What is a postmortem?
A postmortem is a structured review conducted after an incident.
✔ Blameless culture → Focuses on root cause analysis, not blame
✔ Goal: Learn from failures to drive continuous improvement

What is an incident?
An incident is any event where a software system fails to perform as expected, disrupting normal operations.
✔ Impact Areas: Functionality, availability, performance, security
✔ Effects: Service degradation, downtime, outages
✔ Quantifiable Impact: Lost users, service downtime, failed transactions → lost revenue
✔ Unquantifiable Impact: Damage to brand image, morale, legal issues

Incident Metrics
✔ SLI (Service Level Indicator): Measures service reliability (e.g., uptime, latency)
✔ SLO (Service Level Objective): Target performance levels (e.g., 99.9% availability)
✔ SLA (Service Level Agreement): Customer commitments & consequences of breaches

How do we rank the severity of an incident?
✔ P0 (Critical): System-wide failure, immediate response required (ASAP, all hands on deck)
✔ P1 (High): Severe issue affecting functionality, but not all users (urgent fix needed)
✔ P2 (Medium): Minor impact, not urgent but affects service quality (can be scheduled)
✔ P3 (Low): Non-critical, often cosmetic issues (backlog priority)

Lifecycle of an Incident
1️⃣ Occurrence → Incident happens
2️⃣ Detection → Monitoring system identifies it
3️⃣ Acknowledgement → Officially recognized
4️⃣ Escalation → Assign severity & notify teams
5️⃣ Mitigation & Triage → Contain impact & diagnose
6️⃣ Resolution → Fix the issue
7️⃣ Postmortem → Analyze root cause
8️⃣ Action Items (AIs) → Implement improvements

Communication during incidents
✔ War Room Roles:
Incident Commander → Oversees, delegates, coordinates
Communication Lead → Handles updates with stakeholders
Tier 1 On-call → First responders (immediate action)
Tier 2 On-call → Handles complex debugging
✔ Structured Updates:
Recent changes, progress on mitigation, next update time

Roles
Roles and responsibilities differ by company:
- QA engineer
- Site reliability engineer (SRE)
- Software engineer

Incident Examples
✔ SLO Misses: Delays in ETL, slow website loading due to data growth
✔ Human Error: Faulty code, manual process mistakes
✔ Malicious Activity: DDoS attacks, bot activity
✔ Physical Failures: Datacenter outages, network issues

Postmortem template
1️⃣ Status → Current state of the incident
2️⃣ Summary → Brief incident description
3️⃣ Incident Timeline
    Detection → When and how it was identified
    Mitigation Progression → Steps taken to reduce impact
    Resolution → How the issue was fixed
4️⃣ Impact Assessment → Effect on users, systems, and business
5️⃣ Root Cause Analysis → Underlying cause of the incident
6️⃣ Corrective Actions → Fixes applied to prevent recurrence
7️⃣ Future Prevention → Long-term improvements and monitoring

Postmortems vs. Retrospectives
✔ Postmortems: Focus on failures, providing a detailed, formal analysis of incidents.
✔ Retrospectives: Improve team processes and performance, not limited to failures.
✔ Both are essential for building resilient systems.

The Swiss Cheese Model
✔ Concept: Incidents occur when all layers of safeguards fail
✔ Layers of Defense:
1️⃣ Good programming practices
2️⃣ Code review
3️⃣ Unit tests
4️⃣ Integration tests
5️⃣ E2E tests
6️⃣ Load/stress tests
7️⃣ Good rollout strategy
📌 Multiple safeguards reduce the risk of failures, but when all fail, incidents happen.

Why? Why? Why?
✔ Method: Repeatedly asking "Why?" helps identify the true root cause of an incident.
✔ Example:
1️⃣ Why did the database crash? → Ran out of memory.
2️⃣ Why? → Query cache kept growing.
3️⃣ Why? → Automatic purging was disabled.
4️⃣ Why? → Misconfiguration in the deployment script.
5️⃣ Why? → Testing didn't cover this scenario.
📌 Many incidents stem from lack of testing & monitoring.

Testing & Monitoring Summary
✔ Testing → First line of defense to prevent incidents.
✔ Monitoring → Second line of defense to detect incidents early.
✔ Why both? Testing is never 100% perfect, so monitoring is essential for catching failures.
📌 Strong testing + effective monitoring = fewer postmortems.

Monitoring vs. Alerting Summary
✔ Monitoring → Tracks & observes system health by collecting metrics, logs, and trends.
✔ Alerting → Notifies engineers when predefined thresholds are met based on monitoring data.
📌 Key Difference: Monitoring analyzes, alerting response.

Monitoring Scenarios
✔ Best Case: Detect incidents before they happen → Engineers fix issues proactively (e.g., job runtime threshold exceeded, adjusted resources).
✔ Average Case: Detect small/low-priority issues early → Engineers respond before major impact (e.g., UI bug causing null values, rollback & fix).
✔ Slightly Bad Case: Detect incidents as they occur → Immediate action needed to minimize damage (e.g., database outage detected via failed transactions).
✔ Worst Case: Detect incidents long after they've caused damage → Business impact, financial loss (e.g., revenue loss due to miscategorized data).
📌 Early detection = faster resolution, lower impact.

Monitoring - Best Practices

✔ SLOs: Use Service Level Objectives to define reliability goals.
✔ Reducing Noise: Ensure alerts are meaningful & actionable, avoid excessive/irrelevant alerts.
✔ Adaptive Monitoring: Use dynamic thresholds that adjust over time instead of fixed values, and regularly update them.
📌 Effective monitoring minimizes noise and adapts to real-world changes.

Logging
✔ Why?: Speeds up debugging by tracing failures and providing a detailed event timeline.
✔ Helps reproduce issues to prevent future incidents.
✔ Enhances monitoring systems by providing better insights.
📌 Good logging = faster debugging & better monitoring.

Unit Testing
✔ Definition: Tests individual components or functions in isolation.
✔ Best Practices:
Every code change should include unit tests.
Use mocking, stubbing, and fake data to simulate dependencies.
Code coverage helps ensure thorough testing.
✔ Example:
AuthService class: Handles user login.
TestAuthService: Uses unittest to check successful login, failed login, and database mocking.
📌 Unit tests catch bugs early and ensure code reliability.

Integration Testing
✔ Definition: Tests interactions between different systems (e.g., databases, APIs).
✔ Key Purpose: Ensures data is correctly passed between modules.
✔ More Complex Than Unit Testing: No mocking or stubbing, tests real components.
✔ Common Issue: Data changes → One service updates its output, breaking downstream services.
📌 Integration tests ensure smooth communication between system components.

E2E Testing
✔ Definition: Tests the entire data flow from start to finish.
✔ Challenges:
Difficult to simulate all system parts as complexity grows.
Small changes in components may break E2E tests.
Broken tests slow down development progress.
📌 E2E tests ensure full system functionality but can be fragile and slow.

Load Testing
✔ Objective: Simulate real-world traffic to assess system performance under load.
✔ Types of Load:
Normal Load: Expected daily traffic.
Peak Load: Sudden traffic spikes.
Stress Load: Pushing the system beyond limits.
📌 Load testing ensures the system can handle real-world demand.

Fault injection
✔ Purpose: Simulate real-world failures to uncover weaknesses in:
Infrastructure reliability
Monitoring effectiveness
Incident response readiness
✔ Netflix 'Chaos Monkey':
Randomly terminates instances/services in production.
Goal: Test system resilience, redundancy, and failover mechanisms.
Success: In 2012, Netflix survived an AWS outage thanks to fault injection testing.
📌 Fault injection helps build failure-resistant systems.

Best Deployment Practices
✔ Deployment Hygiene:
1️⃣ Test in dev environment first.
2️⃣ Deploy in dev with real data for validation.
3️⃣ Deploy in prod after successful testing.
✔ A/B Testing: Compare different versions to optimize performance.
✔ Rollout Plan:
Incremental rollout (Canary Deployment): Deploy to a small subset before full release.
📌 Gradual, well-tested deployments reduce risks.

Incident Case Studies Summary
1️⃣ YouTube's 90-Minute Global Outage (Oct 16, 2018)
✔ Root Cause: Bad network configuration change → Routing failures → Servers inaccessible.
✔ Mitigation: Manual rollback.
✔ Impact:
User frustration (platform down).
Revenue loss (ads & content creators, $11B annual revenue in 2018).
✔ Why it happened:
No staged rollouts → Change applied too broadly.
Insufficient testing → Bad config pushed to production.
No automated rollback → Delay in resolution.
📌 Gradual rollouts, better testing, automated rollback reduce risks.

2️⃣ Equifax Data Breach (Sep 2017)
✔ Root Cause: Failure to patch a known vulnerability in a third-party system.
✔ Mitigation: Security patch.
✔ Impact:
147M users' sensitive data leaked (PII, SSN, credit cards).
$700M settlement & trust loss.
35% stock drop & CEO resigned.
✔ Why it happened:
Poor security management → Missed a publicly known patch.
Poor monitoring → Breach undetected for 76 days.
Lack of network segmentation → Hackers moved freely inside the system.
📌 Proactive security patching, strong monitoring, and network segmentation are critical.

Key Takeaways
✔ Incidents = Learning Opportunities
Focus on root causes, not blame.
Implement action items to prevent recurrence.
✔ Effective Communication Matters
Provide clear, concise, and timely updates.
Use a structured incident command system.
✔ Proper Testing Prevents Downtime
Test all possible failure scenarios.
✔ Monitoring & Alerting Reduce Downtime
Catch issues missed in testing.
Fine-tune alerts to be meaningful & actionable.
✔ Build Quality Engineering Practices
Automate where possible.
Design robust, resilient systems.
📌 Strong testing, monitoring, and communication build reliable systems.