Journal of Computer Science and Technology Studies
ISSN: 2709-104X
DOI: 10.32996/jcsts
Journal Homepage: www.al-kindipublisher.com/index.php/jcsts

JCSTS
AL-KINDI CENTER FOR RESEARCH AND DEVELOPMENT

| RESEARCH ARTICLE

Enabling Testability: Key Step in Automating Automation

Vivek Krishna Choppa
National Institute of Technology, Rourkela, India
Corresponding Author: Vivek Krishna Choppa, **E-mail**: vivekkrishnachoppa@gmail.com

| ABSTRACT
In the modern software development lifecycle, automation has become essential for achieving speed, accuracy, and scalability in testing processes. However, a critical step often overlooked is ensuring the *system's testability* early in its development. This paper introduces the concept of the *Testability Analyser*, a tool that evaluates software systems for their testability. By leveraging AI technologies and integrating with design tools such as AWS documentation, Draw.io, PlantUML, and Creately, the Testability Analyser facilitates early testability evaluations, optimizing systems for automated testing. The paper discusses the importance of testability, the role of AI systems in understanding complex software architectures, and key items to verify testability in software architecture before and after the advent of Large Language Models (LLMs) and Model Context Protocol (MCP).

| KEYWORDS
Testability Analysis, AI-Driven Testing, Model Context Protocol, Software Architecture, Test Automation

| ARTICLE INFORMATION
ACCEPTED: 20 May 2025 **PUBLISHED:** 11 June 2025 **DOI:** 10.32996/jcsts.2025.7.6.10

1. Introduction: Importance of Ensuring Testability Early in the Development Lifecycle

Testability, the ease with which a system can be tested, is a critical aspect of modern software engineering, yet it is often neglected until later stages of the development process [1]. The absence of a testability strategy can lead to significant delays, as systems with poorly designed testability require additional work to refactor or make them suitable for automation. For example, ensuring testability early on is crucial in the payments domain. Consider a payment gateway designed to handle transactions from various platforms (e.g., e-commerce, mobile apps, and banking apps). If the architecture does not account for testability from the outset, integration and end-to-end testing can become exceedingly complex, leading to longer validation cycles and delayed releases [2]. The cost of identifying design flaws after deployment in such high-stakes environments can be substantial, with issues ranging from payment failures to security vulnerabilities. A Testability Analyser would assess the system's architecture, flagging areas where modularization is required or where the introduction of mockable interfaces would streamline testing. By proactively ensuring that the system is designed for testability, teams can avoid costly reworks and ensure that automation is integrated smoothly, accelerating time to market while reducing risk [1]. The Model Context Protocol (MCP) enhances the Testability Analyser by providing context-aware test data provisioning, inter-service coordination, and asynchronous test environment management. This integration ensures more accurate, dynamic, and scalable testing in complex, distributed systems [3].

2. AI Systems that Understand Software Architectures

Modern AI systems have made significant strides in understanding software architectures, including design diagrams, sequence diagrams, flowcharts, and component icons. AI systems, especially those based on Large Language Models (LLMs) and Model Context Protocols (MCPs), can now interpret these artifacts in ways that were previously impossible [5]. While LLMs are primarily text-based models, they can interpret structured text representations of architectural diagrams [4]. For instance, an architecture diagram created using tools like Draw.io or Creately often includes annotations, descriptions, and metadata about components and interactions (e.g., "Service A communicates with Service B via API X"). LLMs can parse these text elements, identify

relationships between components, and analyze the modularity and dependencies in the architecture. Although the diagram itself is visual, LLMs can interact with its textual counterpart (e.g., diagrams exported to formats such as JSON, YAML, or UML text files) to assess testability aspects, including whether the system's components are loosely coupled or if certain services have hidden dependencies [6]. In the context of the Testability Analyser, AI can be trained to interpret complex design elements, such as:

- **Design Diagrams**: AI can parse architectural diagrams (e.g., AWS architecture, microservices diagrams) created with tools like Draw.io or Creately to assess the modularity of services, component interactions, and potential areas for automation.
- **Sequence Diagrams**: AI can analyze sequence diagrams to evaluate whether components communicate in ways that facilitate testability. For example, tight coupling between components or the absence of clear interfaces might make automated testing challenging, and these issues can be flagged early [5].
- **Component Icons and Symbols**: Using specific design patterns and icons can help AI systems identify standard practices influencing testability [4]. For instance, if a service is designed with poor fault tolerance or high dependencies on other services, the AI system can detect these risks and recommend architectural changes.

3. Key Items to Verify Testability in Software Architecture

Ensuring testability in software architecture involves evaluating several key factors influencing how easy it will be to automate testing. Before the advent of LLMs and MCP, verifying testability was manual, requiring deep expertise to examine the architecture for potential testing challenges [7]. However, these technologies have transformed the verification process, enabling more intelligent, automated, and scalable testability assessments. Below are key items to verify, comparing how each item has transformed before and after the introduction of LLMs and MCP.

3.1. Modularization and Separation of Concerns

- **Before LLMs and MCP**: The design process for ensuring modularization required extensive code reviews and manual inspection of architectural diagrams. Developers would assess whether the system adhered to principles of separation of concerns, which was often subjective and time-consuming. In many cases, issues like tight coupling or the lack of clearly defined service boundaries were only identified during later stages of testing.
- **After LLMs and MCP**: AI-powered tools like the Testability Analyser, trained with LLMs and Model Context Protocols, can automatically evaluate whether components are properly modularized. These AI models can parse architecture diagrams (e.g., AWS architecture or UML) to identify where modularization is weak or components are unnecessarily tightly coupled. They can even suggest where services should be decoupled or interfaces should be defined to improve testability, greatly reducing human effort and enabling early intervention in the design phase [7].

The evolution of MCP ensures that systems are analyzed for modularity and optimal resource allocation for testing, automatically provisioning the necessary infrastructure when needed.

3.2. Dependency Injection and Test Hooks

- **Before LLMs and MCP**: Manual inspections of the code and architecture were required to verify whether dependency injection patterns were correctly implemented. Developers also needed to ensure services could be easily mocked or replaced with test doubles during unit tests. This process was prone to oversight and typically only addressed after integration tests failed.
- **After LLMs and MCP**: AI systems can automatically analyze architecture and codebases to identify dependency injection patterns and the availability of test hooks. They can detect whether services are appropriately decoupled and mockable interfaces are in place, making it much easier to ensure that components are testable from the outset. AI tools can even suggest where test hooks are missing, making the process more proactive and automated.

3.3. Exposure of Internal Logic and States

- **Before LLMs and MCP**: Testing internal logic often required deep inspection of the codebase and sometimes even reverse engineering of compiled code. It was difficult to ensure that key internal states were exposed for testing without directly modifying the system.
- **After LLMs and MCP**: With LLMs and Model Context Protocols, AI systems can analyze the architecture and the generated code to ensure that essential internal states and business logic are exposed through well-documented APIs or event streams. These systems can recommend areas of the architecture that need improved logging, state exposure, or access to allow for more efficient and comprehensive testing.

3.4. Test Automation Compatibility

- **Before LLMs and MCP**: Verifying a system's compatibility with automation tools required manually integrating various testing frameworks and confirming that APIs were testable. Additionally, creating test environments often required additional infrastructure setup, which could be cumbersome and error-prone.
- **After LLMs and MCP**: AI models can scan through system architectures and identify potential gaps or friction points in test automation compatibility. They can suggest integrating specific frameworks or testing tools for unit, integration, and end-to-end tests. Moreover, these AI models can automatically generate the necessary configurations for CI/CD pipelines, which reduces setup time and errors.

3.5. Error Handling and Fault Tolerance

- **Before LLMs and MCP**: Ensuring effective error handling and fault tolerance in a system's architecture involved detailed manual analysis of the error handling code and exception management strategies, often leading to reactive fixes after testing failures.
- **After LLMs and MCP**: AI systems now use advanced pattern recognition to analyze a system's entire error-handling framework. They can predict potential failure points, flagging parts of the system where error handling is weak or prone to cascading failures. Furthermore, AI can suggest adopting more robust fault-tolerance patterns such as retries, circuit breakers, and failover mechanisms.

3.6. Identifying the Need for Test Infrastructure and Automating Launch with MCP

- **Before LLMs and MCP**: The identification and provisioning of test infrastructure was traditionally done manually, often requiring teams to determine which environments (e.g., cloud instances, databases, services) were needed for different types of testing. This step was time-consuming and prone to errors, especially when scaling systems or when test environments needed to be set up for specific test cases (e.g., load testing, integration testing, etc.).
- **After LLMs and MCP**: With the integration of LLMs and Model Context Protocols, AI systems can now automatically identify which test infrastructure is required based on the testability analysis of the system architecture. By analyzing the components and dependencies in the system, the Testability Analyser can recommend the necessary infrastructure (e.g., EC2 instances, Docker Images, iOS simulators, Android emulators, RDS databases, S3 buckets for test data) and automatically launch the relevant infrastructure using MCP clients and servers [8]. The MCP clients are configured to interact with the infrastructure automatically, ensuring that all required services are correctly set up before tests are executed. This automation ensures that teams can focus on testing without the manual overhead of infrastructure setup and management, reducing time and risk during the testing phase.

3.7. Provisioning Test Data with MCP Clients and Internal Corp MCP Servers

- **Before LLMs and MCP**: Managing test data often required manual intervention to create, manage, and maintain diverse sets of test data for different testing purposes (e.g., unit tests, integration tests, performance tests). This process was time-consuming and error-prone, often leading to inconsistent or outdated data across different testing environments. Additionally, configuring test data for scenarios involving sensitive information requires extra precautions to maintain security and privacy.
- **After LLMs and MCP**: With the integration of LLMs and Model Context Protocols, AI systems can now automatically provision test data by interacting with internal corporate MCP servers. Based on the testability analysis and the requirements for different tests, the Testability Analyser can recommend the appropriate type of test data required. MCP clients can automatically retrieve or generate realistic, sanitized test data from internal servers and provision it directly to the test environments [8]. This automation ensures that test data is always up-to-date, secure, and aligned with the testing needs, reducing the manual effort involved in preparing test datasets and eliminating the risk of human error. Furthermore, using internal corp MCP servers ensures that sensitive data is managed securely, with privacy protocols enforced during test data provisioning.

4. AI-Driven Test Data Generation Architecture with LLMs and MCP

Innovating on microservices testing, we propose a conceptual architecture that combines Large Language Models (LLMs) with the Model Context Protocol (MCP) to automate test data generation in a platform-agnostic manner. This architecture directly tackles the *testability analyzer* problem by improving test data accuracy, coverage, and scalability across distributed services. Fig. 1 below illustrates the high-level design, where an AI-driven *Test Data Orchestrator* (powered by an LLM) interfaces with microservices and corporate systems via standardized MCP connectors. As recent studies note, microservices' asynchronous, distributed nature creates a pressing need for robust testing approaches, yet current automated test generation tools for such systems remain scarce [9]. Traditional methods struggle to maintain adequate coverage in large, fast-evolving microservice deployments [9]. Our architecture addresses these gaps through a context-aware AI that understands system design artifacts and coordinates data provisioning across services. Key components and innovations include:

- **LLM Understanding of Design Artifacts:** The LLM can parse and interpret software design knowledge, from sequence diagrams and API specifications to database schemas. The model builds a semantic map of service interactions, data flows, and constraints by ingesting these artifacts (e.g., via text-based UML descriptions, OpenAPI/Swagger definitions, or SQL schema dumps). Recent advances in applying LLMs to software engineering tasks (architecture modeling and testing) demonstrate their capability to explain system behavior and requirements [10]. For example, the LLM can read a sequence diagram of an order-processing workflow and discern which services are involved, what data is exchanged, and in what order. It can interpret an API spec similarly to learn input parameter types, value ranges, and error conditions. Internally, the architecture might transform graphical models into structured prompts (or use OCR for text within diagrams) so that the LLM "sees" the design in narrative form. This understanding enables the LLM to generate test data that respects each service's intended interactions and data formats.
- **Context Integration via Model Context Protocol (MCP):** The architecture leverages MCP as a universal interface to connect the LLM with live knowledge from distributed services and internal systems. MCP is an open protocol that standardizes how applications provide context to LLMs, analogous to a *USB-C for AI* [11]. Our Test Data Orchestrator can consistently and securely plug into various data sources and tools through MCP. For instance, one MCP adapter may fetch real-time service contract details or sample messages from a service registry, while another can query a corporate database for schema metadata. The MCP framework follows a client-server model in which lightweight *MCP servers* expose specific capabilities (APIs, databases, file systems) to any AI *host* client [11]. This design makes the solution platform-agnostic – the same LLM-driven agent can run against different tech stacks or cloud platforms by simply switching MCP connectors (e.g., swapping an Oracle DB adapter with a MongoDB adapter, or pointing to a different API endpoint) [11]. All context retrieval and tool invocation is handled through this uniform protocol, simplifying integration. The LLM uses the context gathered (service schemas, configuration settings, etc.) to generate test data in the actual system state and rules.
- **Automated Discovery of Test Data Requirements:** A crucial innovation is using AI to *discover* what test data is needed, not just generate data blindly. The LLM-powered orchestrator analyzes service contracts and past testing or runtime behavior to identify test data requirements. It inspects API contracts for each microservice to determine input fields, boundary values, and invariants (for example, an age field might be an integer 0–120, or an accountType parameter may allow "SAVINGS", "CHECKING", etc.). It also reviews database schemas to understand referential integrity (e.g., a customer_id in the Order service must match an ID from the Customer service). By mining historical test results and logs, the system learns which input scenarios have been covered and where gaps exist – for instance, if past tests never provided a null value for an optional field or never triggered certain error responses. The architecture's analyser component (aided by the LLM) automatically flags such gaps. It might produce a *Test Data Requirements Matrix* that maps each service endpoint to needed test data variations (valid cases, boundary cases, error-inducing cases, rare combinations, etc.), derived from both the explicit contract and implicit usage patterns. This AI-driven discovery ensures that test data generation is *targeted*, focusing on high-risk or under-tested scenarios rather than random data. It effectively builds a knowledge base of what needs to be tested for each service, which can drive the generation of those specific inputs.
- **Deterministic & Synthetic Test Data Generation:** With requirements identified, the LLM orchestrator produces the actual test data in a deterministic and synthetic fashion. **Deterministic** means that the generated data is reproducible given the same context and requirements – an essential property for reliable regression testing. The architecture achieves this by controlling the generation process (for example, using fixed random seeds in any data fuzzing, or instructing the LLM to output structured data sets rather than free-form text). Meanwhile, **synthetic** data is artificially generated to meet the testing needs, rather than production data. The LLM's deep knowledge base is a game-changer: it can encode domain-specific rules and constraints that traditional dummy data generators often miss. For instance, an LLM can "know" that a valid French phone number must start with 06 or 07 and follow a certain format [10] – it will generate values accordingly, ensuring realism and validity. Studies have shown that LLMs can produce fake, remarkably realistic test data compliant with domain constraints [10]. In our architecture, the LLM may either output the raw data (e.g., a JSON payload with fields filled out) or even generate small utility programs (scripts) to produce larger volumes of data. Notably, researchers found that an LLM can generate executable test data generators that can be plugged into test frameworks directly [10]. This approach supports functional testing (providing well-crafted inputs that hit different functional paths) and non-functional testing (scaling up data volume or complexity to stress performance, security, etc.). The deterministic nature ensures that if a performance test needs 10,000 records, the same 10,000 can be regenerated consistently for tuning or comparison. The synthetic nature guarantees that no sensitive real customer data is used while mimicking its statistical properties or edge cases.
- **Coordinated, Context-Aware Data Provisioning:** A standout feature is the coordination across distributed services made possible by the LLM's system-wide view. Because the AI has ingested the relationships and interactions from design diagrams and contracts, it can generate consistent data across services. For example, suppose a test scenario involves a user placing an order that propagates through an Order service, Inventory service, and Notification service. In

that case, the orchestrator can generate interrelated data: a fake user profile valid in the User service, an order request that references that user's ID, an inventory entry for the ordered product, etc. The MCP-connected agents may call each service's test API or database to inject or retrieve the necessary pieces, all guided by the LLM's plan. This ensures that asynchronous or loosely coupled components receive data that aligns logically, avoiding the "missing link" problem (e.g., testing one service with data that another service cannot recognize). The architecture can also coordinate the timing of data injection. For instance, first seed a particular record in Service A, then trigger an event in Service B that expects that record to exist. By orchestrating such sequences, the solution tackles the challenges of testing event-driven and async workflows, which are traditionally hard to control [9]. Every test data is generated with full context, so each microservice sees inputs that make sense in the larger business process.

- **Enhanced Testability through Observability and Controllability:** This LLM+MCP-driven approach ultimately boosts the *testability* of the system under test. Software testability is known to hinge on factors like *observability* (how easily one can observe the system's internal states and outputs) and *controllability* (how easily one can control the state of the system during testing) [10]. Our architecture improves controllability by enabling direct control over distributed components via MCP – the tester (here, the AI agent) can set up specific states or inject events at will in any service, something not feasible in a black-box integration test. It also improves observability by encouraging instrumentation and analysis of outcomes. For example, the orchestrator can use MCP to query a service's state after a test input is processed or to retrieve logs from each service to verify that an asynchronous transaction completed. In practice, we can extend each microservice with test hooks or assertions (in line with Binder's classic recommendation of adding specification-based assertions to improve testability) [12] that the LLM agent knows how to trigger and validate. By improving these properties, the solution makes the system more amenable to thorough testing. The result is a higher fault-detection rate and confidence in quality: as the LLM generates diverse and edge-case inputs, previously untested behaviors are exposed, and any anomalies can be observed across the service chain. This aligns with industry findings that LLM-powered test agents can significantly increase test coverage and efficiency [10]. In essence, the architecture generates data and creates a feedback loop. If certain test data causes an unexpected result in one service, the LLM notices it (via logs/outputs). It can suggest additional tests to investigate or pinpoint the issue, a level of intelligent adaptability beyond conventional scripts.

Innovative Impact and Future Trends: By uniting an LLM's reasoning with the integration capabilities of MCP, this architecture represents a forward-looking solution in AI-assisted software testing. It combines *model-based testing* (using design models to derive tests) and *data-driven AI* to achieve a breadth and depth of test coverage previously unattainable in complex microservice ecosystems. The approach is inherently platform-agnostic, since MCP abstracts the specifics of underlying technologies, and the LLM can be swapped or fine-tuned for different domains. It also complements ongoing trends in autonomous testing – for example, using AI to generate test cases, monitor system behavior, and even self-heal tests as services evolve. Researchers like Garousi and Felderer have highlighted that improving testability (through better control and observability) is key to enabling more effective automation [10], and our proposed architecture directly addresses this by design. Moreover, early evidence from industry and academia suggests that LLM-driven testing can enhance the accuracy and scalability of testing processes [10], which our solution amplifies by systematically incorporating system context. Going forward, such an architecture could be extended with learning capabilities – e.g. the LLM could continuously learn from new production incidents or user behavior to suggest new test data scenarios, making the testing process increasingly proactive. Overall, the LLM+MCP test data generation architecture serves as a novel blueprint for improving microservice testability, offering a high degree of automation and intelligence to keep quality assurance in pace with rapid, distributed development.

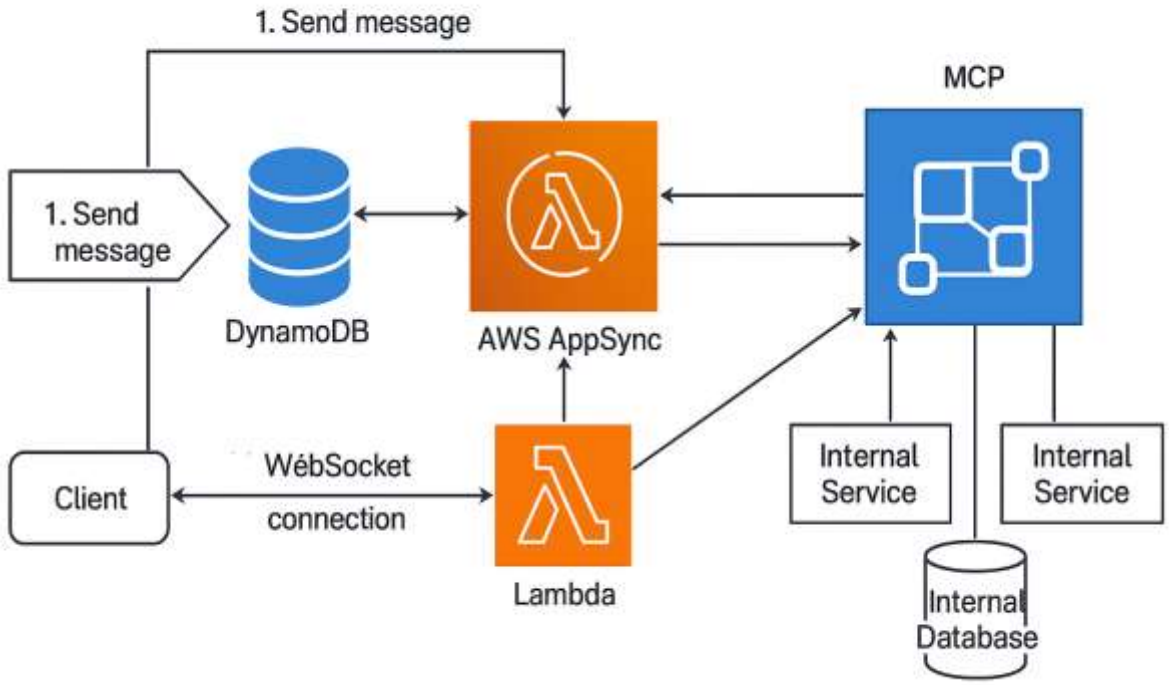

Fig. 1: High-level design of real-time test data generator for enabling testability.

5. Conclusion

Ensuring testability early in the software development lifecycle is a key enabler of successful test automation. The introduction of AI systems, particularly those powered by Large Language Models (LLMs) and Model Context Protocols (MCPs), has significantly transformed how testability is assessed. With these technologies, testability evaluations are no longer manual, error-prone, but are automated, predictive, and proactive. The Testability Analyser leverages these AI advancements to assess software architecture and design diagrams for key testability aspects, enabling teams to implement automation more effectively and efficiently. By verifying modularization, dependency injection, exposure of internal logic, and automation compatibility early in the lifecycle, teams can minimize friction during testing phases, accelerating development cycles and improving software quality. Additionally, the automatic identification and provisioning of test infrastructure, powered by MCP, and the automated provisioning of test data further streamline the testing process and reduce the setup overhead. For further details on how AI and MCP work together to facilitate testing, additional resources are available in technical documentation and research papers that examine the protocol specifications and implementation challenges in applied generative AI systems.

Funding: This research received no external funding.

Conflicts of Interest: The authors declare no conflict of interest.

Publisher's Note: All claims expressed in this article are solely those of the authors and do not necessarily represent those of their affiliated organizations, or those of the publisher, the editors and the reviewers.

References

[1] Benoit B et al., (n.d) Generative AI to Generate Test Data Generators, arXiv, KTH Royal Institute of Technology. Available: https://arxiv.org/pdf/2401.17626

[2] Designing and Implementation Phase, *International Journal of Science and Research*, Volume 4 Issue 4, 2015. https://www.researchgate.net/profile/Harsha-Ratnani/publication/277713636_Object_Oriented_Software_Testability_Survey_at_Designing_and_Implementation_Phase/links/5571484908aee701d61cc024/Object-Oriented-Software-Testability-Survey-at-Designing-and-Implementation-Phase.pdf

[3] Dr. Pushpa R. S and Harsha S, (2015) Object Oriented Software Testability Survey at

[4] Fatih E and Galip A, (2017) Data Classification with Deep Learning using Tensorflow, in 2017 International Conference on Computer Science and Engineering (UBMK), 2017, pp. 755-758. https://www.kresttechnology.com/krest-academic-projects/krest-mtech-projects/CSE/M.%20Tech%20Java%20%202019-20/Mtech%20-Java-%20Basepaper%202019/Basepapers-ML/14.Data%20classification.pdf

[5] Florian H et al., (2014) Software paradigms, assessment types and non-functional requirements in model-based integration testing: A systematic literature review, Researchgate 2014. https://www.researchgate.net/publication/266658757_Software_paradigms_assessment_types_and_non-functional_requirements_in_model-based_integration_testing_A_systematic_literature_review

[6] Michael F et al., (2015) Model-Based Security Testing: A Taxonomy and Systematic Classification, Software Testing, Verification and Reliability, 00(2):1-29, 2015. https://www.researchgate.net/publication/280510415_Model-Based_Security_Testing_A_Taxonomy_and_Systematic_Classification

[7] Model Context Protocol, (n.d) "Introduction." [Online]. Available: https://modelcontextprotocol.io/introduction

[8] Muhammad R S and Lydie D B, (2010) Survey of source code metrics for evaluating testability of object oriented systems, [Research Report] RR-LIG-005, 2010. https://inria.hal.science/hal-00953403/PDF/RR-LIG-005.pdf

[9] Murugesan L. and Balasubramanian P., (2014) Cloud-based mobile application testing, in 2014 IEEE/ACIS 13th International Conference on Computer and Information Science (ICIS), 2014. https://ieeexplore.ieee.org/document/6912148

[10] Pourya N et al, (2010) An Evaluation for Model Testability Approaches, *International Journal of Computers & Technology,* 9(1):938-947, 2010. https://www.researchgate.net/publication/324987553_An_Evaluation_for_Model_Testability_approaches

[11] Robert V. B, (1994) Design for Testability in Object-Oriented Systems, Communications of the ACM 37(9):87-101, 1994. https://www.researchgate.net/publication/220424587_Design_for_Testability_in_Object-Oriented_Systems

[12] Robert V. B, (1999) Testing Object-Oriented Systems: Models, Patterns, and Tools, Addison-Wesley Professional, 1999. Available: https://www.informit.com/store/testing-object-oriented-systems-models-patterns-and-9780201809381

[13] Tingshuo M et al., (2025) Systematic Mapping Study of Test Generation for Microservices: Approaches, Challenges, and Impact on System Quality, Electronics 2025, 14(7), 1397; 2025. Available: https://www.mdpi.com/2079-9292/14/7/1397