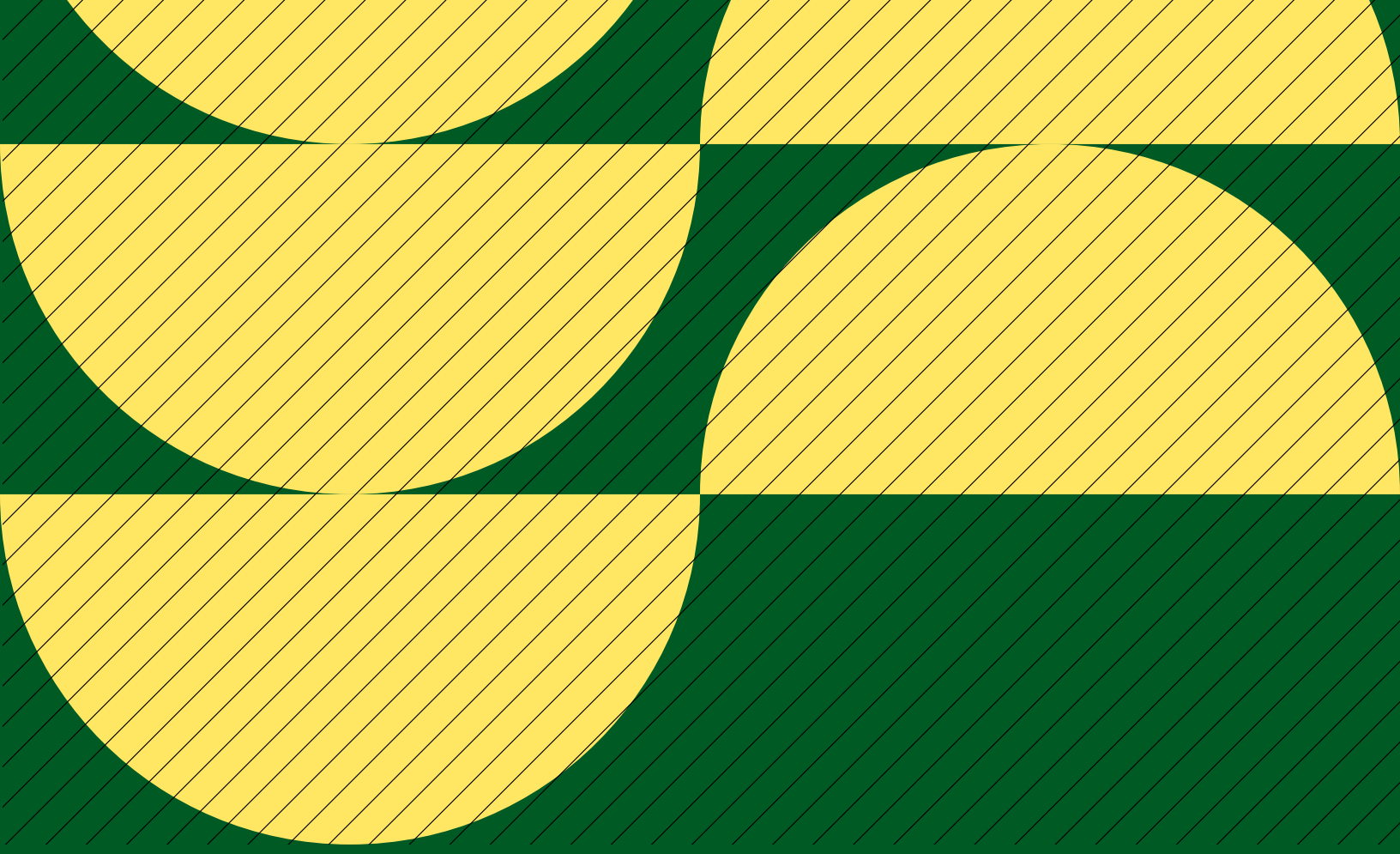


PagerDuty

9 Steps to Owning Your Code

You code it? You own it.

A Developer's Guide to Managing Your Code

In this guide, we'll share strategies around successfully operating services, making on-call as stress-free as possible, and resolving and preventing issues efficiently. These best practices will help prepare you to excel in the new software-defined, developer-driven world order.

Contents

Own Your Code

As technology increasingly shapes business objectives, the role of the software developer is also rapidly changing. Digital natives are disrupting every industry vertical and businesses are expected to innovate, respond, and be available to their customers 24x7. To meet these demands, developers — the architects of customer experiences — are being pushed towards owning their own code. Developers know their code and can fix it faster than anyone else and are able to ship more performant code when they are accountable for managing it. It makes sense that leading organizations are tasking developers with both development and operational responsibilities as the best way to maximize speed, agility, and application performance and quality.

Managing your code is something to be excited about. Establishing this accountability empowers you with the information and control to ensure the services you build are production-ready and high-performing. It ensures you are doing high value work, as you have direct line of sight into how your product or service is actually performing and impacting your customers' day-to-day. And as the customer experience is in your hands, you drive the success of your organization, as well as your own destiny.

This all sounds great, but what exactly does this mean for **you**?

It means that on-call and first-level response is no longer just the responsibility of a centralized NOC or an individual operations team on the other side of a wall. Rather, being on-call will inevitably be a fundamental expectation of you as a developer (if it isn't already), regardless of your organization's size or operational framework. It is now best practice for those who build services to also be accountable for the success of those services in production. This, of course, is in addition to existing, daily software development responsibilities.

Clearly, this introduces significant challenges and the notion of being on-call can be anxiety-inducing on both professional and personal levels. From talking to thousands of developers, we've found that these are some of the things that are top of mind:

HOW DO I FULLY **UNDERSTAND EVERYTHING THAT IMPACTS MY SERVICE** AND MY CUSTOMERS' EXPERIENCES?

HOW DO I ASSESS THE IMPACT **AS QUICKLY AS POSSIBLE**?

HOW DO I ENSURE I ***DON'T WASTE MY TIME*** **DOING AUTOMATABLE THINGS?**

HOW DO I FIND OUT ABOUT MY PROBLEMS ***BEFORE MY CUSTOMERS NOTICE?***

HOW DO I **MAKE SURE** THIS ISSUE **DOESN'T HAPPEN AGAIN?**

WHEN I NEED HELP, HOW DO I **PULL IN THE RIGHT PEOPLE** AND GET THEM UP TO SPEED AS QUICKLY AS POSSIBLE?

HOW DO I ENSURE I'M ***ONLY*** WOKEN UP FOR **CUSTOMER-IMPACTING ISSUES?**

CAN I WORK MY WAY AND WITH THE TOOLS ***I LIKE?***

HOW CAN I CONSTANTLY LEARN TO BOTH **IMPROVE RESPONSE** AS WELL AS BUILD A MORE RESILIENT SERVICE?

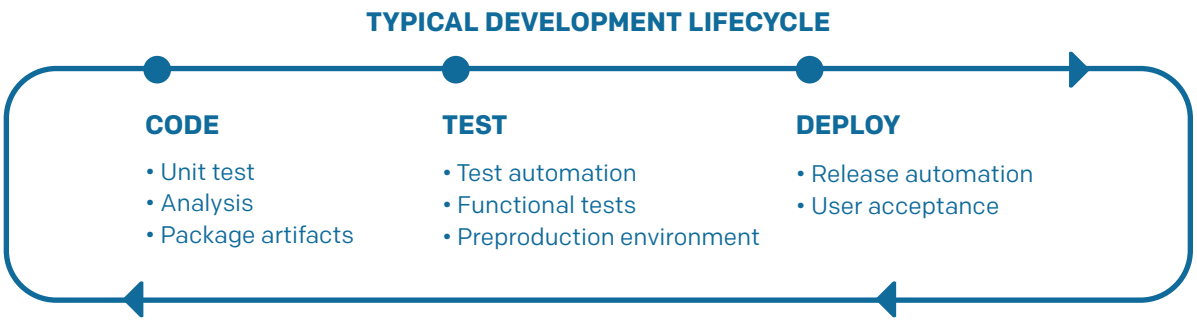
No one wants more unplanned work, or to get paged at 3 a.m. about an issue that could take anywhere from a few minutes to a few hours to solve. Also complicating the equation is the fact that identifying the root cause of an issue is increasingly unpredictable. With continuous integration and continuous delivery, as well as the rise of microservices, developers today are dealing with more infrastructure complexity and faster release velocities than ever before. Technology stacks are becoming so complicated that many teams are outsourcing the provisioning of infrastructure components to keep up with demands for speed and scale. Increased infrastructure complexity results in more things to monitor as well as more unknowns. Moreover, siloed monitoring data stems from tools that don't talk, creating an unprecedented volume of alert storms and responder noise. In this context, it may seem nearly impossible to expect anyone to manage and troubleshoot unplanned issues across complex systems processing hundreds, thousands, or even millions of transactions every hour.

The good news, however, is that you are not alone. Developers at thousands of organizations have already managed this expectation for great results: they are able to minimize performance issues, and improve both operational efficiency and customer satisfaction. Ultimately, the organizations that can deliver the best results to customers are those that turn on-call responsibilities into a key area of strength — organizations in which developers are empowered and personally invested in the entire lifecycle and the success of the services that they build.

As a developer, you code it, you build it, you test it, you deploy it, you manage it. Above all, you own it. The customer experience is in your hands. This is an awesome responsibility, and the best developers embrace it and thrive. And by putting the right processes in place, you can spend more time innovating and doing the things you love, instead of on manual tasks related to fixing production issues. You will even be able to ship higher quality code and spend less time on-call. All of these factors lead to a better work-life balance.

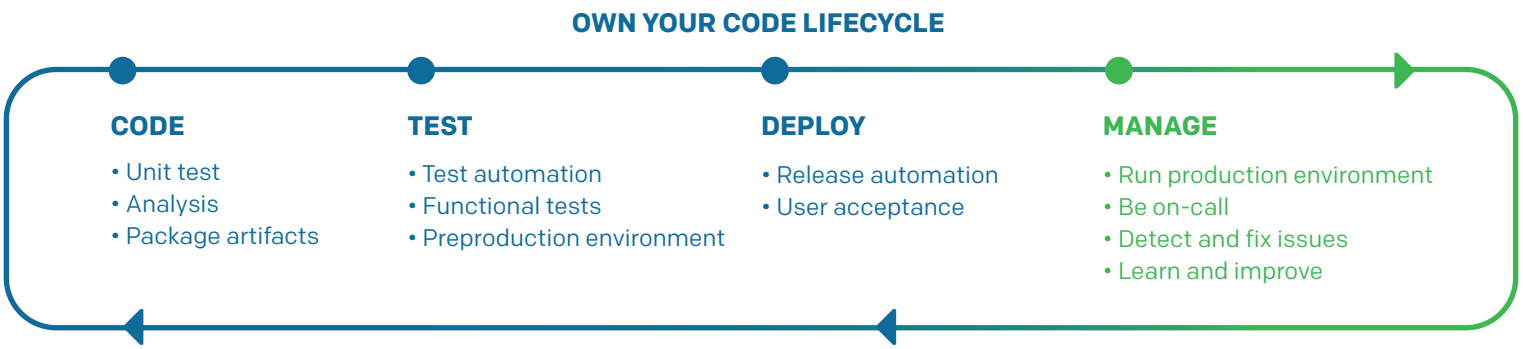
How the Developer's Role Is Changing

To understand how the role of the developer is changing, let's first take a look at the stages of the typical software development lifecycle.



Typically, after the last "deploy" stage, the development team might throw the code over the proverbial wall to the operations team, to manage it and fix issues once it's out in the wild. Day-to-day performance monitoring, issue detection, maintenance and debugging can sometimes be a shared responsibility between development, operations, or the NOC. Regardless, it's important to remember that for developers, the software development lifecycle shouldn't just end once their code has been deployed. The best developers don't just turn a blind eye — they take responsibility for and care about continuously improving the end customer experience.

The additional column in green summarizes responsibilities that are crucial to a developer's role. This updated graph reflects all the hats developers must wear across the software development lifecycle — from coding all the way to owning.



9 Steps to Successfully Owning Your Code

Now that we've established the responsibilities of a developer, let's address how to make them actionable. At PagerDuty, we've spent almost a decade engaging with hundreds of thousands of developers from organizations that represent every stage in the operational maturity continuum, and we've distilled their experiences into 9 key steps.

01

Understand your services and your customers' experiences.

Centralize all data sources: Your service is impacted by many factors — your external or internal cloud hosting environment, the network environment and other infrastructure, and services outside your control. There are many moving pieces in the stack that have the potential to impact the performance of your service. If you're able to see clusters of events across all relevant sources that indicate an issue, it can help shape your development decisions such as dependencies to account for, how much memory you actually need, etc. Having a comprehensive, live view of all data sources (application performance monitoring data, network health, social media feeds, etc.) is essential for building more resilient services and preventing issues.

02

Find out about problems before your customers notice.

Monitor: Establish monitoring systems to monitor usage trends, user behavior, application performance, logs, resource metrics (CPU, network, disk, memory), and other critical system and infrastructure indicators. Leverage several or redundant sources of monitoring data (both human-generated such as tickets or phone calls, and machine-generated such as logs or tracing) to ensure that issues are never missed.

Aggregate: Consolidate all event data in a centralized location so you can proactively identify patterns and anomalies across the entire infrastructure. This is especially important for understanding the blast radius of an issue and root cause, as it correlates data across vendors that don't talk to each other.

Detect and prioritize: Structure events by urgency so that they automatically take on the desired behavior and workflow. For instance, send an email if it's low-urgency, such as a generic health metric. But when an event is detected that potentially impacts customers—for example, an uptime monitoring alert that goes off at 3 a.m. indicating your payment service is down—you should get called, texted, and notified to take immediate action.

Notify: Have a system in place that automates on-call staff scheduling and escalations to the next line of defense. This establishes accountability as well as ensures issues requiring a response are always routed to the right person in real-time.

03

Ensure you are only woken up for customer-impacting issues.

Adjust tuning: Fine tune your alerting thresholds and notification behavior to match the indicators that your team cares about. For example, instead of getting notified on a single request failure, you might choose to get notified only if 50 requests fail in 5 minutes, as crossing that threshold is more indicative of a real problem that needs attention as opposed to a random blip.

Define notification urgencies and routing: Set high-urgency issues to notify and escalate via predetermined, high-urgency rules. For example, you can set a rule that SEV-1, customer-impacting outages automatically ring the on-call, notify stakeholders from customer support and marketing/PR, and escalate more quickly to IT management and executives.

Suppress non-actionable alerts: Just as you've defined notification urgencies for actionable alerts, you should also suppress non-actionable alerts (such as warnings, self-healing events, etc.) so you can minimize alerting noise and focus on what matters. However, while you don't want to get paged, you also don't want to throw away this data as it can be highly valuable for identifying simmering problems and is key to understanding the overall health of your infrastructure.

Consolidate actionable information: A single issue can often cause degradations among several services, firing off a ton of alerts simultaneously that are all symptoms of the same problem. If all of these page out, this lack of aggregation quickly creates many issues: a huge amount of responder noise and alert fatigue, manual work in parsing through and acting on every alert, and highly disjointed resolution and communication streams. As such, being able to consolidate related context into a single object is essential.

04

Assess the impact as quickly as possible.

Normalize events: Normalizing inbound event data into a common format is critical for ensuring all the engineers on your team can quickly and correctly interpret events, as many vendors have different schema (for example, "alias" in Zabbix is shorthand for any configuration term, but in Nagios it is a given name for a host). Critical information (such as source component, location, etc.) must be translated into common fields that everyone can understand.

Identify (potential) root cause: Identifying the root cause can be difficult due to unknown unknowns, but leveraging real-time visualizations that correlate the events across your infrastructure can help. This step is critical in helping you identify which downstream services may have been affected by an issue.

Assess the blast radius: Identifying the affected services and their level of importance to the business, and/or the number and type of affected customer accounts, helps you classify the severity and urgency of the issue at hand. Pulling in alternate data streams such as real-time social media feeds, support tickets, etc., also create a holistic illustration of the extent of the outage impact.

05

Recruit the right people quickly.

Real-time collaboration: When all the right context (related alerts, metadata, monitoring iframes, graphs, the timeline of activity, runbooks, etc.) is in a single location, getting additional responders up to speed is significantly easier.

Automate responder and stakeholder recruitment: In the midst of the firefight, you don't want to be trying to manually digging through the company directory to find, contact, and recruit the right subject matter experts to help. During a critical outage, every minute spent trying to kick off the response workflow could cost the business thousands of dollars. Create groups of the right experts beforehand that have their correct contact details embedded.

06

Prevent recurring issues.

Surface remediation information: Make remediation information as easy to find as possible. Ideally, real-time runbook information is pulled into incidents directly through the API.

Evaluate the likelihood of an issue resurfacing, and prioritize long-term resolution based on its impact/severity.

Defect remediation: Defect remediation aims to fix the underlying issue itself. This can be done in several ways. You can try to make the service fault-tolerant or self-healing. This also applies to the engineering process itself (for example, the team didn't check the code for syntax errors/conduct proper linting before every commit).

Additionally, it's possible that an issue only came up in the production environment and not in the dev or test environments, because all those environments weren't all as similar to each other as possible.

Owning the services you build in production helps significantly here. You gain a better sense of what could go wrong in production under real workloads because you actually feel the pain and have to fix the problem when something breaks. Without visibility into the capacity of the machines in the production environment during peak workloads, if you're simply throwing code over the wall and it's not performant, your code will not pass stress tests.

Escape prevention remediation: The goal here is to ensure issues are detected in the test environment and that they don't escape undetected into the production environment. The main way to fix this long-term is by using automated testing wherever possible in favor of manual testing.

Production detection remediation: When an issue is not detected properly in production, it is usually due to lack of proper alerting or too much noise in system. This type of issue could already be impacting customers before you find out about its existence —it might even be reported in by a customer. The key to preventing issues from going undetected in production is to ensure the right alerts are getting through getting escalated, and getting routed appropriately so that they don't go unnoticed again.

07

Work your way.

Leverage an API to customize and build workflows: With an API, you can optimize your existing tools by creating workflows that make the most sense for you. For example, in a continuous deployment example, you can hook in your deployment data via an API, embed real-time data from different sources, identify who you want it to route to, and automate that notification behavior. Certain actions can ideally also be automated through the API to facilitate an improved end-to-end workflow (such as creating and resolving incidents, merging incidents, normalizing inbound event data, and more).

Fix issues in context: Eliminate the need to waste time toggling in between tools, by having all events and pipe into a single location within your tool of choice. Staying in a single tool for real-time notifications, collaboration and knowledge sharing minimizes context switching, and thus maximizes speed and productivity.

Use ChatOps: Improve operational efficiency and save time by automating many ops-related tasks with slash commands and chatbots.

08

Don't waste your time. Automate.

Automate manual tasks: Automate repetitive tasks wherever possible so that you don't have to deal with them in the heat of an incident. This removes a lot of the burden and stress so you can focus on response. One example is:

> **Automatically surface the right information:** One of the hardest parts may just be getting all the right context together. For example, when you get paged you might immediately start toggling through Splunk, Datadog, etc. to try to pull up the right metrics and parse through to find the information you actually need. This involves trying to remember which query you need to use for Splunk, or whether or not you're looking at the right dashboard in Datadog, etc. By putting steps in place to automatically surface runbook information or embed the right real-time monitoring iframes and metrics in your view, you can minimize the cognitive load of those additional tasks.

09

Build more resilient services.

Introduce intentional failures: Injecting failure is one of the best ways to improve the resilience of your services. Failure tolerant design is great, but you can't plan for everything. The best way to manage the unexpected is to get really good at finding failure. This helps the team proactively identify and remediate deficiencies that could otherwise eventually impact customers. Of course, be sure to put the right measures in place to ensure that failovers work as expected and don't prevent core services from operating as usual.

Learn from post-mortems: Post-mortems are a given for all high-performing teams, as they document what went wrong and what can be learned from every incident. Capture critical pieces of context such as what happened, root cause, the level of impact (length, severity, # of users affected, etc.), what went well, what didn't go so well, the complete timeline of events, and action items stemming from the incident.

Ultimately, operating services and being on-call is essential for you to deliver real value by driving customer outcomes.

It is a fundamental best practice for shipping better and more performant code and will inevitably help you grow your career. As you take on more responsibility on your personal journey towards becoming a better developer, implementing these best practices will help you reduce the on-call pain that exists as a result of unplanned or redundant work. By delivering stress-tested and resilient code, proactively preventing repetitive issues, and silencing non-actionable alerts; you will spend less time fixing issues while on-call and more time being productive and creative. And when issues do arise, you will be prepared to resolve them as quickly as possible by easily surfacing remediation details and consolidating related context, automating manual processes, and customizing workflows that enable you to work how you like, with the tools you like.

Being on-call should be a huge source of empowerment, not stress. By making on-call seamless through implementing these best practices will position you to do the most impactful work of your career that directly drives the success of your customers and your organization.

Try PagerDuty Free for 14 Days

About PagerDuty

PagerDuty is the leading digital operations management platform for businesses, that integrates with ITOps and DevOps monitoring stacks to improve operational reliability and agility. From enriching and aggregating events to correlating them into actionable incidents, PagerDuty provides insights so you can intelligently respond to critical disruptions for exceptional customer experience. With hundreds of native integrations with monitoring and collaboration tools, automated scheduling, advanced reporting, and guaranteed reliability, PagerDuty is trusted by thousands of organizations globally to increase business and employee efficiency.

We are proud to be the first vendor that introduced the tools and APIs that developers need to optimize their environments for on-call success. From 2009 to today, we've supported hundreds of thousands of developers on our platform. Developers on PagerDuty can leverage a single platform for event normalization, customizable event management at scale, numerous and well-documented API endpoints, ChatOps extensibility and collaboration workflows, response automation, and more.

You code it? You own it. You need PagerDuty.

Learn more about PagerDuty for developers or start a free trial.

PagerDuty