Building an MCP Server for pytest-Selenium Automation with GitHub Copilot

Executive Summary

This comprehensive guide demonstrates how to create a Model Context Protocol (MCP) server that integrates with GitHub Copilot to generate pytest-selenium automation scripts. The server loads your existing test framework structure (conftest.py, pytest.ini, and utility methods from commands.py) and uses this context to generate automation scripts that follow your framework's patterns and conventions.

Table of Contents

1. Introduction to MCP and Architecture

What is Model Context Protocol (MCP)?

Model Context Protocol (MCP) is an open standardized protocol that enables Large Language Models (LLMs) to connect with external tools, data sources, and services [1] [2] [3]. Think of MCP as "USB-C for AI tools" - it provides a unified way to expose capabilities regardless of the host application.

MCP Architecture Components

The MCP architecture consists of three main components [3] [4]:

MCP Host: The AI application or environment (e.g., GitHub Copilot, Claude Desktop, VS Code) that contains the LLM and serves as the user's interaction point.

MCP Client: Located within the host, it translates the LLM's requests for the MCP server and converts the server's responses back for the LLM. Each server has its own client for isolation.

MCP Server: The external service that provides specific capabilities (tools, data access, prompts) to the LLM. This connects to various data sources like databases, file systems, and web services.

Communication Protocol

MCP uses JSON-RPC 2.0 messages for communication between client and server [5] [3]. The protocol supports two primary transport methods:

- **stdio (Standard Input/Output)**: For local server processes
- **HTTP/SSE (Server-Sent Events)**: For remote or web-based servers

The communication lifecycle includes three phases [6]:

1. **Initialization**: The MCP client connects to the server and initiates the handshake
2. **Message Exchange**: Request-response patterns for tool invocation and resource access
3. **Termination**: Clean shutdown when the session ends

Why MCP for Test Automation?

Traditional test automation code generation faces several challenges:

- **Inconsistent code patterns**: Generated code doesn't match existing framework conventions
- **Missing context**: AI tools lack awareness of project-specific utilities and patterns
- **Manual adaptation required**: Generated scripts need significant editing to fit the framework

MCP solves these problems by [5] [4]:

- **Providing context**: Loading framework files (conftest.py, pytest.ini, commands.py) into the AI's context
- **Standardizing access**: Using tools to expose browser automation and code generation capabilities
- **Ensuring consistency**: Generated code follows the patterns defined in your framework files

2. MCP Server Components and Capabilities

Core MCP Server Primitives

MCP servers expose three types of components, each serving different interaction patterns [4] [7]:

Tools (Model-Controlled)

Tools are executable functions that the LLM can invoke during conversations [8] [9]. The server exposes these to clients, representing dynamic operations that can modify state or interact with external systems.

Key characteristics:

- Invoked by the AI model based on conversation context
- Accept parameters validated against JSON schemas
- Return structured results to the LLM

- Can perform actions like navigating browsers, extracting data, or generating code

Example tools for our use case:

- `load_framework_context`: Load conftest.py, pytest.ini, and commands.py
- `navigate_and_extract_elements`: Open URL in browser and extract page elements
- `generate_pytest_selenium_script`: Create test script following framework patterns
- `validate_script`: Check generated code against framework conventions

Resources (Application-Controlled)

Resources provide data and content that can be read by clients and used as context for LLM interactions [4] [7]. The client application decides how and when resources should be accessed.

Key characteristics:

- Provide static or dynamic data
- Support URI-based addressing (e.g., `framework://conftest`, `examples://login-test`)
- Can include templates for dynamic parameter substitution
- Read-only by design

Example resources for our use case:

- `framework://conftest`: Template from loaded conftest.py
- `framework://pytest-ini`: Configuration from pytest.ini
- `framework://commands`: Utility methods from commands.py
- `examples://login-test`: Example test patterns

Prompts (User-Controlled)

Prompts are predefined templates and workflows that users can invoke through UI elements [7]. They're exposed from servers to clients with the intention of explicit user selection.

Key characteristics:

- User-initiated through slash commands or UI actions
- Return formatted message templates
- Support parameterization
- Help structure requests to the LLM

Example prompts for our use case:

- `generate_test_prompt`: Create structured prompt for test generation
- `debug_script_prompt`: Troubleshoot generated test issues
- `refactor_to_pom`: Convert standard test to Page Object Model

FastMCP Framework

FastMCP is a high-level Python framework that simplifies MCP server creation[2] [10] [11]. It handles protocol complexities automatically, allowing developers to focus on implementing business logic.

Key advantages of FastMCP[10] [8]:

1. **Automatic schema generation**: Extracts type hints to create JSON schemas
2. **Simple decorator syntax**: `@mcp.tool()`, `@mcp.resource()`, `@mcp.prompt()`
3. **Built-in validation**: Validates parameters against function signatures
4. **Transport flexibility**: Supports stdio and HTTP transports
5. **Context injection**: Provides access to request context and session information

Installation:

```
pip install fastmcp
```

Basic server structure:

```
from fastmcp import FastMCP

mcp = FastMCP(
    name="server-name",
    dependencies=["package1", "package2"]
)

@mcp.tool()
def my_tool(param: str) -&gt; dict:
    """Tool description for the LLM"""
    return {"result": "value"}

if __name__ == "__main__":
    mcp.run()  # stdio transport by default
```

Context Injection and Framework Loading

One of the most powerful features for our use case is context injection[12] [13]. The MCP server loads framework files at startup and makes this context available to all code generation operations.

Context loading pattern:

```
framework_context = {
    "conftest": None,
    "pytest_ini": None,
    "commands": None,
    "page_objects": []
}

@mcp.tool()
def load_framework_context(
```

```
    conftest_path: str,
    pytest_ini_path: str,
    commands_path: str
) -&gt; dict:
    """Load framework files into context"""
    with open(conftest_path, 'r') as f:
        framework_context['conftest'] = f.read()

    with open(pytest_ini_path, 'r') as f:
        framework_context['pytest_ini'] = f.read()

    with open(commands_path, 'r') as f:
        framework_context['commands'] = f.read()

    return {"status": "success"}
```

This context is then used during code generation to:

- Extract fixture patterns from conftest.py
- Identify utility methods from commands.py
- Apply pytest configuration from pytest.ini
- Match naming conventions and import patterns

3. Implementation Guide

Prerequisites

Software requirements:

- Python 3.8 or higher
- pip package manager
- Chrome browser (for Selenium)
- Text editor or IDE with GitHub Copilot support

Python packages:

```
pip install fastmcp&gt;=0.3.0
pip install selenium&gt;=4.0.0
pip install pytest&gt;=7.0.0
pip install pytest-selenium&gt;=4.0.0
pip install webdriver-manager&gt;=3.8.0
```

Project Structure

Organize your pytest-selenium framework with the following structure:

```
pytest-selenium-framework/
├── mcp_selenium_server.py       # Main MCP server implementation
```

```
├── config/
│   ├── conftest.py              # pytest fixtures (driver setup)
│   ├── pytest.ini               # pytest configuration
│   └── commands.py              # utility methods
├── tests/
│   ├── test_example.py          # example existing tests
│   └── (generated test files)
├── pages/
│   └── (page object classes)
├── screenshots/
├── reports/
└── requirements.txt
```

Core Server Implementation

The complete MCP server consists of several key components working together. Here's the full implementation:

```
# mcp_selenium_server.py
from fastmcp import FastMCP
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
import os
import json
from typing import Dict, List, Optional

# Initialize MCP server with dependencies
mcp = FastMCP(
    name="pytest-selenium-generator",
    dependencies=["selenium", "pytest", "pytest-selenium"]
)

# Global context store for framework files
framework_context = {
    "conftest": None,
    "pytest_ini": None,
    "commands": None,
    "page_objects": []
}

@mcp.tool()
def load_framework_context(
    conftest_path: str,
    pytest_ini_path: str,
    commands_path: str
) -&gt; Dict[str, str]:
    """
    Load framework files to provide context for code generation.

    This tool reads your existing pytest-selenium framework files
    and stores their content for use during test script generation.
```

```
    Args:
        conftest_path: Absolute path to conftest.py file
        pytest_ini_path: Absolute path to pytest.ini file
        commands_path: Absolute path to commands.py utility file

    Returns:
        Dictionary with loading status and file paths
    """
    try:
        # Load conftest.py (fixtures and driver setup)
        with open(conftest_path, 'r') as f:
            framework_context['conftest'] = f.read()

        # Load pytest.ini (configuration)
        with open(pytest_ini_path, 'r') as f:
            framework_context['pytest_ini'] = f.read()

        # Load commands.py (utility methods)
        with open(commands_path, 'r') as f:
            framework_context['commands'] = f.read()

        return {
            "status": "success",
            "message": "Framework context loaded successfully",
            "files_loaded": [conftest_path, pytest_ini_path, commands_path]
        }
    except FileNotFoundError as e:
        return {
            "status": "error",
            "message": f"File not found: {str(e)}"
        }
    except Exception as e:
        return {
            "status": "error",
            "message": f"Error loading framework context: {str(e)}"
        }

@mcp.tool()
def navigate_and_extract_elements(
    url: str,
    headless: bool = True
) -&gt; Dict:
    """
    Navigate to a URL and extract all interactive page elements.

    Opens the specified URL in a browser, identifies all interactive
    elements (inputs, buttons, links, etc.), and extracts their
    attributes for use in test script generation.

    Args:
        url: The URL to navigate to
        headless: Run browser in headless mode (default: True)

    Returns:
        Dictionary containing page info and element details
    """
```

```
    try:
        # Setup Chrome options
        options = webdriver.ChromeOptions()
        if headless:
            options.add_argument('--headless')
            options.add_argument('--disable-gpu')
        options.add_argument('--no-sandbox')

        # Initialize driver
        driver = webdriver.Chrome(options=options)
        driver.get(url)

        # Extract interactive elements
        elements = []
        interactive_tags = ['input', 'button', 'a', 'select', 'textarea']

        for tag in interactive_tags:
            web_elements = driver.find_elements(By.TAG_NAME, tag)
            for elem in web_elements:
                # Extract element attributes
                element_info = {
                    "tag": tag,
                    "id": elem.get_attribute("id"),
                    "name": elem.get_attribute("name"),
                    "class": elem.get_attribute("class"),
                    "type": elem.get_attribute("type"),
                    "text": elem.text[:50] if elem.text else "",
                    "placeholder": elem.get_attribute("placeholder"),
                    "value": elem.get_attribute("value")
                }
                elements.append(element_info)

        # Compile page information
        page_info = {
            "url": url,
            "title": driver.title,
            "elements": elements[:50],  # Limit to first 50
            "element_count": len(elements)
        }

        driver.quit()
        return page_info

    except Exception as e:
        return {
            "status": "error",
            "message": f"Error extracting elements: {str(e)}"
        }

@mcp.tool()
def generate_pytest_selenium_script(
    test_name: str,
    url: str,
    actions: List[Dict],
    use_page_object: bool = True
) -&gt; str:
```

```
    """
    Generate a pytest-selenium automation script.

    Creates a complete test script following your framework's
    patterns, conventions, and utility methods. Can generate
    either Page Object Model or standard test structure.

    Args:
        test_name: Name for the test function (e.g., "test_login")
        url: Base URL for the test
        actions: List of action dictionaries with keys:
                 - action: "click", "type", "select", etc.
                 - element_name: descriptive name for element
                 - by: locator strategy ("ID", "NAME", "XPATH", etc.)
                 - locator: locator value
                 - value: (optional) value to enter for type actions
                 - method_name: (optional) custom method name
        use_page_object: Generate POM structure (default: True)

    Returns:
        Complete Python test script as string
    """

    # Extract utility methods from commands.py
    utility_methods = extract_utility_methods(
        framework_context.get('commands', '')
    )

    # Generate imports based on framework patterns
    imports = generate_imports(framework_context)

    # Generate appropriate script structure
    if use_page_object:
        script = generate_pom_script(
            test_name, url, actions, utility_methods, imports
        )
    else:
        script = generate_standard_script(
            test_name, url, actions, utility_methods, imports
        )

    return script

def extract_utility_methods(commands_content: str) -&gt; List[str]:
    """
    Extract utility method names from commands.py content.

    Parses the commands file to identify available utility
    methods that can be used in generated tests.
    """
    methods = []
    if commands_content:
        lines = commands_content.split('\n')
        for line in lines:
            if line.strip().startswith('def '):
                # Extract method name
```

```
                method_name = line.split('def ')[^1].split('(')[^0]
                methods.append(method_name)
    return methods

def generate_imports(context: Dict) -&gt; str:
    """
    Generate import statements based on framework context.

    Creates appropriate imports by analyzing the loaded
    framework files and identifying required modules.
    """
    imports = []
    imports.append("import pytest")
    imports.append("from selenium import webdriver")
    imports.append("from selenium.webdriver.common.by import By")
    imports.append("from selenium.webdriver.support.ui import WebDriverWait")
    imports.append("from selenium.webdriver.support import expected_conditions as EC")

    # Add custom imports if commands.py exists
    if context.get('commands'):
        imports.append("from utils.commands import *")

    return '\n'.join(imports)

def generate_pom_script(
    test_name: str,
    url: str,
    actions: List[Dict],
    utility_methods: List[str],
    imports: str
) -&gt; str:
    """
    Generate Page Object Model based test script.

    Creates a test following POM design pattern with
    separate page class and test class.
    """

    page_class_name = f"{test_name.replace('test_', '').title()}Page"

    # Build script header and imports
    script = f'''"""
Generated pytest-selenium automation script
Test: {test_name}
URL: {url}
Pattern: Page Object Model
"""

{imports}

class {page_class_name}:
    """Page Object for {url}"""

    def __init__(self, driver):
        self.driver = driver
        self.url = "{url}"
```

```
    # Locators
'''

    # Add locator constants from actions
    for i, action in enumerate(actions):
        locator_name = action.get('element_name', f'element_{i}')
        by_type = action.get('by', 'ID')
        locator_value = action.get('locator', '')
        script += f'    {locator_name.upper()} = (By.{by_type}, "{locator_value}")\n'

    # Add page load method
    script += '''
    def load(self):
        """Navigate to the page"""
        self.driver.get(self.url)
        return self

'''

    # Generate action methods for each interaction
    for action in actions:
        method_name = action.get('method_name',
                                 f"perform_{action.get('action', 'action')}")
        element_name = action.get('element_name', 'element')

        if action.get('action') == 'click':
            script += f'''    def {method_name}(self):
        """Click {element_name}"""
        element = self.driver.find_element(*self.{element_name.upper()})
        element.click()
        return self

'''
        elif action.get('action') == 'type':
            script += f'''    def {method_name}(self, text):
        """Enter text into {element_name}"""
        element = self.driver.find_element(*self.{element_name.upper()})
        element.clear()
        element.send_keys(text)
        return self

'''

    # Add test class
    script += f'''
@pytest.mark.usefixtures("driver")
class Test{test_name.replace('test_', '').title()}:
    """Test class for {test_name}"""

    def test_{test_name.replace('test_', '')}(self, driver):
        """Test case implementation"""
        # Initialize page object
        page = {page_class_name}(driver).load()

'''
```

```
    # Add test step calls
    for action in actions:
        if action.get('action') == 'click':
            script += f"        page.{action.get('method_name', 'click')}()\n"
        elif action.get('action') == 'type':
            value = action.get('value', 'test_value')
            script += f"        page.{action.get('method_name', 'type')}('{value}')\n"

    # Add assertion placeholder
    script += '''
        # Add your assertions here
        assert driver.current_url
'''

    return script

@mcp.resource("framework://conftest")
def get_conftest_template():
    """Provide loaded conftest.py content as resource"""
    return framework_context.get('conftest',
                                 'No conftest.py loaded. Use load_framework_context first.

@mcp.resource("framework://pytest-ini")
def get_pytest_ini_template():
    """Provide loaded pytest.ini content as resource"""
    return framework_context.get('pytest_ini',
                                 'No pytest.ini loaded. Use load_framework_context first.'

@mcp.resource("framework://commands")
def get_commands_template():
    """Provide loaded commands.py utility methods as resource"""
    return framework_context.get('commands',
                                 'No commands.py loaded. Use load_framework_context first.

@mcp.prompt()
def generate_test_prompt(url: str, test_description: str) -&gt; str:
    """
    Generate a structured prompt for test creation.

    Creates a detailed prompt that includes context from
    loaded framework files and guides the LLM to generate
    appropriate test code.

    Args:
        url: URL to test
        test_description: What the test should do

    Returns:
        Formatted prompt string
    """
    return f'''
Create a pytest-selenium automation script with the following requirements:

**Target URL**: {url}
**Test Objective**: {test_description}
```

```
**Framework Context**:
Please follow the patterns and conventions defined in our framework:
- Use fixtures from conftest.py for driver setup
- Follow pytest.ini configuration and markers
- Leverage utility methods from commands.py where appropriate
- Use consistent naming conventions (test_*, Page suffix for POM)
- Include appropriate waits and error handling

**Code Quality**:
- Add descriptive docstrings
- Use meaningful variable names
- Include assertions to verify expected behavior
- Follow PEP 8 style guidelines

Generate a complete, runnable test that integrates seamlessly with our existing framework
'''

if __name__ == "__main__":
    # Run the MCP server (stdio transport by default)
    mcp.run()
```

Framework Configuration Files

conftest.py Example

The conftest.py file defines pytest fixtures used across your test suite [14] [15]:

```
import pytest
from selenium import webdriver
from selenium.webdriver.chrome.service import Service
from webdriver_manager.chrome import ChromeDriverManager

@pytest.fixture(scope="function")
def driver(request):
    """
    WebDriver fixture for tests.
    Creates a new driver instance for each test.
    """
    options = webdriver.ChromeOptions()
    options.add_argument('--start-maximized')
    options.add_argument('--disable-extensions')
    options.add_argument('--disable-infobars')

    driver = webdriver.Chrome(
        service=Service(ChromeDriverManager().install()),
        options=options
    )

    driver.implicitly_wait(10)

    # Provide driver to test
    yield driver
```

```
    # Cleanup after test
    driver.quit()

@pytest.fixture(scope="function")
def driver_headless(request):
    """Headless WebDriver fixture for CI/CD environments"""
    options = webdriver.ChromeOptions()
    options.add_argument('--headless')
    options.add_argument('--disable-gpu')
    options.add_argument('--no-sandbox')

    driver = webdriver.Chrome(
        service=Service(ChromeDriverManager().install()),
        options=options
    )

    yield driver
    driver.quit()

@pytest.fixture(scope="session")
def base_url():
    """Base URL for application under test"""
    return "https://example.com"
```

pytest.ini Example

The pytest.ini file configures pytest behavior [16] [17]:

```
[pytest]
# Test markers for categorization
markers =
    smoke: Quick smoke tests
    regression: Full regression suite
    ui: UI automation tests
    api: API tests
    integration: Integration tests

# Default command-line options
addopts =
    -v
    --tb=short
    --strict-markers
    --html=reports/report.html
    --self-contained-html
    --capture=no

# Test discovery patterns
testpaths = tests
python_files = test_*.py
python_classes = Test*
python_functions = test_*

# Logging configuration
log_cli = true
log_cli_level = INFO
```

```
log_cli_format = %(asctime)s [%(levelname)s] %(message)s
log_cli_date_format = %Y-%m-%d %H:%M:%S
```

commands.py **Example**

The commands.py file contains reusable utility methods [18] [19]:

```
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
from selenium.webdriver.common.by import By
from selenium.common.exceptions import TimeoutException, NoSuchElementException
import logging

logger = logging.getLogger(__name__)

def wait_for_element(driver, by, locator, timeout=10):
    """
    Wait for element to be present in DOM.

    Args:
        driver: WebDriver instance
        by: Locator strategy (By.ID, By.XPATH, etc.)
        locator: Locator value
        timeout: Maximum wait time in seconds

    Returns:
        WebElement if found

    Raises:
        TimeoutException if element not found
    """
    try:
        element = WebDriverWait(driver, timeout).until(
            EC.presence_of_element_located((by, locator))
        )
        logger.info(f"Element found: {by}={locator}")
        return element
    except TimeoutException:
        logger.error(f"Element not found within {timeout}s: {by}={locator}")
        raise

def wait_and_click(driver, by, locator, timeout=10):
    """
    Wait for element to be clickable and click it.

    Args:
        driver: WebDriver instance
        by: Locator strategy
        locator: Locator value
        timeout: Maximum wait time

    Returns:
        Clicked WebElement
    """
    element = WebDriverWait(driver, timeout).until(
```

```
        EC.element_to_be_clickable((by, locator))
    )
    element.click()
    logger.info(f"Clicked element: {by}={locator}")
    return element

def wait_and_type(driver, by, locator, text, timeout=10):
    """
    Wait for element and enter text.

    Args:
        driver: WebDriver instance
        by: Locator strategy
        locator: Locator value
        text: Text to enter
        timeout: Maximum wait time

    Returns:
        WebElement after text entry
    """
    element = wait_for_element(driver, by, locator, timeout)
    element.clear()
    element.send_keys(text)
    logger.info(f"Entered text into {by}={locator}")
    return element

def get_element_text(driver, by, locator, timeout=10):
    """
    Get visible text from element.

    Args:
        driver: WebDriver instance
        by: Locator strategy
        locator: Locator value
        timeout: Maximum wait time

    Returns:
        Text content of element
    """
    element = wait_for_element(driver, by, locator, timeout)
    text = element.text
    logger.info(f"Retrieved text: '{text}' from {by}={locator}")
    return text

def is_element_visible(driver, by, locator, timeout=5):
    """
    Check if element is visible on page.

    Args:
        driver: WebDriver instance
        by: Locator strategy
        locator: Locator value
        timeout: Maximum wait time

    Returns:
        True if visible, False otherwise
```

```
    """
    try:
        WebDriverWait(driver, timeout).until(
            EC.visibility_of_element_located((by, locator))
        )
        return True
    except TimeoutException:
        return False

def scroll_to_element(driver, element):
    """
    Scroll element into view.

    Args:
        driver: WebDriver instance
        element: WebElement to scroll to
    """
    driver.execute_script("arguments[^0].scrollIntoView(true);", element)
    logger.info("Scrolled to element")

def take_screenshot(driver, filename):
    """
    Capture screenshot and save to file.

    Args:
        driver: WebDriver instance
        filename: Name for screenshot file (without extension)
    """
    filepath = f"screenshots/{filename}.png"
    driver.save_screenshot(filepath)
    logger.info(f"Screenshot saved: {filepath}")
    return filepath

def switch_to_iframe(driver, iframe_locator, by=By.ID, timeout=10):
    """
    Switch driver context to iframe.

    Args:
        driver: WebDriver instance
        iframe_locator: Locator for iframe
        by: Locator strategy
        timeout: Maximum wait time
    """
    iframe = wait_for_element(driver, by, iframe_locator, timeout)
    driver.switch_to.frame(iframe)
    logger.info(f"Switched to iframe: {by}={iframe_locator}")

def switch_to_default_content(driver):
    """Switch driver context back to default content"""
    driver.switch_to.default_content()
    logger.info("Switched to default content")
```

4. Configuration and Deployment

Installation Steps

1. **Create project directory**:

```
mkdir pytest-selenium-framework
cd pytest-selenium-framework
```

2. **Create virtual environment** (recommended):

```
python -m venv venv
source venv/bin/activate  # On Windows: venv\Scripts\activate
```

3. **Install dependencies**:

```
pip install fastmcp selenium pytest pytest-selenium webdriver-manager
```

4. **Create directory structure**:

```
mkdir -p config tests pages screenshots reports
```

5. **Save server implementation**:
   Save the MCP server code to `mcp_selenium_server.py` in the project root.
6. **Create framework files**:

- Save conftest.py example to `config/conftest.py`
- Save pytest.ini example to `config/pytest.ini`
- Save commands.py example to `config/commands.py`

Testing the MCP Server Locally

Before integrating with GitHub Copilot, test the server independently [20] [21]:

Option 1: Using MCP Inspector (Recommended)

The MCP Inspector provides an interactive interface for testing tools:

```
# Terminal 1: Run server in dev mode
mcp dev mcp_selenium_server.py

# Terminal 2: Launch inspector
mcp inspector
```

This opens a web interface where you can:

- View available tools, resources, and prompts

- Test tool invocations with different parameters
- Inspect request/response payloads
- Debug issues in real-time

Option 2: Programmatic Testing

Create a test client script:

```
# test_mcp_client.py
from fastmcp import Client
import asyncio
import json

async def test_server():
    # Connect to server
    client = Client("mcp_selenium_server.py")

    async with client:
        # Test ping
        await client.ping()
        print("✓ Server connected successfully")

        # Test loading framework context
        result = await client.call_tool(
            "load_framework_context",
            {
                "conftest_path": "config/conftest.py",
                "pytest_ini_path": "config/pytest.ini",
                "commands_path": "config/commands.py"
            }
        )
        print(f"✓ Framework context loaded: {result.data}")

        # Test extracting page elements
        elements = await client.call_tool(
            "navigate_and_extract_elements",
            {
                "url": "https://the-internet.herokuapp.com/login",
                "headless": True
            }
        )
        print(f"✓ Extracted {elements.data['element_count']} elements")

        # Test generating script
        script = await client.call_tool(
            "generate_pytest_selenium_script",
            {
                "test_name": "test_login",
                "url": "https://the-internet.herokuapp.com/login",
                "actions": [
                    {
                        "action": "type",
                        "element_name": "username",
                        "by": "ID",
```

```
                        "locator": "username",
                        "value": "tomsmith",
                        "method_name": "enter_username"
                    },
                    {
                        "action": "type",
                        "element_name": "password",
                        "by": "ID",
                        "locator": "password",
                        "value": "SuperSecretPassword!",
                        "method_name": "enter_password"
                    },
                    {
                        "action": "click",
                        "element_name": "login_button",
                        "by": "CSS_SELECTOR",
                        "locator": "button[type='submit']",
                        "method_name": "click_login"
                    }
                ],
                "use_page_object": True
            }
        )
        print("✓ Test script generated successfully")
        print("\nGenerated Script:")
        print("=" * 60)
        print(script.data)

if __name__ == "__main__":
    asyncio.run(test_server())
```

Run the test:

```
python test_mcp_client.py
```

GitHub Copilot Integration Configuration

VS Code Configuration

GitHub Copilot in VS Code supports MCP servers through the settings[22] [23]:

1. **Open VS Code Settings**: Press `Cmd/Ctrl + ,`
2. **Edit settings.json**: Click the "Open Settings (JSON)" icon
3. **Add MCP server configuration**:

```
{
  "github.copilot.advanced": {
    "mcp": {
      "servers": {
        "pytest-selenium-generator": {
          "command": "python",
```

```
          "args": [
            "/Users/yourname/pytest-selenium-framework/mcp_selenium_server.py"
          ],
          "env": {},
          "cwd": "/Users/yourname/pytest-selenium-framework"
        }
      }
    }
  }
}
```

Important: Use absolute paths for both the script and working directory.

4. **Restart VS Code** to apply changes
5. **Verify connection**: Open Copilot Chat and type `@pytest-selenium-generator` - you should see it in the autocomplete suggestions

Cursor Configuration

Cursor supports MCP through its configuration file [24] [25]:

Global Configuration (applies to all projects):

- Location: `~/.cursor/mcp.json`

Project Configuration (applies to single project):

- Location: `.cursor/mcp.json` in project root

Create or edit the file with:

```
{
  "mcpServers": {
    "pytest-selenium-generator": {
      "command": "python",
      "args": [
        "/absolute/path/to/mcp_selenium_server.py"
      ],
      "cwd": "/absolute/path/to/pytest-selenium-framework",
      "env": {}
    }
  }
}
```

Activation steps:

1. Save the configuration file
2. Open Cursor → Settings → MCP
3. Ensure the server appears and is enabled
4. Restart Cursor if necessary

Claude Desktop Configuration

Claude Desktop can integrate MCP servers for enhanced capabilities[22] [24]:

Configuration file locations:

- macOS: `~/Library/Application Support/Claude/claude_desktop_config.json`
- Windows: `%APPDATA%\Claude\claude_desktop_config.json`

Edit or create the file:

```
{
  "mcpServers": {
    "pytest-selenium-generator": {
      "command": "python",
      "args": [
        "/absolute/path/to/mcp_selenium_server.py"
      ],
      "cwd": "/absolute/path/to/pytest-selenium-framework"
    }
  }
}
```

Activation:

1. Save the configuration
2. Fully quit Claude Desktop (don't just close the window)
3. Restart Claude Desktop
4. Look for MCP indicator in the chat interface
5. Click the indicator to browse available tools

5. Integration with GitHub Copilot

Understanding GitHub Copilot's Context System

GitHub Copilot uses multiple sources of context to generate code suggestions[26] [27]:

Local context:

- Lines before and after cursor position
- Open files in the editor
- Active document content
- Code selection

Workspace context:

- Repository structure and file paths
- Frameworks and dependencies detected

- Language and libraries used

MCP context (when configured):

- Exposed tools from MCP servers
- Resources available through MCP
- Prompts defined in MCP servers

When you integrate an MCP server, GitHub Copilot can invoke its tools during agent mode to access capabilities beyond its training data[28] [22] [23].

Using MCP Tools in GitHub Copilot Chat

Once your MCP server is configured, you can interact with it through GitHub Copilot Chat using agent mode[28] [22]:

Activating Agent Mode:

1. Open Copilot Chat in your IDE
2. Click the "Agent" mode toggle in the chat interface
3. The agent can now access MCP tools

Invoking MCP Tools:

Example 1: Load Framework Context

In Copilot Chat, type:

```
@pytest-selenium-generator load_framework_context with paths:
- conftest: config/conftest.py
- pytest_ini: config/pytest.ini
- commands: config/commands.py
```

Copilot will invoke the tool and confirm loading.

Example 2: Extract Page Elements

```
@pytest-selenium-generator navigate to https://example.com/login
and extract all interactive elements
```

Response will include element details like:

- Input fields with IDs, names, types
- Buttons with text and locators
- Links and other interactive elements

Example 3: Generate Complete Test

```
@pytest-selenium-generator create a pytest test for login functionality:
- Navigate to https://example.com/login
- Enter username "testuser" in field with id="username"
- Enter password "testpass" in field with id="password"
- Click submit button with id="login-btn"
- Verify successful login by checking for dashboard element
- Use Page Object Model pattern
```

The generated test will:

- Follow your framework's structure from conftest.py
- Use utility methods from commands.py
- Apply pytest.ini configuration
- Match your naming conventions
- Include proper imports and fixtures

Best Practices for Prompting

Be specific about requirements[27]:

```
Generate a test that:
- Uses the driver fixture from conftest.py
- Implements Page Object Model
- Uses wait_and_click utility from commands.py
- Includes @pytest.mark.ui marker
- Has descriptive method names
```

Reference framework context:

```
Following the patterns in our conftest.py and using the utility methods
from commands.py, generate a test for...
```

Iterate and refine:

```
# First prompt
Generate a basic login test

# Second prompt (after reviewing)
Update the test to use POM pattern and add error handling

# Third prompt
Add assertions to verify login success and failure cases
```

6. Usage Examples and Best Practices

Complete Workflow Example

Let's walk through a complete example of using the MCP server with GitHub Copilot to create a login test.

Step 1: Load Framework Context

First, ensure your framework files are loaded into the MCP server context:

```
In Copilot Chat:

@pytest-selenium-generator load the framework context from:
config/conftest.py, config/pytest.ini, and config/commands.py
```

Step 2: Analyze Target Page

Extract elements from the page you'll be testing:

```
@pytest-selenium-generator navigate to
https://the-internet.herokuapp.com/login
and show me all the form elements
```

MCP server response will include:

```
{
  "url": "https://the-internet.herokuapp.com/login",
  "title": "The Internet",
  "elements": [
    {
      "tag": "input",
      "id": "username",
      "type": "text",
      "name": "username"
    },
    {
      "tag": "input",
      "id": "password",
      "type": "password",
      "name": "password"
    },
    {
      "tag": "button",
      "class": "radius",
      "type": "submit",
      "text": "Login"
    }
  ]
}
```

Step 3: Generate Test Script

Now generate the complete test:

```
@pytest-selenium-generator create a comprehensive login test:

Test Name: test_valid_login
URL: https://the-internet.herokuapp.com/login
Pattern: Page Object Model

Actions:
1. Enter "tomsmith" into username field (id=username)
2. Enter "SuperSecretPassword!" into password field (id=password)
3. Click the login button (css=button[type='submit'])
4. Verify success message appears

Use our framework patterns:
- driver fixture from conftest.py
- wait_and_type and wait_and_click from commands.py
- @pytest.mark.smoke marker
- Proper docstrings and comments
```

Generated Output:

```
"""
Generated pytest-selenium automation script
Test: test_valid_login
URL: https://the-internet.herokuapp.com/login
Pattern: Page Object Model
"""

import pytest
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
from utils.commands import *

class LoginPage:
    """Page Object for login functionality"""

    def __init__(self, driver):
        self.driver = driver
        self.url = "https://the-internet.herokuapp.com/login"

    # Locators
    USERNAME = (By.ID, "username")
    PASSWORD = (By.ID, "password")
    LOGIN_BUTTON = (By.CSS_SELECTOR, "button[type='submit']")
    SUCCESS_MESSAGE = (By.CSS_SELECTOR, "#flash.success")
    ERROR_MESSAGE = (By.CSS_SELECTOR, "#flash.error")

    def load(self):
        """Navigate to login page"""
```

```
        self.driver.get(self.url)
        return self

    def enter_username(self, username):
        """Enter username in the username field"""
        wait_and_type(self.driver, *self.USERNAME, username)
        return self

    def enter_password(self, password):
        """Enter password in the password field"""
        wait_and_type(self.driver, *self.PASSWORD, password)
        return self

    def click_login(self):
        """Click the login button"""
        wait_and_click(self.driver, *self.LOGIN_BUTTON)
        return self

    def is_success_message_visible(self):
        """Check if success message is displayed"""
        return is_element_visible(self.driver, *self.SUCCESS_MESSAGE)

    def get_success_message(self):
        """Get success message text"""
        return get_element_text(self.driver, *self.SUCCESS_MESSAGE)

@pytest.mark.smoke
@pytest.mark.ui
@pytest.mark.usefixtures("driver")
class TestLogin:
    """Test suite for login functionality"""

    def test_valid_login(self, driver):
        """
        Test successful login with valid credentials.

        Steps:
        1. Navigate to login page
        2. Enter valid username
        3. Enter valid password
        4. Click login button
        5. Verify success message appears
        """
        # Arrange
        page = LoginPage(driver).load()

        # Act
        page.enter_username("tomsmith") \
            .enter_password("SuperSecretPassword!") \
            .click_login()

        # Assert
        assert page.is_success_message_visible(), \
            "Success message should be visible after valid login"
        assert "You logged into a secure area!" in page.get_success_message(), \
            "Success message should contain expected text"
```

Test Script Patterns

Pattern 1: Simple Linear Test

For quick tests without POM:

```
@pytest-selenium-generator create a simple test:
- Don't use Page Object Model
- Just basic selenium commands
- Test name: test_simple_form
- Fill out a contact form and submit
```

Generated:

```
import pytest
from selenium.webdriver.common.by import By
from utils.commands import wait_and_type, wait_and_click

@pytest.mark.smoke
def test_simple_form(driver):
    """Simple form submission test"""
    # Navigate
    driver.get("https://example.com/contact")

    # Fill form
    wait_and_type(driver, By.ID, "name", "John Doe")
    wait_and_type(driver, By.ID, "email", "john@example.com")
    wait_and_type(driver, By.ID, "message", "Test message")

    # Submit
    wait_and_click(driver, By.ID, "submit")

    # Verify
    assert is_element_visible(driver, By.CLASS_NAME, "success")
```

Pattern 2: Data-Driven Test

Using pytest parametrize:

```
@pytest-selenium-generator create a data-driven login test
with multiple credential sets using @pytest.mark.parametrize
```

Generated:

```
import pytest
from selenium.webdriver.common.by import By
from utils.commands import *

@pytest.mark.parametrize("username,password,expected_result", [
    ("tomsmith", "SuperSecretPassword!", "success"),
```

```
    ("invaliduser", "wrongpass", "error"),
    ("", "", "error"),
])
@pytest.mark.regression
def test_login_scenarios(driver, username, password, expected_result):
    """
    Test login with various credential combinations.

    Args:
        username: Username to test
        password: Password to test
        expected_result: Expected outcome (success/error)
    """
    # Navigate
    driver.get("https://the-internet.herokuapp.com/login")

    # Perform login
    if username:
        wait_and_type(driver, By.ID, "username", username)
    if password:
        wait_and_type(driver, By.ID, "password", password)
    wait_and_click(driver, By.CSS_SELECTOR, "button[type='submit']")

    # Verify result
    if expected_result == "success":
        assert is_element_visible(
            driver, By.CSS_SELECTOR, "#flash.success"
        )
    else:
        assert is_element_visible(
            driver, By.CSS_SELECTOR, "#flash.error"
        )
```

Pattern 3: Multi-Page Workflow

Testing across multiple pages:

```
@pytest-selenium-generator create a test for complete checkout flow:
1. Search for product
2. Add to cart
3. Proceed to checkout
4. Fill shipping info
5. Complete payment

Use Page Object Model with separate page classes
```

Framework Integration Best Practices

1. Always load context first [1] [2]

Before generating any tests, load your framework files:

```
# First action in your session
load_framework_context(
    conftest_path="config/conftest.py",
    pytest_ini_path="config/pytest.ini",
    commands_path="config/commands.py"
)
```

2. Leverage utility methods [18] [29]

Generated tests should use your framework's utilities:

```
# Good: Uses framework utility
wait_and_type(driver, By.ID, "email", "user@example.com")

# Avoid: Raw Selenium calls
element = driver.find_element(By.ID, "email")
element.send_keys("user@example.com")
```

3. Follow consistent naming [14] [30]

Match your framework's conventions:

```
# Test functions
def test_descriptive_name(driver):  # Good
def TestCase(driver):  # Avoid

# Page classes
class LoginPage:  # Good
class login_page:  # Avoid

# Methods
def click_submit_button(self):  # Good
def submit(self):  # Less descriptive
```

4. Apply appropriate markers [16] [17]

Use pytest markers from your configuration:

```
@pytest.mark.smoke  # Quick tests
@pytest.mark.regression  # Full suite
@pytest.mark.ui  # UI-specific
@pytest.mark.slow  # Long-running tests
```

5. Include proper waits [18] [30]

Always use explicit waits, not sleep:

```
# Good: Explicit wait
wait_for_element(driver, By.ID, "result", timeout=15)
```

```
# Avoid: Hard-coded sleep
time.sleep(5)
```

6. Add meaningful assertions [31] [32]

Tests should verify specific expected outcomes:

```
# Good: Specific assertion
assert page.get_success_message() == "Login successful"
assert driver.current_url.endswith("/dashboard")

# Weak: Generic assertion
assert True
```

7. Handle errors gracefully [18] [33]

Include error handling and cleanup:

```
try:
    page.perform_action()
    assert expected_result
except Exception as e:
    take_screenshot(driver, f"error_{test_name}")
    logger.error(f"Test failed: {str(e)}")
    raise
```

Code Review Checklist

Before committing generated tests, verify:

- [ ] Framework context was loaded
- [ ] Test follows POM or agreed pattern
- [ ] Utility methods from commands.py are used
- [ ] Fixtures from conftest.py are properly applied
- [ ] pytest.ini markers are included
- [ ] Descriptive docstrings present
- [ ] Meaningful variable and method names
- [ ] Explicit waits instead of sleeps
- [ ] Specific assertions with clear messages
- [ ] Error handling and cleanup
- [ ] Test is independent and can run in any order
- [ ] No hard-coded sensitive data

7. Advanced Features and Customization

Adding Custom Tools

You can extend the MCP server with additional tools for specific needs:

Example: Screenshot Comparison Tool

```
@mcp.tool()
def capture_baseline_screenshot(
    url: str,
    screenshot_name: str,
    element_selector: Optional[str] = None
) -&gt; Dict:
    """
    Capture baseline screenshot for visual regression testing.

    Args:
        url: Page URL
        screenshot_name: Name for screenshot file
        element_selector: Optional CSS selector for specific element

    Returns:
        Dictionary with screenshot path and metadata
    """
    options = webdriver.ChromeOptions()
    options.add_argument('--headless')
    options.add_argument('--window-size=1920,1080')

    driver = webdriver.Chrome(options=options)
    driver.get(url)

    if element_selector:
        element = driver.find_element(By.CSS_SELECTOR, element_selector)
        screenshot_path = f"baselines/{screenshot_name}_element.png"
        element.screenshot(screenshot_path)
    else:
        screenshot_path = f"baselines/{screenshot_name}_full.png"
        driver.save_screenshot(screenshot_path)

    metadata = {
        "url": url,
        "screenshot_path": screenshot_path,
        "timestamp": datetime.now().isoformat(),
        "viewport": "1920x1080"
    }

    driver.quit()
    return metadata
```

Example: Test Data Generator

```
@mcp.tool()
def generate_test_data(
    data_type: str,
    count: int = 1
) -&gt; List[Dict]:
    """
    Generate realistic test data for automation.

    Args:
        data_type: Type of data (user, product, order)
        count: Number of records to generate

    Returns:
        List of generated data dictionaries
    """
    from faker import Faker
    fake = Faker()

    if data_type == "user":
        return [{
            "first_name": fake.first_name(),
            "last_name": fake.last_name(),
            "email": fake.email(),
            "phone": fake.phone_number(),
            "address": fake.address()
        } for _ in range(count)]

    elif data_type == "product":
        return [{
            "name": fake.catch_phrase(),
            "price": round(fake.random.uniform(10, 1000), 2),
            "description": fake.text(max_nb_chars=200),
            "sku": fake.ean13()
        } for _ in range(count)]

    return []
```

Adding Custom Resources

Resources provide static or dynamic reference data:

Example: Common Locator Library

```
@mcp.resource("locators://common")
def get_common_locators():
    """Provide library of common locator patterns"""
    return '''
# Common Locator Patterns

## Login Forms
- Username: By.ID, "username" or By.NAME, "username"
- Password: By.ID, "password" or By.NAME, "password"
- Submit: By.CSS_SELECTOR, "button[type='submit']"
```

```
## Navigation
- Header: By.TAG_NAME, "header"
- Footer: By.TAG_NAME, "footer"
- Nav Menu: By.CSS_SELECTOR, "nav" or By.TAG_NAME, "nav"

## Messages
- Success: By.CLASS_NAME, "alert-success"
- Error: By.CLASS_NAME, "alert-error"
- Warning: By.CLASS_NAME, "alert-warning"

## Forms
- Input fields: By.CSS_SELECTOR, "input[type='text']"
- Checkboxes: By.CSS_SELECTOR, "input[type='checkbox']"
- Dropdowns: By.TAG_NAME, "select"
'''
```

Example: Test Templates

````
@mcp.resource("templates://crud-test")
def get_crud_test_template():
    """Provide template for CRUD operation tests"""
    return '''
# CRUD Test Template

Use this pattern for Create-Read-Update-Delete tests:

```python
@pytest.mark.regression
class TestCRUD:

    def test_create_record(self, driver):
        """Test creating a new record"""
        # Navigate to create page
        # Fill form with data
        # Submit form
        # Verify success message
        # Verify record appears in list

    def test_read_record(self, driver):
        """Test reading/viewing a record"""
        # Navigate to list
        # Click on record
        # Verify details displayed correctly

    def test_update_record(self, driver):
        """Test updating an existing record"""
        # Navigate to record
        # Click edit
        # Modify fields
        # Save changes
        # Verify updated values

    def test_delete_record(self, driver):
        """Test deleting a record"""
        # Navigate to record
````
````

```
        # Click delete
        # Confirm deletion
        # Verify record removed from list
```

'''

```
### Adding Custom Prompts

Prompts help structure complex requests:

**Example: Refactoring Prompt**

```python
@mcp.prompt()
def refactor_to_pom(test_code: str) -> List[Message]:
    """
    Generate prompt to refactor existing test to POM pattern.

    Args:
        test_code: Existing test code to refactor

    Returns:
        Structured messages for LLM
    """
    from mcp.server.fastmcp.prompts import base

    return [
        base.Message(
            role="user",
            content=[
                base.TextContent(
                    text=f'''
Refactor the following test to use Page Object Model pattern:

```python
{test_code}
```

Requirements:

1. Create a separate page class with locators
2. Add descriptive method names for each action
3. Use method chaining where appropriate
4. Move all element interactions to page class
5. Keep test logic clean and readable
6. Follow our framework patterns from conftest.py
7. Use utility methods from commands.py

Provide the complete refactored code with both page class and test class.
'''
```
```

)
]
)
]

```
### Validation and Quality Checks

Add tools to validate generated code:

```python
@mcp.tool()
def validate_generated_script(script_content: str) -> Dict:
    """
    Validate that generated script follows framework standards.

    Performs static analysis and pattern matching to ensure
    generated code adheres to framework conventions.

    Args:
        script_content: Generated Python test script

    Returns:
        Validation results with issues and recommendations
    """
    issues = []
    warnings = []

    # Check required imports
    required_imports = ['pytest', 'selenium']
    for imp in required_imports:
        if imp not in script_content:
            issues.append(f"Missing required import: {imp}")

    # Check for fixture usage
    conftest_content = framework_context.get('conftest', '')
    if '@pytest.fixture' in conftest_content:
        if 'def driver' in script_content and \
           '@pytest.fixture' not in script_content:
            issues.append(
                "Test should use driver fixture, not create driver directly"
            )

    # Check test naming convention
    if not any(line.strip().startswith('def test_')
               for line in script_content.split('\n')):
        issues.append("Test functions must start with 'test_'")

    # Check for hard-coded waits
    if 'time.sleep' in script_content:
        warnings.append(
            "Avoid time.sleep, use explicit waits instead"
        )

    # Check for utility method usage
```
```

```
    commands_content = framework_context.get('commands', '')
    if commands_content:
        utility_methods = extract_utility_methods(commands_content)
        if utility_methods and 'driver.find_element' in script_content:
            warnings.append(
                f"Consider using utility methods: {', '.join(utility_methods[:3])}"
            )

    # Check for assertions
    if 'assert' not in script_content:
        issues.append("Test should include at least one assertion")

    # Check for docstrings
    if '"""' not in script_content and "'''" not in script_content:
        warnings.append("Add docstrings to classes and methods")

    return {
        "valid": len(issues) == 0,
        "issues": issues,
        "warnings": warnings,
        "issue_count": len(issues),
        "warning_count": len(warnings),
        "recommendations": [
            "Review framework patterns in conftest.py",
            "Use utility methods from commands.py",
            "Add descriptive docstrings",
            "Include specific assertions"
        ] if issues or warnings else []
    }
```

Integration with CI/CD

Configure the server for continuous integration:

```
@mcp.tool()
def generate_ci_config(
    framework: str = "github_actions",
    python_version: str = "3.9"
) -&gt; str:
    """
    Generate CI/CD configuration for pytest-selenium tests.

    Args:
        framework: CI platform (github_actions, gitlab_ci, jenkins)
        python_version: Python version to use

    Returns:
        CI configuration file content
    """
    if framework == "github_actions":
        return f'''
name: Selenium Tests

on:
  push:
```

```
    branches: [ main, develop ]
  pull_request:
    branches: [ main ]

jobs:
  test:
    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v3

    - name: Set up Python {python_version}
      uses: actions/setup-python@v4
      with:
        python-version: {python_version}

    - name: Install dependencies
      run: |
        python -m pip install --upgrade pip
        pip install -r requirements.txt

    - name: Install Chrome
      run: |
        sudo apt-get update
        sudo apt-get install -y google-chrome-stable

    - name: Run tests
      run: |
        pytest tests/ --html=report.html --self-contained-html

    - name: Upload test results
      if: always()
      uses: actions/upload-artifact@v3
      with:
        name: test-results
        path: report.html
'''

    elif framework == "gitlab_ci":
        return f'''
image: python:{python_version}

stages:
  - test

selenium_tests:
  stage: test
  before_script:
    - pip install -r requirements.txt
    - apt-get update &amp;&amp; apt-get install -y wget gnupg
    - wget -q -O - https://dl-ssl.google.com/linux/linux_signing_key.pub | apt-key add -
    - echo "deb http://dl.google.com/linux/chrome/deb/ stable main" &gt;&gt; /etc/apt/sou
    - apt-get update &amp;&amp; apt-get install -y google-chrome-stable
  script:
    - pytest tests/ --html=report.html --self-contained-html
  artifacts:
```

```
    when: always
    paths:
      - report.html
    expire_in: 30 days
'''

### Performance Optimization

Add tools for performance testing:

```python
@mcp.tool()
def measure_page_load_time(url: str, iterations: int = 3) -> Dict:
    """
    Measure page load performance metrics.

    Args:
        url: Page URL to test
        iterations: Number of measurements to take

    Returns:
        Performance metrics and averages
    """
    import time
    from statistics import mean, stdev

    load_times = []
    options = webdriver.ChromeOptions()
    options.add_argument('--headless')

    for i in range(iterations):
        driver = webdriver.Chrome(options=options)

        start_time = time.time()
        driver.get(url)

        # Wait for page to be fully loaded
        WebDriverWait(driver, 30).until(
            lambda d: d.execute_script('return document.readyState') == 'complete'
        )

        end_time = time.time()
        load_time = end_time - start_time
        load_times.append(load_time)

        driver.quit()

    return {
        "url": url,
        "iterations": iterations,
        "load_times": load_times,
        "average_load_time": mean(load_times),
        "std_deviation": stdev(load_times) if len(load_times) > 1 else 0,
        "min_load_time": min(load_times),
        "max_load_time": max(load_times)
    }
```
```

Conclusion

This comprehensive guide has covered the complete process of building an MCP server for pytest-selenium automation with GitHub Copilot integration. The key benefits of this approach include:

Context-Aware Generation: By loading your framework files (conftest.py, pytest.ini, commands.py), the MCP server ensures generated tests match your existing patterns and conventions [1] [2] [5].

Standardized Communication: MCP provides a protocol-based approach that works across multiple AI platforms (GitHub Copilot, Claude, Cursor) [3] [4].

Extensibility: The modular design allows easy addition of custom tools, resources, and prompts for your specific needs [8] [9].

Quality Assurance: Built-in validation tools help maintain code quality and framework consistency [18] [30].

Developer Productivity: Automating test script generation while maintaining framework standards significantly accelerates test development [23] [26].

Next Steps

To continue improving your MCP-powered test automation:

1. **Expand tool library**: Add tools for specific testing scenarios (API tests, mobile tests, visual regression)
2. **Enhance validation**: Implement more sophisticated code quality checks and linting
3. **Add test data management**: Integrate with test data generators and databases
4. **Implement reporting**: Create tools for test result analysis and reporting
5. **Support multiple frameworks**: Extend beyond pytest to support other testing frameworks
6. **Add Playwright support**: Implement browser automation using Playwright alongside Selenium
7. **Create template library**: Build a comprehensive library of test templates for common patterns
8. **Integrate with test management**: Connect to tools like TestRail or Zephyr for test case management

References

[1] https://northflank.com/blog/how-to-build-and-deploy-a-model-context-protocol-mcp-server
[2] https://aiengineering.academy/Agents/MCP/CreateMCPServe/
[28] https://www.youtube.com/watch?v=HN47tveqfQU
[5] https://simplescraper.io/blog/how-to-mcp
[34] https://github.com/modelcontextprotocol/create-python-server
[22] https://docs.github.com/en/copilot/how-tos/provide-context/use-mcp/use-the-github-mcp-server
[3] https://cloud.google.com/discover/what-is-model-context-protocol
[35] https://auth0.com/blog/build-python-mcp-server-for-blog-search/
[24] https://learn.microsoft.com/en-us/azure/app-service/tutorial-ai-model-context-protocol-server-node
[4] https://www.digitalocean.com/community/tutorials/model-context-protocol

[20] https://www.digitalocean.com/community/tutorials/mcp-server-python
[36] https://github.com/skills/integrate-mcp-with-copilot
[6] https://www.youtube.com/watch?v=kOhLoixrJXo
[37] https://www.youtube.com/watch?v=j5f2EQf5hkw
[38] https://learn.microsoft.com/en-us/azure/app-service/tutorial-ai-model-context-protocol-server-java
[39] https://modelcontextprotocol.io/docs/develop/build-server
[40] https://www.youtube.com/watch?v=-8k9lGpGQ6g
[25] https://docs.github.com/copilot/customizing-copilot/using-model-context-protocol/extending-copilot-chat-with-mcp
[41] https://www.datacamp.com/tutorial/mcp-model-context-protocol
[21] https://realpython.com/python-mcp/
[42] https://masteringbackend.com/posts/parameterized-testing-with-pytest-and-selenium
[14] https://qaautomation.expert/2024/04/02/page-object-model-implementation-of-python-with-selenium-pytest/
[16] https://docs.pytest.org/en/stable/example/simple.html
[43] https://www.linkedin.com/posts/zeeshanasgharr_testautomation-selenium-python-activity-7366696295639511040-8Uy8
[15] https://www.guvi.in/blog/python-selenium-page-object-model-explained/
[44] https://www.lambdatest.com/blog/end-to-end-tutorial-for-pytest-fixtures-with-examples/
[18] https://www.selenium.dev/documentation/test_practices/design_strategies/
[45] https://stackoverflow.com/questions/76983730/using-conftest-python-file-but-unable-to-access-self-driver-methods-from-test-c
[46] https://dev.to/m4rri4nne/automating-your-api-tests-using-python-and-pytest-23cc
[30] https://www.browserstack.com/guide/best-practices-in-selenium-automation
[19] https://www.browserstack.com/guide/page-object-model-in-selenium-python
[31] https://realpython.com/pytest-python-testing/
[17] https://www.softoneconsultancy.com/folder-structure-python-selenium-framework-with-pytest/
[47] https://www.youtube.com/watch?v=qBK5I_QApCg
[48] https://circleci.com/blog/pytest-python-testing/
[29] https://jignect.tech/best-practices-for-efficient-test-automation-with-selenium-and-python/
[49] https://nareshnavinash.github.io/Selpy-Python/
[50] https://www.blazemeter.com/blog/pytest-framework
[51] https://www.lambdatest.com/learning-hub/selenium-pytest-tutorial
[52] https://github.com/qxf2/qxf2-page-object-model
[53] https://playbooks.com/mcp/raghu6798-browser-scrape
[10] https://gofastmcp.com/tutorials/create-mcp-server
[12] https://gofastmcp.com/servers/context
[54] https://webscraping.ai/faq/scraping-with-mcp-servers/how-do-i-perform-browser-automation-using-mcp-servers
[11] https://thepythoncode.com/article/fastmcp-mcp-client-server-todo-manager
[13] https://socradar.io/mcp-for-cybersecurity/core-concepts/how-does-context-injection-work-in-practice/
[55] https://brightdata.com/blog/ai/web-scraping-with-mcp
[8] https://mcpcat.io/guides/adding-custom-tools-mcp-server-python/
[7] https://modelcontextprotocol.io/specification/2025-06-18/server/prompts
[56] https://www.youtube.com/watch?v=Rzu9_maHqn4

[57] https://www.speakeasy.com/mcp/framework-guides/building-fastapi-server
[23] https://code.visualstudio.com/docs/copilot/overview
[32] https://www.geeksforgeeks.org/python/writing-tests-using-selenium-python/
[26] https://github.com/features/copilot
[33] https://www.functionize.com/automated-testing
[27] https://docs.github.com/en/copilot/get-started/best-practices

[58] [59] [60] [61] [62] [63] [64] [65] [66] [67] [68] [69] [70] [71] [72] [73] [74] [75] [76] [77] [78] [79] [80]

⁂

1. https://northflank.com/blog/how-to-build-and-deploy-a-model-context-protocol-mcp-server
2. https://aiengineering.academy/Agents/MCP/CreateMCPServe/
3. https://cloud.google.com/discover/what-is-model-context-protocol
4. https://www.digitalocean.com/community/tutorials/model-context-protocol
5. https://simplescraper.io/blog/how-to-mcp
6. https://www.youtube.com/watch?v=kOhLoixrJXo
7. https://modelcontextprotocol.io/specification/2025-06-18/server/prompts
8. https://mcpcat.io/guides/adding-custom-tools-mcp-server-python/
9. https://gofastmcp.com/servers/tools
10. https://gofastmcp.com/tutorials/create-mcp-server
11. https://thepythoncode.com/article/fastmcp-mcp-client-server-todo-manager
12. https://gofastmcp.com/servers/context
13. https://socradar.io/mcp-for-cybersecurity/core-concepts/how-does-context-injection-work-in-practice/
14. https://qaautomation.expert/2024/04/02/page-object-model-implementation-of-python-with-selenium-pytest/
15. https://www.guvi.in/blog/python-selenium-page-object-model-explained/
16. https://docs.pytest.org/en/stable/example/simple.html
17. https://www.softoneconsultancy.com/folder-structure-python-selenium-framework-with-pytest/
18. https://www.selenium.dev/documentation/test_practices/design_strategies/
19. https://www.browserstack.com/guide/page-object-model-in-selenium-python
20. https://www.digitalocean.com/community/tutorials/mcp-server-python
21. https://realpython.com/python-mcp/
22. https://docs.github.com/en/copilot/how-tos/provide-context/use-mcp/use-the-github-mcp-server
23. https://code.visualstudio.com/docs/copilot/overview
24. https://learn.microsoft.com/en-us/azure/app-service/tutorial-ai-model-context-protocol-server-node
25. https://docs.github.com/copilot/customizing-copilot/using-model-context-protocol/extending-copilot-chat-with-mcp
26. https://github.com/features/copilot
27. https://docs.github.com/en/copilot/get-started/best-practices
28. https://www.youtube.com/watch?v=HN47tveqfQU
29. https://jignect.tech/best-practices-for-efficient-test-automation-with-selenium-and-python/

30. https://www.browserstack.com/guide/best-practices-in-selenium-automation
31. https://realpython.com/pytest-python-testing/
32. https://www.geeksforgeeks.org/python/writing-tests-using-selenium-python/
33. https://www.functionize.com/automated-testing
34. https://github.com/modelcontextprotocol/create-python-server
35. https://auth0.com/blog/build-python-mcp-server-for-blog-search/
36. https://github.com/skills/integrate-mcp-with-copilot
37. https://www.youtube.com/watch?v=j5f2EQf5hkw
38. https://learn.microsoft.com/en-us/azure/app-service/tutorial-ai-model-context-protocol-server-java
39. https://modelcontextprotocol.io/docs/develop/build-server
40. https://www.youtube.com/watch?v=-8k9lGpGQ6g
41. https://www.datacamp.com/tutorial/mcp-model-context-protocol
42. https://masteringbackend.com/posts/parameterized-testing-with-pytest-and-selenium
43. https://www.linkedin.com/posts/zeeshanasgharr_testautomation-selenium-python-activity-7366696295639511040-8Uy8
44. https://www.lambdatest.com/blog/end-to-end-tutorial-for-pytest-fixtures-with-examples/
45. https://stackoverflow.com/questions/76983730/using-conftest-python-file-but-unable-to-access-self-driver-methods-from-test-c
46. https://dev.to/m4rri4nne/automating-your-api-tests-using-python-and-pytest-23cc
47. https://www.youtube.com/watch?v=qBK5I_QApCg
48. https://circleci.com/blog/pytest-python-testing/
49. https://nareshnavinash.github.io/Selpy-Python/
50. https://www.blazemeter.com/blog/pytest-framework
51. https://www.lambdatest.com/learning-hub/selenium-pytest-tutorial
52. https://github.com/qxf2/qxf2-page-object-model
53. https://playbooks.com/mcp/raghu6798-browser-scrape
54. https://webscraping.ai/faq/scraping-with-mcp-servers/how-do-i-perform-browser-automation-using-mcp-servers
55. https://brightdata.com/blog/ai/web-scraping-with-mcp
56. https://www.youtube.com/watch?v=Rzu9_maHqn4
57. https://www.speakeasy.com/mcp/framework-guides/building-fastapi-server
58. https://devblogs.microsoft.com/dotnet/build-a-model-context-protocol-mcp-server-in-csharp/
59. https://modelcontextprotocol.io/examples
60. https://modelcontextprotocol-security.io/ttps/command-injection/code-injection/
61. https://github.com/punkpeye/awesome-mcp-servers
62. https://www.firecrawl.dev/blog/fastmcp-tutorial-building-mcp-servers-python
63. https://www.practical-devsecops.com/mcp-security-vulnerabilities/
64. https://hexmos.com/freedevtools/mcp/browser-automation/tatn--mcp-server-fetch-python/
65. https://github.com/jlowin/fastmcp
66. https://www.blazemeter.com/blog/ui-test-automation

67. https://learn.microsoft.com/en-us/sql/tools/visual-studio-code-extensions/github-copilot/code-generation?view=sql-server-ver17
68. https://saucelabs.com/resources/blog/selenium-with-python-for-automated-testing
69. https://applitools.com/blog/transform-user-actions-into-cypress-tests-with-applitools-testgenai/
70. https://www.browserstack.com/guide/python-selenium-to-run-web-automation-test
71. https://www.prolifics-testing.com/news/genai-test-automation-concept
72. https://stackoverflow.com/questions/76741410/how-to-invoke-github-copilot-programmatically
73. https://www.headspin.io/blog/how-to-write-an-automated-test-script-using-selenium
74. https://playwright.dev/docs/codegen
75. https://www.reddit.com/r/GithubCopilot/comments/1gox5ng/how_to_feedprovide_documentations_to_github/
76. https://chromewebstore.google.com/detail/selenium-auto-code-genera/ocimgcpcnobcnmclomhhmjidgoiekeaf?hl=en
77. https://github.blog/ai-and-ml/github-copilot/github-for-beginners-building-a-rest-api-with-copilot/
78. https://www.selenium.dev/documentation/webdriver/getting_started/first_script/
79. https://www.lambdatest.com/blog/automatic-test-case-generation/
80. https://stackoverflow.com/questions/67645196/how-do-i-generate-python-selenium-code-from-a-gui