Gremlin

Post-Incident Playbook for Building Resilience

HOW TO IMPLEMENT PRACTICES THAT PREVENT INCIDENTS

BUILDING RESILIENCE

EBOOK

Post-Incident Playbook for Building Resilience

The outage is finally resolved and the incident dust has settled. So you do a postmortem, craft a plan to address the issues and deploy it. Now you can finally move on and get back to making progress on your roadmap.

At least until another outage takes your system down and the whole circus starts again.

Sound familiar?

It's an all-too-common loop that's painful for engineers and companies. Unreliable systems mean engineers are constantly pulled off important work, delaying releases and contributing to burnout, while outages lose revenue, damage brands, and even lead to regulatory fines.

Theoretically, we should be building more resilience with every incident, but in practice, the rapid deployment of code means we're always behind—and it's only gotten worse with AI-driven development.

It's time to break the cycle with a proactive reliability management practice built around preventative testing and risk resolution with provable results.

And the conversation needs to happen now as part of the postmortem process. Not next quarter or next year when your organization is "more mature," but when everyone has just felt the impact of poor resilience and the outage is fresh in your minds. Because if you don't, then it's only a matter of time until the next failure.

Following this approach and these best practices, hundreds of leading companies have improved their availability, decreased outages, and broken the endless cycle of disruptive incidents.

Resilience has to be verified with real failures

Every piece of code undergoes some testing before it's deployed. But it's impossible to do fully exhaustive QA testing, and there are many failure modes that don't show up when you test code in isolation. The only way to really know whether your service will be resilient is for it to face failure in a production-like environment with real traffic levels.

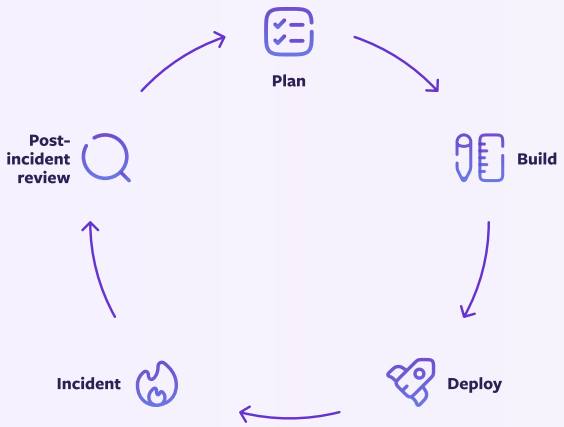
Which leaves you with two options:

You can wait until an actual failure mode occurs in production and causes an incident, then react as fast as you can.

You can safely simulate those failure conditions and verify resilience before it causes an outage.

The first option is, sadly, more common, but the second option is the more prudent one. It's also the only way to break the firefighting cycle. By using fault injection testing, you can create the conditions of failure, such as a dependency being unavailable, increased latency, or a region outage, see how your systems react, then safely restore back to the previous state.

And by integrating the tests with your observability tool's existing monitors and alerts, you can create a pass/fail gate based on signals you care about. If metrics don't hit the alert threshold, then the code passes and is resilient. If it does, then you've identified the point of failure and have an opportunity to fix the issue before it causes an outage.



Standard cycle

Resilience testing cycle
(incident prevented)

When applied correctly, this kind of testing can help you avoid outages outright. And if you test regularly with automation, a natural cultural shift will start to happen where engineers proactively build better resilience and are more efficient.

How to determine which failures to test

The first thing you should test is the failure that just occurred. Yes, you patched it, and it's tempting to assume this means it won't happen again...but you can't be 100% sure. And that means you can't prove resilience to leadership and the rest of the company.

As an example, say a dependency outage of your buyer rewards service caused cascading failures with your checkout service. After resolving the incident, you built circuit breakers to prevent the situation from happening again.

It's logical to assume that a circuit breaker will do the job, but if you want to prove it, then you should run a test that simulates the same outage by cutting off all traffic to the buyer rewards service. If the circuit breaker engages and the purchase can still go through, then your fix was successful. If it doesn't, or if it engages and the service still goes down, then it's far better to find out now than when a customer tries to check out.

Once you're done testing that failure, you should also branch out to similar failures. After all, just because the checkout service can withstand the failure of one dependency doesn't mean it's resilient to all dependency failures.

Outage Cause	% of companies affected	How to test for resilience
Network failure	35%	Verify service redundancy and failover by blocking network traffic
Third-party or cloud provider failure	28%	Verify resource/dependency redundancy by blocking network traffic
Deploying software changes	28%	Run a standardized group of tests in staging to verify resilience before deployment
Someone making a change to the environment	26%	Automate test suites to run in production and verify continued resilience
Security failure	24%	Follow standard security practices
Hardware failure	23%	Verify resource redundancy by blocking network traffic to a node
Power failure	22%	Verify resource redundancy by blocking network traffic to an availability zone or region
Capacity constraint	20%	Verify service resilience by blocking traffic to a percentage of nodes in a cluster
Unexpected traffic surge	18%	Verify scalability by consuming resources (CPU, memory, disk I/O, etc.)
DNS issue	18%	Verify DNS redundancy by blocking network traffic to your primary DNS provider
Certificate expiration	13%	Ensure no SSL/TLS certificates are expiring soon with a passive check, including third-party dependencies

SOURCE: New Relic 2025 Observability Forecast

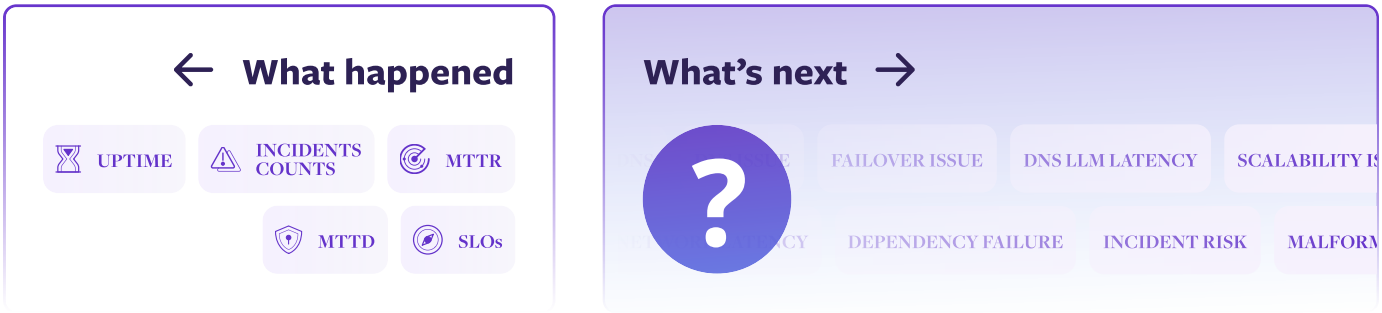
The failures in the table above match the most common outage causes with tests to prove resilience. Obviously, there's an entire other practice for 24% of security outages, but the majority of the other causes can be tested using resilience testing. To truly improve the reliability of your application and its resilience to failure, you should test all of these other causes to eliminate as many reliability risks as possible.

Measuring reliability and automating at scale

No individual service exists in a vacuum. Every application and system out there is a web of interconnected dependencies, and one that's almost constantly changing. You might increase the reliability of the checkout service, but that can only go so far if the payment processing service isn't also resilient.

Improving reliability has to happen across the whole application, which means you need a systematic way to track changes in the reliability of its individual pieces. And for that, you need a single metric that can apply to all services.

Unfortunately, it's impossible to measure reliability with incident response metrics like MTTR, uptime, or incident counts. All of these are backward-looking: they'll show you what already happened, but not what will happen.



Instead, you can use the test results to create a forward-looking reliability metric. After all, the whole point of the test is to show you how your service will react in the future.

Start by setting policies around a standard group of tests. The most common failures listed above are a great place to start, and give you a broad range of coverage. Once you have results from those tests, you can use them to produce a reliability percentage.

Say you had a group of ten tests you ran. Passing all of them would indicate that the service is 100% resilient against those failures, which means a passed test would be worth 10 percentage points. However, a failed test is worth zero, as it means there's no resilience against that failure mode. So if your service passed seven tests and failed three, then it's 70% resilient against those particular failures. If you fix something and now it passes eight tests, it becomes 80% resilient. (Note: For this scoring, an inconclusive test, such as one that didn't run correctly, should also be scored zero because you can't be sure your service would've passed.)

As you track these results over time, you get a metric that shows you when the reliability of the service improves or worsens. And if you automate the same group of tests to run across all of your services, now you have a consistent reliability metric that shows you, and your leadership team, the resilience of your application and which outages you're actively avoiding.

That's how you really break the loop: systematic, standardized reliability measurement across your organization. Because when you can predict your system's response to failure, you're no longer just reacting to outages—you're preventing them.

Going from postmortem plan to program execution

Organization-wide resilience means organization-wide buy-in, and the postmortem is a great place to kick that off. During a postmortem, you have the attention of key engineering stakeholders, exactly the people who can sign off on a program and have the accountability to make it truly impactful.

But how do you get that sign off? Over the years, the successful programs we've seen at Gremlin start by connecting the impact directly to the business and the person in leadership who is most responsible for that part of the business. That way, instead of a fascinating engineering side project to find bugs, it becomes a key business initiative to improve resilience and reliability.

When you break down your case for the new program, make sure you answer these key questions:

What's currently broken?
During an RCA/postmortem, you've already found something wrong with your system, but what was wrong with the processes that let that issue make it out into production? There are many possible reasons. Things might have been changing too quickly, assumptions may have been made about the reliability of third-party dependencies, or AI coding agents may have introduced configuration drift.
But all of these come down to the same issue: no one tested how the systems would respond to real failure conditions. Because risks will happen, and it's up to you to make sure they're found and fixed.

What is the resilience testing approach?
As explained above, resilience testing is different from the standard battery of QA tests. And that's the whole point. Resilience testing takes a holistic approach grounded in the entire system's response to real failure conditions, which means it catches things other tests can't.
Your goal with this program isn't to replace other testing methods, but to supplement them to close the gaps and truly verify the resilience of your code.

What is the business impact?
There are general approaches you can use to figure out the ROI of reliability, such as looking at the average cost of downtime per minute, including the cost of engineering time, then comparing it to the amount spent on reliability to get an ROI percentage. (For more, check out the blog post What is the ROI of reliability?)

But this is even easier right after an incident or outage. You've all just seen what an outage can cost in terms of spent engineering time, brand impact, lost sales, and more. Now you need to show that regular resilience testing and a built-out program could have uncovered the risk before the outage. If you can show that, and that the testing would have found the causes behind other recent outages, then you've proven a positive business impact for the program.

Program Proposal Template

On a broad level, your proposal should end up looking something like the template below. This lays out the program details at the top, then a cost-benefit analysis at the bottom.

For the purposes of this template, we've filled in some average global numbers, but you'll want to get those numbers as specific to your company as possible. When you do, be sure to include the labor costs as well.

For example, a retail company might track the cost of downtime or lowered performance to the minute in terms of lost sales, but not necessarily the amount of engineering time it costs to repair the incident. With sometimes hundreds of outages every year, each pulling in a dozen or more engineers, that cost can easily add up to millions above and beyond the impact of lost sales.

Reliability Management Program Proposal

What does implementation look like?	The key part here is ownership. Historically, programs without ownership or accountability struggle to get results. Answer questions like, "Who is going to hold the group accountable?" and "Who will review the numbers and ask hard questions?" For more, check out the Best-in-Class Reliability Program assessment to see how your plan matches up.
What is it?	Describe the reliability management program as a way to verify, track, and report resilience organization-wide in order to prevent incidents and future outages.
What problem is it addressing?	Be specific! Tie this directly to the outage that just occurred, as well as other recent outages to show how the issues slipped through current processes.
How will it address the problem?	Give specific examples of how the tests could have uncovered the failures that caused the outages. Not sure exactly which tests to use? Check out the Gremlin docs or reach out to us for help!

Cost-Benefit Analysis

Outage Costs		
Hrs of downtime last year *(or similar tracking time period)*		175 hrs/yr (98% uptime)
Cost of downtime/hr (Use company information or industry averages)		$411,480/hr (Avg. for 2,500-5,000 employee companies)
Downtime costs *(Cost/hr * hour of downtime)*	**~$72,000,000**	
Program costs & benefits		
Testing Tool License Fees	**$100,000**	
Predicted decrease in incidents (Gremlin customers report 10-20% decrease yearly)		15%
Predicted Savings (Total outage costs * % decrease)	**$10,800,000**	
TOTAL PROGRAM BENEFITS	**$10,700,000**	

(Not sure which numbers to use? Check out our interactive ROI calculator at gremlin.com/tools/roi)

Manage reliability safely and easily at scale with Gremlin

The Gremlin reliability management platform was built specifically to break the constant firefighting cycle of repeat outages. Its ability to safely test failure conditions at scale has been proven at hundreds of companies, including global organizations in zero-tolerance industries like financial services or retail.

In Gremlin, the most common outage causes are collected into an out-of-the-box suite of tests that makes it easy to roll out testing across services. And it includes robust reporting for tracking and proving reliability across your team.

And that's just the beginning. Gremlin's AI capabilities analyze the test results, compare it against a wealth of data gathered from millions of tests, then recommend a possible fix for your team or your agentic AI to implement.

It's time to break out of the firefighting cycle. Start by proving the resilience to the outage that just happened, then expand efforts to improve resilience and reliability across your entire organization.

Gremlin

Gremlin is the reliability management platform trusted by the world's largest enterprises across financial services, SaaS, retail, and media to help meet uptime and performance goals.

Instead of lagging indicators that show past reliability, Gremlin gives engineering teams predictive reliability data so they can systematically measure, manage, and improve reliability.

By combining failure testing, risk detection, and dependency mapping with automation, standardization, and in-depth reporting, Gremlin makes it easy to see where systems are at risk, prioritize what to fix, and track progress across teams over time to prove the results.

Learn how to manage your reliability at gremlin.com