[Section 1]
Question1
What are the key security concerns when it comes to DevOps?

- **Code Security:** Ensuring secure coding practices, code review, and static code analysis.
- **Configuration Management:** Managing configurations securely to prevent unauthorized changes.
- **Access Control:** Implementing strict access controls and role-based access control (RBAC).
- **Infrastructure as Code (IaC):** Ensuring IaC templates are secure and free from vulnerabilities.
- **CI/CD Pipeline Security:** Securing the build and deployment pipelines to prevent unauthorized access and tampering.
- **Monitoring and Logging:** Continuous monitoring and logging to detect and respond to security incidents.
- **Cultural Resistance and Too much focus on tools:** With the ever increasing tools and technologies that DevOps engineers employ to make their works easier, security in devops is a shared responsibility. Convincing teams to join in and makes implementing security in DevOps practices easier.
- **Legacy environments:** Inasmuch as DevOps is a relatively new concept, most tools are not compatible with legacy environments which hosts most services. This presents the dilemma of incompatibility with these systems and might take some time to find a compatible solution to these systems.

  Cloud security: As Cloud computing becomes the defacto the the day, the security risk with deploying to cloud environments increases with the ever changing cybersecurity landscape and the proliferation of AI. This poses the challenge to ensure proper security measures are put in place to mitigate the risks of accidentally exposing sensitive data in cloud environments.

  New and Complex security implementations for Containers and k8s environment: Change is the only constant in this ever changing technology industry. With the introduction of new tools and technologies on the daily, the need arises for proper security implementation in utilizing these tools for critical production environments. Most of these tools come with their custom security configurations and setup which requires expertise and learning before efficient automation can be implemented using DevOps.

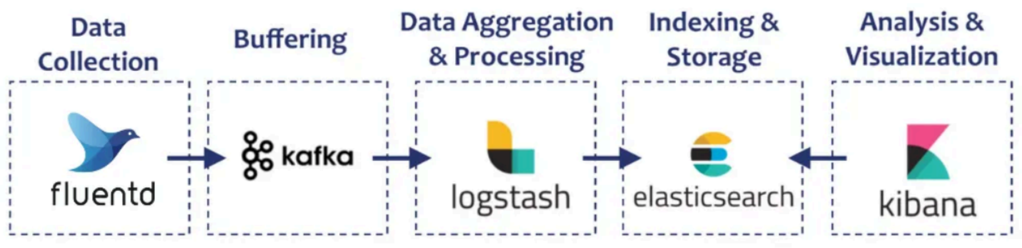
<u>Question2</u>
How do you design a self-healing distributed service?

- **Health Checks and Heartbeats:** Implementing regular health checks to monitor the status of services.
- **Redundancy and Replication:** Using redundant components to ensure availability and avoid single points of failure. This can be implemented using loadbalancer,backup databases or microservices.
- **Implement Monitoring and Logging:** Utilizing tools like prometheus,grafana or Elastic search to collect and visualize data about a systems health and performance. You can also use CloudWatch, CloudTrail or Logstash to capture and analyze logs and events from your systems.
- **Apply feedback and control loops:** Effectively using feedback and control loops to scale your system based on demand(autoscaling), to balance load across your services(loadbalancing), to switch to alternative resources(failover). This can be implemented by using technologies like kubernetes, AWS Auto Scaling,etc.
- **Loose coupling(Event Driven Architectures):** Designing various components of your distributed service to be loosely coupled, eliminating dependencies and bottlenecks that comes with scaling. This can be implemented using message brokers, queues, consumer-producer pattern, publisher-subscriber etc.
- **Failure Detection:** Using automated failure detection mechanisms to identify and isolate faulty components with little no human intervention.
- **Leader election:** Utilizing consensus algorithms(eg: raft and paxos) in your distributed architecture to ensure consistency and uniformity in executing tasks.
- **Throttling and Rate Limiting requests:** Implementing throttling algorithms(Leaky bucket, Sliding window, fixed window, token bucket,etc) to streamline requests to the service and efficiently handle peak loads without breaking down your service
- **Chaos engineering:** Continuous improvement and learning from how your service recovers from failure by stress testing your service using tools like chaos monkey, gremlin, etc
- **Checkpoint long-running transactions**: Using checkpoints to provide resiliency if a long-running operation fails.
- **Follow best practices and standards:** Designing modular, decoupled and loosely coupled components. Also enforce consistent naming conventions, documentation and code quality standards. Adhere to DevOps principles such as continuous integration,delivery,improvement and testing.

<u>Question 3</u>

Describe a centralized logging solution and how can you implement logging for a microservice architecture?

Solution: Using a centralized logging solution like the EFK Stack (Elasticsearch, FluentD, Kibana) or Fluentd with Grafana



Implementation:

Intalling log agents on associated systems and services(FluentD)

Standardizing Log Formats(JSON, necessary logs)

Using correlation IDs and Distributed Tracing tools like OpenTelemetry to map the flow of requests through the system.

Centralizing Log Collection by using a log aggregator(Logstash)

Implementing Real-Time processing(Apache Kafka)
Ensuring scalability(Horizontal scaling and Load balancing)

<u>Question 4</u>
What are some of the reasons for choosing Terraform for DevOps?
Answers:

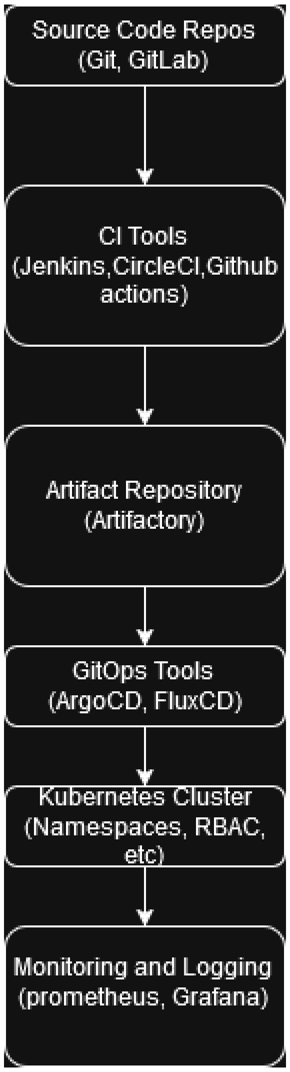
- **Infrastructure as Code**: Allows defining infrastructure using code, making it easy to version control and manage.

- **Cloud Agnostic**: Supports multiple cloud providers (AWS, Azure, GCP, etc.).
- **Automation**: Automates the provisioning and management of infrastructure.
- **Consistency:** Ensures consistent environments across different stages (development, staging, production).

Question 5

How would you design and implement a secure CI/CD architecture for microservice deployment using GitOps? Take a scenario of 20 microservices developed using different languages and deploying to an orchestrated environment like Kubernetes. (You can add a low-level architectural diagram)
Solution:

CI/CD pipeline;

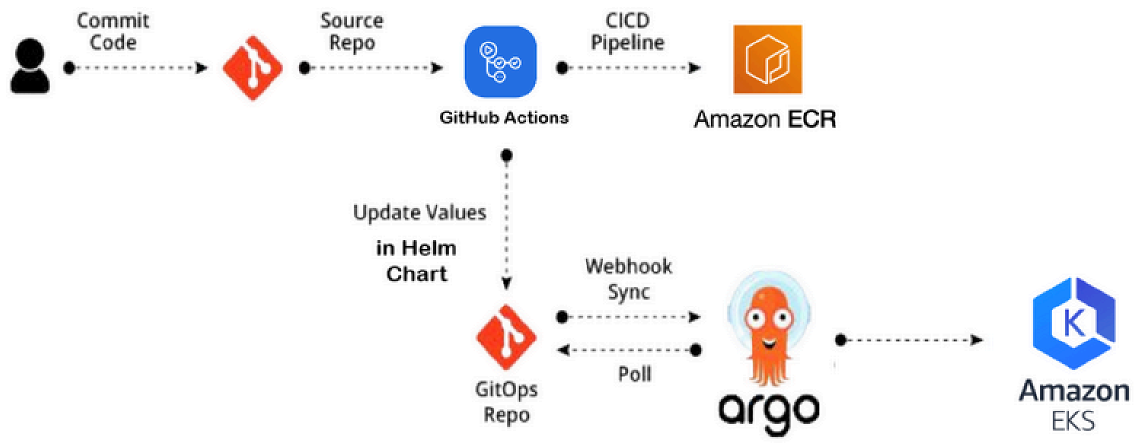

Source Code Repos (Git, GitLab)
CI Tools (Jenkins,CircleCI,Github actions)
Artifact Repository (Artifactory)
GitOps Tools (ArgoCD, FluxCD)
Kubernetes Cluster (Namespaces, RBAC, etc)
Monitoring and Logging (prometheus, Grafana)

Architecture:

- Using Git repositories as the single source of truth for deployment configurations.

- Implementing RBAC and network policies in Kubernetes to secure the cluster.
- Using CI/CD tools like ArgoCD or FluxCD for GitOps.
- Automating builds and deployments using CI tools like Jenkins or GitHub Actions.
- Implementing monitoring and alerting using Prometheus and Grafana.

Architecture Diagram;



Question 6
You notice React Native builds are failing intermittently. What's your debugging process?

Debugging intermittent React Native build failures requires a systematic approach to identify and resolve the root cause. Here's my detailed step-by-step process:

1. Understanding the Symptoms

Error Logs: Gathering details from the build logs to identify error patterns. Look for specific error messages, stack traces, or failed commands.
Frequency: Determine how often the failure occurs (e.g., specific builds, CI pipelines, local builds).
Environment: Checking if the failures happen only in certain environments:
Local/development machines, Continuous Integration (CI) pipelines, Specific operating systems (e.g., macOS for iOS builds).

2. Reproduce the Issue

Attempt to reproduce the build failure:
Clean the project and rebuild

Run the build multiple times to observe if it fails consistently under specific conditions.

3. Analyze Logs

Examining the build logs carefully for:
- Dependency errors: Look for issues with npm, Yarn, Gradle, or CocoaPods dependencies.
- Version mismatches: Identify if incompatible library versions are causing failures.
- Out-of-memory (OOM) errors: Check for memory-related build issues.
- Network errors: Look for problems downloading dependencies or assets.

4. Check Dependency Management

Node Modules:
- Delete node_modules and reinstall dependencies:

```
rm -rf node_modules
npm install
```

Ensure I'm using a consistent package manager (npm or Yarn).

CocoaPods (iOS):

Delete the Pods folder and reinstall:

```
cd ios
rm -rf Pods Podfile.lock
pod install
```

Gradle (Android):

Clear the Gradle cache:

```
./gradlew cleanBuildCache
```

Ensure Gradle wrapper version matches the React Native version.

5. Check for Version Conflicts

React Native Version:
- Ensure the project uses a compatible version of React Native with other libraries.

Node.js:
- Verify the Node.js version is within the supported range for React Native.

Dependencies:
- Look for mismatched or deprecated dependencies in package.json.

Use tools like npm-check or yarn outdated to check outdated packages.
Build Tools:
Check for mismatches in Xcode, Android SDK, or Gradle versions.

6. Check CI/CD Environment

Environment Variables:
Verify that all necessary environment variables (e.g., ANDROID_HOME, JAVA_HOME, signing keys) are correctly set.
Cache Issues:
Clear CI build caches for Node.js, Gradle, and CocoaPods.
Resource Limits:
Check if the CI machine has sufficient memory and CPU for the build.

7. Inspect Native Code

iOS:
Check AppDelegate.m, Podfile, and Xcode project settings for incorrect configurations.
Android:
Inspect MainApplication.java, build.gradle, and AndroidManifest.xml for issues.

8. Test with a Fresh Clone

Clone the repository into a new directory and build the project:
This helps identify issues related to local environment setup or stale files.

9. Check Third-Party Services

API Keys and Services:
Ensure required keys for services like Firebase, Maps, or analytics are properly configured.
External Dependencies:
Check if the failure is due to downtime or changes in third-party services.

10. Common Fixes

Lock File:
Ensure package-lock.json or yarn.lock is committed to maintain consistent dependency versions.
Hermes Engine (if used):
Rebuild the app with Hermes disabled to check if it’s causing the failure.
Disable Hermes in android/app/build.gradle or ios/Podfile.
Incremental Builds:
Disable incremental builds for Android by adding org.gradle.caching=false to gradle.properties.

11. Monitor Build Resources

Memory Usage:
Monitor memory usage during builds. For Android, increase the heap size in gradle.properties:

```
org.gradle.jvmargs=-Xmx4g
```

Parallel Builds:

Disable parallel builds to identify issues:

```
org.gradle.parallel=false
```

12. Use Debugging Tools

Flipper:
Use Flipper to debug React Native builds and view logs.
Verbose Logging:
Enable verbose logs for detailed error messages:

```
npx react-native run-android --verbose
npx react-native run-ios --verbose
```

13. Isolate Problematic Code

Comment out recently added dependencies or code changes to isolate the issue.

14. Consult Documentation and Forums

Check React Native GitHub issues and forums for similar problems.
Verify compatibility of libraries using the React Native Directory.

15. Contact Support

If the issue persists, contact the relevant support teams (e.g., CI provider, React Native community, library maintainers).

[Section 2]
<u>**Section 2.**</u>

Question 7
Write a Prometheus exporter in Python that connects to a specified RabbitMQ HTTP API(it's the management plugin) and periodically reads…

Answer;
https://github.com/belisky/lemonade_devops/blob/main/prometheus_exporter.py

Question 8:
Write a script to restart the Laravel backend service if CPU usage exceeds 80%;
Answers;
Python script version:
https://github.com/belisky/lemonade_devops/blob/main/laravel_restart_python.py

Bash script version:
https://github.com/belisky/lemonade_devops/blob/main/laravel_restart_bash.sh

Question 9:
A Postgres query is running slower than expected. Explain your approach to troubleshooting it.

Answer;

1. Understanding the Context

- **Query Details**:
  - What does the query do (e.g., SELECT, INSERT, UPDATE)?
  - Is it part of an application workflow or a one-off query?
  - Are there specific inputs causing slowness?
- **History**:
  - Has the query always been slow, or is this a recent change?
  - Have there been recent changes to the database schema, indexes, or data volume?
- **Environment**:
  - Are other queries running slower, or is it isolated to this query?
  - Is the issue specific to certain times, workloads, or environments (e.g., development, production)?

2. Reproduce and Measure

- Run the query in an isolated environment (e.g., psql) to rule out external factors like network latency.
- Use EXPLAIN and EXPLAIN (ANALYZE) to inspect the query execution plan:
  EXPLAIN ANALYZE SELECT * FROM my_table WHERE condition;

- Look for high-cost operations such as sequential scans, nested loops, or large sort operations.

3. Identify Performance Bottlenecks

a. Query Structure

- Check for:
    - Inefficient JOINs or WHERE conditions.
    - Use of functions on indexed columns (e.g., WHERE LOWER(column) = 'value' prevents index usage).
    - Missing or redundant columns in SELECT (e.g., SELECT * can fetch unnecessary data).

b. Indexes

- Verify if appropriate indexes exist for the query:

```
SELECT *
FROM pg_indexes
WHERE tablename = 'my_table';
```

If a sequential scan is being used, consider adding an index to columns used in WHERE, JOIN, GROUP BY, or ORDER BY clauses.

c. Table Size and Statistics

- Check table size and the number of rows:

```
 SELECT pg_size_pretty(pg_relation_size('my_table'));
SELECT COUNT(*) FROM my_table;
```

Ensure table statistics are up to date:

```
ANALYZE my_table;
```

d. Locks and Contention

- Check for locks or contention from other queries:

```
SELECT * FROM pg_stat_activity;
```

- Look for long-running transactions or blocked queries.

e. Hardware/Resource Issues

- Monitor system resources (CPU, memory, disk I/O) on the database server.
- Use PostgreSQL's statistics views:

```
SELECT * FROM pg_stat_database WHERE datname = 'my_database';
```

4. Optimize the Query

a. Indexing

- Add indexes for frequently queried columns

b. Query Rewriting

- Rewrite complex queries to use subqueries, CTEs, or temp tables for better clarity and execution.
- Limit data retrieved by the query using LIMIT or OFFSET where possible.

c. Partitioning

- For very large tables, consider partitioning:

d. Avoid Unnecessary Operations

- Remove unnecessary DISTINCT, ORDER BY, or GROUP BY clauses.

5. Use PostgreSQL Tools

- **Auto-Tuning**: Use tools like pg_stat_statements to analyze query performance over time.

6. Monitor and Test

- After making changes, test the query with realistic data and workloads.
- Compare query execution times before and after optimization.

7. Scale if Necessary

If the issue is resource-related, consider:

- Upgrading hardware (e.g., more memory, faster storage).
- Implementing read replicas for read-heavy workloads.
- Using connection pooling (e.g., PgBouncer) to handle high concurrency.

Question 10

Write a Dockerfile to containerize a Laravel application.

Answer;

Optimized laravel Dockerfile:
https://github.com/belisky/lemonade_devops/blob/main/Dockerfile_Laravel.git

Multistage Optimized laravel Dockerfile:

https://github.com/belisky/lemonade_devops/blob/main/Dockerfile_laravel_multistage.git

[Section 3]

Question 11

How would you set up monitoring for the React Native mobile app's API endpoints?

Answer:

1. Defining Monitoring Objectives

Focus on:

- **Uptime**: Ensuring the API is accessible.
- **Performance**: Measure latency, response times, and throughput.
- **Errors**: Track error rates and response codes (e.g., 4xx, 5xx).
- **Usage Patterns**: Monitor the number of requests, popular endpoints, and user behaviors.

2. Setting Up Server-Side Monitoring

Using tools to monitor the APIs directly from the server:

- **Application Performance Monitoring (APM)**: Implementing tools like **New Relic**, **Datadog**, or **Dynatrace** to provide insights into API performance, database queries, and server health.
- **Logs**:
    - Using a centralized logging service like **ELK Stack** (Elasticsearch, Logstash, Kibana) or **AWS CloudWatch**.
    - Loging requests and responses, including headers, payload, and response times.
    - Including metadata such as user ID, app version, and device type for React Native-specific insights.

3. Implementing API Health Checks

- Using tools like **Postman Monitor** or **Pingdom** to schedule health checks.
- Automating checks for:
    - Endpoint availability.
    - Expected responses.
    - Authentication and authorization flows.

4. Instrumenting the React Native App

To gather client-side API metrics, instrument the React Native app for:

a. Network Interception

Using libraries to intercept API calls and capture data:

- **Axios Interceptors** (for Axios-based apps)

- **React Native Network Inspector** or custom middleware for fetch.

b. Error Reporting

Using error reporting tools like **Sentry**, **Bugsnag**, or **Crashlytics**:

- Logging API request errors and categorize them by endpoint, error type, and affected users.

c. Performance Monitoring

- Using tools like **Firebase Performance Monitoring** to track API call latency and errors.
- Adding custom traces for critical API endpoints.

5. Visualizing Metrics

- **Dashboards**:
  - Using platforms like **Grafana**, **Datadog**, or **AWS CloudWatch** to visualize metrics like response times, errors, and throughput.
  - Creating specific dashboards for endpoints critical to the app (e.g., user login, payment processing).
- **Alerts**:
  - Configuring alerts for:
    - High response times (e.g., >500ms).
    - Error rates above a threshold (e.g., >5% for 5xx responses).
    - Downtime.
  - Using tools like **PagerDuty**, **OpsGenie**, or **Slack notifications** for alert delivery.

6. Simulating User Behavior

- **Synthetic Monitoring**:
  - Using tools like **New Relic Synthetic Monitoring** or **Checkly** to simulate user interactions and monitor API behavior under realistic conditions.
- **Load Testing**:
  - Using tools like **Apache JMeter**, **k6**, or **Gatling** to identify performance bottlenecks under stress.

7. Enabling Distributed Tracing

Using distributed tracing tools like **Jaeger** or **Zipkin** to trace API requests across services. This helps:

- Identify bottlenecks across microservices.
- Measure end-to-end response times.

8. Monitoring Backend Resources

Monitoring the backend infrastructure to ensure it supports API requests:

- **Server Metrics**: CPU, memory, disk usage, and network throughput.
- **Database Performance**: Query response times, connection pool usage, and indexes.
- **Caching Layers**: Cache hit/miss rates if using tools like Redis or Memcached.

9. Enabling Real-User Monitoring (RUM)

RUM tools like **Datadog RUM** or **New Relic RUM** provide insights into:

- Actual API response times as experienced by users.
- Errors and performance issues on specific devices or regions.

10. Continuous Iteration and Improvement

- Continuously analyze logs and metrics to identify trends.
- Perform post-mortem analyses after incidents to prevent recurrence

Question 12

Explain how you would debug high latency in the Node.js microservices.

Answer;

- **Log Analysis:** Check logs for error messages and performance metrics.
- **Performance Profiling:** Use tools like Node.js profiler or Clinic.js to identify bottlenecks.
- **Resource Monitoring:** Monitor resource usage (CPU, memory) to identify potential issues.
- **Database Queries:** Ensure database queries are optimized and not causing delays.
- **Network Latency:** Check for network-related issues that may be causing high latency.

[Section 4]
Question 13
Describe a time you improved the performance of an infrastructure system. What challenges did you face?

Answer;

The Situation

While managing the infrastructure for a high-traffic e-commerce platform hosted on Amazon EKS (Elastic Kubernetes Service), we noticed a significant performance degradation during peak traffic hours. Key issues included:

- **Slow API responses**: Response times exceeding Service Level Agreements (SLAs).
- **Resource contention**: Pods frequently evicted due to insufficient resources.
- **Unstable scaling**: Delays in adding new nodes to handle increased load.

The system needed to handle traffic spikes during promotional campaigns while maintaining high availability and performance.

My Role

As a Cloud DevOps Engineer, I was responsible for diagnosing the performance issues, identifying bottlenecks, and implementing optimizations to meet SLA requirements.

The Actions Taken

1. Diagnose the Problem

- **Logs and Metrics**: Used Prometheus and Grafana to analyze CPU, memory, and disk usage. Identified pods competing for resources and some underutilized nodes.
- **Scaling Delays**: Reviewed Cluster Autoscaler logs and discovered it was slow in scaling nodes due to under-configured thresholds.
- **API Bottlenecks**: Used AWS X-Ray for tracing requests and pinpointing slow services, which were database-heavy operations.

2. Address Immediate Resource Contention

- **Pod Resource Requests and Limits**:
  - Audited resource requests/limits and optimized them to match actual usage patterns.
  - Removed over-provisioning that caused wasted resources.
- **Node Taints and Tolerations**:
  - Configured taints to prioritize critical workloads and prevent eviction during contention.

3. Optimize Scaling

- **Karpenter Integration**:
  - Replaced Cluster Autoscaler with **Karpenter**, an open-source cluster autoscaler optimized for EKS.
  - Tuned scaling thresholds to respond faster to traffic spikes and reduced scale-down delays.
- **Spot Instances**:
  - Incorporated Spot Instances for cost efficiency while ensuring reliability with a mix of On-Demand and Spot Instances.

4. Optimize Database and Application Performance

- **Query Optimization**:
  - Worked with the development team to optimize slow SQL queries identified through AWS RDS Performance Insights.
- **Caching**:
  - Introduced Redis caching for frequent queries, reducing database load by 40%.
- **Horizontal Pod Scaling**:
  - Configured Horizontal Pod Autoscaler (HPA) to scale pods based on CPU and custom metrics (e.g., request rate).

5. Stress Testing and Monitoring

- Performed load testing with k6 to validate the improvements.
- Set up detailed monitoring dashboards in Grafana to track scaling events, pod utilization, and API latency in real time.

The Challenges

1. **Lack of Immediate Visibility**: Initial metrics and logs were insufficient, requiring significant effort to instrument better monitoring.
2. **Balancing Cost and Performance**: Ensuring the platform scaled effectively without over-provisioning resources.
3. **Coordinating Changes Across Teams**: Collaborated closely with developers, database admins, and DevOps engineers to implement holistic improvements.
4. **Production Risks**: Making changes in a live environment while ensuring minimal disruption.

The Results

1. **Improved Latency**: Reduced API response times from 1.5 seconds to under 500ms during peak traffic.
2. **Scalability**: Achieved seamless scaling during promotional events, handling 3x traffic spikes without downtime.

3. **Cost Optimization**: Reduced overall infrastructure costs by 20% through better utilization and Spot Instance adoption.
4. **Better Observability**: Introduced real-time dashboards and alerts, enabling proactive issue detection.

Key Takeaways

This experience reinforced the importance of:

- Comprehensive observability for diagnosing and monitoring systems.
- Leveraging modern tools like Karpenter to simplify scaling in dynamic environments.
- Cross-functional collaboration for system-wide performance optimization.

Question 14
How do you prioritize tasks when multiple urgent issues arise?

Answer;
Assess Impact: Evaluate the impact and urgency of each issue.
Prioritize: Address the highest impact issues first.
Delegate: Delegate tasks if possible to ensure quick resolution.
Communicate: Keep stakeholders informed about progress and any changes in priorities.