Data-Driven API Test Case Generation Using AI and Model Context Protocol

Harold Wilson and Joseph Petsche

hwilson@CovertCoders.DEV and Joe@CovertCoders.DEV

Covert Coders LLC

Abstract

In the rapidly evolving landscape of software development, Application Programming Interfaces (APIs) are critical for enabling seamless integration and functionality across diverse systems. Ensuring their reliability and robustness is essential, yet traditional testing methods often struggle with efficiency and comprehensive test coverage, particularly in addressing complex interactions and edge cases. This paper presents an innovative approach that leverages Artificial Intelligence (AI) and Model Context Protocol (MCP) to automate the generation of test cases for APIs. By employing advanced Machine Learning (ML) algorithms and MCP interfaces, our system analyzes API specifications, historical usage patterns, and test data to intelligently generate a diverse and thorough set of test cases. This AI-driven methodology accelerates the testing process and enhances coverage by identifying and addressing edge cases that traditional testing might overlook.

Biography

Harold Wilson is a distinguished software quality assurance professional with extensive experience leading QA teams. His career highlights include serving as a Test and Reliability Consultant for the United States Space Force and as Director of Quality Assurance at Entercom Digital (radio.com). Harold has expertise in software testing, reliability, and security including compliance with PCI and HIPAA standards, Harold has consistently developed robust QA processes from the ground up. He began his career as an Electronics Technician in the United States Navy and is a decorated Gulf War veteran. Harold holds a Bachelor of Science in Computer Science from the College of Santa Fe.

Joseph Petsche is a seasoned Software Architect with over two decades of experience, specializing in automation, integration, and quality assurance. Currently serving as an Automation Architect at EverDriven, he streamlines DevOps pipelines and fosters cross-team collaboration to deliver high-quality software. His career includes automating data flows at WebMD and building resilient software through automated testing at Red Rock Tech Solutions. Joseph is passionate about solving complex problems and ensuring client satisfaction. An avid mountain climber, he approaches challenges with strategic determination. Joseph holds a degree in Computer Science & Engineering from the University of Toledo.

1 Introduction

1.1 Problem Statement and Motivation

The growing complexity of modern API ecosystems presents significant challenges for software quality assurance teams. As Application Programming Interfaces (APIs) have become the backbone of modern software architecture, enabling seamless integration and functionality across diverse systems, the traditional approaches to API testing are increasingly proving inadequate. Manual test case creation faces inherent limitations in scalability and consistency, while coverage gaps in edge-case testing leave critical vulnerabilities undetected. Furthermore, the maintenance overhead associated with traditional test suites becomes prohibitive as API specifications evolve and expand, creating a bottleneck that impedes development velocity and compromises software reliability.

1.2 Research Question and Current State

Existing testing methodologies struggle to keep pace with the rapid evolution of API-driven applications, particularly in addressing complex interactions and identifying subtle edge cases that manual testers might overlook. This gap between testing capabilities and system complexity raises a fundamental research question: How can Artificial Intelligence (AI) and Model Context Protocol (MCP) be leveraged by testers to improve API test case generation efficiency and enhance test coverage? The current state of API testing relies heavily on static, predefined test scenarios that fail to adapt to changing specifications and usage patterns, highlighting the urgent need for intelligent, data-driven approaches that can automatically generate comprehensive and relevant test cases while reducing the burden on testing teams.

2 Background and Related Work

2.1 API Testing Fundamentals

API testing has emerged as a critical component of modern software quality assurance, driven by the exponential growth of web and mobile applications and their significant role in software architecture. Effective API testing methodologies follow a multi-layered approach aligned with the test pyramid concept (Cohn 2009), including unit tests, integration test, and system-level testing coverage. Best practices emphasize the importance of implementing both automated and manual testing strategies (Briggs, et al. 2019), where automated testing provides rapid feedback and regression detection and manual verification offers flexibility and the ability to identify subtle security and business logic issues that automated tools might miss. Organizations are increasingly adopting comprehensive testing techniques including golden or happy path testing (TechTarget 2024) for expected scenarios, edge case testing to validate boundary conditions, smoke testing for basic functionality verification, and combinatorial testing to achieve maximum coverage of parameter combinations.

2.2 API Testing Challenges

API testing faces significant challenges that span technical, organizational, and process-related domains. One of the most prominent is the complexity of maintaining test suites as API ecosystems grow and evolve rapidly, especially in environments where developers can deploy code changes to production in Continuous Integration / Continuous Deployment (CI/CD) environments. This acceleration creates pressure for test frameworks to keep pace with development velocity while maintaining comprehensive coverage. This challenge is compounded by the difficulty of achieving effective test automation repeatability.

Environmental consistency presents another major pain point for testers. Testing teams frequently encounter issues where breaking changes introduced in development environments go undetected until late in the development cycle. This is because tests run in different environments can generate indeterministic behavior. Organizations often face the dilemma of balancing automated testing coverage with the resource-intensive nature of comprehensive feature testing. Teams also struggle with the selection and evaluation of testing tools from an overwhelming marketplace of solutions. This leads to decision paralysis and tool choices that often fail to meet organizational needs for features, price, ease of use, community support, and automation capabilities.

2.3 Artificial Intelligence in Software Testing

AI and ML represent a revolutionary approach to automating software testing that mimics how humans learn and make decisions. Think of AI as teaching a computer that behaves intelligently. ML, which is a subset of AI, is the science of getting computers to learn and improve their performance without being explicitly programmed for every possible scenario (MIT Sloan. 2021). AI addresses a fundamental limitation of traditional automated testing: current test scripts are rigid, break easily when applications change, and cannot adapt to new situations the way human testers can. AI-driven testing solutions can perceive application states, act intelligently based on what they observe, and uncover defects through learned behavior rather than pre-written instructions.

AI, a Large Language Model (LLM), serves as the intelligent core of the test generation system, using Claude AI advanced natural language processing and reasoning capabilities to analyze API specifications and generate comprehensive test cases. The engine employs a multi-layered neural network architecture that processes API documentation through both supervised and unsupervised learning approaches (Hou, Xinyi, et al. 2025). The supervised learning component is trained using historical test data and API specification patterns, enabling the system to recognize common endpoint structures to generate testing scenarios. Meanwhile, the unsupervised learning mechanisms excel at discovering edge cases and unusual parameter combinations that traditional testing approaches may miss.

2.4 Model Context Protocol

The MCP Server integration layer, shown in Figure 1, bridges the gap between various API specification formats and the AI engine's processing components. The MCP layer supports OpenAPI specifications directly (APImatic. 2023). The MCP Servers allow the AI to implement intelligent context extraction mechanisms that capture not just the structural information from API specifications but also the relationships between endpoints, data dependencies, and business logic patterns.

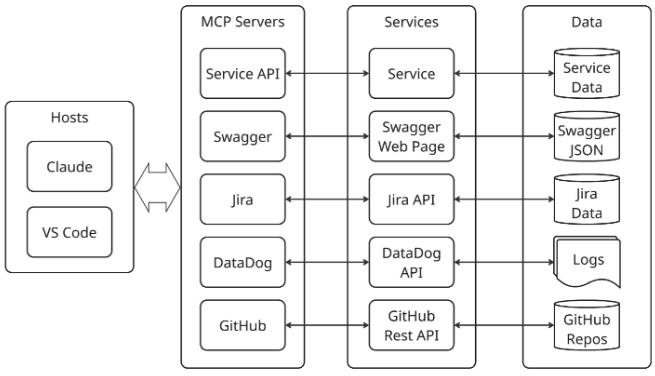

Figure 1: MCP Architecture Layers

Advanced parsing capabilities if the AI include support for complex authentication schemes, nested parameter structures, and conditional request/response patterns that are often found in enterprise API environments. The integration layer also features dynamic content adaptation, where it can adjust its parsing strategies based on the quality and completeness of the source documentation, simultaneous testing of both the delivered API and its associated documentation.

2.5 API Testing Approach

For this research we used the JEST Test Execution framework. This tool is effectively native to our development environment, VS Code. The JEST test execution framework provides a robust and scalable foundation for running AI-generated test cases. JEST offers a JavaScript-based testing environment that implements an emphasis on modularity, maintainability, and extensibility while supporting both synchronous and asynchronous API testing patterns. The framework features intelligent test parallelization capabilities that can dynamically adjust execution based on API endpoint characteristics and system load, significantly reducing overall test execution time while maintaining test isolation and preventing interference between concurrent test runs (Jest Team. 2025). The JEST integration includes custom packages specifically designed for API testing scenarios. This enables more expressive and readable test case assertions and provides detailed failure diagnostics when tests do not pass.

The results analysis and reporting system implements a flexible, multi-tiered approach that accommodates both generic JSON-based output for broad compatibility and specialized integration with Jira Xray for enterprise test management workflows. The specialized Jira Xray integration provides connectivity to enterprise test management processes, automatically creating test execution records, updating test case statuses, and generating requirements traceability matrices that link API testing results back to business requirements in Jira.

3 Methodology

3.1 System Architecture

The test bench and technical stack consist of the following components: Microsoft Azure web hosted environment, Hosted servers to execute the JavaScript automation code, Claude with Sonnet 4 Desktop Application, and Jira test case management for reporting. The test bench is integrated with the following components via direct custom connection to MCP agent/servers: Swagger API Documentation, Actual API endpoints to be tested, log servers, Jira, and the Jira Xray testing plugin.

The data flow and processing pipeline follows two paths. In the first path, See Figure 3, the prompted AI generates test cases based on the current state of the system to be tested, which is based on inputs from Swagger, log servers, load balancers, existing test cases and user stories in Jira, and other documentation from the release i.e. readme files and Claude markdown files, …, etc. The second path, see Figure 2, is the actual test case execution driven by the JEST test case execution framework. In this path, generated test cases and existing test cases are executed against the API's to be tested, and the results of testing are sent to Jira Xray.

3.2 MCP Integration

The MCP interface implements a standardized protocol that abstracts the complexities of direct API specification parsing while providing Claude Sonnet with structured, contextual information about the target APIs. The implementation featured a bidirectional communication system where the MCP server can dynamically load and parse Swagger/OpenAPI documents, transforming them into a normalized context format that Claude Sonnet can efficiently process. Authentication and security considerations for the API are built into the MCP layer.

The context extraction process represents a sophisticated analysis engine that transforms static Swagger documentation into understandable tokens suitable for AI-driven test case generation. The system performs multi-layered parsing that goes beyond simple structural interpretation to understand the relationships between API endpoints, parameters, and business logic patterns. The extraction engine analyzes endpoint hierarchies to identify logical groupings and dependencies, recognizing patterns such as Create, Read Update, and Delete (CRUD) operations, pagination schemes, and nested resource relationships that inform comprehensive test scenario development. Parameter analysis includes deep inspection of data types, validation constraints, enumeration values, and format specifications, while also identifying implicit relationships between parameters across different endpoints that might indicate workflow dependencies or data consistency requirements.

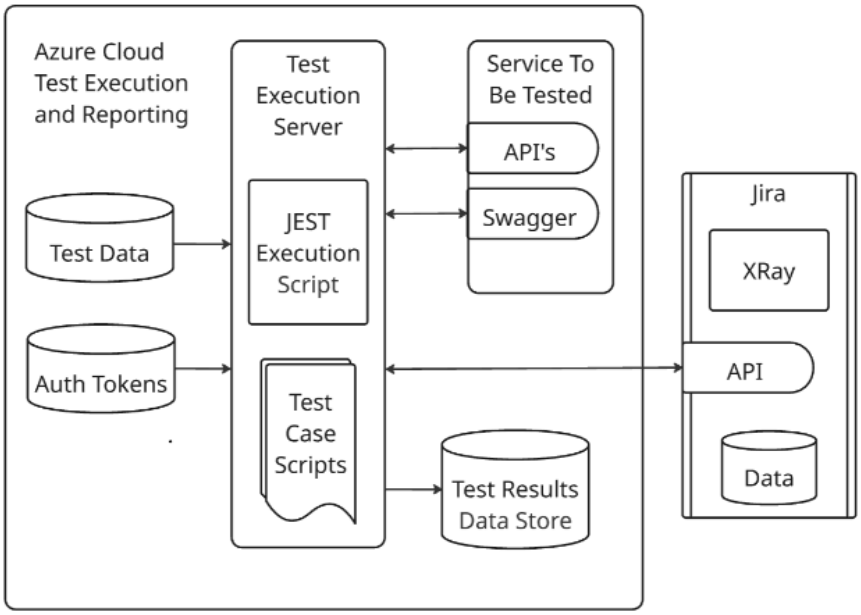

Figure 2: Test Execution Environment

Claude Sonnet's reasoning capabilities combined with extracted API context create intelligent, realistic test data. The system implements an approach that combines constraint-based test case generation, pattern recognition, and business logic understanding to produce comprehensive test datasets. The generator analyzes parameter schemas to understand not just basic data types but, also parameter meaning, generating values that respect format constraints, enumeration limits, and cross-parameter dependencies while creating both valid and invalid inputs for comprehensive positive and negative test case coverage. MCP agents and servers obtain context specific data for testing i.e., car makes and models and state, county and school district names and specific addresses.

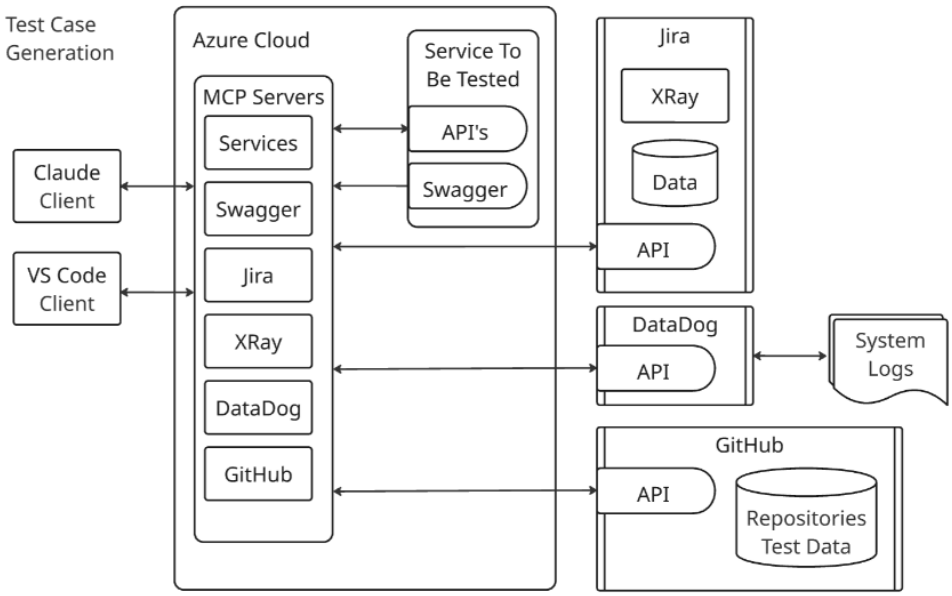

Figure 3: Test Case Generation

3.3 Test Case Generation and Data Flows

Both test case generation data flows are orchestrated as GitHub Actions using YAML files as implementation and execution scripts. Additional guidance on test case development was provided to Claude AI via Claude's markdown files and a Claude's command folder embedded in the test automation source code. The code block, see Listing 1, shows the Claude configuration JSON file for implementing the servers in Figure 2

```
{
"mcpServers":{
    "Jira":{"command":"npx",
            "args":["-y","@modelcontextprotocol/server-filesystem",
"~/MCP_SERVERS"],
            "env":{"NODE_ENV": "staging"}},
    "GitHub":{"command": "npx",
              "args": ["-y","@modelcontextprotocol/server-
filesystem", "~/MCP_SERVERS"],
              "env":{"NODE_ENV": "staging"}},
    "DataDog":{"command": "npx",
               "args": ["-y", "@modelcontextprotocol/server-
filesystem", "~/MCP_SERVERS"],
               "env":{"NODE_ENV": "staging"}},
    "Swagger":{"command": "npx",
               "args": ["-y", "@modelcontextprotocol/server-
filesystem", "~/MCP_SERVERS"],
               "env":{"NODE_ENV": "staging"}},
    "XRay":{"command": "npx",
            "args": ["-y", "@modelcontextprotocol/server-
filesystem", "~/MCP_SERVERS"],
            "env":{"NODE_ENV": "staging"}}}
}
```

Listing 1: Configuration file used to integrate MCP servers with the Claude Client Desktop Application

Keyword descriptions:

- **"filesystem"**: Name for the server that will appear in the Claude Desktop user interface.
- **"command"**: "npx": Interpreter or another tool used to run the server.
- **"-y"**: Automatically confirms the installation integration of the server package.
- **"@modelcontextprotocol/server-filesystem"**: Package name of the Filesystem Server.
- **Additional optional arguments**: Directories the server is allowed to access.

Note: *If your configured server fails to load, and logs indicate an error referring to ${APPDATA} within a path, you may need to add the expanded value of ${APPDATA}.*

4 Experimental Design and Evaluation

4.1 Evaluation Methodology

The hypothesis is that using AI tools like Claude would improve the performance of the QA Team leading to better testing with fewer resources. To prove this hypothesis, we needed to baseline the team's existing performance using traditional manual testing and test automation scripts written by Software Development Engineer in Test (SDET's). Then we compare the performance using AI tooling to the traditional testing approach. The delta between the two become the AI performance enhancement metric that could be measured over time to estimate future performance of the team. The API features have an easily quantifiable test space and coverage metrics.

The following attributes measure the performance of the team's ability to test APIs without AI tooling this includes overall endpoint coverage as a percentage; test cases developed per week. test cases executed per week. creation of new defects, and validation of fixed defects. We are specifically not measuring the quality of the delivered code i.e. defect density. Our initial performance benchmarks a listed in Table 1.

Table 1: Team Performance Metrics

	Manual Tests Without AI	Manual Tests Using AI	Automated Tests Without AI	Automated Tests Using AI
Test Cases Developed Per Week	15	42	5	12
Test Cases Executed Per Week	77	77	356	356
Defects Created Per Week	2	3	0	2
Defects Validated {er Week	1	1	0	0
Coverage	45%	55%	75%	78%

4.2 Test Environments

The testing environment consists of the target of evaluation (the ting to be test) in this case a set of API end points around a particular service and the test automation stack which is largely Postman tests driven by Newman driven by JEST. Manual testing is accomplished my manually executing Postman collections or Curl commands, see Figure 4. There are additional security components related to authorization that are not addressed in this paper.

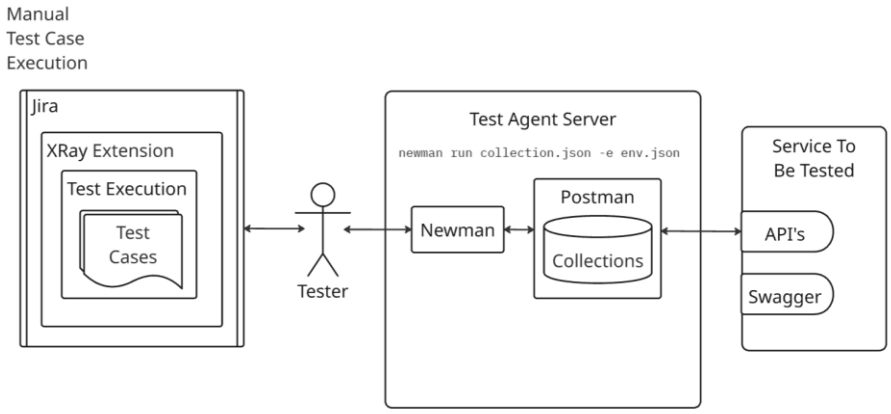

Figure 4: Manual Test Case Execution

4.3 Experiment Metrics and Data Collection

The following metrics were captured for both the initial baseline (without A.I. tooling) and approximately a month later with A.I. tooling. See Table 2 for details and comparisons of test case generation speed metrics, Coverage improvement measures, edge case discovery rates, and false positive/negative analysis. For these metrics, the term coverage is defined as providing a test case for each scenario in a given feature.

Table 2: Test Case Comparisons Traditional vs. AI Enhanced

	Manual Tests Without AI	Manual Tests Using AI	Automated Tests Without AI	Automated Tests Using AI
Happy Path Test Cases	111	220	432	450
Negative Test Cases	217	250	545	562
Edge Cases	0	3	35	72
Performance	0	0	5	7
Security	0	0	14	21
False Positives	2	5	0	10
Coverage	45%	55%	75%	78%

5 Results and Analysis

5.1 Summary

We observed an overall increase in the team's ability to generate and execute test cases when leveraging AI. However, manual test executions were not impacted by the use AI. There was a measurable amount of false positive test results when relying on AI requiring manual validation.

5.2 Qualitative Analysis

While there was no appreciable increase in the overall quality of the product being tested, this is based on a definition of quality related to the defects detected in the final production environment and reported by the end users. The QA Team was able to demonstrate an increase in automated and manual test cases developed. Specifically in the areas of edge cases and test cases addressing security vulnerabilities i.e. HTTP parameter injection and insecure deserialization. See Table 2 for details.

5.3 Quantitative Analysis

Test case generation and test coverage metrics comparing traditional automated testing solutions and manual testing against testing solutions leveraging AI are listed in Table 1. Overall, the team was able to produce thirty-four more test cases per week. However, the number of false positives increased dramatically. In our exercises, AI had no impact on the team's ability to execute test cases either manual or automated.

6 Conclusions

6.1 Key Finding

The current state of AI, as of August 2025, shows that API testing can be a valuable addition to existing testing methods. However, most of the efficiencies gained by leveraging AI were offset by correcting AI's fabrication or Synthetic misinformation. Given Moore's Law of AI (METR 2025) that shows AI's capabilities doubling every seven months, it is expected that AI will be providing more value for the API tester and software testers in general in the very near future.

While AI can increase the number of test cases being developed by a test team, there was no impact on the overall quality of the product delivered and there was only a minimal impact on the team's efficiency. Any efficiency gains realized by leveraging AI in generating test cases were offset by tracking down and troubleshooting false positives.

When the user stories provided to the AI were written using the Gherkin Syntax (Matt Wynne, et al. 2017), AI generated better test cases. The structured description of features and scenarios is user stories appeared to generate higher quality test cases and yielded fewer false positives in later testing.

It is important to note that the AI did provide valuable security tests that the QA Team did not have coverage for in their existing suite of tests. Specifically, the AI suggested and created test cases for Indirect Deserialization vulnerabilities (OWASP Foundation 2025) with multiple API endpoints.

6.2 Lessons Learned

- For now, the best use of AI tooling in API testing is in manual test case generation. The AI was able to discover edge cases that the team had not thought of and provided valuable security tests.
- Connecting the AI to the API's Swagger documentation enabled the AI to generate a large volume of new test cases providing substantial labor efficiencies in test case generation.
- Connecting the IA to the log server did not appear to impact the AI's ability to generate new test cases.

6.3 Limitations, Constraints and Challenges

The support community for Model Context Protocol is chaotic and largely disorganized. MCP is an emerging industry standard that only a few AI vendors support as of August 2025. This necessitated the sole use of Claude AI for our AI tooling and MCP server setup and configuration. It is also almost impossible to keep pace with the pace of AI products, features, and innovations in the AI ecosystem. The implication is that newer and better products may exist in the marketplace that we were unaware of.

6.4 Organizational Impact

The API tested in the six-week period (spanning three agile sprints) showed no measurable increase in overall quality. Neither the defects found in our production environment, nor the defect density changed as a result of leveraging AI. However, we have positioned the organization to take full advantage of AI as it develops over time linking AI's improvement to our own.

References

Cohn, Mike. 2009. *Succeeding with Agile: Software Development Using Scrum*. Addison-Wesley Professional.

Briggs, K., Santiago, D., Adamo Jr., D., Daye, P., & King, T. M. (2019). Semi-Autonomous, Site-Wide A11Y Testing Using an Intelligent Agent. *PNSQC Conference Proceedings*, 356.

TechTarget. 2024. "What is Happy Path Testing? | Definition from TechTarget." https://www.techtarget.com/searchsoftwarequality/definition/happy-path-testing (accessed August 2, 2025).

MIT Sloan. 2021. "Machine learning, explained." *MIT Sloan Ideas Made to Matter*, April 21. https://mitsloan.mit.edu/ideas-made-to-matter/machine-learning-explained (accessed August 2, 2025)

Hou, Xinyi, et al. 2025. "Model Context Protocol (MCP): Landscape, Security Threats, and Future Research Directions." arXiv:2503.23278 [cs.CR], Cornell University. https://arxiv.org/abs/2503.23278

APImatic. 2023. "Top API Specification Trends: 2019-2022." *APImatic Blog*, December 18. https://www.apimatic.io/blog/2022/03/top-api-specification-trends-2019-2022 (accessed August 4, 2025)

Jest Team. 2025. "Jest: Delightful JavaScript Testing." *Jest Documentation*. https://jestjs.io/ (accessed August 2, 2025)

OWASP Foundation. "Insecure Deserialization." *OWASP Community Vulnerabilities.* Available at: https://owasp.org/www-community/vulnerabilities/Insecure_Deserialization (accessed August 13, 2025)

METR. (2025, March 19). Measuring AI Ability to Complete Long Tasks. *METR Blog*. https://metr.org/blog/2025-03-19-measuring-ai-ability-to-complete-long-tasks/ (Accessed August 20, 2025)

Matt Wynne, Aslak Hellesoy, and Steve Tooke. *The Cucumber Book*. 2nd ed., Pragmatic Bookshelf, 2017.