The Dev and Ops Guide to Incident Management

Modern Theory, Process, Team Structure, & Roles

About the Author

Matthew Boeckman

Developer Advocate, VictorOps

Matthew is an 18-year veteran building infrastructure and leading engineering teams. Despite his heavy Ops background, Matthew has been a longtime friend of Developers and considers DevOps his primary passion and focus. Most recently VP of Infrastructure at Craftsy, Matthew is now a Developer Advocate with VictorOps - providing focused consulting and content for teams looking to expand their DevOps practice.

Table of
Contents

Introduction.

What happens when things break down? Does anyone know? Does someone get called? Can that person fix the problem? Is a fix even possible? These fundamental questions occur anytime a system encounters a problem. Information Technology and Software Development are no different. Indeed, with the rapidly growing costs of system downtime, how to manage Incidents and Events is now an essential business focus. As the continued adoption of DevOps, System Reliability Engineering, Agile, and related practices shows, traditional responsibilities are in frequent flux.

"...how to manage Incidents and Events is now an essential business focus."

Team members, traditionally unaccustomed to incident management processes, are now being dropped into loosely defined on-call roles. This creates the risk that the first-line responders are actually unprepared to address incidents quickly and effectively. To be successful in this role, it is necessary to have a basic familiarity with the fundamental concepts of incident management.

This guide provides all levels of teams with a framework and common vocabulary for managing their incident management practice. It is a guide for Development and Operations professionals alike, or anyone wishing to take a larger responsibility for systems. With the increasing complexity of systems and applications; it's not a question of if, but when, things will break down. If ever you are faced with the question: "What happens when things break down?", this guide is for you.

Chapter 1

Incident Management Theory and Practice

1

Chapter 1
Incident Management Theory and Practice

Software and Infrastructure have a fascinating habit of breaking. They break in small ways, and big ways. Sure, sometimes a tree falls in a forest and no one hears, but sometimes a simple keystroke error results in a deafening roar.

Whatever the impact, it all begins with an Incident. An Incident is an arbitrary construct of observational data into a system, and while most systems share common Incident patterns, they tend to be highly application-specific.

Regardless of the specifics, it's helpful to think of an Incident in a dynamic fashion. From this view, Incident Management becomes as much about response as it is about monitoring and analysis. A nice framework for understanding the full scope of the incident lifecycle is found in these five main phases of Incident Management:

- **Detection:** monitoring, metrics, thresholds
- **Response:** alerting, on-call, escalations, access, diagnosis
- **Remediation:** tickets, deployments, dispatches, fixes
- **Analysis:** postmortem, how or why, understanding
- **Readiness:** feedback, improvement, learning

The concepts of **Iteration** and **Empathy** also underpin the success of modern Incident Management. Iterative approaches implicitly seek improvement, and adaptation to patterns. Empathy acknowledges that these things are stressful and humans, whatever their professional background, don't operate well under stress.

"Effective treatment of Incidents requires a working knowledge of the main concepts and phases, as well as how they interrelate."

1 System Concepts

Anyone who has worked alongside an Operations professional likely has a basic grasp of Incident Management. Something goes wrong (with a Monitored system), some other thing notices it (Monitoring system), a notification makes a phone go *beep beep beep* (Incident Management system), and someone's heart rate skyrockets as they jump in to respond.

Let's dive into each of these systems.

Monitored Systems

Monitored systems are just that – the servers, switches, routers, instances, containers, clusters, and/or databases being monitored. Monitoring can generally be broken into a couple categories: System, Application Performance, and Business Objective (or passive).

- **System monitors** tend to be generic and apply to the system resource consumption: CPU, Memory, Storage usage or capacity, Network, etc.. System monitors are historically the realm of the Operations team.
- **Application performance monitors** are focused on application specific metrics: Heap, connection pools, cache utilization, etc.. APM tends to be more interesting to Developers.
- **Business objective monitors** measure business performance as an indirect way to gauge application or infrastructure health. Some common examples are transactions/sec or new user registration/hour.

Regardless of type, monitored system checks share several basic characteristics:

- **Check frequency:** how often the observation occurs
- **Thresholds:** at what value, or rate of change, is something considered "unhealthy"
- **Time periods:** expected results may vary widely by time of day

Of course, with the adoption of complex systems, there is also significant variance in the way systems are monitored. Active/Passive, log based, data inference, and many other approaches are used to isolate a basic truth: **Things are OK**, or **Things are Not OK**.

Monitoring Systems

Monitoring Systems run the checks, receive the results, check them against the expected value, and ultimately trigger an Incident. Oftentimes monitoring systems are also Monitored systems (Quis custodiet ipsos custodes?). Monitoring systems tend to be complicated to configure, and offer flexible rule processing. Some examples of active monitoring include Nagios, Zabbix, Datadog, and Zenoss. Some examples of passive monitoring include Splunk, Sumo Logic, and Logstash.

Incident Management Systems

Incident Management systems apply a secondary logical operation to an Incident. Based on configured rules, an Incident Management system determines:

- Who should be alerted to this particular Incident
- What method of notification (chat, phone, SMS, etc) is appropriate
- The state of the Incident and execution of Escalations or secondary workflows

So far, our examples have covered the Detection phase of Incident Management. From here we will explore Response and Remediation.

This is where the rubber meets the road for both Developers and Operations - it is likely that multiple “someones” are on-call. They are going to be alerted, at any time of day or night, about an Incident. They will be expected to respond in a deliberate and consistent way to Incidents of all types, covering all systems. Regardless of the secondary systems used to manage the Incident, they are absolutely expected to resolve the Incident in a timely manner.

Response

In many ways, Response is the simplest of these phases to implement. Teams agree on some approach to shared responsibility, which reflects in a schedule and rotation. A team of five Front-end developers may agree on a weekly rotation, where each member is on-call for 1 week out of 5. A team of 60 Java developers may agree to split responsibility along functional lines associated with the application, or by service.

All teams will include a mix of seniority, experience, and familiarity with a given technology. Similarly, all individuals will bring varying degrees of availability and dedication to the problem. For any Incident, who should get called if the first responder is not responding, or is incapable of resolving an incident? Escalation policies are the way to approach these questions, and are a key component of the Response phase.

While these concepts may seem obvious, the tactical execution of them is critical to a successful Incident Management process. Every second that passes without the "right people" responding to an Incident extends the service interruption.

Remediation

Nearly everyone, Developers and Operations alike, come to a conversation on Incident Management with the notion that "fixing things" is all there is to talk about. A more sophisticated view is to understand that Incidents, like software, have a lifecycle. Effective practice requires deep focus on Remediation, but also an extended view to understanding the inputs to, and outputs from, this phase.

That said, much of the actual time, if not the excitement, of your Incident will be spent in Remediation. Triage, diagnosis and resolution is, after all, what we're trying to do. An ideal approach to remediation follows a pattern similar to that of the Scientific Method:

1. Make an Observation

this is the Detection phase - some metric is no longer where it should be

2. Ask a question

why is this metric now X?

3. Form a Hypothesis

perhaps because....

4. Test the Hypothesis

look at more metrics, examine logs, deploy new code, change configuration, etc...

5. GOTO Step 1

This methodical approach sets a team up to quickly investigate, identify, and recover from an Incident.

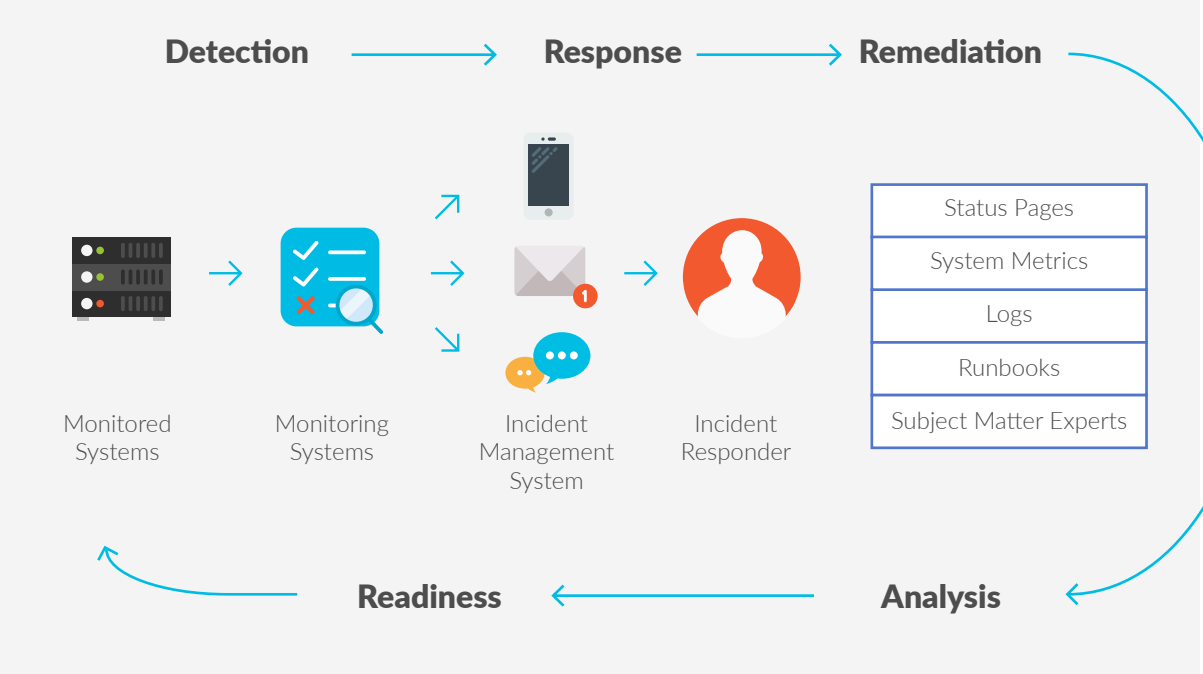
As you can appreciate, the supporting tools involved vary widely by technology stack, severity of Incident, functional role of the responder, and more. Whatever the specific choices, the functional breakdown of remediation tooling can be grouped like this, with a few examples:

- Documentation (Confluence, Visio, Wiki, Swagger)
- Intelligence (Grafana, NewRelic, Splunk)
- Workflow (Jira, ServiceNow, Rally)
- Change or Build management (Jenkins, Puppet, TFS)

The approach, backed by and mediated through the supporting tools, provides responders and on-call teams the ways and means to remediate Incidents.

Concepts in Action

For a visual representation, if you map the **"Five Phases"** of the Incident Management framework to monitored/monitoring systems, you can create a picture like this:



For purposes of illustration, imagine there is one monitored system (test-webhost), one monitoring system (Nagios), and one Incident Management system (VictorOps). Nagios, or any monitoring system, enables some built-in protection against false positives. Anyone facing on-call is rightly concerned about interruptions that "aren't real," or otherwise are triggered from transient failures instead of real Incidents. This is why a properly configured monitoring system is important.

In this example, Nagios is configured to "check http" on test-webhost. While a review of the Nagios documentation to support this task can be overwhelming, the salient pieces are here:

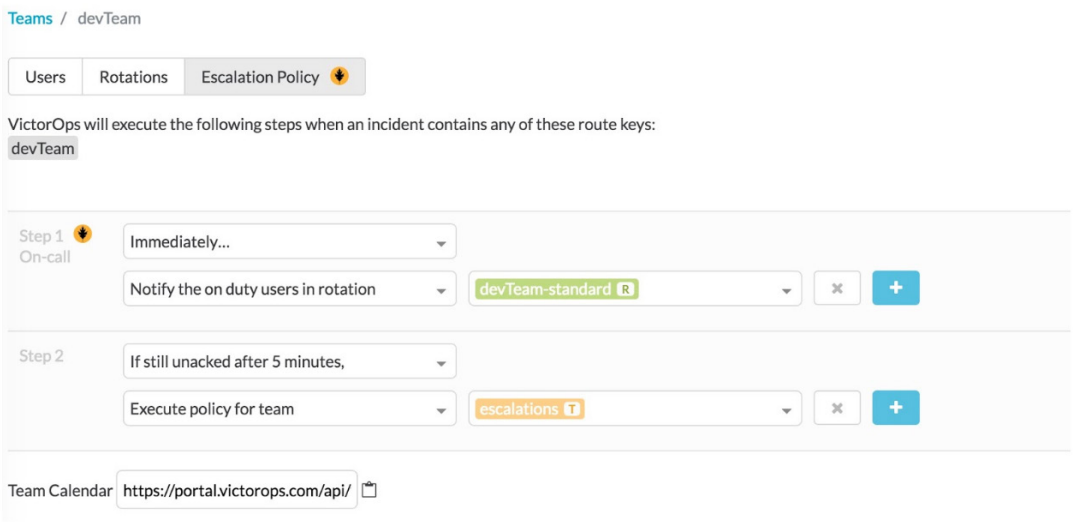
```
check_interval          5
max_check_attempts      3
retry_interval          1
```

This provides some false positive protection: Nagios will check this service every 5 minutes (check_interval), upon a failed check, Nagios will retry again in 1 minute (retry_interval). Nagios will not send a notification of failure until it registers 3 consecutive failures (max_check_attempts).

Meanwhile, the Incident Management system is configured with two teams: DevTeam, and Escalations. This Nagios integration is configured to properly route alerts for that test-webhost to the DevTeam.

Additionally, the DevTeam is configured for escalations. Escalations can work in a variety of ways, but typically the idea is "if no action takes place in X minutes, automatically notify someone else." Escalations are the safety net to any Incident Management system. Even the most responsive team member will occasionally misplace a phone, silence a ringer, or sleep a little too soundly. Escalations provide a built-in defense against such unplanned events.

"Escalations are the safety net to any Incident Management system."



Beep Beep Beep

To test this workflow, I've triggered an alert by disabling http on the test-webhost. Here you can see Nagios in the check/retry_interval logic as it has marked the service "CRITICAL", but has not triggered an Incident. The 6th column lists check ⅓ has failed:

test-webhost	http_check	CRITICAL	01-26-2017 23:34:39	0d 0h 0m 10s	1/3	← Connection refused

Once the alert passes that max_check_attempts (3), Nagios forwards the alert to the Incident Management system, where the on-call team member is notified. If I ignore the alert, or am unable to acknowledge the alert, the escalation kicks in. First, the timeline indicates I've been contacted:

NOTIFY: Trying to contact matthewboeckman for #10, sending EMAIL
Jan 26, 2017 17:02:10 MST

INCIDENT: #10 was OPENED for SERVICE (http_check) on HOST (test-webhost)
Jan 26, 2017 17:02:10 MST

Jan 26, 2017 17:02:09 MST

More Info

NAGIOS	PROBLEM
TIME	Jan 26, 2017 17:02:09 MST
SERVICE	http_check / test-webhost
STATE	CRITICAL
OUTPUT	Connection refused
HOST	test-webhost
STATE	UP
OUTPUT	PING OK - Packet loss = 0%, RTA = 11.83 ms

Then we see the escalations kick in 5 minutes later, preventing unacknowledged or unresolved Incidents from being missed.

NOTIFY: Trying to contact ginatasmanto for #10, sending EMAIL
Jan 26, 2017 17:07:11 MST

NOTIFY: Trying to contact flerghen for #10, sending EMAIL
Jan 26, 2017 17:07:10 MST

NOTIFY: Trying to contact matthewboeckman for #10, sending EMAIL
Jan 26, 2017 17:07:10 MST

1

For an on-call team, ACK(nowledge) is shorthand for "I've got this". An ACK by no means implies a fix, or a resolution of the Incident. At last, I ACK this alert. Note how the ACK cancels further escalations.

NOTIFY: Cancelled paging ginatasmanto Jan 26, 2017 17:11:03 MST

NOTIFY: Cancelled paging flerghen Jan 26, 2017 17:11:03 MST

NOTIFY: Cancelled paging matthrewboeckman Jan 26, 2017 17:11:03 MST

INCIDENT: #10 was ACKED for SERVICE (http_check) on HOST (test-webhost) by @matthewboeckman
Jan 26, 2017 17:11:02 MST

Jan 26, 2017 17:02:09 MST

More Info

NAGIOS	ACKNOWLEDGEMENT
TIME	Jan 26, 2017 17:11:09 MST
SERVICE	http_check / test-webhost
STATE	CRITICAL
OUTPUT	Connection refused
HOST	test-webhost
STATE	UP
OUTPUT	PING OK - Packet loss = 0%, RTA = 11.83 ms

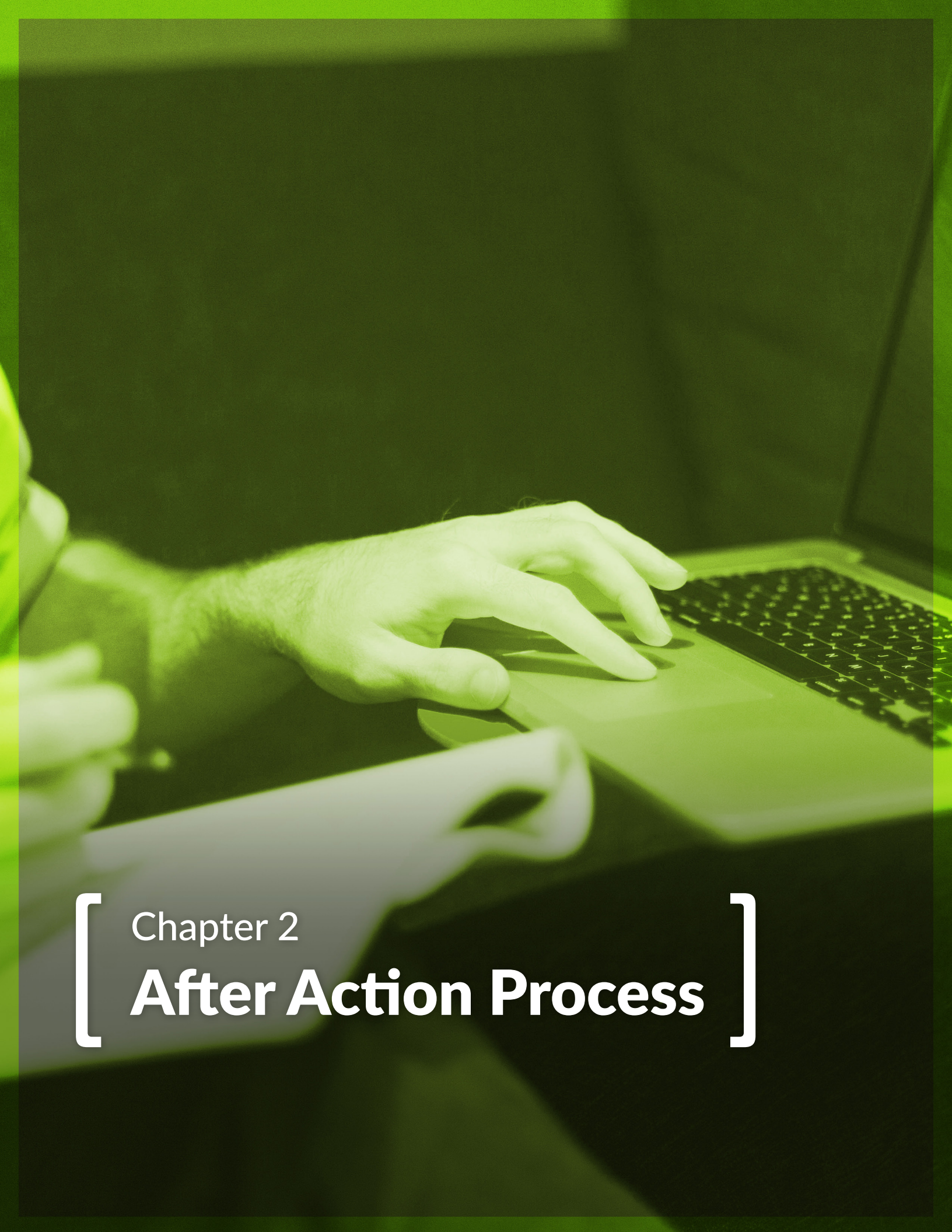
An ACK simply communicates to the team, and the Incident Management system, that the alerted person is responsive and on the case. In many implementations, the ACK is the final word on escalations. However, it is common to see an ACK silence further escalations, but if an Incident remains unresolved for a longer time period, then a separate set of escalations can kick in.

1

Building Trust

This example provides some basic transparency to the ways an Incident may be triggered, and how that Incident may cause your phone to make annoying noises at 3am. For anyone joining their first on-call rotation, having faith in the systems behind the alerts is important. Engineers, as a rule, don't do much on faith. As such, it is encouraged that the Operations team, or whoever maintains the alerting systems, is an active partner with the entire on-call team. Provide clear explanations of how alerting works, what conditions trigger escalations, and set clear expectations around responsiveness (e.g. Service Level Objectives).

By discussing these concepts within your team, you can help responders understand the how and why of what triggers alerts. Keeping these concepts and terminology in mind for postmortems will ensure everyone understands the discussion. Over time, teams arrive at a place of great trust that the systems are monitoring the right things, the right ways, and alerting for the right reasons.



Chapter 2

After Action Process

Chapter 2

After Action Process

2

In the preceding chapter, we discussed a practical example representing parts of the first three phases of our "Five Phases" approach. Let us turn then, to the remaining two: Analysis and Readiness.

The practice of a Postmortem (or Retrospective) is a fundamental component of the analysis phase of modern Incident Management. While it can be structured in many different ways, at it's highest level a postmortem is a detailed review of an Incident that clearly identifies what caused it and how it was corrected. Postmortems build on these reviews to create a more full understanding of the system as a whole. A postmortem review should be conducted blamelessly, and in service of an objective review of systems, processes, and teams following an Incident.

"The Practice of a Postmortem is a fundamental component of the analysis phase of modern Incident Management."

Teams often have a standing time set aside to review Incidents that were small in nature or impact. As needed, a postmortem can (and should) be called to review significant events, or Incidents that may portend hard times ahead. The output of a postmortem then becomes the leading input for the Readiness phase.

Iteration is key in all software or operations endeavors, and none more so than Incident Management. Following a successful postmortem, action becomes expected in Readiness. Adjusting system or infrastructure configuration is a common action item: adding space to a disk volume, increasing thread count on a pool, or tuning a cache. Similarly, teams may spend time adjusting thresholds, policies, or checks in their monitoring systems. Updating runbooks, documentation, and procedures is also a highly fruitful endeavor and one of the many ways teams can improve not only their systems, but their practice as well.

Chapter 3

Actionable Alerts

Chapter 3

Actionable Alerts

3

Incident response teams are frequently hamstrung by an unproductive cycle of alerting against arbitrary thresholds, instead of alerting on actionable failures. A classic example of this is the Linux Load metric - most default settings in Alerting systems will trigger an alert at an arbitrary value of 3. What does 3 signify? Doesn't matter. Wake someone up.

Given the business impact of failure in critical systems, it is no wonder that many organizations err on the side of over-alerting. This translates into a fundamental fear shared by all on-call teams: #FOMO or the Fear of Missing Out. #FOMO frequently sets teams up for alert-fatigue, and general dissatisfaction with the on-call process.

Unpacking the problem, we can see that an alert's actionability has to exist on two dimensions. First, the alert must require an action, differentiated from something that's merely informational. Second, the alert must route to someone who has the access, permission, and skills to be able to adequately perform said action.

If either of those things are untrue, you're going to have a bad time.

One of the most common examples of this being done poorly are setups where the Operations team receives all Incidents, and is asked to "just escalate" when the source of the alert is outside their domain. CPU load is high on a host, the offending process is the application itself, and Operations needs Developers to triage and repair. Why did Operations get involved at all?

A different example illustrating common frustration running the other way: Development team receives a page indicating application health checks are failing. Operations has restricted Developers from tools or systems necessary to action the alert (root, shell access, access to Splunk, etc). Great! Guess we'll wake an admin to fix the problem.

Actionable Change

Depending on where your team specifically falls on this prickly issue, there are some simple and easy to implement things you can do to get better.

1. **First and foremost –** ensure you have some implementation of "teams" configured in your Monitoring or Incident Management systems. Teams should align with systems either by virtue of functional alignment (Java Developers on-call for java application) or by responsibility (teams on the Registration team own registration and identity services).

2. **Second,** revisit default configurations in Monitoring systems. Either as part of scheduled postmortem time, or a separate project, analyze current thresholds for alerts. If you can't answer the question "what should someone do about this", change the threshold or remove the alert entirely.

3. **Lastly,** a compromise can be struck by assigning alerts of uncertain status to a severity level of "Informational". In most platforms, this creates an alert, but does not trigger a primary communication method. Stated differently - you get an email, but not a phone call.

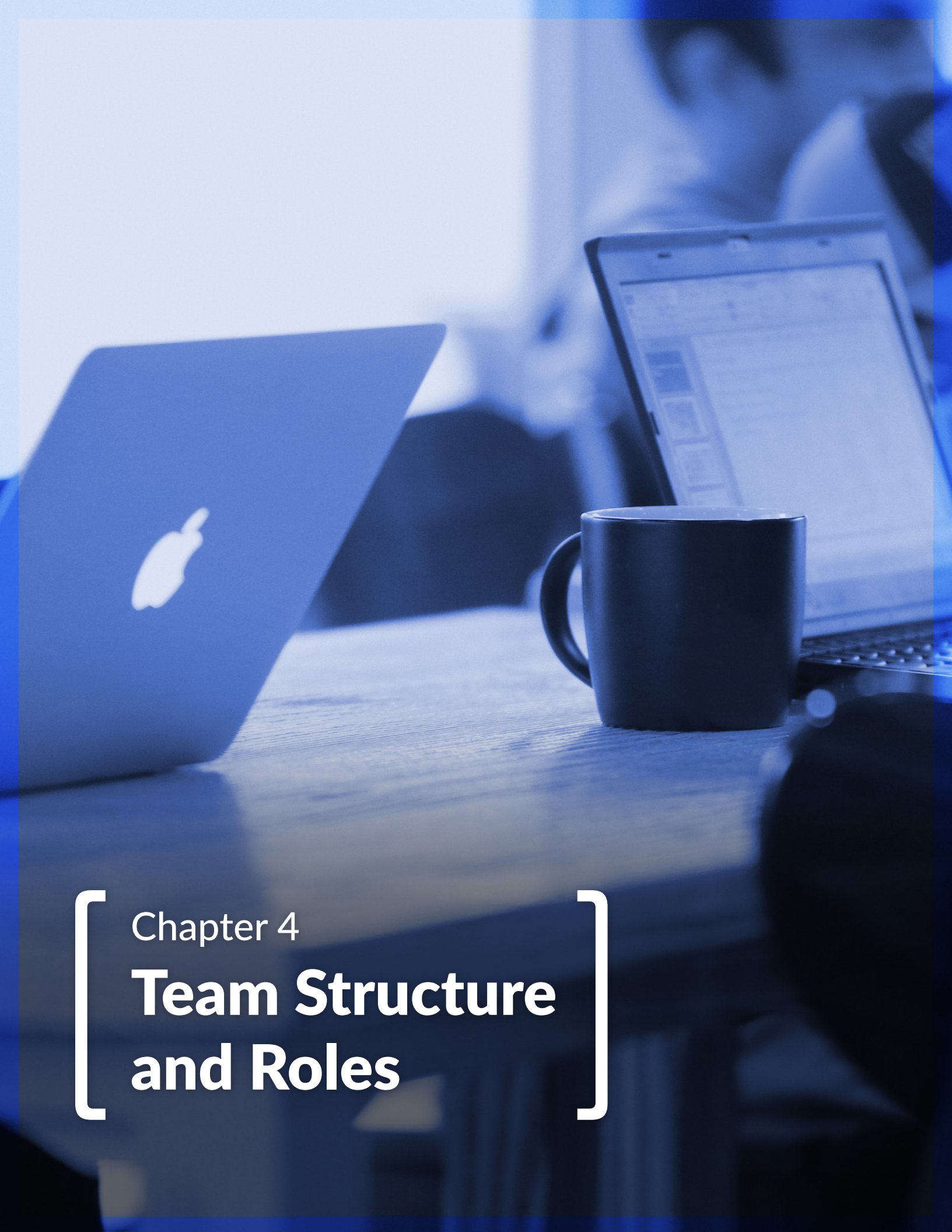
This approach is a fantastic way to evaluate thresholds and explore alerts without creating fatigue.

Alert Evolution

If you accept the myriad of things in-play, you must also accept that there is no single, static answer to the problem. Similarly, you have to recognize that where your team nests is only a point-in-time. Every Postmortem should include some discussion of the alerting and routing associated with the Incident, keeping the conversation focused on the Detection and Response phases:

- Was it actionable? Or just informational?
- Did it get to the right team?
- Did they have tools, access, and skills necessary to action a remedy?

Inevitably that answer will occasionally come up “no”, and that’s where you, the on-call team, must embrace personal action to improve the system. Continuous improvement only happens when teams communicate clearly, and devote iterations to tuning and refining these systems.



Chapter 4

Team Structure and Roles

Chapter 4

Team Structure and Roles

Organizing an on-call roster is tough. The problem of aligning skills, experience, and availability with specific application technologies is magnified by the dynamic nature of teams today. In most cases you settle for "close enough" and hope smart people make good decisions. Skills and scheduling are really only the beginning; effective Incident Management requires focus on how your on-call team operates.

4

> **"How we treat each other matters."**

Think of team interactions on two dimensions:

First, there is the structural organization of the team–people playing roles, workflows, and escalation paths. This is important to on-call teams because it impacts the procedural behavior of Incident Management.

Second, there are the cultural aspects. This dynamic is important to on-call teams because we're all likely to interact in high pressure moments in the middle of the night.

Team Mechanics

Many responses in Incident Management can be managed down to simple runbooks or automated completely. These simple, low diagnostic threshold Incidents are the bread and butter of an on-call team. Nearly anyone skilled enough to be in rotation can meaningfully respond to the Incident. Java OOM? Restart the app.

Sometimes, the cause of an alert or the necessary diagnosis exceeds a specific responder's abilities. While (hopefully) less frequent, these kinds of alerts tend to indicate something is wrong in a larger sense. Whether this is just a tricky problem, or Rome is actually burning, the first responder has to know how to get help.

4

"Every moment the response team spends notifying others, or escalating, is a moment of effective diagnostic or remediation wasted."

It's important to provide that escalation path in the simplest, easiest method possible. At 3am, while a database implodes, you don't want a Unix admin breaking out a call-down tree. Every moment the response team spends notifying others, or escalating, is a moment of effective diagnostic or remediation wasted. How you setup your on-call team for this moment will turn the tide.

An Embarrassment of Helpers

We can all imagine the panic suffered by a first responder when they call for help and no one comes. Paradoxically, it's almost worse when the first responder calls for help, and everyone shows up. Without direction, a dozen engineers trying to crowd solve a problem is a recipe for disaster. This is where two new escalation roles begin to show value: an **Incident Commander** (or Quarterback), and an **Incident Communicator** (or Scribe).

The Quarterback

4

The Quarterback is responsible primarily for directing and de-duplicating the efforts of the team. They provide ad-hoc organization to the swarm, and ensure that some of the team is focused on resolution of the proximate cause, and some of the team focuses on remediation or mitigation efforts if a timely resolution is not likely. The Quarterback also plays the role of risk advisor and assessor. If a fix is risky, they're making the call.

The Scribe

The Scribe manages inbound and outbound communication with others in the engineering team, or others in the business. A Scribe can also manage internal documentation of the event itself–capturing investigatory threads, who's looked at what, and changes made "in the heat of battle" in a simple, easy-to-digest document. As new team members join the response party, scrolling back the last 200 lines of chat is not practical, but **having a running synopsis of what's going on is.**

A (Non) Hypothetical Example

To further illustrate the idea, let's take an Incident pattern and walk through the firefight.

04:07 - 04:15

Series of alerts are triggered indicating slow response times on application servers. Alerts routed to an individual on the platform on-call team.

04:09 - 04:23

Series of alerts are triggered indicating high CPU and Memory utilization on application servers, high CPU and high Disk I/O on a database system. Alerts routed to an individual on the Operations on-call team.

4

Question

What's going on? Clearly something bad.

04:30 - 04:35

Both teams declare an emergency and escalate to the Quarterback and Scribe on-call. They join the chat and begin getting up to speed.

Question

What (if any) queries are responsible for the high DB load? Is the DB healthy? What (if any) application calls are churning through that CPU? Are the App servers healthy? Any recent changes going out? At a high level, is traffic to the application within expected norms for the time period?

04:35 - 04:45

Scribe begins manual escalation and notification of SME's or others deemed necessary to help. Scribe begins the running summary document, or otherwise documenting the Incident. Public facing status pages are updated. They notify Customer Support, and key business stakeholders of the event, and invite them to monitor the summary document. Backend team isolates high CPU calls to a few specific operations, each hammering the DB, and all triggered from a few common URL patterns. Database team confirms the database is generally healthy, but high volume of expensive queries coming from aforementioned application operations. Quarterback directs new level of inquiry focused on the application; while directing the database team to investigate reducing query execution time. Quarterback requests additional escalations to network team to investigate potential traffic rerouting, Scribe works through getting them up to speed.

Question

Where is the traffic coming from? Why is it triggering this condition now? Did new code go out recently? Any system changes? Any security concerns with the traffic?

I won't continue the tedious play-by-play in this example, you get the idea. Lots of people, running in different directions, quickly. Multiple independent investigative threads are going, avenues explored and abandoned, individuals are focused and refocused against new questions. All of which must be captured to prevent duplicative effort, and provide a clear narrative for future postmortem analysis and new team members joining the fight.

As this example plays out, each team is focused on their functional area of expertise and responsibility. Transparency to stakeholders is maintained without distracting the responders, and a strong narrative is built enabling handoffs, efficient triage, and an eventual postmortem analysis.

Training Opportunities

While a Quarterback is almost always a senior member of the team, the Scribe role is a perfect place to introduce junior or new team members to the on-call process. Scribes can come from any background, and get an opportunity to observe the diagnostic and remediation work without bearing the technical burden personally. Lastly, a Scribe is an active, meaningful participant, who will both add more value in, and extract more out of, than passive observers or ride-a-longs.

Blameless Escalations

Formal team structure is critical to the success of your firefighting efforts. It sets the team up for success by enabling responders to focus directly on the problem. How we all behave in the moment, however, is far more impactful to the outcome of a specific Incident, and a team's long-term success.

Individuals must be encouraged to escalate early and often.

While much has been said about the importance of keeping after-action analysis blameless, it is doubly important to keep escalations blameless. A lone wolf toiling away makes for a great graphic novel hero, but rarely leads to effective resolution of Incidents in complex systems.

4

> **"They're concerned about how they'll be viewed if they "need help" too often."**

This basic tenet is often missing in on-call team culture. Individuals want others to respect their abilities. They're concerned about how they'll be viewed if they "need help" too often. Previous acrimony creates a reluctance to engage other teams. Managers frequently highlight the work of "Heroes", establishing an anti-pattern and "Hero" culture within the team.

This theme of blameless escalation becomes doubly meaningful to teams expanding on-call to non-traditional teams. Developers, who are unfamiliar with organizational or industry norms, will be pinned between a reluctance to get help and a fear that they lack the skills to adequately respond by themselves. Encouraging blameless escalations enables everyone in a rotation to get help when they need it.

Summary

Distilling the myriad variables and experience of the massively complex topic of Incident Management is no easy task. In any organization the ways and means of Detecting an Incident, Responding, and Remediating, and the formal existence of Analysis and Readiness, will vary wildly. Any established technology team has some approach to Incident Management, whether a formal adoption of ITSM, or a hoary confusion of unspoken agreements, people have a sense of their duty.

While building and/or joining these teams may seem daunting, and personally inconvenient, the benefits are significant. Deep dives in metrics, diagnostic technique, and a deeper understanding of complex behavior are deeply satisfying pursuits (and benefit career growth). For organizations, the benefits of a DevOps or SRE culture are manifest.

Teams with an effective Incident Management focus produce resilient applications, scalable infrastructure, and work better together. Building and maintaining a focus on your Incident Management practice will empower you to be victorious.