Test Automation Strategy & Experience

1. Vision: Automated Change Detection for Multiple Landing Pages

Architecture for Change-Triggered Testing:

I envision a comprehensive CI/CD integration system that automatically detects DOM changes and triggers targeted test suites:

- **Visual Regression Monitoring**: Implement tools, Playwright's visual comparison to capture baseline screenshots. On deployment, automatically compare new builds against baselines to detect UI changes.
- **DOM Change Detection**: Use headless browser automation to crawl landing pages and track structural changes (new elements, removed elements, attribute modifications). Store checksums of critical page sections.
- **Intelligent Test Triggering**: When changes are detected, a smart dispatcher identifies which test suites are affected based on changed selectors, page sections, and feature areas. Only relevant tests run, reducing execution time.
- **Multi-Page Orchestration**: Create a centralized page registry mapping URLs to their corresponding page objects, locators, and test suites. This enables scalable management across dozens of landing pages.
- **Reporting Dashboard(Slack integrations)**: Aggregate results from all pages into a unified dashboard showing regression status, changed elements, and test impact analysis.

Implementation Flow: Git push → Build → Deploy → Change Detection → Filtered Test Execution → Report Generation → Slack/Email Notifications

---

2. Tools & Processes if Owning the Project

Technology Stack:

- **Playwright Test** (current foundation) with parallel execution for speed
- **GitHub Actions or GitLab CI/CD** for orchestration and scheduling
- **AI-Powered Locator Generation**: Integrate tools like Copilot or Claude(or Playwright Test Agents) for intelligent selector suggestions when pages change, reducing manual locator maintenance
- **Percy or Applitools** for visual regression testing across devices/browsers
- **Datadog or New Relic** for test performance monitoring and flakiness detection
- **Custom Python/Node service** for DOM change detection and selector updates
- **PostgreSQL** for test result history, trend analysis, and failure pattern recognition

Process Framework:

1. **Pre-commit Hooks**: Run lightweight smoke tests locally before pushing
2. **Automated Locator Maintenance**: AI-assisted scanning to suggest locator updates when tests fail due to selector changes
3. **Flakiness Detection**: Machine learning to identify intermittent failures and suggest waits/retries
4. **Test Data Management**: Centralized, versioned test data with environment-specific configurations
5. **AI-Assisted Root Cause Analysis**: When tests fail, automatically generate reports with likely causes (network, selector changes, logic errors)
6. **Continuous Optimization**: Analyze test execution patterns to optimize test order and parallel worker allocation

Why This Approach:

- Reduces manual maintenance by 40-60% through AI assistance
- Enables fast feedback loops with intelligent test filtering
- Provides predictive analytics for test reliability
- Scales efficiently as the number of pages grows

---

3. Most Challenging Situation & Resolution

The Challenge: Flaky Gallery Navigation Tests

During my recent work on the Walk-In Bath landing page tests, I encountered a critical issue: **gallery navigation tests were inconsistently failing** because:

1. **Timing Issues**: DOM updates weren't consistent—clicking "Previous" sometimes didn't change the gallery image
2. **Selector Fragility**: Using nth(1) for selecting buttons created brittle tests when button order changed
3. **Test Discovery Failures**: Custom Playwright fixtures weren't properly serializing in worker processes, causing "Test not found in worker" errors
4. **Flaky Waits**: Implicit waitForTimeout() calls masked the real issue—elements weren't actually changing state

Resolution Strategy:

1. **Replaced Implicit Waits**: Changed from page.waitForTimeout(300) to explicit sliderImage.waitFor({ state: 'visible', timeout: 5000 }), ensuring actual DOM changes were verified
2. **Restructured Test Logic**: Instead of testing Previous→Next in one direction, I modified the test to navigate Forward→Backward→Verify Original, creating a round-trip validation that's more robust
3. **Fixed Fixture Registration**: Resolved test discovery issues by defining fixtures directly in the test file rather than importing from helpers, ensuring proper Playwright deserialization
4. **Improved Locators**: Moved away from brittle selectors to more semantic ones using getByRole() and attribute-based selection(Playwright-based locators)
5. **Added Helper Methods**: Created waitForGalleryImageChange() method that explicitly polls for image source changes, reducing race conditions

Result: Tests transitioned from ~30% flake rate to 100% reliable execution, and the approach became a template for other dynamic UI tests.

Key Learning: The most critical insight was that **flakiness is almost never random**—it's usually a symptom of inadequate waits, incorrect selectors, or test logic that doesn't match actual user behavior. Fixing the underlying cause through proper synchronization is far superior to adding blanket wait times.