SOFTWARE TESTING

Complete Learning Guide

Techniques • Methods • Types • Levels

1. What is Software Testing?

Software testing is the process of dynamically verifying that a program provides expected behaviors on a finite set of test cases, suitably selected from the usually infinite execution domain. In simpler terms, it is the process of finding errors in a developed product and checking whether real outcomes match expected results.

1.1 Key Terms in the Definition

Dynamic	The program is actually executed on selected inputs during testing.
Finite	Testing is conducted on a subset of all possible tests, determined by risk and prioritization criteria.
Selected	Risk analysis techniques and software engineering expertise are applied to choose test cases.
Expected	It must be possible to decide whether the observed outcomes of program testing are acceptable or not.

1.2 Verification vs. Validation

These two concepts are fundamental and often confused:

Verification	"Are we building the product right?" - The software should conform to its specification.
Validation	"Are we building the right product?" - The software should do what the user really requires.

✨ **Trust, but Verify. Verify, but also Validate.** ✨

1.3 Other Important Considerations

- Testing is intended to show that a program does what it is intended to do and to discover program defects before it is put into use.
- When you test software, you execute a program using artificial data.
- You check the results of the test run for errors, anomalies, or information about the program's non-functional attributes.
- Testing is part of a more general verification and validation process, which also includes static validation techniques.

2. Testing Paths

There are two fundamental directions testing can take, each with a different goal:

Happy Path Testing

Goal: To demonstrate that the software meets its requirements.
Success means: The system operates as intended under normal, expected conditions.

Sad / Bad Path Testing

Goal: To discover faults or defects where behavior is incorrect or not in conformance with its specification.
Success means: A test that makes the system perform incorrectly, exposing a defect, or validates that exceptions are handled gracefully.

3. Seven Principles of Software Testing

These principles guide how testing should be approached and understood:

1. Testing shows presence of defects	Testing can show that defects exist, but cannot prove that there are no defects. Even thorough testing does not guarantee absence of bugs.
2. Exhaustive testing is impossible	Testing everything is not feasible. Risk analysis and prioritization are used to focus effort where it matters most.
3. Test as early as possible	Starting testing activities as early as possible in the development lifecycle reduces cost of fixing defects later.
4. Defect clustering	A small number of modules typically contain most of the defects. Testing effort should be concentrated in those areas.
5. Pesticide paradox	If the same tests are repeated, they will no longer find new bugs. Tests must be regularly reviewed and updated.
6. Testing is context dependent	Testing is done differently in different contexts. Safety-critical software needs different testing than an e-commerce site.
7. Absence-of-errors fallacy	Finding and fixing defects is useless if the system built does not fulfill the user's needs and expectations.

4. Error, Defect, and Failure

These three terms are frequently confused but have precise, distinct meanings:

Error	A human action that produces an incorrect result. This is the human mistake that starts the chain.
Defect (Bug / Fault)	A manifestation of an error in software. If executed, a defect may cause a failure.
Failure	A deviation of the software from its expected delivery or service. This is what the user experiences.

The Chain

Human Error → Defect in Code → Failure at Runtime

Failure is an event. Defect is a state. Error is the cause.

5. Sources of Defects

Defects can originate from multiple phases of the software development lifecycle:

Requirements Definition	Erroneous, incomplete, or inconsistent requirements lead to building the wrong thing from the start.
Design	Fundamental design flaws in the software architecture or module design.
Implementation	Mistakes in programming, including logic errors, typos, off-by-one errors, or malicious code.
Support Systems	Poor programming languages, faulty compilers and debuggers, or misleading development tools.
Inadequate Testing	Incomplete testing, poor verification, or mistakes made during the debugging process.
Evolution (Maintenance)	Sloppy redevelopment or maintenance; introduction of new flaws when trying to fix old ones; incremental escalation to inordinate complexity.

6. Testing Techniques

Testing techniques describe how tests are designed and approached. The three main categories are:

6.1 Static Testing

Static testing involves examining software work products without executing them. It can be applied to source code, design documents, models, functional specifications, and requirement specifications.

Methods of Static Testing

- Informal Reviews - Casual peer review of code or documents
- Walkthroughs - Author leads the team through a document step by step
- Technical Reviews - Documented, structured process to detect defects
- Inspections - Most formal type: checklists, entry/exit criteria, metrics
- Static Analysis - Automated tool analysis of code structure
  - Data Flow analysis
  - Control Flow analysis

Value of Static Testing

- Early detection of defects prior to test execution
- Early warning about suspicious aspects of code, design, or requirements
- Identification of defects not easily found in dynamic testing
- Improved maintainability since engineers follow documented standards and rules
- Prevention of defects when engineers learn from their errors and practice continuous improvement

6.2 Dynamic Testing

Dynamic testing validates the functionality of an application when the code is actually executed. It involves running the program and observing its behavior.

Structure-Based (White-Box) Testing

Focuses on what is happening inside the system or component. Also called white-box, glass-box, or clear-box testing.

- Statement Coverage - Has every statement in the code been executed?
- Decision Coverage - Has every branch (true/false) been executed?
- Condition Coverage - Has every boolean sub-expression been evaluated both true and false?

- Multiple Condition Coverage - All combinations of conditions tested

Structure-based techniques are a good way of generating additional test cases that are different from existing tests.

Specification-Based (Black-Box) Testing

Focuses on what the software does, not how it does it. Includes both functional and non-functional testing.

- Equivalence Partitioning - Divide input domain into classes where all values in a class are expected to behave the same way
- Boundary Value Analysis - Test at the edges of equivalence partitions
- Decision Tables - Systematic representation of combinations of inputs and resulting outputs
- State Transition Testing - Tests based on the states a system can be in and transitions between them
- Use Case Testing - Test cases derived from use case scenarios

Experience-Based Testing

Relies on the knowledge, skills, and background of the tester. Both technical and business expertise are valuable.

- Exploratory Testing - Simultaneous learning, test design, and test execution
- Error Guessing - Use experience and intuition to guess probable error locations
- Ad Hoc Testing - Informal, unplanned testing without documentation

This may be the only technique used for low-risk systems, but is particularly useful under extreme time pressure.

7. Test Types

Test types define the objective of a specific round of testing for a program or project:

Functional Testing	Tests that the system performs the functions it is supposed to perform. Based on requirements and specifications.
Non-Functional Testing	Tests attributes such as performance, usability, reliability, security, and scalability.
Structural Testing	Tests the internal structure of the system. Based on the code and its design (white-box approach).
Regression Testing	Ensures existing functionality still works after new changes or enhancements. New changes must NOT introduce new bugs.
Retesting	Verifies that a specific defect has been fixed and that the related functionality now works as expected.

Regression vs Retesting

Regression Testing: Runs the full suite of tests to ensure no side effects from changes. Broader scope.

Retesting: Focuses on confirming a specific previously-reported defect has been fixed. Narrower scope.

8. Testing Levels (The V-Model)

Testing levels correspond to phases of software development. Each level maps to a development artifact.

Unit Testing	Tests individual, separately testable software elements (functions, classes, methods) in isolation. Maps to program specification.
Integration Testing	Tests interactions among software components. Architecture-driven; incrementally integrates components based on functional threads. Maps to technical specification.
System Testing	Tests the behavior of the entire system as a whole. Maps to functional specification.
Acceptance Testing	Customers test the system to decide whether to accept it from the developers and deploy it in the customer environment. Primarily for custom products. Maps to requirement specification.

8.1 Extended Testing Levels

Beyond the V-Model, two additional levels are often considered in practice:

Installation Testing

After completion of system and acceptance testing, the software is verified upon installation in the actual target environment. This ensures the product works correctly when deployed in its real setting.

Alpha and Beta Testing

- Before release, software is given to a small, selected group of potential users for trial use.
- Alpha Testing: Takes place at the developer's site. Typically performed by a group independent of the design team.
- Beta Testing: Takes place at the customer's site. Users install and use the software under real-world working conditions.

8.2 The Agile Testing Quadrants

The Testing Quadrants model (Brian Marick) organizes testing activities in an Agile context along two axes:

- Business-Facing vs. Technology-Facing - Does the test support the business or critique the product from a technical angle?

- Supporting the Team vs. Critiquing the Product - Is the test helping development or evaluating the finished product?

Q1 (Automated, Technology-Facing)	Unit Tests and Component Tests. Support the team. Automated.
Q2 (Automated & Manual, Business-Facing)	Functional Tests, Examples, Story Tests, Prototypes, Simulations. Support the team.
Q3 (Manual, Business-Facing)	Exploratory Testing, Scenarios, Usability Testing, UAT, Alpha/Beta Testing. Critique the product.
Q4 (Tools-Based, Technology-Facing)	Performance & Load Testing, Security Testing, 'ility' Testing. Critique the product.

9. Test Automation

Test automation is the use of tools to run tests automatically and compare results against expected outcomes.

- Automated tests are fast, but require time to prepare and maintain.
- Manual testing complements automated tests - it cannot be fully replaced.
- Automation is expected across different test levels.
- Start with the most repeatable tests first for maximum ROI.

9.1 The Test Automation Pyramid

The pyramid shows the ideal distribution and automation extent for different testing methods. More tests at the base, fewer at the top.

Manual & Exploratory Testing (Few, High Cost)
Automated UI Tests
Automated API Tests | Automated Integration Tests | Automated Component Tests
UNIT TESTS (Many, Low Cost, Fast)

10. The Fundamental Testing Process

Every professional testing effort follows a structured set of activities:

1. Plan	Determine scope, risks, and objectives. Choose the test approach. Implement test policy/strategy. Determine required resources. Schedule activities. Define exit criteria.
2. Design	Create test cases based on the test approach. Define test conditions and expected results. Select test data.
3. Implement	Prepare test data and test environment. Write automated test scripts. Organize test cases into test suites.
4. Execute	Run the tests. Log outcomes. Compare actual results against expected results. Report defects.
5. Evaluate	Assess whether exit criteria have been met. Produce a test report summarizing findings, coverage, and quality.

10.1 Test Cases

A test case is a set of conditions and variables used to determine whether a system under test functions correctly. Key fields include:

- ID - Unique identifier
- Title - Short descriptive name
- Priority - High / Medium / Low
- Purpose - What is being verified
- Preconditions - State the system must be in before the test
- Steps - Sequential actions to perform
- Expected Result - What should happen if the system works correctly
- Actual Result - What actually happened (filled in during execution)
- Status - Pass / Fail / Not Run / Blocked

10.2 Gherkin Scenarios (BDD)

Gherkin is a human-readable format for writing test cases that serves as executable documentation. It uses the Given-When-Then structure:

Given	Describes the initial state of the system before an action is taken.
When	Describes the action or event that triggers the scenario.
Then	Describes the expected outcome after the action has been performed.

Example Gherkin scenario:

```
Feature: User Authentication

Scenario: Successful login with valid credentials
  Given the user is already registered to the website
  And the user is on the login page
  When the user inputs the correct email address
  And the user inputs the correct password
  And the user clicks the Login button
  Then the user should be authenticated
  And the user should be redirected to their dashboard
```

11. Testing Tools

A wide range of tools support different aspects of the testing process:

Defect Tracking	Track and manage reported bugs (e.g., Jira, Bugzilla, Azure DevOps).
Test Case Management	Organize, store, and report on test cases and results (e.g., TestRail, TestPad, Xray, PractiTest).
GUI Test Drivers	Automate interactions with graphical user interfaces (e.g., Selenium, Playwright, Cypress).
API Testing Tools	Test REST, SOAP, and GraphQL APIs (e.g., Postman, REST-assured).
Load and Performance	Simulate many users and measure system response under load (e.g., JMeter, Gatling, k6).
Static Analysis	Analyze source code without executing it for potential defects (e.g., SonarQube, ESLint, PMD).
Test Coverage	Measure what percentage of the code is exercised by tests (e.g., JaCoCo, Istanbul/nyc).
Unit Test Frameworks	Write and run unit tests (e.g., JUnit for Java, NUnit for C#, Jest for JavaScript, pytest for Python).
Bundled Tool Suites	All-in-one platforms combining multiple testing functions.

12. Practical Tips

12.1 Tips for Code Review (Static Testing)

1. Do not review your own code. Switch with a peer to get a fresh perspective.
2. Use a checklist for common mistakes to ensure nothing is overlooked systematically.
3. Commit to the task - aim to actually understand the code, not just skim it.
4. Check documentation, tests, and build files as well as the code itself.
5. Keep priorities clear: code functional first, clean and maintainable second, optimized third.
6. Follow up on review comments to ensure they are addressed.

12.2 Tips for Testing Your Project

7. Do not test your own code - switch with a teammate for objectivity.

8. Test from the start. Do not leave testing for the last minute before a deadline.
9. Use a mix of techniques to define test cases and prioritize them by risk.
10. Test both happy path (normal flow) and sad path (error conditions and edge cases).
11. Use realistic, life-like test data. Random strings like 'alksjlkj' are not valid test data.
12. Make no code commits after the last test run. If you commit a change, you must retest.

13. Summary and Key Takeaways

Core Concepts

- Software testing verifies that a program provides expected behaviors on a finite set of test cases.
- **Trust, but Verify. Verify, but also Validate.**
- Defects can originate from any phase of software development: requirements, design, code, tools, testing, or maintenance.
- The 7 testing principles guide the approach to all testing work.
- Happy and sad path testing serve different, complementary goals.
- Static testing examines artifacts without execution; dynamic testing runs the code.
- Testing levels (Unit, Integration, System, Acceptance) map to software development stages.
- The automation pyramid: many unit tests, fewer UI tests, exploratory testing on top.
- The fundamental testing process: Plan -> Design -> Implement -> Execute -> Evaluate.
- Testing is only one part of quality management. Other quality assurance activities must also be implemented.

RESILIENT SYSTEMS

Dependability, Cybersecurity & Design Patterns

KTU Software Engineering • Assoc. Prof. Dr. Tomas Blazauskas

14. Dependability and Resilience

Dependability is a broad property of a system that encompasses several sub-properties. Resilience is one of these sub-properties, specifically focused on maintaining continuity under adverse conditions.

Availability	The ability of the system to deliver services when requested.
Reliability	The ability of the system to deliver services as specified.
Safety	The ability of the system to operate without catastrophic failure.
Security	The ability of the system to protect itself against deliberate or accidental intrusion.
Resilience	The ability of the system to resist and recover from damaging events.

Core Definition of Resilience

The resilience of a system is a judgment of how well that system can maintain the continuity of its critical services in the presence of disruptive events, such as high load, equipment failure and cyberattacks.

14.1 Essential Resilience Ideas

- Some services offered by a system are critical services whose failure could have serious human, social or economic effects.
- Some events are disruptive and can affect the ability of a system to deliver its critical services.
- Resilience is a judgment, not a metric. There are no resilience metrics and resilience cannot be measured. The resilience of a system can only be assessed by experts who examine the system and its operational processes.

14.2 Resilience Engineering Assumptions

- Resilience engineering assumes that it is impossible to avoid system failures and so is concerned with limiting the costs of these failures and recovering from them.
- Resilience engineering assumes that good reliability engineering practices have been used to minimize the number of technical faults in a system.
- It therefore places more emphasis on limiting the number of system failures that arise from external events such as operator errors or cyberattacks.

15. The Four Resilience Activities (4 Rs)

Resilience is maintained through four sequential activities that together form a complete response lifecycle to disruptive events:

1. Recognition	The system or its operators should recognise early indications of system failure. Monitoring and alerting are key here.
2. Resistance	If the symptoms of a problem or cyberattack are detected early, resistance strategies may be used to reduce the probability that the system will fail. This includes both proactive resistance (built-in defences) and reactive resistance (actions taken when a problem is discovered).
3. Recovery	If a failure occurs, the recovery activity ensures that critical system services are restored quickly so that system users are not badly affected by the failure.
4. Reinstatement	In this final activity, all of the system services are restored and normal system operation can continue.

15.1 Resistance in Detail

- Resistance strategies may focus on isolating critical parts of the system so that they are unaffected by problems elsewhere.
- Proactive resistance: defences are included in a system to trap problems before they escalate.
- Reactive resistance: actions are taken when a problem is actually discovered in progress.

15.2 The Resilience Activity Flow

The four activities map to states a system moves through during an incident:

Recognition	Resistance	Recovery	Reinstatement
Normal operating state + Attack recognition	Critical service delivery + Attack resistance	Critical service delivery + System repair	Restricted service delivery + Software and data restoration

16. Cybersecurity

Cybersecurity is a broader topic than system security engineering alone. It is a sociotechnical issue covering all aspects of ensuring the protection of citizens, businesses and critical infrastructures from threats that arise from their use of computers and the Internet.

- Cybercrime is the illegal use of networked systems and is one of the most serious problems facing our society.
- Cybersecurity is concerned with all of an organization's IT assets, from networks through to application systems.

16.1 The CIA Security Goals

Cybersecurity threats target three core security goals:

Confidentiality	Threats where data is not damaged but is made available to people who should not have access to it. The data is exposed without authorization.
Integrity	Threats where systems or data are damaged in some way by a cyberattack. The correctness or completeness of data is compromised.
Availability	Threats that aim to deny use of assets by authorized users. Legitimate users cannot access the service when they need it (e.g., DDoS attacks).

16.2 Factors Contributing to Cybersecurity Failure

- Organizational ignorance of the seriousness of the problem.
- Poor design and lax application of security procedures.
- Human carelessness by employees, operators, or administrators.
- Inappropriate trade-offs between usability and security.

16.3 Redundancy and Diversity as Cyber-Defence

- Copies of data and software should be maintained on separate computer systems. This supports recovery and reinstatement after a successful cyberattack.
- Multi-stage diverse authentication (MFA) can protect against password attacks. This is a resistance measure.
- Critical servers may be over-provisioned, i.e., more powerful than required to handle expected load, so that attacks can be resisted without serious service degradation.

17. Cyber-Resilience Planning

A cyber-resilience plan is a structured organizational strategy for protecting assets and recovering when attacks succeed. It feeds into every stage of the 4 Rs model.

17.1 Planning Steps

Asset Classification	The organization's hardware, software and human assets are examined and classified depending on how essential they are to normal operations.
Threat Identification	For each of the assets (at least the critical and important ones), threats to that asset are identified and classified.
Threat Recognition	For each threat or asset/threat pair, you identify how an attack based on that threat might be recognised early.
Threat Resistance	For each threat or asset/threat pair, possible resistance strategies are identified. These may be technical (embedded in the system) or procedural (relying on operational procedures).
Asset Recovery	For each critical asset or asset/threat pair, work out how that asset could be recovered in the event of a successful cyberattack.
Asset Reinstatement	A more general process of asset recovery where procedures are defined to bring the entire system back into normal operation.

17.2 Streams of Work in Resilience Engineering

- Identify business resilience requirements.
- Plan how to reinstate systems to their normal operating state.
- Identify system failures and cyberattacks that can compromise a system.
- Plan how to recover critical services quickly after damage or a cyberattack.
- Test all aspects of resilience planning.

18. Resilient Systems Design

18.1 Design Principles

- Identifying critical services and assets: Critical services and assets are those elements of the system that allow a system to fulfill its primary purpose. For example, in an ambulance dispatch system, the critical services are those concerned with taking calls and dispatching ambulances.
- Designing system components that support recognition, resistance, recovery and reinstatement: For example, in an ambulance dispatch system, a watchdog timer may be included to detect if the system is not responding to events.

18.2 Survivable Systems Analysis

Survivability analysis is a four-stage iterative process to identify and address vulnerabilities in critical systems:

1. System Understanding	Review the goals of the system (mission objectives), the system requirements, and the system architecture.
2. Critical Service Identification	Identify the services that must always be maintained and the components required to maintain those services.
3. Attack Simulation	Identify scenarios or use cases for possible attacks, along with the system components that would be affected by these attacks.
4. Survivability Analysis	Identify components that are both essential and compromisable by an attack. Define survivability strategies based on resistance, recognition, and recovery.

19. Design Patterns for Resilient Systems

Design patterns are reusable, proven solutions to commonly occurring problems. The following patterns are specifically applicable to building resilient distributed systems. All are documented in the Microsoft Azure Architecture Center.

19.1 Retry Pattern

Category: Recovery

When an application attempts an operation on a hosted service and the request fails with a transient error (such as a temporary overload or network blip), the Retry pattern instructs the application to wait and try again rather than immediately failing.

- Attempt 1: Request fails with HTTP 500 (internal server error). Wait a short interval.
- Attempt 2: Request fails again with HTTP 500. Wait a longer interval (exponential backoff).
- Attempt 3: Request succeeds with HTTP 200 (OK).

Reference: https://docs.microsoft.com/en-us/azure/architecture/patterns/retry

19.2 Circuit Breaker Pattern

Category: Resistance and Recovery

Unlike the Retry pattern which keeps attempting, the Circuit Breaker pattern stops sending requests to a failing service to prevent cascading failures and allow the service time to recover. It operates through three states:

Closed (normal)	Requests pass through. If too many failures occur (failure threshold reached), the circuit opens.
Open (failing fast)	All requests immediately return an error without contacting the service. A timeout timer runs.
Half-Open (testing)	One test request is allowed through. If it succeeds, the circuit closes. If it fails, the circuit opens again.

Reference: https://docs.microsoft.com/en-us/azure/architecture/patterns/circuit-breaker

19.3 Compensating Transaction Pattern

Category: Recovery and Reinstatement

In long-running distributed transactions that involve multiple steps (e.g., booking flights F1, F2, F3 and hotels H1, H2 in sequence), a failure partway through leaves the system in an inconsistent state. The Compensating Transaction pattern addresses this by recording a counter-operation (compensation) for each step. If the overall transaction fails, the compensating logic runs in reverse to undo all completed steps.

Reference: https://docs.microsoft.com/en-us/azure/architecture/patterns/compensating-transaction

19.4 Health Endpoint Monitoring Pattern

Category: Recognition

An external monitoring agent (Agent) periodically calls a dedicated health-check endpoint on the application (e.g., GET /health on port 80 or 443). The application responds with the status of all its internal dependencies: storage, database, and downstream services. The agent aggregates these statuses and can trigger alerts or automated recovery actions when a component reports unhealthy.

Reference: https://docs.microsoft.com/en-us/azure/architecture/patterns/health-endpoint-monitoring

19.5 Throttling Pattern

Category: Resistance

Throttling limits the consumption of resources by controlling how many requests a particular client or tenant can make. When a request rate exceeds a configured threshold (e.g., 150 requests per second for a tenant named Contoso), further requests are rejected with a throttled error. This prevents a single bad actor or spike from consuming all available resources and degrading service for other users.

Combined with autoscaling, throttling can also be used as a load-leveling mechanism: the system throttles while autoscaling provisions additional capacity, then relaxes the throttle once sufficient capacity is available.

Reference: https://docs.microsoft.com/en-us/azure/architecture/patterns/throttling

19.6 Queue-Based Load Leveling Pattern

Category: Resistance and Recovery

Instead of tasks calling a service directly (which can overwhelm it during traffic spikes), tasks post messages into a persistent message queue. The service consumes messages from the queue at its own steady pace. This decouples the rate at which tasks arrive (variable) from the rate at which the service processes them (consistent), preventing overload and data loss.

A real-world example is a simulation service: the client software inserts a simulation request into the queue and polls for status. The service processes the request when capacity is available and posts the result back. The client then retrieves the result and removes the request.

Reference: https://docs.microsoft.com/en-us/azure/architecture/patterns/queue-based-load-leveling

19.7 Pattern Summary Table

Pattern	4R Category	Core Idea
Retry	Recovery	Automatically retry transient failures with increasing wait times.
Circuit Breaker	Resistance / Recovery	Stop sending requests to a failing service; fast-fail while it recovers.
Compensating Transaction	Recovery / Reinstatement	Undo completed steps of a failed multi-step transaction.
Health Endpoint Monitoring	Recognition	Expose a health endpoint; monitor it externally to detect failures early.
Throttling	Resistance	Limit request rates per client to protect shared resources.
Queue-Based Load Leveling	Resistance / Recovery	Decouple producers and consumers via a queue to smooth out traffic spikes.

20. Resilient Systems: Key Takeaways

Summary

- Resilience is a judgment of how well a system can maintain continuity of critical services in the presence of disruptive events.
- **Resilience should be structured around the 4 R's model: Recognition, Resistance, Recovery, and Reinstatement.**
- Processes should be flexible and adaptable to allow operators to cope with unexpected problems. Excessive automation can make it harder for humans to intervene.
- An important part of design for resilience is identifying critical services first. Protect them, and recover them as quickly as possible after failure.

- Cybersecurity threats target confidentiality, integrity, and availability (CIA). Common failure causes include ignorance, poor design, human error, and bad usability/security trade-offs.
- Redundancy and diversity in hardware, software processes, and software systems is essential to the development of dependable systems.
- Formal methods, where a formal model of a system is used as a basis for development, help reduce the number of specification and implementation errors.
- When you face a resiliency problem, look at the established Resiliency Patterns catalog (Azure Architecture Center) for proven solutions.