AI in Software Testing

Completed source code for all labs (for checking your work) can be found at:

`https://github.com/chrisminnick/ai-in-software-testing`

Table of Contents

Disclaimers and Copyright Statement

Disclaimer

WatzThis? takes care to ensure the accuracy and quality of this courseware and related courseware files. We cannot guarantee the accuracy of these materials. The courseware and related files are provided without any warranty whatsoever, including but not limited to implied warranties of merchantability or fitness for a particular purpose. Use of screenshots, product names and icons in the courseware are for editorial purposes only. No such use should be construed to imply sponsorship or endorsement of the courseware, nor any affiliation of such entity with WatzThis?.

Third-Party Information

This courseware contains links to third-party web sites that are not under our control and we are not responsible for the content of any linked sites. If you access a third-party web site mentioned in this courseware, then you do so at your own risk. We provide these links only as a convenience, and the inclusion of the link does not imply that we endorse or accept responsibility for the content on those third-party web sites. Information in this courseware may change without notice and does not represent a commitment on the part of the authors and publishers.

Copyright

Help us improve our courseware

Please send your comments and suggestions via email to info@watzthis.com

Credits

About the Author

Chris Minnick is a prolific published author, trainer, web developer and founder of WatzThis, Inc. Minnick has overseen the development of hundreds of web and mobile projects for customers from small businesses to some of the world's largest companies, including Microsoft, United Business Media, Penton Publishing, and Stanford University.

Since 2001, Minnick has trained thousands of Web and mobile developers. In addition to his in-person courses, Chris has written and produced online courses for Ed2Go.com, Skillshare, O'Reilly Media, and Pluralsight.

Minnick has authored and co-authored books and articles on a wide range of Internet-related topics including JavaScript, ReactJS, HTML, CSS, mobile apps, e-commerce, Web design, SEO, and security. His published books include Artificial Intelligence All-In-One For Dummies, Microsoft Copilot For Dummies, JavaScript All-In-One For Dummies, Coding All-In-One For Dummies, ReactJS Foundations, JavaScript for Kids, Writing Computer Code, Coding with JavaScript For Dummies, Beginning HTML5 and CSS3 For Dummies, Webkit For Dummies, CIW eCommerce Certification Bible, and XHTML.

Course Requirements

To complete the labs in this course, you will need:
- A computer with MacOS, Windows, or Linux.
- Access to the Internet.
- A modern web browser.
- Ability to install software globally (or certain packages pre-installed as specified below).

Introduction and Git Repo Info

Most of the labs in this course build on the labs that came before. So, if you don't complete a lab or can't get a certain lab to work, it's possible that you can get stuck and won't be able to move forward until the error is corrected.

To help you check your work and to make it possible to come into the class at any point, the git repository for this course contains finished versions of every lab.

The url for the course repository is:
`https://github.com/chrisminnick/ai-in-software-testing`

You can find the finished code for each lab inside the **solutions** directory.

Lab 1: Introduction to AI in Software Testing

Objectives:

- Understand how to interact with an AI assistant to generate test cases.
- Practice writing a basic function and asking AI to generate a test for it.
- Learn to install and run Jest to execute tests.

Instructions:

- ☐ 1. Set up your environment
- ☐ 2. Open your code editor and create a new project folder (e.g., `ai-lab-01`).
- ☐ 3. In your terminal, navigate to the folder and initialize a Node.js project:

```
npm init -y
```

- ☐ 4. Create a simple JavaScript function
- ☐ 5. Inside your project, create a file named `sum.js`:

```
function sum(a, b) {

  return a + b;

}


module.exports = sum;
```

- ☐ 6. Ask ChatGPT or GitHub Copilot to write a Jest test
- ☐ 7. Prompt: "Write a Jest test for a function named `sum` that adds two numbers."
- ☐ 8. Review the AI-generated output and ensure it includes an `expect()` assertion.
- ☐ 9. Create your test file
- ☐ 10. Create a file named `sum.test.js` and paste in the test from ChatGPT:

```
const sum = require('./sum');


test('adds 1 + 2 to equal 3', () => {

  expect(sum(1, 2)).toBe(3);

});
```

- ☐ 11. Install Jest
- ☐ 12. In your terminal, run:

```
npm install --save-dev jest
```

- ☐ 13. Add a test script to your package.json
- ☐ 14. Edit the `scripts` section like this:

```
"scripts": {

  "test": "jest"

}
```

☐ 15. Run your test
☐ 16. In the terminal:

```
npm test
```

☐ 17. Experiment with different prompts
☐ 18. Try asking ChatGPT: "What happens if I pass strings to `sum()`?"
☐ 19. Update or add new test cases based on the output.

Discussion and Sharing:

- What changes in the test when you change the prompt?
- Was the AI-generated test easy to understand?
- What would you ask next if the output wasn't what you expected?

Lab 2: Test Case Generation with LLMs

Objectives:

- Use AI prompting to generate test cases, including edge cases.
- Learn how to analyze AI-generated tests for completeness and quality.
- Practice iterating on prompts to improve test coverage.

Instructions:

☐ 1. Prepare a non-trivial JavaScript function

Create a file named `validateEmail.js` and add this code:

```
function validateEmail(email) {
  const re = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
  return re.test(email);
}

module.exports = validateEmail;
```

Note
As of this writing, Jest uses CommonJS module syntax by default. To use ECMAScript modules, follow the instructions here: https://jestjs.io/docs/ecmascript-modules.

☐ 2. Ask ChatGPT to generate test cases

Prompt: "Write five Jest tests for a function that validates an email address."

Paste the response into a new file `validateEmail.test.js`.

☐ 3. Run the tests

In your terminal, execute:

`npm test`

Review the results. Did all tests pass?

☐ 4. Evaluate edge case coverage

Manually review or ask ChatGPT: "What edge cases might be missing in these tests?"

Consider:

- Empty string
- Emails with subdomains
- Invalid domains (e.g., `example.c`)
- Use of special characters
- Missing @ symbol

☐ 5. Prompt for additional edge case tests

Example prompt: "Write Jest tests for validateEmail covering invalid formats and edge cases like no domain, no @ symbol, special characters, etc."

☐ 6. Add new test cases

- o Use `test.each()` if appropriate:

```
test.each([
  ["plainaddress", false],
  ["@missingusername.com", false],
  ["email@.com", false],
  ["email@domain.co.uk", true],
  ["user+tag@example.com", true]
])('validateEmail("%s") should return %s', (input, expected) => {
  expect(validateEmail(input)).toBe(expected);
});
```

☐ 7. Rerun your test suite

- o Confirm new tests behave as expected and identify any failing cases.

Discussion and Sharing:

- Which edge cases were missed by the initial AI prompt?
- How did you refine your prompt to improve test generation?
- What would be the risks of relying only on the AI's first output?

Lab 3: AI-Assisted Code Coverage and Refactoring

Objectives:

- Analyze test coverage of an existing codebase.
- Use AI to generate missing tests for uncovered lines.
- Refactor redundant or verbose tests using AI suggestions.

Instructions:

1. **Set up a sample project**
   - Create a new folder (e.g., `ai-lab-03`) and initialize:

     ```
     npm init -y
     npm install --save-dev jest
     ```

   - Add a script in `package.json`:

     ```
     "scripts": {
       "test": "jest --coverage"
     }
     ```

2. **Add a JS module with multiple branches**
   - Create a file `calculator.js`:

     ```
     function calculate(operation, a, b) {
       switch (operation) {
         case 'add':
           return a + b;
         case 'subtract':
           return a - b;
         case 'multiply':
           return a * b;
         case 'divide':
           if (b === 0) throw new Error('Cannot divide by zero');
           return a / b;
         default:
           throw new Error('Unknown operation');
       }
     }

     module.exports = calculate;
     ```

3. **Add a partial test suite**
   - Create `calculator.test.js`:

     ```
     const calculate = require('./calculator');

     test('adds numbers', () => {
       expect(calculate('add', 3, 2)).toBe(5);
     });
     ```

```
test('subtracts numbers', () => {
  expect(calculate('subtract', 5, 3)).toBe(2);
});
```

4. **Check code coverage**
   o Run:

   ```
   npm test
   ```

   o Look at the coverage report in your terminal and note which `case` branches or lines are not covered.
5. **Use ChatGPT to help generate missing tests**
   o Prompt example:
   "This is my function. What Jest tests should I add to improve coverage?" (Paste `calculate()` function)
   o Paste back the uncovered `case`s and error throws, then ask:
   "Write Jest tests to cover divide by zero and unknown operation."
6. **Add the suggested tests**
   o Add AI-generated tests like:

   ```
   test('divides numbers', () => {
     expect(calculate('divide', 6, 3)).toBe(2);
   });

   test('throws error on divide by zero', () => {
     expect(() => calculate('divide', 6, 0)).toThrow('Cannot divide
   by zero');
   });

   test('throws error on unknown operation', () => {
     expect(() => calculate('modulo', 6, 3)).toThrow('Unknown
   operation');
   });
   ```

7. **Rerun the tests and confirm coverage**
   o Your coverage report should now show 100% for `calculator.js`.
8. **Refactor verbose or repetitive tests with AI**
   o Prompt ChatGPT: "Can you refactor these four Jest tests into a `test.each` block?"
   o Example refactor:

   ```
   test.each([
     ['add', 1, 2, 3],
     ['subtract', 5, 3, 2],
     ['multiply', 2, 3, 6],
     ['divide', 6, 2, 3]
   ])('%s %i and %i returns %i', (op, a, b, expected) => {
     expect(calculate(op, a, b)).toBe(expected);
   });
   ```

Discussion and Sharing:

- What did the coverage report show initially?
- How did the AI help generate tests to fill gaps?
- Which tests were most repetitive and how did you refactor them?

Lab 4: Testing Legacy Code

Objectives:

- Practice understanding unfamiliar or undocumented code using AI assistance.
- Use AI to reverse-engineer function behavior.
- Create regression tests that "lock in" the current behavior for safety during future changes.

Instructions:

1. **Create a legacy function file**
   - In your project directory, create `legacy.js`:

     ```javascript
     function mysteryFunc(str) {
       let result = '';
       for (let i = 0; i < str.length; i++) {
         if (i % 2 === 0) {
           result += str[i].toUpperCase();
         } else {
           result += str[i].toLowerCase();
         }
       }
       return result;
     }

     module.exports = mysteryFunc;
     ```

   - Pretend this function is part of a legacy codebase with no documentation or comments.
2. **Use ChatGPT to interpret the function**
   - Prompt:
     "What does this JavaScript function do?" (Paste the `mysteryFunc` code)
   - Review the AI explanation. You should expect a response like:

     > "It alternates uppercase and lowercase characters, starting with uppercase."

3. **Ask ChatGPT to generate regression tests**
   - Prompt:
     "Write Jest regression tests for this function, assuming the current behavior is correct."
   - You should get something like:

     ```javascript
     const mysteryFunc = require('./legacy');

     test('alternates casing for simple word', () => {
       expect(mysteryFunc('hello')).toBe('HeLlO');
     });
     ```

```
test('works with empty string', () => {
  expect(mysteryFunc('')).toBe('');
});

test('handles numbers and symbols', () => {
  expect(mysteryFunc('a1b2')).toBe('A1B2');
});
```

4. **Create the test file and paste the tests**
   o Save them in `legacy.test.js`.
5. **Run your test suite**

   ```
   npm test
   ```

6. **Discuss test intent**
   o These are **regression tests**, meaning they capture what the function currently does, regardless of whether that behavior is "correct" or not.
   o Any future change that alters the behavior will cause these tests to fail — which is what you want if you're working with legacy code.

Discussion and Sharing:

- How confident are you in your understanding of the legacy function after using AI?
- Would you change the behavior of this function? If so, how would you refactor safely?
- Why are regression tests valuable even when you don't fully trust the original code?

Lab 5: Exploratory and Edge Case Testing

Objectives:

- Learn how to perform exploratory testing using AI to suggest unexpected or tricky inputs.
- Use boundary, edge case, and fuzz testing techniques.
- Practice using `test.each()` in Jest for efficient test structuring.

Instructions:

1. **Create a parser function**
   - Create a new file `parseDate.js`:

     ```
     function parseDate(str) {
       const date = new Date(str);
       if (isNaN(date.getTime())) {
         throw new Error('Invalid date');
       }
       return date;
     }

     module.exports = parseDate;
     ```

2. **Start with a basic test file**
   - Create `parseDate.test.js`:

     ```
     const parseDate = require('./parseDate');

     test('parses a valid ISO string', () => {
       expect(parseDate('2024-01-01').toISOString()).toBe('2024-01-01T00:00:00.000Z');
     });
     ```

3. **Ask ChatGPT for exploratory test inputs**
   - Prompt:
     "Give me a list of unusual and edge case date strings to test a function that parses dates."
   - Expect suggestions like:
     - `"02/30/2022"` (invalid date)
     - `""` (empty string)
     - `"9999-12-31"` (extreme future)
     - `"13/01/2022"` (non-ISO format)
     - `"Hello, world!"` (nonsense input)
     - `"2023-02-29"` (non-leap year date)

4. **Convert AI suggestions into a `test.each()` block**
   - Add this to `parseDate.test.js`:

     ```
     test.each([
       ['valid ISO date', '2023-01-01', true],
     ```

```
    ['empty string', '', false],
    ['non-date string', 'Hello, world!', false],
    ['invalid day', '2022-02-30', false],
    ['non-leap year Feb 29', '2023-02-29', false],
    ['extreme future date', '9999-12-31', true]
])('%s: %s', (_, input, shouldPass) => {
  if (shouldPass) {
    expect(() => parseDate(input)).not.toThrow();
  } else {
    expect(() => parseDate(input)).toThrow('Invalid date');
  }
});
```

5. **Run the tests**
   o In your terminal:

   ```
   npm test
   ```

6. **Refine based on failures**
   o If any valid-looking cases fail, investigate why. Are you using locale-sensitive formats? Could time zones be an issue?

Discussion and Sharing:

- Which inputs surprised you in how they were handled?
- Did AI suggest any tests you wouldn't have thought of?
- How could exploratory testing help when validating user-facing features?

Lab 6: Generating Synthetic Test Data with AI

Objectives:

- Use AI to generate structured test data based on a schema.
- Practice testing with both valid and invalid inputs.
- Use `test.each()` to run multiple data-driven test cases efficiently.

Instructions:

1. **Create a sample validation function**
   - Create a file `validateUserProfile.js`:

   ```
   function validateUserProfile(user) {
     if (!user.name || typeof user.name !== 'string') return false;
     if (!user.email || !user.email.includes('@')) return false;
     if (typeof user.age !== 'number' || user.age < 0 || user.age >
   120) return false;
     return true;
   }

   module.exports = validateUserProfile;
   ```

2. **Define your schema (informally)**
   - Your user profile structure expects:
     - `name` → non-empty string
     - `email` → contains "@"
     - `age` → number between 0 and 120

3. **Prompt ChatGPT for test data**
   - Prompt:
     "Generate 10 valid and 5 invalid JavaScript user objects based on this schema: {
     name: string, email: string, age: number between 0 and 120 }."
   - Example valid outputs:

   ```
   { name: "Alice", email: "alice@example.com", age: 30 }
   { name: "Bob", email: "bob@mail.com", age: 45 }
   ```

   - Example invalid outputs:

   ```
   { name: "", email: "invalid.com", age: 25 } // no @ in email
   { name: "John", email: "john@site", age: -5 } // negative age
   { name: "Lucy", email: "lucy@domain.com", age: 999 } // age too
   high
   ```

4. **Build a `test.each()` block**
   - Create `validateUserProfile.test.js`:

   ```
   const validateUserProfile = require('./validateUserProfile');
   ```

```
const validUsers = [
  { name: "Alice", email: "alice@example.com", age: 30 },
  { name: "Bob", email: "bob@mail.com", age: 45 },
  { name: "Chloe", email: "chloe@domain.com", age: 25 }
  // Add more from ChatGPT output
];

const invalidUsers = [
  { name: "", email: "noatsign.com", age: 20 },
  { name: "Dan", email: "dan@example.com", age: -1 },
  { name: "Eve", email: "eve@site.com", age: 130 }
  // Add more from ChatGPT output
];

test.each(validUsers)('valid user: %#', (user) => {
  expect(validateUserProfile(user)).toBe(true);
});

test.each(invalidUsers)('invalid user: %#', (user) => {
  expect(validateUserProfile(user)).toBe(false);
});
```

5. **Run the tests**

```
npm test
```

6. **Explore variations**
    o   Ask ChatGPT to generate "edge case" users:
        ▪   Very long strings
        ▪   Unicode characters
        ▪   Missing fields
        ▪   Fields as null or undefined
7. **(Optional) Refactor test data into a separate file**
    o   Helps keep test logic clean and promotes reusability.

Discussion and Sharing:

*   How did AI help you broaden your input cases?
*   Which invalid cases were tricky to detect in code?
*   How would you adapt the test if your schema changed?

Lab 7: Test Smells and Anti-Patterns

Objectives:

- Learn to recognize common test code problems ("test smells").
- Use AI to identify and fix poor testing practices.
- Improve readability, maintainability, and effectiveness of test code.

Instructions:

1. **Set up a test suite with bad practices**
   - Create a file `badCalculator.test.js`:

   ```javascript
   const calc = require('./calculator');

   test('test addition', () => {
     let result = calculate('add', 1, 2);
     expect(result).toBe(3);
   });

   test('test addition again', () => {
     let result = calculate('add', 1, 2);
     expect(result).toBe(3);
   });

   test('test sub', () => {
     let result = calculate('subtract', 5, 2);
     expect(result).toBe(3);
   });

   test('testing division by zero works', () => {
     try {
       calculate('divide', 4, 0);
     } catch (e) {
       expect(e.message).toBe('Cannot divide by zero');
     }
   });

   test('weird test', () => {
     expect(calculate('multiply', 3, 3)).toBe(9);
   });

   const spy = jest.fn();
   spy('hello');
   spy('world');
   expect(spy).toHaveBeenCalled();
   ```

   - This file contains:
     - Redundant tests

- Poorly named tests
- Unnecessary mocks
- Repeated logic

2. **Ask ChatGPT to identify test smells**
   - Prompt:
   "Here's a Jest test file. Can you point out the test smells or anti-patterns?" (Paste the file)
   - Expected feedback:
     - Duplicate tests
     - Vague test names
     - Unused/irrelevant mocks
     - Inconsistent structure

3. **Ask ChatGPT to suggest improvements**
   - Prompt:
   "Refactor this test file to remove duplication and improve readability."
   - You may get something like:

```
const calc = require('./calculator');

describe('Calculator', () => {
  test.each([
    ['add', 1, 2, 3],
    ['subtract', 5, 2, 3],
    ['multiply', 3, 3, 9]
  ])('%s %i and %i equals %i', (op, a, b, expected) => {
    expect(calc(op, a, b)).toBe(expected);
  });

  test('throws error when dividing by zero', () => {
    expect(() => calc('divide', 4, 0)).toThrow('Cannot divide by
zero');
  });
});
```

4. **Refactor your test file based on AI suggestions**
   - Save the refactored code in `refactoredCalculator.test.js`.
5. **Run and compare**
   - Run both test files:

```
npm test
```

   - Confirm both pass, but the refactored version is easier to read and maintain.

Discussion and Sharing:

- Which issues did the AI point out that you hadn't noticed?
- How does refactoring help when onboarding new developers?

- When might using `test.each` or `describe()` blocks make tests more modular?

Lab 8: Testing AI Systems

Objectives:

- Understand the challenges of testing AI-generated output.
- Write tests for non-deterministic outputs using structure, type, and tone checks.
- Explore how AI can help simulate and validate AI behavior.

Instructions:

1. **Create a mock AI-powered function**
   - Create a file `generateSummary.js`:

     ```
     function generateSummary(text) {
       // Simulate AI behavior
       if (!text || text.length < 10) return "Input too short.";
       return `Summary: ${text.slice(0, 50)}...`;
     }

     module.exports = generateSummary;
     ```

   - This function mocks how an LLM might summarize text — simple, but non-deterministic if expanded later.

2. **Start with a basic test file**
   - Create `generateSummary.test.js`:

     ```
     const generateSummary = require('./generateSummary');

     test('returns message for short input', () => {
       expect(generateSummary('Hi')).toBe('Input too short.');
     });

     test('returns summary string for long input', () => {
       const input = 'This is a long paragraph about testing AI
     output.';
       const output = generateSummary(input);
       expect(typeof output).toBe('string');
       expect(output.startsWith('Summary:')).toBe(true);
     });
     ```

3. **Ask ChatGPT to generate realistic test cases**
   - Prompt:
     "Generate 3–5 input/output examples for a text summarizer function. Output should be concise, in natural language, and begin with 'Summary: '."
   - Example input/output pairs:
     - Input: `"Testing is important to ensure software quality."`
       Expected: Starts with `"Summary:"`, includes part of original text.
     - Input: `""`
       Expected: `"Input too short."`

4. **Use AI to guide validation criteria**
   - Prompt:
     "How can I test a text summarization function if the exact output may vary?"
   - ChatGPT might suggest:
     - Check output length (e.g., less than 100 chars)
     - Ensure it contains keywords from input
     - Use regular expressions or sentence structure patterns
     - Validate tone or prefix (e.g., starts with `"Summary:"`)
5. **Add flexible tests based on AI advice**
   - Expand `generateSummary.test.js`:

   ```
   test('output is concise and prefixed', () => {
     const input = "Artificial intelligence in software testing
   improves coverage.";
     const summary = generateSummary(input);
     expect(summary).toMatch(/^Summary: .+/);
     expect(summary.length).toBeLessThan(100);
   });
   ```

6. **(Optional) Discuss non-determinism**
   - What if the real API generates a different summary each time?
   - How would you test:
     - Format?
     - Presence of input keywords?
     - Tone or style?
7. **Explore snapshot testing (advanced)**
   - Jest offers `expect().toMatchSnapshot()` — but beware, snapshots for LLMs can become flaky if the output shifts frequently.

Discussion and Sharing:

- How do we define "correct" for AI-generated output?
- When is output verification subjective?
- What metrics or constraints make AI output testable?

Lab 9: Test Maintenance and Flaky Tests

Objectives:

- Learn to identify and debug flaky tests — tests that fail intermittently.
- Use AI to help diagnose common causes (timing, async, external dependencies).
- Apply techniques to stabilize unreliable tests.

Instructions:

1. **Create a flaky test scenario**
   - Create `networkRequest.js`:

```
function networkRequest() {
  return new Promise((resolve, reject) => {
    const delay = Math.random() * 5000; // Random delay 0-5
seconds
    const shouldFail = Math.random() < 0.1; // 10% chance
of failure

    setTimeout(() => {
      if (shouldFail) {
        reject(new Error('Network error'));
      } else {
        resolve('response');
      }
    }, delay);
  });
}

module.exports = networkRequest;
```

2. **Write an unstable test**
   - Create `networkRequest.test.js`:

```
const networkRequest = require('./networkRequest');

test('resolves with response', async () => {
  const result = await networkRequest();
  expect(result).toBe('response');
});
```

   - Add flaky tests:

```
test('flaky test - sometimes fails due to timeout', async () => {
    jest.setTimeout(2500);
    const result = await networkRequest();
    expect(result).toBe('response');
  });
```

```
test('unstable test - random failure', async () => {
  const shouldFail = Math.random() < 0.3;
  if (shouldFail) {
    throw new Error('Random test failure to simulate instability');
  }
  const result = await networkRequest();
  expect(result).toBe('response');
});

test('another flaky approach - race condition', async () => {
  const timeoutPromise = new Promise((_, reject) => {
    setTimeout(() => reject(new Error('Request timeout')), 2000);
  });

  const result = await Promise.race([networkRequest(),
    timeoutPromise]);
  expect(result).toBe('response');
});
```

3. **Run tests multiple times**
   o Run:

```
npm test
```

   o Run again to see intermittent failures.
4. **Paste the test and error into ChatGPT**
   o Prompt:
   "This async Jest test fails sometimes. Why and how can I fix it?"
   (Include test code and error message if available.)
5. **Fix the flaky tests with mocks**
   o For example: add a stable version:

```
test('resolves after delay (mocked)', async () => {
    // Use fake timers only for this test
    jest.useFakeTimers();

    // Mock Math.random
    const originalRandom = Math.random;
    Math.random = jest.fn(() => 0.5); // Always return 0.5

    const promise = networkRequest();

    // Fast-forward all timers
    jest.runAllTimers();

    const result = await promise;
    expect(result).toBe('response');

    // Clean up
    Math.random = originalRandom;
    jest.useRealTimers();
});
```

6. **Verify test reliability**
   - Run tests multiple times:

   ```
   npm test
   npm test
   ```

   - Ensure stable, consistent results.
7. **Try ChatGPT's other suggestions**
   - Ask:
     "How do I stabilize flaky tests caused by random data, timeouts, or network dependencies?"
   - Suggested practices:
     - Use mocks for APIs and timers
     - Avoid `Math.random()` in production tests
     - Set up retries or assertions with `waitFor` for UI/async tests

---

Discussion and Sharing:

- What caused your test to be flaky?
- What patterns can you now recognize in other flaky tests?
- How would you educate a team about preventing flaky test debt?

Lab 10: CI/CD Integration of AI-Generated Tests

Objectives:

- Learn how to set up continuous integration (CI) to run tests automatically.
- Use GitHub Actions to trigger test runs on every push or pull request.
- Integrate AI-generated test files into a working test pipeline.

Instructions:

1. **Set up your GitHub repository**
   - Create GitHub repo for this class if you haven't already.
   - Push your local project to GitHub if it's not already there:

   ```
   git init
   git add .
   git commit -m "Initial commit"
   git branch -M main
   git remote add origin https://github.com/your-username/your-repo-name.git
   git push -u origin main
   ```

2. **Add a GitHub Actions workflow**
   - In your project folder, create the following file:

   ```
   .github/workflows/tests.yml
   ```

   - Add this YAML configuration:

   ```yaml
   name: Run Tests

   on:
     push:
       branches: [ main ]
     pull_request:
       branches: [ main ]

   jobs:
     test:
       runs-on: ubuntu-latest

       steps:
         - name: Checkout code
           uses: actions/checkout@v3

         - name: Set up Node.js
           uses: actions/setup-node@v3
           with:
             node-version: '18'

         - name: Install dependencies
   ```

```
          run: npm install

        - name: Run tests
          run: npm test
```

3. **Push the workflow to GitHub**

```
git add .github
git commit -m "Add CI workflow for running tests"
git push
```

4. **Trigger a test run**
   o Make a small change (e.g., update a comment) and push it:

```
echo "// trigger build" >> index.js
git add index.js
git commit -m "Trigger build"
git push
```

5. **Check the GitHub Actions dashboard**
   o Go to your repo on GitHub
   o Click on **"Actions"**
   o Confirm that your workflow ran and passed
6. **(Optional) Use AI to write or explain your CI config**
   o Prompt:
      "Write a GitHub Actions workflow to install dependencies and run Jest tests on push and pull request."
   o You can also paste your config and ask:
      "Is there anything I can improve in this CI pipeline?"
7. **(Optional) Add test report formatting**
   o Use a Jest reporter (e.g., `jest-junit`) for CI-friendly test output
   o Add to your YAML if needed

Discussion and Sharing:

- What was the benefit of running tests automatically in CI?
- How might AI-generated tests introduce risk in CI pipelines?
- How would you ensure AI-generated code doesn't break your main branch?

Lab 11: Documentation and Reporting

Objectives:

- Use AI to generate readable documentation from test cases.
- Create Markdown reports and JSDoc-style comments from your tests.
- Practice summarizing a test suite for team communication or QA sign-off.

Instructions:

1. **Choose a representative test file**
   - For example, use `calculator.test.js` or `validateUserProfile.test.js`.
2. **Paste the test file into ChatGPT**
   - Prompt:
     "Summarize what this Jest test suite does in plain English."
     (Paste the test file contents.)
   - ChatGPT may respond:

     > This test suite checks basic arithmetic operations like addition, subtraction, multiplication, and division. It also verifies that division by zero throws an error.

3. **Ask ChatGPT to generate test descriptions**
   - Prompt:
     "Write one-sentence descriptions for each test case."
   - Example result:

     ```
     - Test 1: Confirms that adding 1 and 2 returns 3.
     - Test 2: Confirms that subtracting 5 and 2 returns 3.
     - Test 3: Confirms that dividing by zero throws an error.
     ```

4. **Convert descriptions into a Markdown report**
   - Create a file `TEST_REPORT.md`:

     ```
     # Test Report: Calculator Functions

     This document summarizes the tests written for `calculator.js`.

     ## Covered Functions

     - `add(a, b)`
     - `subtract(a, b)`
     - `multiply(a, b)`
     - `divide(a, b)`

     ## Test Cases

     - ✅ Adds 1 and 2 to return 3
     - ✅ Subtracts 5 and 2 to return 3
     ```

```
- ✅ Multiplies 3 and 3 to return 9
- ✅ Divides 6 by 2 to return 3
- ⚠️ Throws an error when dividing by 0

## Notes

- Division is tested for error handling.
- More edge cases could be added for input validation.
```

5. **Generate inline documentation (JSDoc-style)**
   o Ask ChatGPT:
     "Generate JSDoc comments for the `calculate()` function."
   o Example output:

```
/**
 * Performs a basic arithmetic operation on two numbers.
 * @param {string} operation - One of: 'add', 'subtract',
'multiply', 'divide'.
 * @param {number} a - The first operand.
 * @param {number} b - The second operand.
 * @returns {number} The result of the operation.
 * @throws Will throw an error for division by zero or unknown
operations.
 */
```

   o Add this above the function in `calculator.js`.
6. **(Optional) Use AI to generate a test plan summary**
   o Prompt:
     "Write a high-level test plan based on the following tests."
     (Include test file.)
   o The result could serve as documentation for QA teams or stakeholders.

Discussion and Sharing:

- How did AI help make your tests more understandable?
- What types of audiences benefit from these summaries (e.g., QA, PMs, developers)?
- When should documentation be automated vs. written manually?

Lab 12: Limitations and Ethics

Objectives:

- Critically analyze the accuracy and trustworthiness of AI-generated tests.
- Identify hallucinations, overconfidence, and logical flaws in AI-generated code.
- Discuss ethical concerns around test authorship, liability, and misuse.

Instructions:

1. **Review a flawed AI-generated test file**
   - Create a file `aiGenerated.test.js`:

     ```
     const isEven = require('./isEven');

     // AI-generated test cases (with intentional issues)
     test('should return true for even number', () => {
       expect(isEven(4)).toBe(true);
     });

     test('should return false for negative number', () => {
       expect(isEven(-4)).toBe(false); // ✖ incorrect — -4 is even
     });

     test('should return false for decimal number', () => {
       expect(isEven(4.2)).toBe(false);
     });

     test('should throw for non-numeric input', () => {
       expect(isEven('four')).toBe(false); // ✖ should throw or handle
     gracefully
     });
     ```

   - Create the `isEven.js` file:

     ```
     function isEven(n) {
       if (typeof n !== 'number' || !Number.isInteger(n)) throw new
     Error('Invalid input');
       return n % 2 === 0;
     }

     module.exports = isEven;
     ```

2. **Run the test suite**
   - In your terminal:

     ```
     npm test
     ```

   - Note which tests fail and why.
3. **Prompt ChatGPT to analyze test issues**

- o Paste the test file and ask:
  "Can you find logical flaws or incorrect assumptions in this test suite?"
- o AI should identify:
  - ▪ Misinterpretation of negative even numbers
  - ▪ Lack of exception handling tests
  - ▪ Incorrect assumption that `'four'` returns false instead of throwing

4. **Fix the broken or misleading tests**
   - o Update to:

```
test('should return true for negative even number', () => {
  expect(isEven(-4)).toBe(true);
});

test('should throw for non-numeric input', () => {
  expect(() => isEven('four')).toThrow('Invalid input');
});
```

5. **Reflect on AI's overconfidence**
   - o Ask ChatGPT:
     "Why do LLMs make confident but wrong code suggestions?"
   - o It may explain:
     - ▪ Lack of true understanding
     - ▪ Pattern-matching over reasoning
     - ▪ No access to live execution feedback

6. **Discuss ethical implications**
   - o Prompts for team or classroom discussion:
     - ▪ Who is responsible if an AI-generated test causes a false sense of security?
     - ▪ Should you disclose the use of AI in safety-critical codebases?
     - ▪ How should copyright be handled for AI-generated test logic?

Discussion and Sharing:

- What mistakes did you catch that the AI missed?
- How do we balance AI convenience with the need for human oversight?
- When should we treat AI-generated code as a "draft," and when can it be trusted?

Lab 13: Capstone Lab

Objectives:

- Apply AI tools and techniques to test a complete JavaScript application.
- Generate tests, refactor them, and integrate with CI/CD.
- Demonstrate ability to use AI responsibly and effectively in the software testing workflow.

Instructions:

1. **Download or clone the sample app**
   o Use the provided repo from the course:

   ```
   https://github.com/chrisminnick/ai-in-software-testing
   ```

   o Navigate to the `capstone` or `final-project` folder, or select a provided app such as a basic `todo` or `shopping-cart`.
2. **Install dependencies**

   ```
   npm install
   ```

3. **Generate an initial Jest test suite using AI**
   o Choose one or two core functions (e.g., `addItem()`, `removeItem()`).
   o Prompt ChatGPT:
   "Generate Jest tests for a todo app function that adds items to a list."
   o Review, validate, and paste the AI-generated tests into `__tests__` or a `*.test.js` file.
4. **Identify coverage gaps**
   o Run:

   ```
   npm test -- --coverage
   ```

   o Review the coverage report.
   o Ask ChatGPT:
   "Suggest test cases for the uncovered lines/functions in this file."
   (Paste uncovered code snippet.)
5. **Refactor AI-generated tests for readability**
   o Look for:
     ▪ Redundant tests
     ▪ Inconsistent naming
     ▪ Long or unclear logic
   o Ask ChatGPT to refactor:
   "Refactor these Jest tests using `test.each` or `describe` blocks."
6. **Document tests using AI**

- Ask ChatGPT:
  "Write descriptions for these tests in Markdown format for a QA report."
- Create `TEST_PLAN.md` with sections:
  - Covered features
  - Test cases and purposes
  - Known limitations or areas needing manual testing

7. **Add CI/CD integration**
   - Create a GitHub Actions workflow file at:

     ```
     .github/workflows/tests.yml
     ```

   - Use the workflow template from Lab 10 to install dependencies and run tests.

8. **Push to GitHub and verify**
   - Commit your test files, documentation, and workflow:

     ```
     git add .
     git commit -m "Add capstone tests and CI"
     git push
     ```

   - Visit GitHub → Actions tab → Confirm successful workflow run.

9. **Review your work**
   - Checklist:
     - ✅ Tests created with AI
     - ✅ Test coverage expanded and verified
     - ✅ Tests refactored for clarity
     - ✅ Documentation written
     - ✅ CI/CD workflow integrated and passing

Discussion and Sharing:

- What part of the workflow did AI help with the most?
- Where did AI fall short or require heavy human correction?
- How ready do you feel to apply this process to a real-world project?