🕶️ Negative Tests Guide: The Agents' Challenge

"What you must learn is that these rules are no different from the rules of a computer system. Some of them can be bent. Others can be broken. In negative testing, we deliberately try to break the system to find its weaknesses."

Chapter 1: The Nature of Negative Testing

What Are Negative Tests?

Agent Smith's Purpose: Just as Agent Smith exists to find and eliminate threats to the Matrix, negative tests exist to find and expose weaknesses in our system. They are the necessary adversaries that make our software stronger.

The Dark Mirror: If positive tests are like Neo's successful missions, negative tests are like encounters with Agent Smith - challenging scenarios designed to test the system's limits and failure handling.

Core Philosophy:

- Test what happens when things go wrong
- Validate error handling and system robustness
- Find edge cases and unexpected behaviors
- Ensure graceful degradation under stress

Chapter 2: The Agents' Architecture - System Vulnerabilities

Class Structure Analysis

```python
class TestBlueOriginNegative:
    """Negative test scenarios for Blue Origin career website"""
```

The Agents' Council: This class represents the collective intelligence of the Agents - systematically probing for weaknesses in our Matrix (the Blue Origin website).

BrowserStack Configuration - The Agent Network

```python
BSTACK_USER = "serinezargaryan_P7fD2M"
BSTACK_KEY = "XkSkNCoqLRuqo5PTbisp"
BSTACK_URL = f"https://{BSTACK_USER}:{BSTACK_KEY}@hub-cloud.browserstack.com/wd/hub"
```

⚠️ **Security Alert:** Unlike the positive tests, this code exposes credentials directly in the source code!

Agent Smith Would Say: "*This is a vulnerability, Mr. Anderson. Credentials should never be hardcoded.*"

Proper Approach (What Should Be Done):

```python
BSTACK_USER = os.getenv("BSTACK_USER")
BSTACK_KEY = os.getenv("BSTACK_KEY")
```

Why This Matters: Hardcoded credentials are like telling Agent Smith your location - a security risk.

Browser Capabilities - Agent Variants

```python
BSTACK_CAPABILITIES = {
    "chrome": {
        "browserName": "chrome",
        "browserVersion": "latest",
        "os": "Windows",
        "osVersion": "11",
        "buildName": "browserstack-BlueOrigin-Negative-Tests-v1",
        "projectName": "Blue Origin Testing",
        "userName": BSTACK_USER,  # Direct injection (bad practice)
        "accessKey": BSTACK_KEY   # Security vulnerability
    }
}
```

Agent Evolution: Each browser configuration is like a different Agent variant - Smith, Johnson, Brown - each with slightly different capabilities but the same destructive purpose.

Chapter 3: The Setup Method - Summoning the Agents

`setup_method()` - Agent Initialization

```python
@pytest.fixture(autouse=True)
def setup_method(self, request):
```

Agent Spawning: This method is like the Matrix spawning new Agents - it creates the hostile environment needed for negative testing.

Key Differences from Positive Tests:

1. Hardcoded BrowserStack Check

```python
use_browserstack = os.getenv('USE_BROWSERSTACK', 'false').lower() == 'true'

if use_browserstack:
    self.driver = self._get_browserstack_driver(self.browser_name, test_name)
else:
    self.driver = WebDriverFactory.get_driver(self.browser_name)
```

Agent Smith's Logic: The setup is more rigid - less flexible than Neo's approach in positive tests.

2. Different Capability Structure

```python
def _get_browserstack_driver(self, browser_name, test_name):
    capabilities = self.BSTACK_CAPABILITIES.get(browser_name, self.BSTACK_CAPABILITIES['chrome']).copy()
    capabilities['sessionName'] = f"Negative Test: {test_name} - {browser_name}"
```

Agent Identification: Each negative test session is clearly marked as a hostile operation.

Chapter 4: Negative Test Cases - The Agents' Attacks

Test Case 1: `test_tc_n_001_job_count_mismatch_between_systems()` - The Inconsistency Attack

```python
def test_tc_n_001_job_count_mismatch_between_systems(self, browser):
```

Agent Smith's Strategy: "*The Matrix has inconsistencies, Mr. Anderson. Let me show you.*"

Attack Vector Analysis:

Phase 1: Intelligence Gathering (Blue Origin System)

```python
self.driver.get(BlueOriginUrls.CAREERS_SEARCH_URL)
self.helpers.handle_cookie_consent()
time.sleep(3)

blue_origin_count = self.helpers.get_search_results_count()
assert blue_origin_count > 0, "No jobs found on Blue Origin search page"
```

Translation: *"First, we infiltrate the primary system and gather intelligence about job counts."*

Phase 2: Cross-System Reconnaissance (Workday System)

```python
workday_count = self.helpers.get_workday_job_count()
assert workday_count > 0, "No jobs found on Workday careers page"
```

Translation: *"Then we infiltrate the secondary system to compare their data."*

Phase 3: The Contradiction Reveal

```python
print(f"Blue Origin job count: {blue_origin_count}")
print(f"Workday job count: {workday_count}")

assert blue_origin_count == workday_count, f"Job count mismatch detected: Blue Origin ({blue_origin_count}) vs Worko
```

Agent Smith Reveals: *"You see? The systems are inconsistent. The Matrix is flawed."*

Why This Test is "Negative": We expect this test to FAIL because systems often have mismatched data. The failure reveals the inconsistency.

Test Case 2: `test_tc_n_002_numeric_keyword_search_logic_comparison()` - The Logic Probe

```python
def test_tc_n_002_numeric_keyword_search_logic_comparison(self, browser):
```

Agent Brown's Approach: *"Let's test their logic with something unexpected - numbers where they expect words."*

The Numeric Confusion Attack:

Phase 1: Confuse System A (Blue Origin)

```python
search_success = self.helpers.search_for_keyword("123")
assert search_success, "Search input field not found on Blue Origin"
blue_origin_results = self.helpers.get_search_results_count()
```

Confusion Tactic: Searching for "123" is like asking the Oracle a question in binary - technically valid but unexpected.

Phase 2: Confuse System B (Workday)

```python
workday_search_success = self.helpers.search_workday_platform("123")
# Complex fallback logic if first attempt fails...
workday_results = self.helpers.get_workday_search_results_count()
```

Phase 3: Expose Logic Differences

```python
print(f"Blue Origin '123' search results: {blue_origin_results}")
print(f"Workday '123' search results: {workday_results}")

assert blue_origin_results == workday_results, f"Numeric search logic differs"
```

Agent's Discovery: Different systems may handle numeric searches differently - revealing implementation inconsistencies.

Test Case 3: `test_tc_n_003_exact_job_title_search_consistency()` - The Existence Paradox

```python
def test_tc_n_003_exact_job_title_search_consistency(self, browser):
```

Agent Johnson's Trap: "*If a job exists in one system, it should exist in all systems. Let's expose their lies.*"

This is the most sophisticated negative test - it's like a philosophical paradox:

The Paradox Setup:

Step 1: Extract Truth from Source A (Workday)

```python
self.driver.get(BlueOriginUrls.WORKDAY_URL)
time.sleep(2)
cookie_handled = self.helpers.handle_workday_cookie_consent()
exact_job_title = self.helpers.get_first_workday_job_title()

assert exact_job_title is not None, "Could not find any job title on Workday careers page"
```

The Trap: We take a job title that definitely exists (we just saw it) and use it as our weapon.

Step 2: Verify Truth in Source A (Workday Self-Test)

```python
workday_search_success = self.helpers.search_workday_platform(exact_job_title)
workday_results_count = self.helpers.get_workday_search_results_count()

if workday_results_count == 0:
    print(f"CRITICAL: Workday search returned 0 results for a job title that exists on Workday")
    workday_search_issue = True
```

The Self-Contradiction: If Workday can't find a job that exists on Workday, the system contradicts itself!

Step 3: Test Truth in System B (Blue Origin)

```python
self.driver.get(BlueOriginUrls.CAREERS_SEARCH_URL)
search_success = self.helpers.search_for_keyword(exact_job_title)
blue_origin_results_count = self.helpers.get_search_results_count()
```

Step 4: Reveal the Inconsistency

```python
# Multiple assertion levels revealing different types of failures:

# Level 1: System self-contradiction (most critical)
assert workday_results_count > 0, f"CRITICAL BUG: Workday search returned 0 results for job title '{exact_job_title}' that

# Level 2: Cross-system inconsistency
assert blue_origin_results_count > 0, f"Blue Origin search returned 0 results for job title '{exact_job_title}' which exists or

# Level 3: Count mismatch
assert workday_results_count == blue_origin_results_count, f"Job title search results should be identical between platfo
```

Agent's Victory Conditions:

1. **Self-Contradiction:** Workday can't find its own job (system broken)
2. **Cross-System Failure:** Blue Origin doesn't have Workday's job (sync failure)
3. **Count Mismatch:** Different systems return different counts (logic inconsistency)

Test Case 4: `test_tc_n_004_search_robustness_with_special_characters()` - The Chaos Injection

```python
```

```
def test_tc_n_004_search_robustness_with_special_characters(self, browser):
```

Agent Smith's Chaos: "*Order, Mr. Anderson. Systems crave order. Let's give them chaos.*"

The Chaos Attack:

Phase 1: Inject Chaos

```
python

special_query = "  software engineer @@ ##  "
search_success = self.helpers.search_with_special_characters(special_query)
```

Chaos Elements:

- **Leading/trailing spaces:** `"  software engineer  "`
- **Special characters:** `@@ ##`
- **Mixed content:** Valid keywords mixed with noise

Phase 2: Measure System Response

```
python

special_results_count = self.helpers.get_search_results_count()
print(f"Search results with special characters '{special_query}': {special_results_count}")
```

Phase 3: Test System Recovery

```
python

if special_results_count == 0:
    print("System correctly filtered out special characters and returned 0 results")
else:
    print(f"System returned {special_results_count} results for special character search")

# Test if system survived the chaos
normal_search_success = self.helpers.search_for_keyword("engineer")
```

Agent's Analysis:

- **Good System:** Filters chaos, returns 0 results, recovers gracefully
- **Bad System:** Crashes, returns errors, or becomes unresponsive
- **Mediocre System:** Returns random results from chaos input

---

Test Case 5: `test_tc_n_005_career_page_functionality_without_javascript()` - The Power Drain

```python
def test_tc_n_005_career_page_functionality_without_javascript(self, browser):
```

Agent Smith's Ultimate Test: *"What is the Matrix without its power, Mr. Anderson? Let's find out."*

This test is like removing Neo's powers - testing if the website works when JavaScript (its superpowers) is disabled.

The Power Drain Process:

Phase 1: Disable the Matrix Powers

```python
if use_browserstack:
    # BrowserStack approach - recreation with disabled JS
    if browser.lower() == 'chrome':
        prefs = {
            "profile.default_content_setting_values": {
                "javascript": 2  # 2 = disabled
            }
        }
        options.add_experimental_option("prefs", prefs)
    elif browser.lower() == 'firefox':
        options.set_preference("javascript.enabled", False)
```

Power Removal: Like taking away Neo's ability to bend spoons, we remove JavaScript's ability to manipulate the page.

Phase 2: Test Basic Survival

```python
self.driver.get(BlueOriginUrls.CAREERS_URL)
time.sleep(5)

# Test 1: Check that video elements don't work
video_not_working = self.helpers.check_video_elements_disabled()

# Test 2: Test basic HTML functionality
link_navigation_works = self.helpers.test_basic_link_navigation()
```

Survival Tests:

- **Videos disabled:** Confirms JavaScript is actually disabled
- **Links working:** Confirms basic HTML still functions

Phase 3: Evaluate Matrix Integrity

```python
if video_not_working and link_navigation_works:
    print("SUCCESS: JavaScript disabled test passed")
    print("- Videos disabled (JavaScript not working)")
    print("- HTML links functional (basic HTML works)")
```

Agent's Verdict:

- **Strong Matrix:** Works without JavaScript (good accessibility)
- **Weak Matrix:** Completely broken without JavaScript (poor design)
- **Fragile Matrix:** Partially functional (acceptable but not ideal)

Chapter 5: The Philosophy of Negative Testing

Why Agents Exist

The Necessary Opposition: Just as Agent Smith serves a purpose in the Matrix, negative tests serve a crucial purpose in software development:

1. **Expose Weaknesses:** Find bugs before users do
2. **Test Error Handling:** Verify graceful failure modes
3. **Validate Assumptions:** Challenge what we think we know
4. **Improve Robustness:** Force systems to handle edge cases

Agent Smith's Wisdom

"It came to me when I tried to classify your species. I realized that you're not actually mammals. Every mammal on this planet instinctively develops a natural equilibrium with the surrounding environment. But you humans do not."

Testing Translation: *"Every software system should instinctively develop natural error handling with its environment. But most systems do not."*

Common Negative Testing Patterns

1. Boundary Testing

```python
```

```python
# Test with edge cases
search_query = "a" * 1000  # Very long input
search_query = ""          # Empty input
search_query = "    "      # Only spaces
```

2. Invalid Input Testing

```python
# Test with invalid data types
search_query = 123         # Number instead of string
search_query = None        # Null value
search_query = ["test"]    # Array instead of string
```

3. System State Testing

```python
# Test when system is in unexpected states
# - No internet connection
# - Server overloaded
# - Database unavailable
# - JavaScript disabled
```

4. Cross-System Consistency Testing

```python
# Test data consistency between related systems
# - Job counts should match
# - Search results should be identical
# - User data should sync
```

Chapter 6: Error Handling in Negative Tests

Embracing Failure

Agent's Perspective: Unlike positive tests that fear failure, negative tests embrace it. Expected failures are victories.

Expected vs Unexpected Failures

Expected Failure (Good)

```python
```

```
# We expect this to fail - revealing a known issue
assert blue_origin_count == workday_count, f"Job count mismatch detected"
```

Agent Smiles: "*As expected, the systems are inconsistent.*"

Unexpected Failure (Bad)

```
python

# We don't expect this to fail - indicates a new problem
assert blue_origin_count > 0, "No jobs found on Blue Origin search page"
```

Agent Concerned: "*The primary system is completely broken - this is unexpected.*"

Defensive Programming

```
python

try:
    normal_search_success = self.helpers.search_for_keyword("engineer")
    if normal_search_success:
        normal_results = self.helpers.get_search_results_count()
    else:
        print("First attempt at normal search failed, refreshing page...")
        self.driver.refresh()
        retry_search_success = self.helpers.search_for_keyword("engineer")
except Exception as e:
    print(f"Warning: Could not verify normal search functionality: {str(e)}")
    # Don't fail the test - the main assertion already passed
```

Agent's Strategy: Always have a backup plan. If Plan A fails, try Plan B. If Plan B fails, document the failure but don't crash.

Chapter 7: Advanced Agent Techniques

The Multi-Phase Attack

Complex negative tests often follow this pattern:

1. **Reconnaissance:** Gather information about the system
2. **Exploitation:** Attack the discovered weakness
3. **Verification:** Confirm the weakness exists
4. **Recovery Testing:** Ensure system can recover
5. **Documentation:** Record findings for the resistance (developers)

Cross-Browser Agent Deployment

```python
@pytest.mark.parametrize("browser", ["chrome", "firefox", "edge"])
def test_something(self, browser):
```

Agent Multiplication: Like Agent Smith multiplying himself, we run the same attack across multiple browsers to find browser-specific vulnerabilities.

Environment-Aware Attacks

```python
use_browserstack = os.getenv('USE_BROWSERSTACK', 'false').lower() == 'true'

if use_browserstack:
    # Cloud-specific attack vectors
    self.driver = self._get_browserstack_driver(browser_name, test_name)
else:
    # Local-specific attack vectors
    self.driver = WebDriverFactory.get_driver(browser_name)
```

Adaptive Agents: Like Agents adapting to different Matrix environments, our tests adapt to different testing environments.

Chapter 8: Interpreting Agent Reports

Reading Negative Test Results

Victory Report (Test Failed as Expected)

```
test_negative.py::TestBlueOriginNegative::test_tc_n_001_job_count_mismatch_between_systems[chrome] FAILED
AssertionError: Job count mismatch detected: Blue Origin (573) vs Workday (581)
```

Agent's Report: "*Mission accomplished. Inconsistency detected and documented.*"

Unexpected Success (Test Passed When Expected to Fail)

```
test_negative.py::TestBlueOriginNegative::test_tc_n_001_job_count_mismatch_between_systems[chrome] PASSED
```

Agent's Surprise: "*The systems are actually consistent. Either they fixed it, or our intelligence was wrong.*"

System Breakdown (Test Failed for Wrong Reasons)

```
test_negative.py::TestBlueOriginNegative::test_tc_n_001_job_count_mismatch_between_systems[chrome] ERROR
TimeoutException: Could not find search page after 30 seconds
```

Agent's Concern: "*The system is more broken than expected. We can't even begin our attack.*"

Chapter 9: The Agent's Code Quality Analysis

Security Vulnerabilities Found

1. Hardcoded Credentials (Critical)

```python
BSTACK_USER = "serinezargaryan_P7fD2M"  # ❌ Exposed username
BSTACK_KEY = "XkSkNCoqLRuqo5PTbisp"    # ❌ Exposed password
```

Agent Smith: "*Your security is an illusion, Mr. Anderson.*"

2. Mixed Architecture Patterns

The negative test file uses a different setup pattern than the positive tests - inconsistency in the codebase.

3. Implicit Wait Usage

```python
driver.implicitly_wait(10)  # ❌ Global implicit waits are unpredictable
```

Better Approach:

```python
# ✅ Explicit waits are more reliable
WebDriverWait(driver, 10).until(EC.element_to_be_clickable(element))
```

Chapter 10: The Agent's Final Assessment

Strengths of the Negative Testing Approach

1. **Comprehensive Coverage:** Tests multiple failure scenarios
2. **Cross-System Validation:** Compares different implementations
3. **Edge Case Testing:** Special characters, JavaScript disabled
4. **Real-World Scenarios:** Tests what actually happens in production

Weaknesses to Address

1. **Security Issues:** Hardcoded credentials must be fixed
2. **Inconsistent Patterns:** Should match positive test structure
3. **Error Handling:** Some tests could recover more gracefully
4. **Documentation:** Complex tests need better inline documentation

The Agent's Recommendation

Agent Smith's Final Word: "*The negative tests serve their purpose - exposing the flaws in your Matrix. But they themselves are flawed. Fix the security issues, standardize the patterns, and these tests will become truly powerful weapons against bugs.*"

Conclusion: Embracing the Agent Within

"We're not here because we're free. We're here because we're not free. There's no escaping reason, no denying purpose. Because as we both know, without purpose, we would not exist."

The Purpose of Negative Testing:

- To find bugs before users do
- To validate error handling
- To ensure system robustness
- To challenge assumptions
- To make software better through adversity

Remember: Every good software system needs its Agent Smith - something that constantly challenges it, finds its weaknesses, and forces it to evolve.

The Paradox: We create Agents (negative tests) to destroy our Matrix (system), so that we can build a stronger Matrix that can withstand real Agents (user errors, system failures, security attacks).

"Welcome to the desert of the real... testing world. Population: You and your bugs." 🕶️