Chaos engineering is the discipline of intentionally injecting failures into your systems to see how they respond.

It is to move from a reactive to a proactive approach to build resilient systems.

1. Uncovering Hidden Weaknesses
2. Improve System Resilience
3. Reduce Downtime and Costs
4. Validate Monitoring and Alerting

This concept was first started by Netflix in the late 2000s. They created "Chaos Monkey," a tool that randomly terminated production instances.

Polly is a .NET resilience and transient-fault-handling library.

With Polly, you can gracefully handle these issues and make your application robust and self-healing.

Chaos Studio is the "attack," and Polly is the "defense" that you build into your C# code.

Polly allows you to wrap your C# code (e.g., an HttpClient call) inside one or more "policies."

**1. Retry** : This is the simplest and most common policy. If an operation fails, try it again.

* What it does: Catches a specific exception (like a network error) and re-runs your code.
* Best Practice: Use it with a "wait and retry" strategy, often with exponential backoff (e.g., wait 1s, then 2s, then 4s). This prevents you from hammering a service that is already struggling.
* Chaos Studio Context: Chaos Studio adds 500ms of network latency. Your code times out and fails. The Polly Retry policy catches the timeout, waits, and tries again, and this time it succeeds.

**2. Circuit Breaker** : This is a critical pattern for preventing cascading failures.

* What it does: If a specific operation fails too many times in a row, the "circuit opens." For a configured period (e.g., 30 seconds), Polly won't even try to execute your code. It will fail immediately, protecting your application from wasting resources on a service that is clearly down. After the timeout, it will let one test call through (the "half-open" state) to see if the service has recovered.
* Chaos Studio Context: Chaos Studio shuts down a dependency VM. Your first few calls fail, and Polly's Circuit Breaker trips. For the next 30 seconds, your app fails fast without waiting for timeouts, perhaps returning a cached response. This improves your app's performance and stability even when its dependency is offline.

**3. Timeout** : Ensures that one slow operation doesn't stall your entire application.

* What it does: Enforces a time limit on an operation. If it doesn't complete within the specified time, Polly will cancel it.
* Chaos Studio Context: Chaos Studio injects a CPU spike on a dependency, making it slow to respond. Polly's Timeout policy ensures your calling application gives up after a reasonable time instead of hanging indefinitely.

**4. Fallback** : Provides a graceful alternative when an operation fails.

* What it does: If your code fails (and all retries are exhausted), instead of throwing an exception, it executes an alternative piece of code.
* Chaos Studio Context: Your code tries to call a service to get a user's name, but the service is down (thanks to Chaos Studio). The Polly Fallback policy catches the final error and returns a default value like "Guest" instead of crashing the user's session.

Putting It All Together: The C# Developer's Role

Here is the workflow:

1. **Run Experiment**: Azure Chaos Studio injects a fault into your infrastructure.
2. **Observe Failure**: Your application, which lacks resilience logic, crashes or behaves poorly.
3. **Implement Policy**: You, the C# developer, open the code. You identify the fragile call (e.g., httpClient.GetAsync(...)).
4. **Wrap with Polly**: You wrap that call in a Polly policy. You might even combine them: "Retry 3 times, with a 2-second timeout on each attempt. If all retries fail, open the circuit breaker for 30 seconds and execute the fallback logic."