Part 1:

1. Flakiness Issues

What “flaky” means

A flaky test is one that sometimes passes and sometimes fails without any code changes.

Issues here:

- No explicit waits (dynamic dashboard loading)
  - Dashboard elements load dynamically.
  - The test immediately checks URL and UI elements.

- URL assertion happens before navigation completes
  - `page.url == "..."` may run before navigation completes.

- No handling of 2FA
  - Some users require two-factor authentication.
  - Test assumes instant login success.

- Hardcoded credentials
  - Credentials may change or behave differently in CI.

- Browser not run in headless mode (CI mismatch)
  - CI runs headless, locally it may be headed.
  - Screen size differences can affect UI behavior.

- No retry / timeout configuration
  - CI environments are slower and less predictable.

- Using `page.url ==` instead of `expect(page).to_have_url`

- `.all()` used without waiting for elements
  - `.locator(".project-card").all()` is called immediately.

- Same browser context reused without isolation

- Different tenant load times not handled
  - Different tenants have different loading times.

2. Root Causes (short explanations)

Why CI fails but local passes:

Example style:

> In CI/CD, execution is faster, network is slower, UI elements load asynchronously, and screen sizes differ. Without proper waits and assertions, tests become timing-dependent and flaky.
>
> Without proper synchronization and environment-aware handling, these differences make tests unreliable and flaky. These tests behave differently in CI/CD compared to local execution due to differences in execution speed, environment configuration, and resource availability.

1. **Faster execution in CI/CD**
    - CI pipelines execute steps very quickly.
    - Assertions may run before UI elements finish loading.

2. **Slower or unstable network in CI**
    - Dashboard and project data load asynchronously.
    - Network delays cause timing-related failures.

3. **Headless browser execution**
    - CI runs browsers in headless mode.
    - UI rendering timing differs from local headed mode.

4. **Different screen sizes and resolutions**
    - CI uses default viewport sizes.
    - Responsive layouts may behave differently.

5. **Parallel execution in CI**
    - Tests may run simultaneously.
    - Shared test data can cause interference.

6. **Environment differences**
    - CI often runs on different OS, browser versions, or hardware.
    - Local machine is usually more stable and predictable.

7. **Authentication flow variations**
    - 2FA may be enabled for some users in CI environments.
    - Test assumes a single-step login process.

8. **Tenant-specific performance differences**
    - Some tenants load more data.

- This increases page load time unpredictably.

3. Fixed Code (pseudocode)

The following changes focus on improving test stability by adding proper waits, handling dynamic loading, and making assertions more reliable.

Key improvements:

- Added explicit waits for navigation and elements
- Used Playwright's auto-waiting assertions
- Handled dynamic dashboard loading
- Improved tenant-specific validation
- Made the test CI-friendly

```
import pytest
from playwright.sync_api import sync_playwright, expect

def test_user_login():
    with sync_playwright() as p:
        browser = p.chromium.launch(headless=True)
        context = browser.new_context()
        page = context.new_page()

        page.goto("https://app.workflowpro.com/login")

        page.fill("#email", "admin@company1.com")
        page.fill("#password", "password123")
        page.click("#login-btn")

        # Wait for successful navigation
        page.wait_for_url("**/dashboard", timeout=10000)

        # Verify dashboard is loaded
        expect(page.locator(".welcome-message")).to_be_visible()

        browser.close()
```

Corrected Multi Tenant wali test

```python
def test_multi_tenant_access():
    with sync_playwright() as p:
        browser = p.chromium.launch(headless=True)
        context = browser.new_context()
        page = context.new_page()

        page.goto("https://app.workflowpro.com/login")

        page.fill("#email", "user@company2.com")
        page.fill("#password", "password123")
        page.click("#login-btn")

        page.wait_for_url("**/dashboard", timeout=10000)

        # Wait for project cards to load
        project_cards = page.locator(".project-card")
        expect(project_cards.first).to_be_visible()

        projects = project_cards.all()
        for project in projects:
            assert "Company2" in project.text_content()

        browser.close()
```

For users with 2FA enabled, the test would need to either mock the 2FA service, use a pre-authenticated session, or rely on test users with 2FA disabled.

PART 2: Test Framework Design

This framework design focuses on scalability, maintainability, and support for multi-tenant, multi-platform testing.

Suggested Folder structure

```
automation-framework/
│
├── tests/
│   ├── ui/
│   ├── api/
│   ├── mobile/
│   ├── integration/
│
├── pages/
│   ├── login_page.py
│   ├── dashboard_page.py
│   └── project_page.py
│
├── utils/
│   ├── config_loader.py
│   ├── auth_helper.py
│   ├── test_data.py
│   └── browserstack_helper.py
│
├── configs/
│   ├── dev.yaml
│   ├── staging.yaml
│   └── prod.yaml
│
├── fixtures/
│   ├── browser_fixture.py
│   └── api_client.py
│
├── requirements.txt
└── pytest.ini
```

briefly explain each folder in 1 line.

Short explanation (1–2 lines each)

- **tests/** – Contains all test cases grouped by type (UI, API, mobile, integration)
- **pages/** – Page Object Model to keep UI locators and actions separate
- **utils/** – Common utilities such as authentication, config handling, and helpers
- **configs/** – Environment-specific configurations
- **fixtures/** – Pytest fixtures for browser, API clients, and setup/teardown
- **pytest.ini** – Pytest configuration for markers, retries, and execution settings

That's it. No essay.

Configuration Management

- Use **YAML/JSON config files** for environment-specific values
- Store browser, device, and tenant details in config files
- Use **environment variables** for sensitive data (tokens, credentials)
- Parameterize tests for:
    - Browsers (Chrome, Firefox, Safari)
    - Devices (iOS, Android via BrowserStack)
    - Tenants (company1, company2)
- Centralized config loader to read environment settings at runtime

This approach avoids hardcoding values and allows easy scaling across environments.

Missing Requirements (VERY important section)

This is where candidates score.

ask questions like

- How is test data **reset**?
- Test user provisioning?
- How is **test data created and cleaned up** after execution?
- Can tests run **in parallel**, and what are the limits?
- What **reporting tools** are preferred (Allure, HTML, CI-native)?
- Are there **cost constraints** on BrowserStack usage?
- How are **test users** managed per tenant and role?
- Is there a **separate test environment** per tenant?
- How should **failed tests** be retried or reported?
- What is the expected **CI trigger strategy** (PR, nightly, release)?

Show real-world thinking.

PART 3: API + UI Integration Test

(most important)

This test validates the complete project creation flow across API, Web UI, and Mobile while ensuring tenant isolation and cross-platform consistency.

How she should write the test

They literally gave the skeleton. We just need to **expand with comments**.

Testing Strategy Overview

- Use API to create test data quickly and reliably
- Verify the same data through UI and mobile to ensure end-to-end consistency
- Use separate tenant credentials to validate isolation
- Assume authentication tokens and BrowserStack setup are preconfigured

Task 1: Write the Integration Test (Skeleton + Comments)

```
def test_project_creation_flow():
    # Step 1: Create project via API
    # Assumption: Valid auth token and tenant ID are available
    api_response = create_project_api(
        name="Test Project",
        tenant_id="company1"
    )
    project_id = api_response["id"]

    # Step 2: Verify project appears in Web UI
    login_ui(user="admin@company1.com")
    wait_for_dashboard_load()
    verify_project_visible(project_id)

    # Step 3: Verify project is accessible on mobile (BrowserStack)
    launch_mobile_session()
    login_mobile_ui(user="admin@company1.com")
    verify_project_visible_mobile(project_id)

    # Step 4: Verify tenant isolation
    login_ui(user="admin@company2.com")
    verify_project_not_visible(project_id)
```

Task 2: Handle Test Data

- Generate **unique project names** to avoid collisions
- Use API for **setup and cleanup**
- Maintain separate test data per tenant
- Clean up created projects after test execution
- Avoid sharing state between parallel tests

API-based test data creation improves speed and reliability compared to UI-only setup.

Task 3: Cross-Platform Validation

Cross-Platform Testing Approach

- Use Playwright for web validation across multiple browsers
- Use BrowserStack for mobile testing on iOS and Android
- Validate UI consistency and basic responsiveness
- Reuse the same test data across platforms

understand **cost-aware testing**, do not brute-force testing.

Task 4: Tenant Isolation (Security Check)

Tenant Isolation Validation

- Login with a different tenant user
- Ensure project created under company1 is not visible
- Validate backend security boundaries
- Prevent data leakage across tenants

This is **huge** for B2B SaaS. They'll like this.

Edge Cases (don’t skip this)

Mention handling:

- API failure or timeout
- Slow UI loading
- Network latency on mobile
- Project sync delay between API and UI
- Retry logic for transient failures

Explicit waits and retries are used to reduce flakiness in unstable environments.

Even listing them is enough.

Assumptions (VERY IMPORTANT)

Listing things like this to clear up

- Authentication tokens are pre-generated
- Test users exist for each tenant and role
- BrowserStack credentials are configured
- Cleanup jobs are allowed via API

This protects us from ambiguity.