Sample RunBook for Handling Backend System Failures

Table of Contents

Introduction

This RunBook provides detailed procedures for handling common backend system failures, including database connection failures, API endpoint failures, service timeout

issues, and high CPU usage. Each section includes steps for identification, immediate actions, troubleshooting, and resolution. This document is intended for use by Site Reliability Engineers (SREs) and other technical staff responsible for maintaining the reliability and performance of backend systems.

Database Connection Failures

Identification

1. Monitoring Alerts:
    - Check for alerts related to database connectivity issues in your monitoring system (e.g., Prometheus, Datadog).
    - Look for specific error messages such as "Unable to connect to database," "Connection refused," or "Timeout" in the application logs.
2. User Reports:
    - Users may report issues such as "unable to retrieve data," "database timeout," or "application errors related to data retrieval."

Immediate Actions

1. Acknowledge the Alert:
    - Confirm receipt of the alert in the monitoring system to stop further notifications.
2. Check Service Health:
    - Verify the status of the database service (e.g., using `systemctl status mysql` for MySQL, or checking the cloud provider's dashboard for managed databases).

Troubleshooting Steps

1. Network Connectivity:

- Test network connectivity between the application server and the database server using ping and telnet commands:

  ping <database_server_ip>

  telnet <database_server_ip> <database_port>

- Verify firewall rules and network security groups to ensure they allow traffic between the application and database servers.

2. Database Logs:
   - Review database logs for any errors or warnings:

     tail -f /var/log/mysql/error.log

   - Look for messages indicating issues such as resource exhaustion, connection limits, or corrupted data files.
3. Configuration Files:
   - Check database configuration files for any recent changes that could impact connectivity (e.g., /etc/mysql/my.cnf for MySQL).

Resolution

1. Restart Database Service:
   - Restart the database service and verify connectivity:

     sudo systemctl restart mysql

   - Confirm the service is running properly:

     sudo systemctl status mysql

2. Fix Configuration Issues:
   - Correct any misconfigurations found in the database configuration files. Ensure settings such as connection limits and network bindings are correct.
3. Database Health Check:
   - Perform a health check on the database:
     - Check for table corruption using database-specific commands (e.g., CHECK TABLE in MySQL).
     - Run diagnostic queries to ensure the database is performing optimally.
4. Document the Incident:
   - Update the incident ticket with the steps taken and the final resolution.
   - Conduct a post-incident review to identify any preventive measures.

API Endpoint Failures

Identification

1. Monitoring Alerts:
    - Check for alerts related to API endpoint failures in your monitoring system.
    - Look for specific error messages such as "500 Internal Server Error," "404 Not Found," or "503 Service Unavailable" in the application logs.
2. User Reports:
    - Users may report issues such as "API not responding," "unexpected error from API," or "service unavailable."

Immediate Actions

1. Acknowledge the Alert:
    - Confirm receipt of the alert in the monitoring system to stop further notifications.
2. Check Service Health:
    - Verify the status of the API service using system commands (e.g., `systemctl status my-api-service`) or cloud provider dashboards.

Troubleshooting Steps

1. API Logs:
    - Review API logs for any errors or warnings. Use commands such as:

      ```
      tail -f /var/log/my-api-service/error.log
      ```

    - Look for stack traces, exception messages, or other indicators of failure.
2. Dependency Checks:
    - Ensure all dependencies (e.g., databases, external APIs) are operational. Check their status and logs for any issues.
3. Configuration Files:

- Check API configuration files for any recent changes that could impact functionality (e.g., /etc/my-api-service/config.yaml).

4. Code Review:
    - Review recent code changes that could have introduced bugs or issues.

      Use version control tools to identify recent commits:

      git log

Resolution

1. Restart API Service:
    - Restart the API service and verify functionality:

      sudo systemctl restart my-api-service

    - Confirm the service is running properly:

      sudo systemctl status my-api-service
2. Fix Code Issues:
    - Debug and fix any code issues identified in the logs. Deploy hotfixes if necessary.
3. Update Dependencies:
    - Ensure all dependencies are up to date and functioning properly. Update them if necessary.
4. Document the Incident:
    - Update the incident ticket with the steps taken and the final resolution.
    - Conduct a post-incident review to identify any preventive measures.

Service Timeout Issues

Identification

1. Monitoring Alerts:
    - Check for alerts related to service timeouts in your monitoring system.

- Look for specific error messages such as "Request timed out" or "Service unavailable due to timeout" in the application logs.

2. User Reports:
   - Users may report issues such as "service is slow," "request is timing out," or "unable to complete actions."

Immediate Actions

1. Acknowledge the Alert:
   - Confirm receipt of the alert in the monitoring system to stop further notifications.
2. Check Service Health:
   - Verify the status of the affected service using system commands (e.g., `systemctl status my-service`) or cloud provider dashboards.

Troubleshooting Steps

1. Service Logs:
   - Review service logs for any errors or warnings. Use commands such as:

     ```
     tail -f /var/log/my-service/error.log
     ```

   - Look for timeout errors, performance bottlenecks, or resource exhaustion indicators.
2. Network Diagnostics:
   - Perform network diagnostics to identify latency issues:

     ```
     ping <service_endpoint>
     ```

     ```
     traceroute <service_endpoint>
     ```

   - Check for network congestion or routing issues.
3. Resource Utilization:
   - Check resource utilization (CPU, memory, disk I/O) on the server using tools like `top`, `htop`, or `iostat`.
4. Application Metrics:
   - Review application-specific metrics (e.g., request rates, response times) in your monitoring dashboards.

Resolution

1. Increase Timeouts:
    - Adjust timeout settings in the service configuration to allow more time for requests to complete.
2. Optimize Code:
    - Identify and optimize slow code paths. Profile the application to pinpoint performance bottlenecks.
3. Scale Resources:
    - Increase resources (CPU, memory) for the service if necessary. Use auto-scaling features if available.
4. Document the Incident:
    - Update the incident ticket with the steps taken and the final resolution.
    - Conduct a post-incident review to identify any preventive measures.

High CPU Usage

Identification

1. Monitoring Alerts:
    - Check for alerts related to high CPU usage in your monitoring system.
    - Look for spikes in CPU usage graphs and specific error messages indicating CPU exhaustion in the logs.
2. User Reports:
    - Users may report issues such as "system is slow," "high response times," or "application is unresponsive."

Immediate Actions

1. Acknowledge the Alert:
    - Confirm receipt of the alert in the monitoring system to stop further notifications.
2. Check Service Health:

- Verify the status of the affected service using system commands (e.g., `systemctl status my-service`) or cloud provider dashboards.

Troubleshooting Steps

1. Process Analysis:
   - Identify processes consuming high CPU using tools such as `top` or `htop`:

     ```
     top
     ```

     ```
     htop
     ```

   - Look for processes with unusually high CPU usage and identify the corresponding services or applications.
2. Service Logs:
   - Review service logs for any errors or warnings. Use commands such as:

     ```
     tail -f /var/log/my-service/error.log
     ```

   - Look for indicators of resource-intensive operations or infinite loops.
3. Application Profiling:
   - Use profiling tools to identify performance bottlenecks in the application (e.g., `perf`, `py-spy` for Python applications):

     ```
     sudo perf top
     ```

     ```
     py-spy top --pid <pid>
     ```

Resolution

1. Terminate Rogue Processes:
   - Terminate processes that are consuming excessive CPU. Use `kill` or `kill -9` if necessary:

     ```
     sudo kill <pid>
     ```

     ```
     sudo kill -9 <pid>
     ```

2. Optimize Code:
   - Identify and optimize inefficient code paths. Refactor code to improve performance and reduce CPU usage.
3. Scale Resources:

    - Increase CPU resources for the service if necessary. Use cloud provider features to scale vertically or horizontally.
4. Document the Incident:
    - Update the incident ticket with the steps taken and the final resolution.
    - Conduct a post-incident review to identify any preventive measures.

Contact Information

1. On-Call Team:
    - Name: Bob
      Role: SWE
      Contact: +91 91213 12432
2. Escalation Contacts:
    - Name: Tom Cruise
      Role: Senior SWE
      Contact: +91 81924 35216
3. External Support:
    - Vendor Name: Toshiba
      Role: Vendor Support
      Contact: +81 22124 25167
