nightly runs or pre-release smoke suites.

Integration of Security Testing:

- Static Code & Dependency Scanning: To find known vulnerabilities in code and libraries, I incorporate tools like SonarQube, Snyk, and OWASP Dependency-Check into continuous integration.
- Automated API Security Tests: To check for common problems like injection errors, broken authentication, and CORS misconfigurations, I use tools like OWASP ZAP or Postman's security test assertions.
- Authentication/Authorization Checks in Test Cases: To make sure the right security controls are in place, I incorporate negative scenarios in functional and API tests, such as accessing endpoints with insufficient permissions or expired tokens.

Without delaying delivery, I make sure possible bottlenecks or vulnerabilities are identified before they reach production by automating performance and security checks early in the pipeline.

3.2.2) Describe the key metrics and tools you would use for each type of testing.
Testing performance (with Locust):
Locust - This Python-based, lightweight performance testing tool enables the simulation of multiple users and the writing of user behaviour in code.
Key Metrics:
1. Response Time (avg/min/max/percentiles), which gauges how long it takes to process each request. To identify edge case slowdowns, I keep an eye on the 95th and 99th percentiles.
2. Requests Per Second (RPS): This metric aids in gauging system capacity and throughput.
3. Failure Rate/Error Count: Indicates the number of unsuccessful requests (e.g., 500 errors, timeouts).
4. User Load: The quantity of simulated concurrent users and the way the system reacts to an increase in load.
5. Latency vs. Response Time: This is particularly helpful when backend services or databases are the source of lag.

Example: I set up Locust in CI to mimic actual usage patterns (login → browse → checkout, for example) and initiate a test run during pre-release cycles or nightly builds. For long-term tracking, the results are either pushed into Grafana/InfluxDB dashboards or published as HTML reports.

Testing for security:

- OWASP ZAP (Zed Attack Proxy): For automated security checks, particularly on web apps and APIs.
- Snyk/OWASP Dependency-Check: To find security flaws in third-party dependencies and libraries.
- Postman/Newman: For incorporating simple security test assertions based on authentication into API flows.

Key Metrics:

1. Vulnerabilities by Severity (High, Medium, and Low): Monitors serious flaws such as XSS, SQL injection, and unsecured authentication.
2. Dependency Vulnerability Score: derived from the NVD's known CVEs.
3. Authentication & Authorisation Gaps: Indicates whether token validation is inadequate or unauthorised access is feasible.
4. Security Regression Count: Tracks the number of fresh security flaws that have surfaced since the last scan.

Example: Following functional testing, I incorporate ZAP scans as a CI/CD step. The pipeline fails for any medium-to-high-severity problems. Additionally, I plan weekly Snyk dependency scans for both frontend and backend repositories.

3.2.3) Discuss the challenges of automating performance and security tests and provide possible solutions.

Challenges with Performance Test Automation:

1. Environment Restrictions: Production-like environments are required for performance testing. Results from load tests conducted in shared or constrained QA environments are frequently deceptive.
Solution: Perform performance tests using production-like configurations in staging/pre-production environments. To prevent conducting load tests in lower tiers, use environment tagging in your pipeline.
2. Managing Data While Under Load: Reused data can distort results, and large amounts of test data are required for meaningful simulation.
Solution: Use data seeding scripts to reset the environment prior to test runs, or create tests that dynamically create and destroy data.
3. Test Script Complexity: As flows expand, it can become more difficult to write realistic user behaviour in programs like Locust.
Solution: To define user tasks and reuse them in different contexts, use modular Python classes. Strike a balance between performance focus and functional realism.

Challenges with Automating Security Tests:
1. False Positives & Noise: Many low-severity or irrelevant results can be produced by automated scans (such as those from ZAP or Snyk), which can lead to alert fatigue.
Solution: Tailor scan settings to concentrate on important threats. Only fail builds on high- or medium-severity vulnerabilities by implementing filters or rules in CI.
2. Authentication Barriers: If security scans are unable to manage tokens, sessions, or multi-step logins, they frequently fail.
Solution: Make use of pre-saved cookies, token injection, or authenticated ZAP sessions. Automate token generation and login for APIs as part of the test flow.
3. Keeping Up with Dependency Changes: Known vulnerabilities may be reintroduced by routine library or package updates.
Solution: Use OWASP or Snyk tools to schedule weekly automated scans, then incorporate the results into your PR quality checks or backlog.