**SRE Operating Model: Key Scenarios**

**Scenario 1: When an Alert is Received (e.g. high latency or service outage)**

1. **Alert Notification & Acknowledgment:** The monitoring system pages the **On-Call Engineer** (via PagerDuty/Opsgenie, etc.) about an issue (high latency, error rate spike, etc.). The on-call immediately **acknowledges** the alert to take ownership​

[tech.trivago.com](https://tech.trivago.com/post/2022-07-18-sre-on-call-procedure-at-trivago#:~:text=,incident%20is%20not%20left%20unattended)

. This stops further automated escalation and signals that someone is investigating. If the on-call doesn’t respond in time, the alert will escalate to the next engineer or manager by policy​

[tech.trivago.com](https://tech.trivago.com/post/2022-07-18-sre-on-call-procedure-at-trivago#:~:text=we%20do%20when%20we%20receive,incident%20is%20not%20left%20unattended)

.

1. **Initial Triage by On-Call:** The On-Call Engineer **investigates the alert details** – reading the error message, checking linked dashboards and the runbook for the service​

[tech.trivago.com](https://tech.trivago.com/post/2022-07-18-sre-on-call-procedure-at-trivago#:~:text=,%E2%80%9Ccrashloopbackoff%E2%80%9D%20in%20a%20kubernetes%20cluster)

. They assess the situation: confirm if it’s a real incident (not a false alarm) and identify the system components involved. The on-call checks service health metrics (latency, traffic, errors) on monitoring dashboards to **verify the impact** and scope of the issue​

[tech.trivago.com](https://tech.trivago.com/post/2022-07-18-sre-on-call-procedure-at-trivago#:~:text=,business%20need%20to%20be%20communicated)

. For example, is one instance down or is the whole service affected? They also look for any recent changes or related alerts that could hint at the cause (e.g. upstream dependency failures).

1. **Containment and Quick Fixes:** If the issue is straightforward (e.g. known bug or runbook entry), the on-call applies a **quick mitigation**. This might mean restarting a crashed service, rolling back a recent config change, or shifting traffic away from a bad instance. The goal is to **stop the bleeding** and restore service as quickly as possible​

[tech.trivago.com](https://tech.trivago.com/post/2022-07-18-sre-on-call-procedure-at-trivago#:~:text=,in%20charge%20of%20the%20microservice)

. If a temporary fix is used, the on-call notes it for follow-up. Throughout this process, the on-call is mindful of the service’s SLOs – if the incident threatens to violate error budget or uptime targets, it’s treated with highest urgency. In the best-case scenario, the on-call can resolve the incident on their own quickly. In reality, some incidents require a larger team or specialized expertise to resolve​

[atlassian.com](https://www.atlassian.com/incident-management/on-call/escalation-policies#:~:text=When%20an%20incident%20strikes%2C%20the,quickly%20and%20on%20their%20own)

.

1. **Involving the Embedded SRE (if needed):** For complex or unclear issues, the on-call **escalates to the Embedded SRE** or a secondary on-call. The embedded SRE is an engineer deeply familiar with the service (often part of the product team) who can bring additional insight. They jump in to help diagnose deeper problems – analyzing logs, searching for subtle root causes, or reviewing recent code changes. The embedded SRE works with the on-call to form hypotheses (e.g. “is this a database latency issue?”) and test solutions. They might also coordinate with the **development team** if code changes or deeper fixes are needed, since embedded SREs act as a bridge to the dev team​

[rootly.com](https://rootly.com/blog/the-pros-and-cons-of-embedded-sres#:~:text=One%20major%20risk%20is%20that,about%20how%20it%20should%20operate)

. This collaborative response ensures that if the first responder can’t fix it alone, the right experts are engaged quickly.

1. **Communication and Incident Management:** Clear communication is critical during the incident. The on-call (often acting as **Incident Commander**) provides regular updates to stakeholders. For example, they might post in the incident Slack channel or email an update: “Investigating high latency in Service X, mitigation in progress.”​

[tech.trivago.com](https://tech.trivago.com/post/2022-07-18-sre-on-call-procedure-at-trivago#:~:text=when%20there%20is%20an%20incident,of%20an%20even%20bigger%20issue)

This keeps everyone aligned and avoids duplicate efforts. If the impact is user-facing or spans teams, the on-call notifies the **Consumer Teams** (any downstream teams or major customers affected) about the issue and the ongoing efforts. The Embedded SRE may take a role of **Communication Lead** if the situation escalates, ensuring status updates are timely. Internally, the on-call and SRE document what’s happening in an incident ticket or log. Effective communication also means setting expectations – e.g. acknowledging the incident within a few minutes and providing the next update ETA (even if just “under investigation”).

1. **Escalation Procedures:** If the incident is not resolved quickly or is clearly severe (e.g. a **SEV-1** outage), the on-call follows the defined **escalation policy**. This could mean paging a more senior engineer, calling in additional team members, or alerting management. For example, if an alert is unacknowledged or the impact is widespread, the incident management system will automatically page a secondary on-call or tech lead​

[tech.trivago.com](https://tech.trivago.com/post/2022-07-18-sre-on-call-procedure-at-trivago#:~:text=1,the%20Opsgenie%E2%80%99s%20Incident%20Command%20Center)

. In extreme cases, a “war room” is formed – pulling in engineers from various teams on a Zoom bridge or chat room to collaboratively resolve the incident​

[tech.trivago.com](https://tech.trivago.com/post/2022-07-18-sre-on-call-procedure-at-trivago#:~:text=1,the%20Opsgenie%E2%80%99s%20Incident%20Command%20Center)

. The Embedded SRE helps coordinate this, since they know which specialists (database, network, etc.) might need to join. The goal of escalation is to bring in whatever resources or expertise are necessary to restore service **as quickly as possible** when the front-line on-call cannot resolve it alone​

[atlassian.com](https://www.atlassian.com/incident-management/on-call/escalation-policies#:~:text=Of%20course%2C%20in%20the%20real,and%20policy%20for%20incident%20escalation)

.

1. **Resolution and Recovery:** The on-call and any others involved work until the issue is mitigated and the service is back within normal parameters. Once metrics confirm the service is healthy (e.g. latency back to normal, error rate down), the on-call formally **resolves the incident**. They communicate the resolution to stakeholders: for example, posting “Issue resolved – root cause was identified as X and a fix has been applied. Monitoring closely.” The Consumer Teams are informed that the service is healthy again so they can verify on their end or inform end-users if needed. The on-call updates the incident ticket with a summary of the resolution and any immediate remediation done.
2. **Post-Incident Follow-up:** After an alert-driven incident, there are follow-up actions. The **Embedded SRE** often leads the post-incident review. The on-call and SRE will gather to debrief: What was the root cause? Was our monitoring adequate? Did we meet our response SLOs? If the issue was temporarily fixed, they ensure a ticket is filed for the **service owners** (developers) to implement a permanent fix. They might also propose improvements – e.g. tuning alert thresholds or adding a runbook entry if something was missing. For significant incidents (high severity or any user-visible downtime), the team will schedule a **blameless postmortem** (see Scenario 4) to learn from the incident​

[sre.google](https://sre.google/sre-book/being-on-call/#:~:text=Finally%2C%20when%20an%20incident%20occurs%2C,Loo10)

. Any action items (like code fixes, capacity increases, etc.) are tracked to completion so that the same alert won’t happen again. The outcome: the team not only resolved the immediate alert, but also communicated and set up long-term fixes, embodying a full incident response cycle.

**Scenario 2: When a User Reports an Issue (e.g. via ticket or email)**

1. **Ticket Intake and Triage:** A user or **consumer team** submits a report of a problem – for example, a support ticket in ServiceNow or an email saying “Feature Y is returning errors for us.” This report is treated as a potential incident. The on-call SRE (or support engineer) **acknowledges the ticket** and ensures it’s logged in the incident tracking system​

[faun.dev](https://faun.dev/c/stories/squadcast/sre-incident-management-a-guide-to-effective-response-and-recovery/#:~:text=1,to%20pinpoint%20the%20root%20cause)

. They gather initial details: Who is affected (one user or many)? What is the error or symptom? When did it start? The issue is given an initial severity based on user impact (critical if it’s a major customer outage, lower if minor or a single user).

1. **Verification of the Problem:** The **On-Call Engineer** investigates to confirm the user’s report. They check the system’s metrics and logs around the time of the report to see if anything abnormal is showing. For example, if a user reports a slow API response at 10:00 AM, the on-call looks at latency graphs at 10:00 AM for any spikes. They also attempt to **reproduce the issue** if possible – e.g., calling the API endpoint themselves or asking the user for exact steps. This helps distinguish a service issue from a client-side or usage issue. If multiple independent users or teams have reported the same symptom, it’s a strong signal of a service problem. All these findings are noted in the ticket.
2. **Investigation and Diagnosis:** If the issue appears legitimate, the on-call treats it as an incident to diagnose. They use observability tools (logging, distributed tracing, etc.) to dig into the problem​

[faun.dev](https://faun.dev/c/stories/squadcast/sre-incident-management-a-guide-to-effective-response-and-recovery/#:~:text=process%2C%20ensuring%20the%20appropriate%20responders,restored%2C%20the%20incident%20is%20closed)

. For instance, they might find error traces in the logs corresponding to the user’s request ID or see elevated error rates for that function. The **Embedded SRE** can assist here, especially if the problem is complex – they have deeper knowledge of the system internals and recent changes. Together, they analyze possible causes: Is there a bug introduced in the latest release? A misconfiguration for that user’s account? They may also check if this ties into any earlier alerts or known issues (perhaps this user report reveals an ongoing problem that wasn’t caught by automated alerts).

1. **Communication with the Reporter:** While investigating, the team keeps the **user or reporting team** updated. A quick acknowledgment is sent like, “We’ve received your issue and are looking into it.” As they uncover information, they might ask the user for additional details or inform them of progress (“We see an error in module X and are working on a fix”). This ongoing communication is crucial for user-reported incidents – it manages expectations and shows the issue is being addressed. If the issue is impacting a consumer team’s work, the SRE might provide a workaround if one is known, to unblock them temporarily. Keeping a user-centric mindset, as Google’s incident guide notes, is key – the ultimate goal is restoring the user’s experience​

[faun.dev](https://faun.dev/c/stories/squadcast/sre-incident-management-a-guide-to-effective-response-and-recovery/#:~:text=based%20on%20their%20investigation%2C%20continuously,the%20root%20cause%20and%20not)

.

1. **Resolving the Issue:** Depending on the findings, the on-call/embedded SRE takes action to **fix the problem** or involve those who can. If it’s a minor configuration tweak or a known bug, the on-call might solve it on the spot (for example, by redeploying a previous stable version or adjusting a setting). For more complex bugs, the SRE will escalate to the **development team**: for instance, paging the developer on-call or the relevant module owner. They provide all the info gathered (logs, error messages, user impact) so the developers can rapidly pinpoint the flaw. The team then implements a fix. This could be a hotfix deployed to production or a manual correction (like correcting a bad database entry) depending on the issue. Throughout, the SRE ensures the **user’s issue is prioritized** appropriately given its impact.
2. **Validation and Closure:** Once a fix or mitigation is in place, the team validates that the user’s problem is actually resolved. The on-call checks monitoring to confirm errors have subsided or the service behavior has returned to normal. Most importantly, they **confirm with the user** that the issue is fixed – for example, asking the reporting team to retry the operation that was failing. When the user affirms that things are working again, the ticket can be marked resolved​

[faun.dev](https://faun.dev/c/stories/squadcast/sre-incident-management-a-guide-to-effective-response-and-recovery/#:~:text=based%20on%20their%20investigation%2C%20continuously,the%20root%20cause%20and%20not)

. The on-call documents the resolution in the ticket notes (what was found and how it was fixed). If the fix was temporary, a follow-up task is created for a permanent solution. Finally, the SRE communicates a summary to any other stakeholders (for instance, “Customer X’s issue has been resolved. Root cause was a config error, which has been corrected”). After verification and documentation, the user-reported incident is closed.

1. **Follow-up and Prevention:** Even though a user-reported issue might be small, the SRE team treats it as a learning opportunity. They ask: Could we have detected this automatically? If a user found a bug that monitoring didn’t catch, the **Embedded SRE** might add a new alert or improve dashboards so that future occurrences are caught proactively. If the issue was due to a code defect, it gets added to the development backlog (with the SRE pushing for a timely fix or patch if it’s critical). For significant incidents triggered by user reports (for example, a user finds data corruption), a full **postmortem** may be warranted (blamelessly analyzing why it happened and how to prevent it). The SRE ensures any action items (e.g. “improve input validation to prevent this bug”) are assigned and tracked. In essence, the team not only resolves the individual user’s issue but also strengthens the system’s reliability going forward.

**Scenario 3: When a Change is Deployed (new release or config/infrastructure update)**

1. **Pre-Release Reliability Checks:** Before a change goes live, the **Embedded SRE** and development team perform thorough checks to safeguard reliability. This includes code reviews with an eye on stability, updating automated tests, and running performance tests. Often, an SRE will conduct a **production readiness review** for major launches – verifying that the new version meets reliability standards (e.g. has proper monitoring, can handle expected load, fails gracefully). Capacity and rollback plans are reviewed: is there enough headroom if the new release uses more resources? Is there a clear rollback procedure if things go wrong? In some organizations, a change advisory board or an SRE tech lead must sign off on high-risk deployments, ensuring all checks (unit, integration, canary plan, etc.) are in place. The goal is to catch potential issues *before* deployment and ensure the team is prepared to deal with any incident the change might cause.
2. **Deployment Planning:** The team uses **progressive delivery** strategies to limit risk. For example, they might plan a canary release (deploying the update to a small percentage of servers or users first) or a phased rollout (region by region). The Embedded SRE often helps design these rollout plans so that any bad effects can be detected early. They also ensure monitoring is aligned with the deployment: specific metrics or logs are watched more closely during and after the change. Communication is part of planning too – the SRE/Dev team announces the deployment schedule to stakeholders (so on-calls are aware a change is happening). If the change is customer-facing, a heads-up might be given to support teams. In essence, the team treats a deployment as a **planned event that could trigger incidents**, and prepares accordingly​

[googblogs.com](https://www.googblogs.com/reliable-releases-and-rollbacks-cre-life-lessons/#:~:text=The%20release%20of%20a%20new,translates%20to%20three%20basic%20tasks)

(assuming every release may contain bugs).

1. **Monitoring During and After Deployment:** As the change is rolled out, the **On-Call Engineer** (or a designated deployment engineer) closely **monitors system metrics** in real-time. The team looks for any sign of trouble: error rates rising, latency slowing, traffic drops, etc. According to SRE best practices, one of the key tasks is *detecting if a new release is broken* as soon as possible​

[googblogs.com](https://www.googblogs.com/reliable-releases-and-rollbacks-cre-life-lessons/#:~:text=The%20release%20of%20a%20new,translates%20to%20three%20basic%20tasks)

. If they’re doing a canary, the on-call compares the metrics of the canary instances vs. normal ones. If any abnormality is observed, the deployment can be paused. The monitoring plan often includes automated alerts on key SLOs specifically tuned for the deployment window (for example, alert on even a small increase in error rate during rollout). By watching the system like a hawk in the minutes and hours after a release, the team can catch issues before they affect many users. In many SRE teams, the on-call will not consider the change “fully deployed” until a certain bake time has passed with no critical alerts.

1. **Issue Detected – Rollback/Mitigation:** If the deployment causes an incident (for example, users start getting errors or the service becomes unavailable), the team **responds quickly**. The first step is often to **rollback** the change to the last known good state​

[opensource.com](https://opensource.com/article/22/6/change-management-site-reliability-engineers#:~:text=If%20things%20don%27t%20go%20well,solid%20rollback%20mechanism%20in%20place)

. Fast rollback is a cornerstone of SRE change management – “rollback fast and often” if things don’t look right​

[opensource.com](https://opensource.com/article/22/6/change-management-site-reliability-engineers#:~:text=If%20things%20don%27t%20go%20well,solid%20rollback%20mechanism%20in%20place)

. The on-call might initiate an automated rollback script (or manually redeploy the previous stable version). Feature flagging is also used: if the release introduced a new feature behind a flag, the flag can be turned off immediately to mitigate the impact​

[opensource.com](https://opensource.com/article/22/6/change-management-site-reliability-engineers#:~:text=Mechanics%20of%20rollback)

. The Embedded SRE coordinates with devs to decide if rollback is the best course or if a quick fix forward is feasible. (In rare cases, if the bug is trivial and a one-line fix can be pushed in minutes, they might do a **hot-fix forward** instead of a rollback – but only if confident.) Users are often routed away from the bad version during rollback – e.g., using load balancers to drain traffic from the new release. The priority is to **restore service** stability. Thanks to prior planning, this rollback is usually smooth: teams that embrace SRE principles ensure rollback mechanisms are tested and bulletproof​

[opensource.com](https://opensource.com/article/22/6/change-management-site-reliability-engineers#:~:text=Rollback%20fast%20and%20rollback%20often,will%20become%20bulletproof%20over%20time)

​

[opensource.com](https://opensource.com/article/22/6/change-management-site-reliability-engineers#:~:text=The%20entire%20rollback%20could%20be,labels%20that%20are%20clearly%20documented)

.

1. **Communication During Deployment Incidents:** If a deployment is halted or rolled back due to problems, the team treats it like an incident. The on-call/Embedded SRE sends out communications: “Release X version Y has been rolled back due to elevated error rates. Investigating root cause.” This informs other stakeholders (product managers, support, other teams) that the new features might be delayed and that a problem occurred. Often a status page or internal announcement is made if the deployment issue impacted users. Transparency here is important – even though this was a planned change, it became an unplanned incident and is addressed as such.
2. **Post-Deployment Verification:** Assuming the deployment (or its rollback) completes, the team verifies that the system is healthy and all services are running the expected versions. The On-Call watches the system for some extended period after (say, an hour or two) to be confident no slow-burning issues remain. They might run synthetic tests or ask consumer teams to verify everything works with the new version. Only after this verification does the team declare the change deployment a success.
3. **Analysis and Follow-Up:** Any hiccups or incidents caused by the change trigger a follow-up. The **Embedded SRE** and developers will investigate what went wrong with the deployment. For example, if a rollback was needed, they perform a root cause analysis: was there a bug that slipped through tests? Did we miss a dependency in our checks? All this goes into a short incident report or a full postmortem if the incident was severe. Action items are created to prevent similar issues: add test cases for that bug, improve the canary analysis, refine the **release checklist**, etc.

[googblogs.com](https://www.googblogs.com/reliable-releases-and-rollbacks-cre-life-lessons/#:~:text=The%20release%20of%20a%20new,translates%20to%20three%20basic%20tasks)

. SREs emphasize learning from these events to make future releases safer. Additionally, they evaluate the **deployment process** itself: Could monitoring be improved to detect issues sooner? Did the team follow the planned canary procedure, and did it work as expected? Over time, this leads to more mature deployment practices (like automated canary analysis, better feature flagging, etc.). In summary, when a change is deployed, SREs ensure rigorous pre-checks, **closely monitor and quickly mitigate any issues**, and iterate on the process so that deployments can be fast **and** safe​

[opensource.com](https://opensource.com/article/22/6/change-management-site-reliability-engineers#:~:text=1,in%20your%20SRE%20organization%20functionality)

(progressive rollout, strong monitoring, and fast rollback are key to reliable releases).

**Scenario 4: When an Incident Requires a Postmortem (major outage or repeated issues)**

1. **Triggering a Postmortem:** After a **major incident** – for example, a prolonged outage, a severe SLO violation, or a pattern of repeated latency issues – the team initiates a **blameless postmortem**. SRE culture treats postmortems as essential for learning from failures​

[sre.google](https://sre.google/sre-book/postmortem-culture/#:~:text=The%20postmortem%20concept%20is%20well,we%E2%80%99ve%20gained%20over%20the%20years)

. Criteria for requiring a postmortem might include user-visible downtime beyond a certain threshold, significant revenue impact, or any incident where manual intervention was needed due to gaps in automation​

[sre.google](https://sre.google/sre-book/postmortem-culture/#:~:text=%2A%20User,rollback%2C%20rerouting%20of%20traffic%2C%20etc)

. The **Embedded SRE** or team lead typically schedules a postmortem meeting once the incident is resolved and things are stable (often within a few days of the incident while it’s fresh). Relevant stakeholders attend – the on-call engineers involved, the embedded SRE, developers from the affected service, and sometimes product or support representatives if they were impacted.

1. **Blameless Discussion and Timeline:** In the postmortem meeting, the team **reconstructs the incident timeline** in detail: when did the issue start, when was it detected, what actions were taken, and how did it resolve. Everyone contributes facts – from monitoring data, alert timestamps, chat logs, etc. The key is that this discussion is **blameless** – the focus is on *what* happened and *why*, not *who* caused it​

[sre.google](https://sre.google/sre-book/being-on-call/#:~:text=Finally%2C%20when%20an%20incident%20occurs%2C,Loo10)

. For example, instead of saying “Person X rebooted the server and caused an outage,” the phrasing would be “a server reboot command was issued, which triggered an outage; we’ll explore why that command was run.” By avoiding blame, people feel safe to honestly discuss mistakes or system gaps, which leads to better insights. The postmortem document typically includes a narrative of the event, the impact, root causes, and resolutions​

[sre.google](https://sre.google/sre-book/postmortem-culture/#:~:text=The%20postmortem%20concept%20is%20well,we%E2%80%99ve%20gained%20over%20the%20years)

.

1. **Root Cause Analysis:** The team identifies the **primary root cause(s)** of the incident, as well as contributing factors. They ask the “5 Whys” – digging deeper beyond the surface cause. For example, if a deploy caused a crash, the analysis might uncover that a validation was missing in code and tests didn’t catch it, *and* the canary process failed to halt the bad deploy. All those factors are noted. The postmortem also highlights what went well – perhaps monitoring caught the issue early or the on-call’s quick action reduced impact. It’s important to recognize things that **prevented the incident from being worse**, so those practices continue. A good postmortem is a complete record: incident summary, impact on users (e.g. “5% of users experienced errors for 2 hours”), root causes, and how it was fixed​

[sre.google](https://sre.google/sre-book/postmortem-culture/#:~:text=The%20postmortem%20concept%20is%20well,we%E2%80%99ve%20gained%20over%20the%20years)

​

[tech.trivago.com](https://tech.trivago.com/post/2022-07-18-sre-on-call-procedure-at-trivago#:~:text=At%20least%2C%20one%20of%20the,steps%20to%20forestall%20future%20occurrences)

.

1. **Documenting Impact and Learnings:** Part of the postmortem is quantifying the impact and capturing key learnings. The team documents **how many users were affected, how much downtime occurred, and any other damage** (data loss, missed SLA, etc.)​

[tech.trivago.com](https://tech.trivago.com/post/2022-07-18-sre-on-call-procedure-at-trivago#:~:text=Postmortem%20are%20written%20with%20these,questions%20in%20mind)

. They also note how the incident was detected and whether monitoring was sufficient. If there was a delay in response or any missteps, it’s discussed openly as a learning point. For example, “We learned that our runbook was outdated, which slowed down mitigation.” The document may answer specific questions: *Did this incident violate an SLO?* *How did our emergency response process work?* *What did we learn about our system’s behavior?* This ensures the incident has a lasting record that others can reference. Many SRE teams share postmortems widely (internally) so that knowledge spreads and even teams not involved can learn from the incident​

[sre.google](https://sre.google/workbook/postmortem-culture/#:~:text=Our%20example%20postmortem%20was%20shared,tool%20in%20restoring%20shaken%20trust)

.

1. **Action Items and Ownership:** The most important outcome of a postmortem is a list of **concrete action items** to prevent recurrence or reduce impact next time​

[sre.google](https://sre.google/sre-book/postmortem-culture/#:~:text=The%20postmortem%20concept%20is%20well,we%E2%80%99ve%20gained%20over%20the%20years)

. These might include code changes (fixing the bug or adding a missing fail-safe), process changes (updating runbooks, improving monitoring alerts, training on a tool), or other improvements. Each action item is given an owner and a due date. For example: “Update the database backup script to handle network failures – **Owner:** John Doe (Database SRE), due in 2 weeks.” It’s crucial that every action item has a clear owner; without ownership, items are less likely to be completed​

[sre.google](https://sre.google/workbook/postmortem-culture/#:~:text=,less%20likely%20to%20be%20resolved)

. The Embedded SRE often takes responsibility for ensuring these get assigned – they might own some items themselves (especially if related to reliability tooling), while developers might own code fixes. The list is tracked in a task system (like Jira or a spreadsheet). **Blameless accountability** is enforced – owning an action doesn’t mean you caused the incident; it means you’re making sure the follow-up is done​

[newrelic.com](https://newrelic.com/blog/best-practices/incident-postmortems-in-sre-practices#:~:text=Implement%20follow,monitor%2C%20and%20update%20action%20items)

.

1. **Postmortem Review and Closure:** Once the postmortem document is written and action items defined, the team might do a quick peer review of it (sometimes another SRE team reviews to provide outside perspective). Then it’s officially published/stored in the knowledge base. The incident can be considered **closed** once the postmortem is complete and all immediate action items are logged. However, the process isn’t truly over until the action items are completed. SRE teams often have a practice of reviewing outstanding postmortem actions in their weekly meetings or tracking them in a dashboard​

[newrelic.com](https://newrelic.com/blog/best-practices/incident-postmortems-in-sre-practices#:~:text=Implement%20follow,monitor%2C%20and%20update%20action%20items)

. This ensures accountability doesn’t lapse over time. The Embedded SRE monitors progress on these tasks, nudging owners or escalating if necessary to ensure things get done. Some teams even tie completion of action items to objectives/OKRs. The final step might be a **postmortem follow-up meeting** a month or so later to check that all the preventive measures are in place. Through this rigorous postmortem process, the organization turns a painful incident into actionable improvements and shared knowledge, strengthening reliability culture (each failure is “education” rather than just pain)​

[sre.google](https://sre.google/sre-book/postmortem-culture/#:~:text=As%20SREs%2C%20we%20work%20with,an%20essential%20tool%20for%20SRE)

.

**Scenario 5: When Performance Degradation is Detected Proactively (early warning signs, no outage yet)**

1. **Detection of Early Warning Signs:** SREs continuously analyze system health metrics. If **proactive monitoring** shows a worrying trend – for instance, increasing response times, rising memory usage, or error rates creeping up slowly – the **On-Call Engineer** is alerted even though users haven’t complained yet. This might come via a low-severity alert or an SLO report indicating the service is close to burning through its error budget. For example, an alert may fire if latency exceeds a certain threshold for a sustained period, flagging a potential issue. The on-call treats this as an **early incident** signal and begins investigating before a full outage occurs. SRE best practices encourage such SLO-based alerts to catch “slow burn” issues​

[googblogs.com](https://www.googblogs.com/applying-the-escalation-policy-cre-life-lessons/#:~:text=Escalation%3A%20The%20SRE%20oncall%20is,releases%20for%20the%20affected%20service)

​

[googblogs.com](https://www.googblogs.com/applying-the-escalation-policy-cre-life-lessons/#:~:text=Escalation%3A%20SLO,work%20on%20the%20service%E2%80%99s%20reliability)

.

1. **Assessment and Triage:** The On-Call Engineer first verifies the **scope and impact** of the degradation. They check: Is the trend consistent across all instances or just one region? Are certain operations slower than others? The on-call reviews dashboards and logs to pinpoint what’s degrading. For example, they might notice that database query latency has been increasing over the past hour, or that CPU usage on servers is very high, approaching capacity. Since there’s no outage yet, this is the time to be thorough – the on-call can dig into detailed metrics (CPU, memory, I/O, application profiles) to hypothesize the root cause. They might also correlate this with any recent changes or a spike in traffic. Essentially, the on-call is gathering clues to understand **why** performance is slipping.
2. **Engaging Additional Help:** If the on-call suspects a complex cause (e.g., a memory leak or an algorithmic inefficiency), they loop in the **Embedded SRE** or relevant engineers **proactively**. Even though it’s not a full-blown incident, the embedded SRE’s expertise can be valuable to diagnose subtle performance issues. Together, they might run tests or simulations – for instance, using a load testing tool in a staging environment to reproduce the degradation. They also review historical data: Has this slow trend happened before? Is it correlated with a specific build or config? If needed, the SRE might convene a small troubleshooting session with a developer to inspect the code paths involved. This collaborative approach early can prevent a minor degradation from snowballing into a major incident.
3. **Proactive Mitigation:** Once the cause is hypothesized, the team takes steps to **mitigate the issue before it impacts users significantly**. There are a few strategies:
   * **Optimize or Fix**: If they identify an inefficiency (say a newly introduced slow database query), the dev/SRE can deploy an optimized query or code patch. In one Google SRE example, a slow-burning error rate increase was traced to a performance regression, and engineers quickly wrote some optimizations to address it​

[googblogs.com](https://www.googblogs.com/applying-the-escalation-policy-cre-life-lessons/#:~:text=Escalation%3A%20The%20SRE%20oncall%20is,releases%20for%20the%20affected%20service)

.

* + **Scale or Throttle**: If the issue is approaching capacity limits, the SRE might add more resources. For instance, increase the number of server instances or CPU cores to handle the load, buying time to fix the efficiency problem. Alternatively, they might throttle non-critical workloads (like batch jobs) to free up capacity for user-facing requests.
  + **Configuration Tuning**: Sometimes adjusting a setting can help – e.g., increasing a cache size if hit rates are dropping, or raising a connection pool limit if threads are getting starved. These tweaks can improve performance immediately while a long-term fix is developed.
  + **Feature Toggle**: If a new feature or heavy process is causing the slowdown, the team can disable it (toggle it off) until it’s fixed.

The on-call implements the chosen mitigation and then observes the metrics to see if the trend stabilizes or improves. The goal is to **prevent an outage** – for example, if response times were climbing toward a point that would trigger user-visible timeouts, bringing them down early avoids users ever noticing a problem.

1. **Stakeholder Communication:** Even though this is a behind-the-scenes issue, SREs communicate appropriately. The on-call might inform the **product owner or consumer teams** that “We observed some performance degradation in Service X and are applying fixes,” especially if the work could risk slight temporary changes (like using more resources) or if those teams might otherwise wonder about minor slowdowns. This heads-up builds trust that SREs are actively ensuring reliability. If the issue has the potential to become user-facing or to violate SLOs, the SRE may also alert management: “We’re seeing a capacity issue that could cause an incident during peak hours; we’re mitigating it now.” This way, if things did worsen, it wouldn’t be coming out of the blue. In many cases, end-users never know there was a risk – they experience normal service because SRE caught it early. Internally, however, the team treats it with the same rigor as an incident: the fact that an SLO alarm tripped means it’s taken seriously.
2. **Verification and Ongoing Monitoring:** After mitigation, the SREs verify that the performance metrics are back to acceptable ranges. They may run extra tests or continue to watch the system closely for the next day or two, since some fixes (like a code patch) might only show full effect under peak load. They also consider if additional monitoring is needed: for example, if this issue was detected at a late stage, perhaps an earlier warning metric could be put in place. SREs often **tune alerts** based on such experiences – adjusting thresholds or adding new ones so that the next time, they get even more advance notice​

[googblogs.com](https://www.googblogs.com/applying-the-escalation-policy-cre-life-lessons/#:~:text=Escalation%3A%20SLO,work%20on%20the%20service%E2%80%99s%20reliability)

. According to an example scenario from Google, if an error budget burn was not triggering an alert early enough, SREs might set the alert to page faster or at a lower threshold, and treat the investigation as part of ongoing reliability work​

[googblogs.com](https://www.googblogs.com/applying-the-escalation-policy-cre-life-lessons/#:~:text=Escalation%3A%20SLO,work%20on%20the%20service%E2%80%99s%20reliability)

.

1. **Follow-Up and Improvements:** The proactive issue is documented and followed up just like an incident. The team creates a report or adds to an existing **problem record** describing what was observed and how they fixed it. The **Embedded SRE** ensures any long-term fixes are scheduled – e.g. if they did a quick tweak to alleviate symptoms, the underlying bug or performance bottleneck is put in the engineering backlog. They may also update **capacity models** if the issue was capacity-related, to better anticipate growth. This might involve working with product teams to forecast traffic and ensure scaling occurs ahead of demand. All action items (code improvements, infrastructure upgrades, etc.) are tracked. The SRE might not do a full postmortem since no user impact occurred, but they will still hold a short review in the team’s next meeting: “How did we catch this? Did we react optimally? What can we do to avoid even coming close to the edge next time?” In essence, the team treats a proactive degradation alert as a success case of avoiding a real incident, and they solidify the lessons learned. They’ve optimized the system and the monitoring so that reliability is continuously improved **before customers are ever hurt**.​

[googblogs.com](https://www.googblogs.com/applying-the-escalation-policy-cre-life-lessons/#:~:text=Escalation%3A%20SLO,work%20on%20the%20service%E2%80%99s%20reliability)

**SRE Responsibilities in Key Operational Scenarios**

Site Reliability Engineers (SREs) play a crucial role in maintaining service reliability and responding to incidents. This section outlines the SRE’s responsibilities across five key scenarios, ensuring clarity on how SREs engage in each situation.

**When an Alert is Received**

* **Alert Validation and Tuning:** Upon receiving an alert, the SRE verifies its accuracy and relevance. They ensure the alert corresponds to a real issue (not a false alarm) by checking the system’s state. SREs continuously adjust monitoring configurations – setting meaningful thresholds based on service-level objectives (SLOs)​

[gartsolutions.com](https://gartsolutions.com/site-reliability-engineering-best-practices-key-best-practices-for-world-class-reliability/#:~:text=Set%20clear%20and%20meaningful%20alert,acceptable%20tolerances%20for%20system%20behavior)

and refining rules to reduce false positives and noisy alerts​

[gartsolutions.com](https://gartsolutions.com/site-reliability-engineering-best-practices-key-best-practices-for-world-class-reliability/#:~:text=Continuously%20review%20and%20refine%20alerting,false%20positives%20and%20noisy%20alerts)

. This tuning aligns alerts with expected system behavior and prevents alert fatigue.

* **Triage Support and Analysis:** SREs support the on-call engineer in triaging the alert. They dive into system **logs**, **metrics**, and **traces** to pinpoint anomalies or root causes of the alert. By quickly analyzing this telemetry, the SRE can identify what component or change triggered the issue and assess its impact. This detailed analysis helps confirm whether the alert signals a true incident and guides effective mitigation steps.
* **Mitigation and Automated Recovery:** During an alert-driven incident, SREs assist in mitigation efforts. They might execute runbook steps or trigger automated recovery actions (e.g. restarting services, failing over to backups) to remediate the issue. SREs leverage any **automated responses** available for well-understood problems to reduce manual intervention​

[gartsolutions.com](https://gartsolutions.com/site-reliability-engineering-best-practices-key-best-practices-for-world-class-reliability/#:~:text=Aggregate%20related%20alerts%20into%20more,notifications%20to%20avoid%20overwhelming%20responders)

. They also provide recommendations for containment, such as throttling a malfunctioning component or scaling resources, to stabilize the service quickly.

* **Communication and Escalation:** Throughout the incident, SREs communicate status updates and coordinate next steps. They keep stakeholders and affected teams informed of the situation and progress. If the issue is complex or persists, SREs follow escalation procedures – involving senior engineers or specialized teams as necessary​

[gartsolutions.com](https://gartsolutions.com/site-reliability-engineering-best-practices-key-best-practices-for-world-class-reliability/#:~:text=%E2%80%93%20Notify%20the%20relevant%20incident,call%20personnel)

. They maintain clear, timely communication with both internal stakeholders and (if needed) user-facing channels to provide transparency​

[gartsolutions.com](https://gartsolutions.com/site-reliability-engineering-best-practices-key-best-practices-for-world-class-reliability/#:~:text=possible)

. This ensures everyone knows the incident status and that the right experts are engaged.

**When a User Reports an Issue**

* **Issue Verification:** When a user or customer-facing team reports a problem, the SRE collaborates with the on-call engineer to verify its legitimacy. They first determine if the reported symptoms indicate a real service incident (as opposed to a user-specific error or a known ongoing issue). This may involve checking recent alerts or dashboards for any corresponding error spikes. The SRE helps ensure the issue is correctly identified and logged, so it can be addressed promptly.
* **In-Depth Analysis of Logs and Traces:** SREs conduct deeper analysis to diagnose the reported issue. They review historical logs, metrics, and distributed traces around the time of the problem to find any irregularities or trends. By analyzing logs, the SRE can determine when and where the issue occurred and what led up to it​

[wikitech.wikimedia.org](https://wikitech.wikimedia.org/wiki/SRE/Observability/About#:~:text=,be%20quite%20voluminous%2C%20so%20log)

. They look for patterns (such as similar errors in the past or correlated events across services) to understand the root cause. This thorough analysis helps distinguish one-time user glitches from broader system faults.

* **Cross-Team Collaboration:** If the issue spans multiple services or requires domain knowledge from another team, the SRE engages those teams for insights. They coordinate with consumer teams (e.g. the application team experiencing the issue or other dependent service owners) to gather additional context. This collaboration helps in forming a complete picture of the problem – for example, confirming whether a recent change on a consumer side could be contributing, or if other users are reporting similar symptoms. The SRE ensures all relevant parties are involved in solving the issue.
* **Preventative Recommendations:** After assisting in resolving the user-reported issue, the SRE recommends improvements to prevent recurrence. They might suggest changes in process or system design based on the learnings. For instance, if a missing index caused a slow query, the SRE could propose adding that index and improving database monitoring. Every incident or user issue is treated as a learning opportunity – the SRE drives post-incident review discussions to identify root causes and areas for improvement​

[zenduty.com](https://zenduty.com/blog/incident-management-guide/#:~:text=Continuously%20improve%3A)

. They ensure that follow-up actions (like adding new alerts, refining runbooks, or fixing bugs) are taken to avoid the issue in the future.

**When a Change is Deployed**

* **Pre-Release Reliability Checks:** Before a major change or deployment goes live, SREs participate in pre-release reviews to ensure reliability criteria are met. They verify that thorough testing has been done – for example, load testing to confirm the system can handle expected traffic, and failover or chaos testing to validate resilience under failure scenarios. An SRE may review **production readiness checklists** to confirm that monitoring is in place for the new release, rollback procedures are prepared, and that the change won’t violate error budget policies. By gatekeeping releases with these checks, SREs help catch potential reliability issues before they affect users.
* **Post-Deployment Monitoring:** After a deployment, SREs closely monitor system performance and error metrics during the critical post-release window. They watch dashboards and alerts for any regression in latency, error rates, or resource usage that the new change might introduce. Early detection is key – SREs know that effective change management requires robust monitoring to spot problems quickly​

[opensource.com](https://opensource.com/article/22/6/change-management-site-reliability-engineers#:~:text=The%20three%20core%20tenets%20of,and%20safe%20and%20fast%20rollbacks)

. If something looks wrong (e.g. increased error logs or a spike in CPU after the release), the SRE immediately investigates and correlates it with the deployment. This proactive monitoring ensures any issues caused by the change are identified and addressed as soon as possible.

* **Rollback and Hotfix Support:** When a deployment causes user impact or significantly deviates from expected behavior, SREs support swift mitigation, including rolling back the change. They help assess the risk and decide whether to initiate a rollback to the last known good version or to deploy a hotfix. SREs ensure that rollback procedures are documented and can be executed quickly (often automating them to save time)​

[signoz.io](https://signoz.io/guides/sre-best-practices/#:~:text=10%20Essential%20SRE%20Best%20Practices,reduce%20delays%20if%20problems%20occur)

. In the event of a severe problem, the mantra is often “rollback first, diagnose later” – the SRE prioritizes restoring service health. They coordinate with release engineers to initiate the rollback and verify that the system returns to stable conditions. If a quick fix is needed instead, the SRE helps in developing and validating that hotfix under pressure.

* **Updating Deployment Playbooks:** After the deployment (whether it succeeded or had issues), SREs make sure that the knowledge gained is fed back into the deployment processes. They update operational **playbooks or runbooks** to reflect any new steps discovered, so that the next release is smoother. For example, if the deployment revealed a gap in monitoring, the SRE will add a note to instrument that metric next time. If a rollback was performed, they ensure the rollback instructions were clear and adjust them if necessary based on what was learned. Ensuring that deployment runbooks are current and accurate is part of the SRE’s commitment to continual improvement and reliable release practices.

**When an Incident Requires a Postmortem**

* **Leading a Blameless Postmortem:** After a major incident has been resolved, SREs either lead or actively participate in the postmortem process. The emphasis is on a **blameless postmortem** culture – examining what went wrong without blaming individuals​

[sre.google](https://sre.google/sre-book/postmortem-culture/#:~:text=Blameless%20postmortems%20are%20a%20tenet,involved%20in%20an%20incident%20had)

. The SRE helps gather the timeline of events, system metrics, and actions taken during the incident. They ensure a thorough root cause analysis is conducted, focusing on technical factors (e.g. a failed dependency, an insufficient alert) and process factors (e.g. missed test case) that contributed. The goal is to understand **how** and **why** the failure occurred in order to learn from it, rather than who was at fault.

* **Identifying Systemic Improvements:** As part of the postmortem, SREs work to identify any underlying weaknesses in the system or process. They look for things like design flaws, documentation gaps, or inadequacies in testing that allowed the incident to happen or delayed its detection/response. For example, an SRE might determine that a database outage was prolonged due to an missing failover script, or that an alert didn’t fire when it should have because a threshold was too high. The SRE documents these findings and proposes long-term fixes – such as architectural changes, additional automation, or process adjustments – to bolster the system’s resilience. The focus is on addressing root causes and **preventing recurrence**, even if those fixes are non-trivial.
* **Action Item Ownership and Follow-Through:** SREs ensure that the postmortem isn’t just a document, but a driver for real change. They translate the lessons learned into concrete action items (e.g. “add automatic cache invalidation on failover” or “train team on incident command process”). Each action item is assigned an owner and tracked to completion​

[newrelic.com](https://newrelic.com/blog/best-practices/incident-postmortems-in-sre-practices#:~:text=Relic%20newrelic,items%20derived%20from%20these)

. The SRE often takes ownership of many of these remediation tasks or coordinates with the responsible teams to implement them. They may use issue trackers to monitor progress on fixes stemming from the incident. By diligently driving these follow-ups, SREs make sure the organization actually benefits from the insights gained, reducing the chance of similar incidents in the future.

* **Refining Runbooks and Automation:** In the aftermath of an incident, SREs update the operational documentation and automated tooling to incorporate the lessons learned. If runbooks (incident response guides) were found lacking or outdated during the incident, the SRE revises them to be more effective next time. They might add new recovery steps, clarify unclear instructions, or create new runbooks for scenarios that weren’t covered. Similarly, SREs improve automation scripts – for example, writing a script to quickly gather diagnostics if that would have helped, or enhancing self-healing mechanisms that could mitigate the issue automatically next time. Every significant incident is used as feedback to improve procedures and automation​

[gartsolutions.com](https://gartsolutions.com/site-reliability-engineering-best-practices-key-best-practices-for-world-class-reliability/#:~:text=Feedback%20Loop)

. This continuous refinement means the organization is better prepared and the systems more robust when facing future incidents.

**When Performance Degradation is Detected Proactively**

* **Proactive Monitoring of SLIs:** SREs continuously monitor key **service-level indicators (SLIs)** – metrics that reflect the service’s health and performance (such as latency, error rate, throughput, and resource utilization). By keeping a close eye on these signals, SREs can spot early signs of degradation before they escalate into user-visible problems. They fine-tune the monitoring setup and thresholds for these indicators over time so that alerts fire at the right sensitivity. Mastering critical metrics like latency, traffic, errors, and saturation helps the team stay ahead of potential failures​

[pflb.us](https://pflb.us/blog/site-reliability-engineer-sre-roles-responsibilities/#:~:text=Monitoring%20plays%20a%20crucial%20role,teams%20stay%20ahead%20of%20potential)

. In practice, this means the SRE might catch a rising error rate or slower response time and initiate investigation **proactively**, rather than waiting for an incident or user report.

* **Investigating Early Warning Signs:** When SREs detect a trend or anomaly indicating possible performance issues (for example, increasing response times or memory usage creeping up), they delve into detailed analysis using observability tools. They use distributed tracing and profiling tools to follow request paths and identify where slowness or bottlenecks are occurring. They correlate metrics across components to see if a specific microservice or database is the source of the degradation. By analyzing system metrics, logs, and traces together, SREs can pinpoint performance bottlenecks and emerging issues​

[iqpc.com](https://www.iqpc.com/events-observability-sre-summit/downloads/how-observability-enhances-site-reliability-engineering-ian-hawkins#:~:text=How%20Observability%20Enhances%20Site%20Reliability,make%20informed%20decisions%20about)

. This deep dive, done early, often allows them to resolve the underlying cause *before* it impacts customers or breaches SLOs.

* **Automation and Self-Healing:** SREs also implement automated responses for known patterns of performance degradation. If certain thresholds are crossed (CPU too high, queue backups, etc.), automated self-healing actions may kick in – for instance, auto-scaling the service, shedding load, or restarting a stuck process. The SRE designs and maintains these automation rules so that the system can correct itself in some cases without human intervention​

[gartsolutions.com](https://gartsolutions.com/site-reliability-engineering-best-practices-key-best-practices-for-world-class-reliability/#:~:text=Self)

. When a manual response is still required, the SRE might create tools or scripts to make the mitigation faster and consistent. The aim is to minimize performance impacts through quicker-than-human reactions and to handle routine performance fixes automatically.

* **Collaborating on Optimization and Capacity Planning:** For performance issues that indicate deeper capacity or efficiency concerns, SREs engage with the development team to drive long-term optimizations. They might profile the application to suggest code improvements or database query optimizations. If the service is approaching resource limits, the SRE coordinates capacity planning – using data and growth forecasts to ensure enough headroom for future demand. Effective capacity planning and performance tuning are part of the SRE’s remit to keep the system scalable and fast. SREs work continuously on such optimizations to maintain high availability and performance​

[pflb.us](https://pflb.us/blog/site-reliability-engineer-sre-roles-responsibilities/#:~:text=Effective%20capacity%20planning%20helps%20SREs,of%20operations%20to%20meet%20demand)

, often feeding insights back into the design and engineering process. By partnering with development teams on these efforts, SREs help ensure the system remains reliable as it evolves and grows.

**On-Call Engineer Role and Responsibilities**

The On-Call Engineer plays a critical role in maintaining system reliability and ensuring quick recovery from issues. This role is responsible for monitoring systems, responding to incidents, and collaborating with SREs (Site Reliability Engineers) and development teams to keep services running smoothly. Below is a structured, detailed, and actionable overview of the On-Call Engineer’s responsibilities across five key operational scenarios.

**When an Alert Is Received**

* **Acknowledge and Triage Immediately:** Promptly acknowledge incoming alerts and assess their priority to prevent unnecessary escalations. Quick triage helps determine if an alert is critical, urgent, or a false alarm.
* **Perform Initial Diagnostics:** Investigate the alert by checking logs, system metrics, and relevant runbooks. This initial analysis aims to identify obvious causes or confirm whether the alert is valid.
* **Engage SREs for Deeper Analysis:** If the issue isn’t immediately clear or is complex, involve SRE team members for additional expertise. SREs can provide deeper analysis, suggest advanced troubleshooting steps, or confirm suspicions about root causes.
* **Execute Mitigation Steps:** Take action to mitigate the problem using documented procedures or automation tools. This could include restarting services, scaling resources, applying patches, or running predefined scripts to restore normal operations.
* **Communicate and Escalate as Needed:** Continuously communicate status updates to relevant stakeholders (such as team leads or affected product owners) during the incident. If the issue cannot be resolved quickly or is escalating, follow the escalation protocol to bring in additional support or management attention.

**When a User Reports an Issue**

* **Log and Categorize the Issue:** Record user-reported issues in the incident tracking system, noting details like severity, impacted services, and number of users affected. Categorize the issue based on its impact (e.g., minor bug, major outage) to prioritize response.
* **Verify and Correlate with Monitoring Data:** Check monitoring dashboards, logs, and recent deployment changes to verify the user’s report. Confirm if there are corresponding alerts, error logs, or metric anomalies that align with the reported problem.
* **Reproduce the Problem:** If possible, attempt to reproduce the issue in a controlled environment. Reproducing the issue helps in understanding its scope and provides more context, which can lead to a faster and more accurate resolution.
* **Collaborate with Teams to Diagnose:** Work together with SREs and relevant development or support teams to pinpoint the root cause. Share findings from initial diagnostics and user feedback, and use collective expertise to determine the best fix.
* **Provide Updates to Users and Stakeholders:** Keep the reporting user and internal stakeholders informed about progress. Communicate what has been found, what is being done to fix it, and an estimated timeline for resolution. Ensure that once resolved, the user is notified and any necessary follow-up (like a patch or knowledge base update) is completed.

**When a Change Is Deployed**

* **Monitor System Health Around Deployments:** Closely watch system metrics and application logs before, during, and after a deployment. Look for anomalies such as error rate spikes, latency increases, or any unusual behavior that might indicate a deployment issue.
* **Act as First Responder to Failures:** If something goes wrong during or right after a deployment (e.g., new errors, crashes), be the first to respond. Quickly identify if the deployment is the cause by comparing pre- and post-deployment system state and engage the deployment team if necessary.
* **Execute Rollback or Mitigation if Needed:** If the deployment causes a severe issue, initiate a rollback to the previous stable version or execute other mitigation strategies (like feature toggling off a new feature). Follow the predefined rollback procedures to minimize user impact.
* **Document Deployment Outcomes:** Keep detailed deployment logs and note any issues encountered. Update runbooks or deployment documentation with insights gained, such as tweaking monitoring alerts or adding a new check to catch similar issues in the future.
* **Coordinate with Developers and SREs:** Work closely with the developers who performed the deployment and SREs throughout the process. Ensure everyone is aware of system status, and confirm that all post-deployment checks pass before declaring the deployment successful. This collaboration helps ensure a smooth deployment process and quick resolution of any hiccups.

**When an Incident Requires a Postmortem**

* **Document Incident Details:** After a major incident is resolved, compile a detailed timeline of events, including when alerts were received, actions taken by the On-Call Engineer, involvement of other teams, and how the issue was resolved. Clearly note the root cause once identified and any contributing factors.
* **Participate in Blameless Postmortems:** Join the postmortem meeting to discuss what happened openly and without blame. Share your perspective on the incident – what went well, what was challenging – and engage in discussions on how to prevent similar incidents.
* **Identify Gaps and Improvements:** Help pinpoint any gaps in monitoring or alerting that led to a delayed detection, or process issues that hampered the response. For example, maybe a runbook was outdated or an alert threshold was set too high. Collaborate with the team to propose improvements.
* **Update Runbooks and Playbooks:** Incorporate lessons learned into documentation. Update the incident response playbook or specific service runbooks to reflect new knowledge (e.g., adding a troubleshooting step that was discovered or refining an existing procedure). These updates help the next On-Call Engineer handle similar incidents more effectively.
* **Follow Up on Action Items:** Ensure all postmortem action items are assigned to owners and tracked to completion. This might involve changes like adding new monitors, refining deployment processes, or scheduling training. The On-Call Engineer should monitor the progress of these tasks to make sure the organization truly learns from the incident.

**When Performance Degradation Is Detected Proactively**

* **Continuous Performance Monitoring:** Regularly analyze key performance indicators (KPIs) such as response times, throughput, CPU/memory usage, and error rates. Spot trends or early warning signs of degradation before they become user-impacting issues.
* **Exploratory Investigation:** When metrics start to trend in the wrong direction (even if no alert has fired yet), conduct an investigation. Use logs, tracing tools, and dashboards to dig into where the bottleneck or issue might be occurring (for example, a specific database query or an overloaded server).
* **Apply Pre-Approved Mitigations:** If you identify a clear cause for degradation (like a memory leak or an expensive query) and there are known fixes, apply pre-approved mitigation steps. This could involve increasing resources (scaling out/in), clearing caches, or restarting a service to clear a bad state. Always follow change management guidelines even for quick fixes.
* **Escalate Before SLAs Are Breached:** If the degradation is trending towards violating Service Level Agreements (SLAs) or could impact users soon, alert the SREs and possibly the on-call manager proactively. Early escalation ensures more hands and expertise are working on the issue before it becomes a major incident.
* **Collaborate on Long-Term Solutions:** Work with the development team to plan a permanent fix or optimization. This could involve code changes, architectural adjustments, or adding capacity. Also, contribute to capacity planning discussions by sharing insights from the incident (e.g., if traffic is growing faster than anticipated or a component isn’t scaling well).