Outside-In Development and Testing

“Onion/Skeleton Development Process”

- Think about the whole problem first
- Work from the outside in
- Build a skeleton of your entire system first
- Kubernetes->Docker->Skeleton Code->Real Code
- Successive approximation of the final system

Operational Testing

Teaching Operational Testing Is a Challenge

- **Typical response #1 to homework:** “Our system met all challenges!”
- **Typical response #2 to homework:** “We did what you told us to do. Things behaved weirdly. We don’t know why.”

Understand the Purpose of Operational Testing

Know the limits of your system

Find things in your system that need to be improved and improve them.

Using Risks to Guide Operational Testing

Risks

- What are the risks to your system?
- Develop a Risk Register to help you enumerate your risks
- Severity and Probability: help you prioritize efforts
- Mitigation: how do you reduce severity or probability
- Contingency: what do you do if a risk is triggered?

Automatically Triggering Risks Is Key

- Monitor both your software and your operational environment
- Develop a set of triggers for your identified risks
- Examples:
  - Are your services down?
  - Is your code throwing errors?
  - Is your environment running out of resources (high CPU load, low memory, low disk space)?
- If you can articulate it, you can google it: find the tools to spot these risks

Beyond the Risk Registry

- How do you classify types of risks?
- Binary groupings are a good start

Example: Is the risk under your control or outside your control?

Under your control: system performance, scaling, security, etc

Outside your control: Jetstream crashes, Google Drive has an outage, etc

Another Example Categorization: Is the risk triggered suddenly or is it incremental?

- A sudden risk is triggered by a specific event, like network failure or a power outage
- An incremental risk builds up over time, like performance degradation
  - Incremental risks may be punctuated by a crash
- You set thresholds to trigger incremental risks
  - How do you find the thresholds?

Example Risk Associated with Incremental Events

- Your system works well for 25 concurrent users, but response times are twice as long with 50 users, and outright failures are at 1% with 100 users

Incremental Risks Are Often Associated with Non-Functional Requirements

Use Performance Testing to Measure Incremental Risks

Not All Risks Need Performance Testing

- They need to be addressed with better mitigation and contingency plans
- Example: “If IU Jetstream goes down, we’re screwed”
  - Mitigation: Make sure you can move your entire system to a failover location
    - If you have followed the course’s recommendations, this should be doable
  - Mitigation: maintain a separate system for failover cases (just like Blue-Green deployments)
  - Contingency: Move to Amazon and pay the bills

Performance Testing

Understanding your incremental risks

Setting Up Performance Tests

- Apply performance tests to your end-to-end system
- Launch them through the User Interface
- Monitor behavior of the User Interface (response times, correctness)
- Also monitor behavior of services
  - Where are the bottle necks and weak points?
  - Is the system behaving correctly?
- You can also test subsystems (like services that constitute a saga)

Performance Test Examples

- **Load Testing:** how well does the system perform under different load levels? When does performance start to degrade?
- **Stress Testing:** how much load does it take to break the system?
- **Soak Testing:** does the system degrade over time and heavy load?
- **Fault Tolerance Testing:** does the system continue to work if we purposefully cause failures

Load Testing

- What is load?
  - Number of user doing things at normal user speed
- How do you test load?
  - Simulate usage through the user interface
- How do you increase load?
  - Increase number of users
- What do you expect to see as you increase load?
  - Test UI correctness and response time
  - Monitor performance of services and subsystems

Stress Testing

- Like Load Testing, only you really want to break the system
- Sample protocol
  - Test with 10 concurrent users
  - If test passes, increase by order of magnitude (100 users)
  - Continue to increase by order of magnitude (1000, 10,000, ...) until failure
  - If test fails, confirm still working or restart, and then test at the halfway point
    - (10,000-1000)/2, for example
  - Repeat until you have an upper bound
- What do you measure?
  - Correctness
  - Response time

Soak Testing

- Like Load Testing, only for a long time (hours, days, ...)
- Use a heavy but doable load
- Why?
  - Identify "leaks" that cause increases over time in CPU, memory, file system or I/O, and network usage
- What are you monitoring?
  - Correctness of response: did failures occur?
  - End-to-end performance: did it remain constant over time?
- Component correctness and performance are also important
  - Which components degrade over time?
  - Which components are bottlenecks?

Fault Tolerance Testing

- Does my system continue to work when I inject partial failures?
- What is the performance like when I have partial failures?

How to Run Tests

- Microservice environments are highly variable
- You may get different behaviors at different times
- Each test is a measurement or sample
- Collect measurements and calculate statistics: average and standard deviation

In Conclusion, LMGTFY

- Once you have a name for a concept, you can find it
- Load Testing: JMeter is a popular tool, but we’d love to find others
- System Monitoring: Prometheus is a popular tool