Splunk Processing Language (SPL) — often just called **SPL** — is the **query language** used in **Splunk** to search, analyze, and visualize data stored in Splunk indexes.Think of SPL as **Splunk's version of SQL**, but instead of querying tables and columns, you query **machine-generated log and event data**.SPL allows you to:**Search** through large volumes of log or event data.**Filter** and **extract fields** from raw data.**Transform** and **aggregate** data (e.g., count, sum, average, stats).**Visualize** data in charts, reports, and dashboards.**Correlate** data from multiple sources

How SPL Works :- SPL commands are processed from left to right, where each command's output becomes the input for the next command — similar to a Unix/Linux pipeline (|).

4. Common SPL Commands

Command	Purpose	Example
search	Filter data	search error
stats	Aggregates (sum, count, avg)	stats count by source
timechart	Time-based graph	timechart avg(cpu_usage)
sort	Sort results	sort - count
table	Show specific columns	table host, status, message
top	Show top values	top error_code
dedup	Remove duplicates	dedup user_id
rename	Rename fields	rename response_time as latency
where	Conditional filter	where status_code=500
eval	Create calculated field	eval

1. Search Description: Filters events based on keywords or field values.
Example: index=app_logs error OR failure
Finds all events in app_logs index containing the words error or failure.

🔹 2. stats

Description: Calculates statistical summaries like count, avg, sum, etc.
Example:
index=web_logs | stats count by status

Counts the number of web requests for each HTTP status code.

🔹 3. timechart

Description: Creates time-based visualizations (trend over time).
Example:
index=api_logs | timechart count by status

Shows API traffic count over time for each status code.

🔹 4. eval

Description: Creates or modifies fields using expressions or calculations.
Example:
index=server_logs | eval response_sec = duration/1000

Converts response time from milliseconds to seconds.

🔹 5. where

Description: Filters results based on conditions (like a SQL WHERE clause).
Example:
index=db_logs | where duration > 2000

Returns only events where duration is greater than 2000 ms.

🔹 6. top

Description: Displays the most frequent values for a field.
Example:
index=system_logs | top source

Lists the top sources generating the most logs.

🔹 7. sort
Description: Sorts results by a specific field (ascending or descending).
Example:
index=web_logs | stats count by host | sort -count

Sorts hosts by highest event count.

🔹 8. dedup
Description: Removes duplicate events based on a field value.
Example:
index=user_activity | dedup user

Shows only the first event per unique user.

🔹 9. rex
Description: Extracts fields using regular expressions from event text.
Example:
index=app_logs | rex "user=(?<username>\w+)"

Extracts usernames from log messages.

🔹 10. table
Description: Displays specific fields in a tabular format.
Example:
index=app_logs | table user, status, duration

Shows only user, status, and duration columns.

🔹 11. fields
Description: Includes or excludes specific fields from results.
Example:
index=system_logs | fields host, source

Shows only the host and source fields.

🔹 12. head / tail
Description: Limits the number of results returned.
Example:
index=web_logs | head 10

Shows only the first 10 events.

🔹 13. transaction
Description: Groups related events into a single transaction (e.g., session).
Example:
index=auth_logs | transaction user startswith="login" endswith="logout"

Combines login and logout events for each user.

🔹 14. eventstats
Description: Adds aggregated values (like count, avg) to each event without collapsing rows.
Example:
index=api_logs | eventstats avg(duration) as avg_time

Adds average duration to each event as a new field.

🔹 15. lookup
Description: Enriches data by matching fields with an external CSV or lookup table.
Example:

index=network_logs | lookup ip_location ip OUTPUT location
Adds geographic location for each IP from a lookup file.

1. What is SPL in Splunk?
Answer: SPL stands for Search Processing Language.It's the query language used in Splunk to search, filter, analyze, and visualize machine data (logs, metrics, events).
Example:
index=prod_logs error | stats count by host
This query counts the number of errors on each host.

2. What is an index in Splunk?
Answer: An index is like a database in Splunk.It stores raw log data and makes it searchable.Each log event is saved under a specific index.
Example SPL:
index=security_logs
This searches only within the "security_logs" index.

3. What is the timechart command used for?
Answer: Used for time-based visualization (trends over time).
Example:index=web_logs status=500 | timechart count
Shows number of errors over time (line graph).

4. What is the eval command used for?
Answer: Used to create or modify fields dynamically.
Example: index=api_logs | eval latency=response_time/1000 | table latency
Converts milliseconds into seconds.

5. How do you search for specific error messages in SPL?
Answer: index=prod_logs "error" OR "exception"
Searches logs containing either the word error or exception.
You can also add filters: index=prod_logs "error" host=server1 earliest=-1h
➡️ Searches only from server1 in the last one hour.

6. How do you create an alert in Splunk using SPL?
Answer: You can save a query as an alert and set trigger conditions.
Example SPL for alert condition:
index=prod_logs "error" earliest=-15m | stats count | where count > 100
➡️ Trigger alert if more than 100 errors occur in the last 15 minutes.

7. How do you monitor failed logins or security violations?
Answer: index=security_logs action="login" result="failure" | stats count by user
➡️ Counts how many times each user failed to log in.

Useful SPL Shortcuts (for Support Engineers)

Task	**SPL Example**
Check all errors in last 1 hour	index=app_logs "error" earliest=-1h
Find slow APIs	`index=api_logs
Count events by source type	stats count by sourcetype
CPU usage trend	index=system_metrics metric_name=cpu_usage
Filter unique transactions	dedup transaction_id

8. Types / Categories of Commands
SPL commands are classified by how they operate (where in the pipeline they are used, how much data they process, etc.). Some broad categories:

1. Streaming commands
Operate on each event individually, passing them downstream.
Examples: eval, fields, where, rename, replace

9 Transforming / Aggregating commands
Aggregate or summarize data, often reducing the number of events.
Examples: stats, timechart, top, chart, eventstats

3. Generating commands
Commands that generate new events (not derived from the original).
Example: makeresults, inputlookup

4. Orchestrating / Control / Flow commands
Commands that control search flow or combine results.
Examples: join, append, union, subsearches ([...])

🔹 1. search
Definition: Fetches and filters events from specified indexes.
Syntax: index=<index_name> [conditions]
Example:index=web_logs status=500 host=app-server-1
Real Use Case: Troubleshoot application errors on a specific host.
Interview Q: “What is the first command in every SPL query?”
Answer: search — it filters events from indexes.
🔹 2. fields
Purpose: Include or exclude specific fields from results.
Syntax:
| fields <field1>, <field2>
| fields - <field_to_exclude>
Example:
| fields host, status, url
Use Case: Optimize searches and dashboard performance.
Interview Tip: Mention that fields early in a query improves efficiency.

🔹 3. table
Purpose: Display data in tabular form for reports/dashboards.
Example:
| table _time, host, status
Use Case: Visualize structured output for reports.

🔹 4. stats
Purpose: Performs statistical operations like count, sum, avg.
Example:
| stats count by status
Use Case: Summarize total events or requests by status code.
Interview Q: “How do you calculate total or average events?”
A: stats count, stats avg(field) etc.

🔹 5. timechart
Purpose: Show time-based trends (great for dashboards).
Example:
| timechart count by status
Use Case: Monitor error trend hourly or daily.

🔹 6. eval

Purpose: Create or modify fields using expressions.
Example:
| eval response_sec = response_time/1000
Use Case: Convert units, create labels, or conditionally modify fields.
Interview Q: “How to create a derived field?” → Use eval.

🔹 7. where
Purpose: Filter results based on logical condition.
Example:
| where status=500 AND response_time>2
Use Case: To further filter results after computation.

🔹 8. sort
Purpose: Sort events ascending/descending.
Example:
| sort -count
Use Case: Identify top hosts with most requests.

🔹 9. top / rare
purpose: Show most or least frequent values.
Example:
| top url limit=5
Use Case: Find most visited pages or rare errors.

🔹 10. dedup
Purpose: Remove duplicate events based on field(s).
Example:
| dedup session_id
Use Case: Show only unique sessions or users.

🔹 11. rename
Purpose: Rename field names for readability.
Example:
| rename response_time AS resp_time_ms
Use Case: Cleaner field names in dashboards.

🔹 12. rex
Purpose: Extract data from raw logs using regex.
Example:
| rex field=_raw "user=(?<username>\w+)"
Use Case: Extract custom values (like usernames, IDs).
Interview Q: “How do you extract custom fields?” → Use rex.

🔹 13. regex
Purpose: Filter events matching regex pattern.
Example:
| regex url=".login."
Use Case: Find URLs containing “login”.

🔹 14. transaction
Purpose: Group events into transactions (start & end).
Example:
| transaction session_id maxspan=30m
Use Case: Analyze session flow, duration, user activity.

Interview Tip: Often asked in monitoring/user session tracking scenarios.

🔹 15. join
Purpose: Combine results from two datasets.
Example:
index=orders | join order_id [search index=payments]
Use Case: Correlate related data like orders + payments.
Performance Tip: Use stats instead of join when possible — join is expensive.

🔹 16. append
Purpose: Add results from another search below current results.
Example:
index=prod_logs | append [search index=nonprod_logs]
Use Case: Compare logs between environments.

🔹 17. lookup
Purpose: Enrich data using lookup files or static datasets.
Example:
| lookup countries.csv code AS country_code OUTPUT country_name
Use Case: Add human-readable info (e.g. country name for code).

🔹 18. eventstats
Purpose: Add statistical value (avg, sum) to each event.
Example:
| eventstats avg(response_time) AS avg_resp
Use Case: Compare each row's value against overall average.

🔹 19. fillnull
Purpose: Replace null values.
Example:
| fillnull value="N/A"
Use Case: Avoid blanks in dashboards or reports.

🔹 20. chart
Purpose: Pivot-style aggregation for visualization.
Example:
| chart count over status by host
Use Case: Create pivot tables or charts.

🗒️ Quick Reference Table

Command	Category	Use Case	Interview Hint
search	Generating	Filter events	Always first in query
stats	Transforming	Aggregation	Use for counts/sum
eval	Streaming	Calculations	Derived field creation
rex	Extraction	Parse logs	Field extraction
transaction	Transforming	Session tracking	Group start-end logs
join	Transforming	Correlation	Heavy command
lookup	Enrichment	Add external info	Link static data
fillnull	Cleanup	Replace blanks	Dashboard consistency

🧩 1. Basic Log Search

Example

```
index=web_logs status=500
```

Explanation

Finds all logs in the `web_logs` index where the HTTP status code is `500` (server errors).

✅ Use case: Detect backend issues or server crashes.

🔍 2. Count Events by Status Code

```
index=web_logs | stats count by status
```

Explanation

Counts how many events occurred for each HTTP status (200, 404, 500, etc.).

✅ Use case: Quickly check error rate distribution.

⏱️ 3. Trend of Traffic Over Time

```
index=web_logs | timechart count by status
```

Explanation

Shows a time-based chart of log count, grouped by status codes.

✅ Use case: Monitor traffic spikes, identify when errors increased.

⚠️ 4. Find Top 10 IP Addresses Generating Errors

```
index=web_logs status=500 | top limit=10 clientip
```

Explanation

Lists the top 10 client IPs that caused most 500 errors.

✅ Use case: Detect problematic clients, bots, or misconfigured systems.

📈 5. Calculate Average Response Time

```
index=web_logs | stats avg(response_time) as avg_response_time
```

Explanation

Calculates the average API or page response time.

✅ Use case: Measure application performance.

🕒 6. Failed Logins in Last 1 Hour

```
index=auth_logs action="failed login" earliest=-1h
```

Explanation

Finds all failed login attempts from the past hour.

✅ Use case: Detect brute-force or unauthorized access attempts.

🧠 7. Detect Users with More than 5 Failed Logins

```
index=auth_logs action="failed login"
| stats count by user
| where count > 5
```

Explanation

Finds users who failed to log in more than 5 times.

✅ Use case: Security monitoring or lockout automation.

🕵️ 8. Track Unique Visitors

```
index=web_logs | stats dc(clientip) as unique_visitors
```

Explanation

Uses `dc()` (distinct count) to calculate number of unique IPs.

✅ Use case: Website traffic analysis.

⚙️ 9. Join Two Sources (e.g., Errors + Transactions)

```
index=errors | join transaction_id [ search index=transactions ]
```

Explanation

Joins related data from two indexes based on a shared field `transaction_id`.

✅ Use case: Correlate failed transactions with corresponding error logs.

🧾 10. Alert on CPU Usage Above 90%

```
index=system_metrics sourcetype=cpu | stats avg(usage_percent) as cpu_usage by host
| where cpu_usage > 90
```

Explanation

Finds hosts with average CPU usage > 90%.

✅ Use case: Infrastructure monitoring alert.

📬 11. Extract and Parse Fields

```
index=access_logs | rex "user=(?<username>\w+)"
| stats count by username
```

Explanation

Uses `rex` to extract the `username` field dynamically from logs.

✅ Use case: When logs don't have a structured field.

🧮 12. Combine and Compare Two Time Ranges

```
index=web_logs earliest=-1h@h latest=@h | stats count as last_hour
| appendcols [ search index=web_logs earliest=-2h@h latest=-1h@h | stats count as prev_hour ]
| eval percent_change=((last_hour - prev_hour)/prev_hour)*100
```

Explanation

Compares last hour's traffic to previous hour.

✅ Use case: Detect sudden traffic drops or spikes.

🚦 13. Find Slowest 5 Endpoints

```
index=api_logs | stats avg(response_time) as avg_time by endpoint
| sort -avg_time | head 5
```

Explanation

Finds top 5 slowest API endpoints.

✅ Use case: Performance tuning for APIs.

🔄 14. Transaction Grouping (e.g., Login Sessions)

```
index=auth_logs | transaction user startswith="login attempt" endswith="login success"
```

Explanation

Groups logs into a transaction per user login session.

✅ Use case: Measure how long each login took or detect incomplete logins.

🧰 15. Combine Multiple Conditions

```
index=web_logs (status=500 OR status=404) uri_path="/api/*"
| stats count by status
```

Explanation

Filters errors (404/500) from API routes only.

✅ Use case: Focused API error tracking.

Real-World SPL (Search Processing Language) Examples

1️⃣ search

Purpose: Find events/logs matching criteria.

Example:

index=upi_app sourcetype=transaction_logs status="FAILED"

Real-world Use:

At ABC Bank, you use this to find all failed UPI transactions during a specific time window to analyze downtime.

Interview Tip:"We use the search command to quickly isolate failed transactions or 500 errors from large log volumes."

2️⃣ stats

Purpose: Calculate metrics (count, avg, sum, etc.).

Example:

index=upi_app sourcetype=transaction_logs
| stats count by status

Use Case: You get total transactions grouped by their status (SUCCESS/FAILED/PENDING).

Used in daily SLA reports or dashboard KPIs.

Interview Tip: "We use stats count by status to measure UPI success rate and failure trends for reporting."

3️⃣ eval

Purpose: Create/transform fields dynamically.

Example:

| eval total_amount = amount * quantity
| eval transaction_type = if(amount>10000, "HIGH_VALUE", "NORMAL")

Use Case: When analyzing high-value transactions separately for fraud monitoring or RBI compliance.

Interview Tip:"I used eval to tag transactions above ₹10,000 as 'HIGH_VALUE' for audit dashboards."

4️⃣ where

Purpose: Filter data using conditions.
Example:
| where amount > 10000 AND status="FAILED"
Use Case: Identify failed high-value UPI payments to raise priority incident tickets.
Interview Tip:"We filtered failed high-value transactions to trigger alerts for VIP customers."

5 table
Purpose: Display selected fields neatly.
Example:
| table _time, user_id, transaction_id, status, amount
Use Case: For support analysis or sharing investigation data in Excel to business teams.
Interview Tip: "I used table to create clean tabular outputs for RCA reports."

6 top / rare
Purpose: Find most/least frequent values.
Example:
| top error_code
Use Case: Identify top 10 recurring error codes in failed transactions — helps prioritize fixes.
Interview Tip:"We used top error_code weekly to find recurring API issues."

7 dedup
Purpose: Remove duplicate entries.
Example:
| dedup transaction_id
Use Case: During incident analysis, remove duplicate UPI retries caused by network failures.
Interview Tip: "Dedup helped us count unique transactions after retries."

8 rex
Purpose: Extract data using regex patterns.
Example:
| rex "user_id=(?<userid>\w+)"
Use Case: Logs like user_id=abc123 action=login → extract userid dynamically.
Interview Tip: "Rex helped extract IDs from unstructured logs when fields were not indexed."

9 lookup
Purpose: Enrich logs with external CSV reference data.
Example:
| lookup branch_details.csv branch_code OUTPUT branch_name, region
Use Case: Match transaction branch_code with human-readable branch name for MIS reports.
Interview Tip: "We used lookups to add metadata (branch/region) from CSVs for analytics."

🔟 join
Purpose: Merge data from two sources.
Example: index=txn_logs | join user_id [search index=user_master]
Use Case: Combine UPI transactions with user KYC info for compliance reports.
Interview Tip:"I used join for occasional correlation between payment logs and customer master data."

11 timechart
Purpose: Create time-based trend data.
Example: | timechart count by status span=1h
Use Case: Visualize hourly success/failure trend for UPI transactions during downtime.
Interview Tip: "We used timechart to plot hourly UPI success rate trends in Splunk dashboards."

12 transaction
Purpose: Combine related logs into a single logical flow.
Example: | transaction session_id startswith="login" endswith="logout"
Use Case: Calculate total session duration per user — useful for session timeout RCA.
Interview Tip:"Transaction helped us group multiple UPI request-response pairs into one flow."

13 fillnull

Purpose: Replace missing values.
Example: | fillnull value="N/A"
Use Case: Avoid blanks when exporting Splunk data to CSV for auditors.
Interview Tip:"We used fillnull to make dashboards consistent without blank fields."

14️⃣ sort
Purpose: Sort the data ascending or descending.
Example: | sort -amount
Use Case: View top 10 highest payment transactions for fraud monitoring.
Interview Tip: "Sort helped to identify unusually large transfers in real time."

15️⃣ fields
Purpose: Keep only relevant fields (faster performance).
Example: | fields user_id, status, amount
Use Case: Reduce log size while debugging in production.
Interview Tip:"Using fields helped improve query performance during live RCA."

16️⃣ spath
Purpose: Parse JSON structured logs.
Example: | spath input=response path=transaction.status
Use Case: Extract JSON field "status" from API responses like {"transaction":{"status":"SUCCESS"}}.
Interview Tip:"Since our API logs were JSON, I used spath to parse and filter specific keys."

17️⃣ eventstats

Purpose: Add computed values per event.
Example: | eventstats avg(amount) as avg_amount
Use Case: Compare each transaction's amount to average for anomaly detection.
Interview Tip: "We used eventstats to compare real-time transaction against average benchmarks."

18️⃣ bin
Purpose: Group by time/numeric intervals.
Example: | bin _time span=5m
Use Case: Group transactions every 5 minutes for monitoring dashboards.
Interview Tip: "We used bin span=5m for short-interval UPI performance analysis."

19️⃣ filldown
Purpose: Fill missing field values from previous rows.
Example: | filldown transaction_id
Use Case:Used in data correction when partial log entries were missing transaction IDs.

20️⃣ rename
Purpose: Rename fields for readability.
Example: | rename user_id as Customer_ID
Use Case: Prepare readable dashboards for business teams.

🚀 Real-World SPL Scenario Example
Scenario:
Users complained UPI transactions were failing from 10:00–10:15 AM.
SPL Query:
index=upi_app sourcetype=transaction_logs earliest=10:00 latest=10:15
| stats count(eval(status="SUCCESS")) as Success count(eval(status="FAILED")) as Failed
| eval FailureRate = (Failed*100)/(Success+Failed)

Output Insight:
Failure rate = 22%.
Triggered P1 Incident → RCA showed one backend API timeout.
Interview Tip: "We used SPL queries to measure failure rate during an incident and validated recovery after fix."

⚙️1️⃣ Scenario: UPI Transaction Failure Spike

Problem:At 10:00 AM, the number of failed UPI transactions increased suddenly.
Approach using SPL:
index=upi_app sourcetype=transaction_logs
| timechart count(eval(status="SUCCESS")) as success count(eval(status="FAILED")) as failed span=5m
| eval failure_rate = round((failed*100)/(success+failed),2)
Analysis:
Found spike between 10:05–10:10 AM — failure rate jumped to 45%.
Then correlate with e**rror_code**:

index=upi_app sourcetype=transaction_logs status="FAILED"
| top error_code
Outcome:
Top error → ERR_TIMEOUT → backend API latency issue.
Interview Answer (How to say): "I used timechart with eval to calculate failure rate trends and correlated them with error codes to isolate a backend timeout issue."

⚙️2️⃣Scenario: API Latency Increased Suddenly
Problem:UPI Payment response time increased from 2 sec to 8 sec.
SPL Command:
index=upi_app sourcetype=api_logs
| stats avg(response_time) as avg_rt by endpoint
| where avg_rt > 5
Result: /upi/processPayment showed 8.1s latency.
Root Cause:One downstream CBS node (Core Banking System) was overloaded.
Interview Tip: "I used SPL stats avg() with where to isolate high-latency endpoints."

⚙️3️⃣Scenario: Missing Data in Dashboard
Problem: Business dashboard showing "0 transactions" for 1 hour.
Approach:
index=upi_app sourcetype=transaction_logs earliest=-1h@h latest=@h
| stats count by status
Observation:
No data → possible ingestion issue.
Checked _internal index for parsing errors:
index=_internal sourcetype=splunkd component=Metrics OR component=ParsingQueue
| timechart avg(queue_size) by component
Root Cause:
Parsing queue was full — forwarder delay.
Interview Answer: "When dashboards showed zero data, I used _internal logs to verify if ingestion delay occurred."

⚙️4️⃣Scenario: Duplicate Events in Report
Problem:Transaction reports show duplicate transaction IDs.
Approach:
index=upi_app sourcetype=transaction_logs
| dedup transaction_id
| stats count by status
Root Cause: App retried same transaction twice — code-level issue.
Interview Tip: "I used dedup in SPL to remove duplicate records and confirmed retry patterns."

⚙️5️⃣Scenario: Compare Yesterday vs Today's Volume
Goal:Find if transaction count dropped by >20%.
SPL:
index=upi_app sourcetype=transaction_logs
| eval today=strftime(_time,"%Y-%m-%d")
| stats count by today
Then compare via:
| delta count as diff
| eval drop_percent = (diff*100)/count
| where drop_percent > 20
Interview Tip: "We used delta and eval in SPL to monitor daily transaction drop alerts."

⚙️6️⃣Scenario: Alert Setup for Error Code Spike
SPL (for saved alert search):
index=upi_app error_code=*
| stats count by error_code
| where count > 100
Use Case:
Trigger alert if any error crosses threshold (say >100 per 5 min).
Integrated via Splunk Alert Actions → ServiceNow Ticket.
Interview Tip: “We created SPL-based alerts for error spikes to auto-create P2 tickets.”

⚙️7️⃣Scenario: API Timeout RCA
Problem: Random timeout errors observed in payment API.
SPL:
index=upi_app sourcetype=api_logs error_code="ERR_TIMEOUT"
| stats count, avg(response_time) as avg_rt, max(response_time) as max_rt by endpoint
Result: Max response_time = 120s for /verifyCustomer.
Root Cause: Network latency between UPI and CBS.
Interview Tip: “Using stats I analyzed timeouts by endpoint and found spikes caused by CBS slowness.”

⚙️8️⃣Scenario: Identify Top 5 Users with Most Failures
SPL:
index=upi_app sourcetype=transaction_logs status="FAILED"
| top user_id limit=5
Use Case: Identify problematic or malicious users for security investigation.
Interview Tip: “We use top to detect repetitive failures per user for fraud checks.”

⚙️9️⃣Scenario: Correlate Payment and Refund Logs
Problem:
Business asked if refunds are initiated for all failed payments.
SPL:
index=upi_payments sourcetype=txn_logs status="FAILED"
| join transaction_id [search index=upi_refunds sourcetype=refund_logs]
| stats count by transaction_id
Outcome: Some transactions missing refunds → triggered refund batch fix.
Interview Tip: “We used join between payment and refund logs to identify unprocessed refunds.”

⚙️ 🔟 Scenario: JSON Parsing from API Response
Problem: Logs stored as JSON; need to extract nested values.
SPL:
index=upi_api sourcetype=json_logs
| spath input=response path=transaction.status output=txn_status
| stats count by txn_status
Use Case: Count all transactions based on JSON key “status”.
Interview Tip: “We used spath to parse JSON and quickly categorize API responses.”

⚙️ 1️⃣1️⃣Scenario: Disk Usage Alert from Infrastructure Logs
Problem:One Splunk forwarder was filling disk rapidly.
SPL:
index=os sourcetype=df host=*
| stats max(use_percent) by host
| where max_use_percent > 80
Use Case: Trigger email alert if disk >80%.
Interview Tip: “I wrote SPL queries on OS-level logs to monitor disk and CPU metrics.”

⚙️ 1️⃣2️⃣Scenario: User Login Failure Investigation
SPL:
index=auth_logs sourcetype=login_events action="failure"
| top user limit=10
Use Case:
Detect brute-force attempts or repeated login failures.
Interview Tip: “We correlated login failures with IPs to detect potential account lockouts.”

⚙️ 1️⃣3️⃣ Scenario: Identify Source of High CPU Utilization
SPL:
index=os sourcetype=cpu_metrics
| stats avg(cpu_usage) by process_name
| sort -avg(cpu_usage)
| head 5
Use Case: Find top 5 CPU consuming processes.
Interview Tip: “We used SPL to identify CPU-heavy services before they cause outages.”

⚙️ 1️⃣4️⃣ Scenario: Application Restart Validation
Problem:
App was restarted; need to confirm if services recovered.
SPL:
ndex=system_logs message="Application started successfully"
| timechart count by host
Use Case: Confirm recovery time post-restart.
Interview Tip: “I validated app restarts through SPL logs after change deployment.”

⚙️ 1️⃣5️⃣ Scenario: SLA Compliance Check
Goal: Measure percentage of successful transactions.
SPL:
index=upi_app sourcetype=transaction_logs
| stats count(eval(status="SUCCESS")) as success count(eval(status="FAILED")) as failed
| eval success_rate=(success*100)/(success+failed)
Use Case: If success_rate < 98%, trigger an SLA breach alert.
Interview Tip: “We automated SLA checks using SPL and alerts integrated with monitoring tools.”

⚙️ 1️⃣6️⃣ Scenario: Outlier Detection in Transaction Amount
SPL:
index=upi_app sourcetype=transaction_logs
| eventstats avg(amount) as avg_amt, stdev(amount) as std_amt
| where amount > avg_amt + (2*std_amt)
Use Case: Detect unusually large transactions for fraud prevention.
Interview Tip: “We used eventstats and deviation logic for anomaly detection.”

⚙️ 1️⃣7️⃣ Scenario: Monitoring Batch Job Completion
Problem:
Daily reconciliation batch logs missing.
SPL:
index=batch_jobs sourcetype=recon_logs
| stats count by job_name, status
| where status!="COMPLETED"
Use Case: Identify failed or missing batch jobs quickly.
Interview Tip: “We used SPL to check batch status and ensured all critical jobs completed successfully.”

⚙️ 1️⃣8️⃣ Scenario: Correlate Logs Across Multiple Servers
SPL:
index=upi_app sourcetype=transaction_logs OR sourcetype=api_logs
| stats count by host, status
Use Case:
Compare transaction patterns across servers to detect node-level issues.
Interview Tip: “We correlated logs across app nodes to identify load imbalance.”

⚙️ 1️⃣9️⃣ Scenario: Error Trend Visualization for 24 Hours
SPL:
index=upi_app status="FAILED"
| timechart count by error_code span=1h
Use Case:
Create a visualization to track top errors hourly.
Interview Tip: “We used timechart-based dashboards to proactively detect repeating error trends.”

⚙️ 20️⃣ Scenario: Log Volume Anomaly Detection

SPL:

```
index=upi_app
| bin _time span=10m
| stats count as log_volume by _time
| eventstats avg(log_volume) as avg_vol, stdev(log_volume) as std_vol
| where log_volume < avg_vol - (2*std_vol) OR log_volume > avg_vol + (2*std_vol)
```

Use Case: Detect unusual log drops (ingestion issues) or sudden spikes (attacks).

Interview Tip:"We used statistical SPL analysis for proactive log anomaly detection."

1️⃣ HTTP 500 Errors — Web Application Outage

Situation: Users reported that the website was returning "Internal Server Error (500)" intermittently during peak hours.

Task: Identify which endpoints were failing and resolve the issue quickly to restore service.

Action:

- Used Splunk to filter logs:

```
index=web_logs status=500 | stats count by uri_path
```

- `index=web_logs` → search in web logs
- `status=500` → filter only server errors
- `stats count by uri_path` → count 500 errors per API endpoint
- Correlated timestamps with backend logs.

Result: Identified `/api/payment/initiate` as the failing endpoint due to DB connection timeouts. Increased DB pool size → 500 errors stopped.

Takeaway: Splunk **pinpointed the failing endpoint**, reducing MTTR and preventing customer impact.

2️⃣ Spike in API Latency

Situation: Users complained about slow response times in the `/user/profile` API.

Task: Determine which APIs were slow and when the latency spike occurred.

Action:

```
index=api_logs | timechart avg(response_time) by endpoint
```

- `timechart avg(response_time) by endpoint` → plots latency trends over time for each API
- Observed spike after the last deployment.

Result: Rolled back recent release → latency normalized.

Takeaway: Splunk enabled **quick correlation between deployment and performance issues**.

3️⃣ Repeated Failed Logins (Security Alert)

Situation: Security team noticed multiple failed login attempts.

Task: Detect suspicious users or IPs for potential brute-force attacks.

Action:

```
index=auth_logs action="failed login"
| stats count by user, src_ip
| where count > 5
```

Counted failed logins per user/IP.

Result: Identified one IP performing repeated failed logins. Blocked IP → prevented unauthorized access.

Takeaway: Splunk helps **detect security anomalies** in real-time.

4️⃣ Database Connection Errors

Situation: Some application nodes intermittently failed to connect to the database.

Task: Isolate which nodes were affected.

Action:

```
index=app_logs "DBConnectionException" OR "Connection refused"
| timechart count by host
```

Filtered DB errors, grouped by host → visually identified faulty node.

Result: Restarted DB connection pool on the failing node → issue resolved.

Takeaway: Splunk helps **isolate failing nodes quickly**, minimizing downtime.

5️⃣ Memory Leak Detection

Situation: Applications were restarting due to OutOfMemory errors.

Task: Identify the hosts experiencing memory issues.

Action:

```
index=app_logs "OutOfMemoryError" | stats count by host
```

Result: Identified one host repeatedly failing → analyzed heap dump → fixed infinite loop in code.

Takeaway: Splunk helps **detect recurring issues and prioritize root cause analysis**.

6️⃣ Scheduled Job Failure Tracking
Situation: Daily batch jobs were failing, unclear which ones.
Task: Identify failing jobs to prevent delays in reporting.
Action:

```
index=batch_logs "Job Failed" | rex "JobName=(?<job>[^,]+)"
| stats count by job
```

- Extracted job names using `rex`
- Counted failures per job.

Result: Found `ReportSyncJob` failing due to missing file → fixed dependency.
Takeaway: Splunk enables **parsing unstructured logs and identifying recurring job failures**.

7️⃣ Slow SQL Queries
Situation: Reports were running slowly.
Task: Identify top slow queries to improve performance.
Action:

```
index=db_logs duration>5000 | top 5 query
```

- Filtered queries taking >5s, showed top slowest queries.

Result: Added missing index → query time reduced by 80%.
Takeaway: Splunk helps **pinpoint database performance bottlenecks**.

8️⃣ Post-Deployment Component Errors
Situation: After a new release, multiple modules were failing.
Task: Identify which component caused the highest errors.
Action: `index=app_logs earliest=-1h "ERROR" | stats count by component`
Result: Found `OrderService` failing → rolled back that module.
Takeaway:
Splunk is useful for **post-deployment verification and quick rollback decisions**.

9️⃣ 404 Broken Links Detection
Situation: Users reported missing pages.
Task: Identify the most common broken URLs.
Action: `index=web_logs status=404 | top uri_path limit=10`
Result: Fixed outdated sitemap URLs.
Takeaway : Splunk provides **visibility into broken endpoints**, improving user experience.

🔟 Transaction Flow Trace
Situation: A user's transaction failed but issue was unclear.
Task: Trace the transaction across services.
Action: `index=* transaction_id=abc123`
Pulled all logs related to that transaction across multiple services.
Result: Identified failure in downstream payment gateway → notified vendor → transaction resolved.
Takeaway: Splunk enables **end-to-end traceability**, crucial for troubleshooting in microservices.

1️⃣1️⃣ High CPU Usage on Application Hosts

Situation:
Monitoring tool triggered alerts for high CPU usage on some app servers, causing user slowness.

Task: Identify which hosts were consuming excessive CPU and what processes were responsible.

Action:

Ran Splunk query:

```
index=system_metrics sourcetype=cpu
| stats avg(usage_percent) as avg_cpu by host
| where avg_cpu > 90
```

1. `index=system_metrics` → fetch server metrics
   - `avg(usage_percent)` → average CPU
   - `where avg_cpu > 90` → list only overloaded hosts

Correlated with application logs:

```
index=app_logs host="web02" ERROR OR WARN
```

2. Found repeated thread dumps and GC pauses.

Recovery Steps: Restarted the specific node (`web02`).Cleared old temp data and tuned JVM heap.

Result: CPU stabilized below 60%.

Prevention: Added JVM memory tuning, configured autoscaling, and created a Splunk alert when CPU > 85% for 10 mins.

12️⃣ Duplicate Order Creation

Situation: Customer support raised tickets for duplicate order entries in the database.

Task: Find whether duplicate order IDs were generated by the app and why.

Action:
```
index=order_service "Order Created"
| stats count by order_id
| where count > 1
```

`stats count by order_id` → groups events by order_id `where count > 1` → find duplicates
Found retry logic in Kafka consumer reprocessed the same message multiple times.

Recovery Steps: Stopped the consumer.Cleared duplicate records via DB script.Restarted consumer after adjusting “exactly-once” semantics.

Result: No further duplicates observed.

Prevention: Added Kafka offset checkpointing and alert in Splunk for duplicate order patterns.

13️⃣ Top Error Messages (Error Prioritization)

Situation: App logs had thousands of errors — needed to prioritize which to fix first.

Task: Identify the most frequent error messages.

Action:
```
index=app_logs level=ERROR
| top limit=10 error_message
```

`top limit=10` → shows top recurring errors

Found `"NullPointerException in PaymentService"` repeating 70% of the time.

Recovery Steps: Debugged service → missing null check for card type.Patched the code, redeployed.
Result: Error volume dropped by 70%.

Prevention: Added Splunk alert for any new error pattern >100 occurrences/hour.

14️⃣ Job Scheduler Downtime

Situation:
Reports didn’t generate at midnight — suspected job scheduler failure.

Task:
Confirm if jobs triggered and where they failed.

Action:
```
index=scheduler_logs "Job Triggered" OR "Job Completed"
| timechart count by message
```

Gaps in “Job Completed” logs indicated scheduler downtime.
Recovery Steps: Restarted the job scheduler service.Re-ran missed jobs manually.

Result: All reports generated successfully.

Prevention: Created Splunk alert when no “Job Triggered” logs are found for >30 mins.

15️⃣ Disk Space Full on Log Server

Situation: Application stopped writing logs; Splunk ingestion halted.

Task: Verify which partitions had reached threshold.

Action:
```
index=system_metrics sourcetype=df
| stats latest(use_percent) as disk_usage by mount_point
| where disk_usage > 85
```

Identified `/var/log` at 97%.
Recovery Steps: Cleared old archived logs.Extended volume by 20 GB.
Result: Logging resumed immediately.

Prevention: Splunk alert every 6 hours for disk usage >80%.

---

16 Payment Declines Surge

Situation: Sudden increase in “Payment Declined” messages seen during sales event.

Task: Confirm if issue was with our service or payment gateway.

Action:
```
index=payment_logs "Payment Declined"
| stats count by gateway
| where count > 50
```

Found 90% of declines from one third-party gateway.

Recovery Steps: Switched traffic to backup gateway.Informed vendor of outage.
Result: Payment success rate returned to normal.

Prevention: Splunk real-time alert on decline count >30/minute for any gateway.

---

17 Missing Heartbeat Logs

Situation: One microservice (notification-service) stopped responding, no alerts triggered.

Task: Check last heartbeat time to confirm outage.

Action:
```
index=service_logs source="heartbeat"
| timechart count span=5m by service_name
```

Found heartbeat stopped from 02:15 AM to 02:45 AM.
Recovery Steps: Restarted service in Kubernetes.Validated heartbeat resumed.
Result: Service restored.

Prevention: Added Splunk scheduled search to detect gaps >10 minutes in heartbeat logs and send Slack alert.

---

18 Alert Noise (Same Issue Repeated)

Situation: Multiple duplicate Splunk alerts for same database outage caused alert fatigue.

Task: Identify repetitive alert patterns.

Action:
```
index=alerts "Database Unreachable"
| stats count by host
| where count > 10
```

Found alert spam every 1 min for same host.

Recovery Steps: Updated Splunk alert configuration to suppress duplicates within 15 mins.
Result: Reduced alert noise by 80%.

Prevention: Added throttling in alert actions and linked to incident management tool (ServiceNow).

---

19 Missing API Calls Between Services

Situation: Service A triggered a request, but Service B didn’t receive it.

Task: Verify if request logs existed between timestamps.

Action:
```
index=serviceA_logs OR index=serviceB_logs transaction_id=*
| transaction transaction_id
| where eventcount < 2
```

Finds transactions not completed across both services.
Recovery Steps: Found network issue in load balancer. Restarted HAProxy.
Result: Requests started flowing.

Prevention: Splunk alert when incomplete transactions >10 in 5 mins.

2️⃣0️⃣ Application Restart During Deployment

Situation: After deployment, users saw “Service Unavailable (503)” errors.

Task: Check how long app downtime lasted during restart.

Action: `index=web_logs status=503 | timechart count span=1m`

Found downtime window of 6 minutes.
Recovery Steps: Adjusted deployment strategy to rolling restarts.Verified using same Splunk query — no 503s afterward.

Result: Zero-downtime deployments achieved.

Prevention: Automated pre- and post-deployment Splunk checks.

AWS - IAM,EC2,VPC,S3

1. AWS IAM (Identity and Access Management)

🧠 Simple Explanation:

IAM is like the **security guard** of AWS.
It **controls who can access what** inside your AWS environment.

Think of it as:

- **Users:** Employees (e.g., Dev, QA, Support)
- **Groups:** Teams (e.g., Admins, Developers)
- **Roles:** Temporary access cards for AWS services
- **Policies:** The rules written on those cards (permissions)

💡 Example:

> Suppose you’re in a project where developers need access to S3 to upload logs,
> but production support should only view them, not delete.

You can:

- Create an **IAM group "Developers"** → attach policy `AmazonS3FullAccess`
- Create an **IAM group "Support"** → attach policy `AmazonS3ReadOnlyAccess`
- Assign users accordingly.

💬 How to Explain to Interviewer:

> “In my project, we used IAM to manage access control.
> For example, we restricted EC2 termination permission only to the admin team using IAM policies.
> This prevented accidental shutdowns of production servers.”

✅ Common Use Cases:

- Limit access to critical EC2 or S3 resources.
- Generate temporary credentials for application scripts.
- Enable SSO (Single Sign-On) for team members.

💻 2. AWS EC2 (Elastic Compute Cloud)

🧠 Simple Explanation:

EC2 is just a **virtual server in the cloud**.
It’s where you **run your applications, scripts, or services** — just like your own computer, but hosted by AWS

Example:

Suppose your Java microservice runs on a Tomcat server.
Instead of a physical server, you launch an **EC2 instance**, install Java and Tomcat, and deploy your WAR file.

⚙️ Example Command:

You might connect to EC2 like this:

```
ssh -i mykey.pem ec2-user@54.201.123.10
```

💬 How to Explain to Interviewer:

"We hosted our production backend API on EC2 instances.
Once during a high traffic spike, CPU utilization went above 90%.
We logged into the EC2 via SSH, analyzed logs using Splunk (via CloudWatch logs shipping), identified a memory leak, restarted the service, and later increased instance type to t3.medium."

✅ Common Use Cases:

- Hosting applications, databases, scripts.
- Running cron jobs for automation.
- Scaling servers during high load.

🧱 3. AWS VPC (Virtual Private Cloud)

🧠 Simple Explanation:

A **VPC** is like your **own private network inside AWS**.
Think of it as your **secured office LAN**, but on the cloud.

It allows you to control:

- Who can access your servers
- Which servers can talk to each other
- Whether they're exposed to the internet or private

💡 Example:

You create a VPC with:

- **Public subnet** (for web servers — accessible via internet)
- **Private subnet** (for databases — internal only)
- **Internet Gateway** for outbound access
- **Security Groups** like firewalls.

💬 How to Explain to Interviewer:

"In our production setup, our application servers ran in the public subnet and databases in the private subnet inside a VPC.
One time, we faced API timeout issues — we checked the VPC route table and security groups and realized inbound port 443 was blocked.
After updating the rule, communication was restored."

✅ Common Use Cases:

- Isolate environments (Prod, QA, Dev)
- Secure data using private subnets
- Control inbound/outbound access with route tables & NACLs

📦 4. AWS S3 (Simple Storage Service)

🧠 Simple Explanation:

S3 is like a **Google Drive for AWS**, but **for your applications**.
You can store **files, logs, backups, images, videos, etc.**

It stores data as **objects** inside **buckets**.

💡 Example:

You can create a bucket called `my-app-logs` and store all your application log files there.
Splunk or CloudWatch can pull logs from S3 for analysis.

👩‍💻 Example CLI Command:

```
aws s3 cp app.log s3://my-app-logs/
```

💬 How to Explain to Interviewer:

"We used S3 for log storage and backup.
Whenever our EC2 instance generated daily logs, we pushed them to S3 using cron jobs.
This helped us in incident investigation — we retrieved older logs from S3 to analyze historical patterns using Splunk."

✅ Common Use Cases:

- Store logs, backups, static website files.
- Integration with CloudFront (CDN).
- Versioning and lifecycle management (auto-delete old files).

🔁 How They All Work Together (End-to-End Real-Time Scenario)

📘 Scenario:

You have a **Java-based web application** deployed in AWS.

Flow:

1. **IAM:** Controls who can access AWS services.
2. **VPC:** Provides a secure private network for your app and database.
3. **EC2:** Hosts your backend application.
4. **S3:** Stores your application logs and backups.

🧩 Real-Time Problem & Recovery Example:

Issue: Users reported the app was slow.
Step 1: Logged into AWS console → checked **CloudWatch metrics** for EC2 → saw CPU usage 95%.
Step 2: SSH into EC2, tailed logs → confirmed a huge number of API calls from one region.
Step 3: Using **Splunk** dashboards (logs from S3 → Splunk), analyzed traffic pattern — found an API loop from a buggy client app.
Step 4: Used **IAM** to revoke that client's access key temporarily.
Step 5: Scaled up EC2 instance size → CPU stabilized.
Step 6: Restored IAM access after client fix.

✅ **Result:** Application stabilized within 15 minutes.
✅ **Splunk helped** find the real cause (API loop traffic).
✅ **AWS services (IAM, EC2, VPC, S3)** helped isolate, secure, and resolve efficiently.

🧠 How to Answer in Interview (Sample Script)

"In my role as an L3 Support Engineer, I frequently used AWS services for troubleshooting.
For example, I used EC2 to host the application servers, S3 to store logs, IAM to control who can access production, and VPC to isolate environments.
Once we had a high CPU issue — I used CloudWatch and Splunk to analyze logs, found the cause, scaled EC2, and prevented similar issues by fine-tuning IAM and autoscaling policies.
This end-to-end visibility and control is why AWS and Splunk are powerful for support teams."

🧱 1. EC2 – Application Server Down (Real-Time Issue)

🧩 Scenario:

Your production API suddenly stopped responding. Users are getting "503 Service Unavailable".

⚙️ Troubleshooting Steps:

Checked application URL → timeout.
Logged into AWS → EC2 instance status = running, but CPU = 100%.

Used Splunk (logs shipped from EC2 via CloudWatch → S3 → Splunk):
index=app_logs host="prod-api01" "OutOfMemoryError"

✅ Found repeated OutOfMemoryError.

🧠 Root Cause:

A recent deployment introduced a memory leak in a background job.

✅ Resolution:

Restarted the Java service.
Increased instance type from t3.small → t3.medium.
Added CloudWatch alarm: CPU > 80% → send SNS alert to support group.

🎯 Interview Explanation:

"In production, we faced an API outage due to high CPU. I used CloudWatch to check instance metrics and Splunk to analyze application logs. Found an OutOfMemoryError causing repeated restarts. After fixing the service and increasing instance size, the issue was resolved. Later, we set CPU alarms for proactive detection."

🔐 2. IAM – Access Denied for Jenkins Deployment

🧩 Scenario:

Jenkins CI/CD pipeline failed during deployment — AccessDenied: Not authorized to perform s3:PutObject.

⚙️ Troubleshooting Steps:

Checked Jenkins job logs → S3 upload step failed.

Verified IAM role attached to Jenkins EC2 instance.

Used AWS IAM Policy Simulator to test permissions.

🧠 Root Cause:

DevOps had updated the IAM policy, removing s3:PutObject from the Jenkins role.

✅ Resolution:

Edited IAM role → added:

```
{
 "Effect": "Allow",
 "Action": ["s3:PutObject"],
 "Resource": "arn:aws:s3:::app-deployment-bucket/*"
}
```

-
- Re-ran deployment → success.

🎯 Interview Explanation:

"We had a deployment failure due to missing S3 write permission.
I verified IAM roles, used Policy Simulator, and restored the missing permission.
IAM helped us enforce least privilege and quickly identify the root cause."

🌐 3. VPC – Application Unreachable from Internet

🧩 Scenario:

Users report they can't access the production web app hosted on EC2.

⚙️ Troubleshooting Steps:

Ping and curl from inside instance → working internally.

Checked VPC security group and route tables.

Generated new access key via IAM.

Updated credentials in /home/ec2-user/.aws/credentials.

Added IAM policy for automatic key rotation alert.

Backfilled missing logs manually to S3, then reindexed in Splunk.

Verified EC2 instance running fine.

🧠 Root Cause:

Ingress rule for port 443 (HTTPS) was missing in the security group after recent hardening.

✅ Resolution:

Added inbound rule:

Type: HTTPS

Port: 443
Source: 0.0.0.0/0

-
- Application accessible again.

🎯 Interview Explanation:
“In one incident, the web app was unreachable externally.
I verified the instance and traced the issue to missing HTTPS inbound rules in the VPC security group.
Once added, traffic was restored.
This experience helped me understand how VPC controls secure communication.”

📦 4. S3 – Missing Log Files
🧩 Scenario:
Your Splunk dashboard stopped showing daily log data from EC2 servers.
⚙️ Troubleshooting Steps:
Checked cron job on EC2 that uploads logs to S3:

```
aws s3 cp /var/log/app.log s3://prod-app-logs/
```

-
- Found it failing silently due to expired IAM access key.

🧠 Root Cause:
The access key used by the log uploader script expired and was not auto-rotated.
✅ Resolution:
🎯 Interview Explanation:
“Splunk stopped receiving logs one morning.
I found the EC2 log upload to S3 failed due to expired IAM keys.
I regenerated credentials, restored logs, and implemented key rotation alerts.
S3 served as our log storage backbone for Splunk ingestion.”

⚙️ 5. EC2 – High CPU & Auto Scaling Recovery
🧩 Scenario:
During a flash sale, the website slowed down drastically.
⚙️ Troubleshooting Steps:
CloudWatch → EC2 CPU = 95%

Application logs in Splunk showed thousands of checkout API calls.

Auto Scaling group set to min=1, max=2, but scaling threshold was at 90%.

🧠 Root Cause:
Scaling delay — autoscaling rule took time to add new instance.
✅ Resolution:
Increased ASG limits → max=4.

Adjusted threshold → scale-out at 70% CPU.

Verified ELB health checks, traffic normalized.

🎯 Interview Explanation:
“Our EC2 instances were overwhelmed during a sale.
CloudWatch and Splunk helped us correlate traffic and CPU metrics.
I tuned the Auto Scaling policy and validated load balancing — which restored performance without downtime.”

🔐 6. IAM – Developer Locked Out of AWS Console
🧩 Scenario:
A developer couldn’t log in to the AWS console after password reset.
⚙️ Troubleshooting Steps:
IAM policy for developer group restricted iam:ChangePassword.

Used admin role to check CloudTrail logs → saw login attempt failures.

🧠 Root Cause:
The new password violated company’s IAM policy (minimum length 12 chars).
✅ Resolution:

Guided developer to reset password following policy.
Updated password policy documentation in Confluence.

🎯 Interview Explanation:

"IAM policies caused access lockouts for a developer due to strict password rules.
I checked CloudTrail and IAM settings, resolved the issue, and documented password requirements for future onboarding."

🌐 7. VPC – Database Connection Timeout

🧩 Scenario:

App server could not connect to RDS (database).

⚙️ Troubleshooting Steps:

- Verified database up and running.
- Pinged from EC2 → timeout.
- Checked VPC → found database in private subnet, app in another subnet with no route.

🧠 Root Cause:

Missing route between subnets (private ↔ app subnet).

✅ Resolution:

- Updated route table → added route to DB subnet.
- Allowed port 3306 (MySQL) in DB security group.
- Connection successful.

🎯 Interview Explanation:

"An API outage was due to app servers unable to reach DB.
I diagnosed routing issue in VPC and fixed subnet routes and security groups — restoring DB connectivity."

📦 8. S3 – Unexpected Storage Cost Spike

🧩 Scenario:

AWS bill increased unexpectedly due to S3 usage.

⚙️ Troubleshooting Steps:

- Used S3 Storage Lens → found old log files > 1 year.
- Some objects were duplicated by backup scripts.

🧠 Root Cause:

Missing S3 lifecycle policy for auto-deletion of old files.

✅ Resolution:

- Implemented lifecycle rule: delete files older than 90 days.
- Enabled S3 Intelligent-Tiering for cost optimization.

🎯 Interview Explanation:

"We faced a sudden cost spike in S3 storage.
I used S3 analytics to identify redundant log data and applied lifecycle policies.
This optimized our monthly storage cost significantly."

🧩 9. End-to-End Incident (Combining All 4 Services)

📘 Scenario:

API outage at 2 AM — users getting 504 Gateway Timeout.

🧭 Step-by-Step Recovery:

Splunk Check

index=prod_api_logs error OR timeout

1. Found DB connection timeout errors.
2. EC2 Check
   - Instance up but 80% memory usage.
3. VPC Check
   - Security group had a missing inbound rule for DB port (3306).
4. S3 Check
   - Verified logs in S3 to confirm timeline of failure.
5. IAM Check
   - Ensured no permission revocations for RDS role.

✅ Root Cause:

A recent security group change blocked DB access from EC2 to RDS.

✅ Resolution:

- Added proper rule in VPC security group.

- Restarted service.
- Verified connectivity via Splunk logs and application test.

✅ Outcome:
System restored in 15 mins; later added change review step for VPC edits.

🧠 How to Answer in Interview:

"In production support, we used AWS and Splunk together daily.
AWS gave us the infrastructure visibility — EC2 metrics, S3 storage, IAM access, and VPC routing — while Splunk gave us log-level insights.
Combining both, we could identify issues like high CPU, missing permissions, routing errors, and even optimize costs.
These tools together helped us reduce incident resolution time and improve reliability."

AWS Service	Purpose	Key Components	Real-World Analogy
IAM	Controls who can access what	Users, Roles, Policies	Security guard of AWS
EC2	Runs your applications	Instance, AMI, EBS	Virtual server/computer
VPC	Creates secure private networks	Subnets, Route Tables, IGW	Office network (LAN)
S3	Stores data (files/logs)	Buckets, Objects, Lifecycle	Google Drive for AWS

"AWS provides core building blocks for cloud infrastructure.
IAM handles identity and access, ensuring only authorized users or services can access resources.
EC2 provides compute capacity to host applications.
VPC creates a secure networking environment for those instances.
S3 offers scalable storage for logs, backups, and data.
Together, they form the foundation of most AWS architectures.
In my experience, I've used these services daily for troubleshooting production issues, managing access, storing logs, and maintaining application uptime."

Incident Management:-

- Incident — An unplanned interruption or reduction in the quality of an IT service (e.g., website down). The goal is to restore service fast.
- Problem — The underlying cause of one or more incidents. The goal is identify root cause and prevent recurrence.
- Change — Any modification to infrastructure, code, config, process or docs (e.g., apply patch, DB schema change). The goal is controlled, auditable, low-risk implementation.

Term	Meaning
SLA (Service Level Agreement)	Timeframe to resolve incidents (e.g., P1 → 1 hour).
MTTR (Mean Time to Repair)	Average time to restore service.
Workaround	Temporary fix until permanent resolution.
PIR (Post Incident Review)	Analysis meeting after major incidents.
RCA (Root Cause Analysis)	Process to find why the issue happened.

Problem Ticket	Raised if similar incidents keep recurring.

1. **Incident Raised by User / Monitoring Tool**
   - Ticket lands in L1 queue.
2. **L1 Support**
   - Performs initial triage.
   - Checks logs, restarts service if needed.
   - If not resolved → escalates to **L2**.
3. **L2 Support**
   - Deep dive analysis: validate logs, DB, application behavior.
   - Fix if possible (e.g., run script, clear cache, adjust config).
   - If issue found in **application code**, escalate to **L3**.
4. **L3 Support**
   - Debugs code, identifies root cause.
   - Provides a **permanent fix (hotfix / patch)**.
   - Works with L2 for deployment validation and RCA documentation.

Level	**Action**
L1	Checks server status → service running fine. Escalates to L2.
L2	Reviews logs → finds intermittent DB timeout errors. Tries increasing DB connection pool → temporary relief. Still failing occasionally → escalates to L3.
L3	Reviews code → finds that connection isn't properly closed in one thread → memory leak over time. Fixes code, tests, deploys patch. RCA documented.
L2	Monitors after patch → confirms stability. Updates ticket and closes incident.

Criteria	**Level 2 (L2)**	**Level 3 (L3)**
Primary Responsibility	Handles functional and technical issues not resolved by L1. Focus on application behavior and configuration.	Handles deep-rooted or code-related issues that require developer intervention.
Technical Depth	Good understanding of the application logic, middleware, databases, and logs.	Full understanding of source code, APIs, architecture, and frameworks.
Typical Actions	- Reproduce issues- Analyze logs (App / Server)- Restart services- Execute SQL queries- Modify configuration files- Apply known scripts/fixes- Raise bugs to L3	- Debug code- Identify and fix code-level defects- Develop hotfix or patch- Optimize code performance- Handle major incident RCA- Participate in release or build deployment
Access Level	Application & database read access, sometimes limited write access	Full access to source code, repository, build & deployment pipelines
Tools Used	Application monitoring (e.g., Dynatrace, Splunk, AppDynamics), SQL clients, log analyzers	IDEs (Eclipse, IntelliJ), version control (Git), CI/CD tools (Jenkins), debugging tools
Escalation Criteria	If the issue needs a code change or product defect fix, escalate to L3	If the issue is caused by infrastructure, configuration, or data, it can go back to L2 after code fix
Root Cause Analysis (RCA)	RCA at application level (e.g., wrong config, job failed, DB timeout)	RCA at code or architectural level (e.g., null pointer, memory leak, API logic bug)
Example Issues	- API returns 500 error due to bad configuration- Batch job failed due to DB deadlock- Queue backlog in Kafka- Restarting failed microservice	- API throws NullPointerException in code- Memory leak in service- Incorrect business logic in calculation module- Thread contention bug
Ownership	Maintains uptime, stability, and performance of production	Maintains codebase, enhancements, and bug fixes

In Application/Production Support, Incident Management means identifying and resolving unplanned issues that impact production. to restore normal service as quickly as possible with minimum business impact. For

example, if users report payment failures, Splunk alerts confirm 500 errors from the API. We immediately log a P1 incident in ServiceNow, join a bridge call, and analyze logs to identify the root cause. Our goal is to restore service as quickly as possible — even with a temporary workaround — and communicate updates to stakeholders. Once service is stable, we perform RCA and link the incident to a Problem ticket for a permanent fix. This structured approach minimizes downtime and improves reliability."

Why Incident Management is Important

- Ensures **minimum downtime** for business applications.
- Helps teams **respond quickly and systematically** to issues.
- Provides **clear communication** between technical teams and business users.
- Creates **audit trail** for future analysis and RCA (Root Cause Analysis).
- Reduces **repeated issues** by identifying trends.
- Restore normal service ASAP (minimize MTTR).
- Communicate status to stakeholders.
- Triage to the right team (L1 → L2 → L3).
- Capture data for later Problem Management / RCA.

What is it? A process to *detect, record, classify, escalate, resolve and close* incidents with minimal business impact.

Priority	Description	Example
P1 (Critical)	Complete outage of a production system.	Application is down for all users.
P2 (High)	Major functionality broken, limited workaround.	Payments failing for 50% of users.
P3 (Medium)	Minor impact, workaround available.	UI bug, data delay, slow response.
P4 (Low)	Cosmetic or non-urgent issues.	UI color issue, small typo.

- **Priority P1** — Critical, full outage, business-critical: immediate 24x7 response.
- **Priority P2** — Major degradation, partial outage: high-priority response during business hours.
- **Priority P3** — Minor functionality impacted: fix in normal SLAs.
- **Priority P4** — Cosmetic / no impact: low priority.

PART 1: What is Level 1 (L1) Support?

- **Level 1 (L1)** is the **first line of defense** in IT support or service desk operations.
- Their job: **receive**, **log**, **categorize**, **prioritize**, and **resolve** common and low-complexity issues.
- If an issue is too technical or critical → they **escalate** to **L2** or **L3**.

Flow of How a Ticket Comes to L1

Stage	Step	Explanation	Example
1	**User faces a problem**	**End user (employee or customer) faces an issue.**	**"VPN not connecting."**
2	**User reports issue**	**User reports via portal, email, chat, or phone call.**	**Email to `ithelpdesk@company.com` or via ServiceNow portal.**
3	**Ticket automatically created**	**The ITSM (IT Service Management) tool (like ServiceNow, Jira, Freshservice) logs it as a ticket with a unique ID.**	**Ticket ID: *INC0012345* created automatically.**

4	**Categorization & Assignment**	**The system or shift lead assigns it to the L1 support queue (based on category like "Network," "Access," "Email").**	**Category: *Network Issue* → *L1 Network Team*.**
5	**L1 Analysis**	**L1 checks logs, error messages, or monitoring tools to see where the problem originates (user system, network, or backend).**	**L1 checks VPN gateway status or user credentials.**
6	**Resolution or Escalation**	**If L1 can solve → resolves. If not → escalates to L2 or L3.**	**L1 resets credentials → VPN works again.**
7	**Ticket Closure**	**User confirms issue fixed → L1 closes ticket with resolution notes.**	**"Resolved: VPN credentials reset, user verified connectivity."**

How L1 Knows "From Where the Issue is Coming"
L1 uses 4 main sources to identify the root area of issue:

1. Ticket details / description — user's own words (error messages, screenshots).
2. Monitoring tools — like Nagios, SolarWinds, Grafana (show if server/network down).
3. Knowledge Base (KB) — pre-defined solutions or troubleshooting steps.
4. Incident categorization — tells which system/service the issue belongs to.

👉 So, if user says "Email not working," L1 checks:

- Is it a local Outlook issue?
- Is Exchange server down?
  Are credentials expired?

Examples of Incidents L1 Can Solve (with Explanation + Use Case)

---

1. Password Reset / Account Locked
Scenario: User can't log in to system.
Flow:

- User emails helpdesk: "Can't log in to portal."
- Ticket auto-created in "Access Management" queue.
- L1 checks Active Directory → sees account locked.
- L1 unlocks it / resets password.
- User logs in successfully → ticket closed.

✅ Solved by L1 because it's procedural and has predefined steps.

---

2. VPN Connection Not Working
Scenario: Remote employee can't connect to office VPN.
Flow:

- Ticket created under "Network → VPN."
- L1 checks VPN gateway status (if online via monitoring tool).
- If VPN OK → resets user's VPN profile or credentials.
- If multiple users affected → escalates to L2 (network).

✅ L1 knows issue source by comparing *user issue* + *tool status*.

---

3. Email Not Syncing (Outlook Issue)
Scenario: User's Outlook not syncing new emails.
Flow:

- L1 asks user to restart Outlook, clear cache, check server status.
- If still failing, L1 reconfigures Outlook profile.

  ✅ Solved if user-specific; escalated if mail server issue.

---

4. Printer Not Printing
Scenario: Printer in office not responding.
Flow:

- Ticket in "Hardware → Printer."
- L1 checks printer online status from admin panel.

- L1 restarts print spooler service or reconnects printer IP.

  ✅ L1 resolves unless printer hardware physically damaged.

---

5. Application Access Request
Scenario: User needs access to SAP or internal tool.
Flow:

- User raises “Access Request.”
- L1 verifies manager approval → creates access via AD or portal.

  ✅ Solved by L1 because process-driven.

---

6. Software Installation Request
Scenario: User needs Notepad++ installed.
Flow:

- Ticket created → “Software Request.”
- L1 validates software approved → installs remotely via SCCM.

  ✅ Solved by L1; escalated if license or permission issue.

---

7. Disk Space Full on User’s Laptop
Scenario: “System showing low disk space.”
Flow:

- L1 connects remotely → clears temp files, old logs, recycle bin.
- Runs Disk Cleanup or removes old profiles.

  ✅ L1 resolves without escalation.

---

8. Website Not Loading (User Complaint)
Scenario: “Company site not opening on my PC.”
Flow:

- L1 checks:
  - If site loads on other systems (to isolate user machine or DNS issue).
  - Runs `ping` and `tracert`.
  - Clears browser cache or resets DNS (`ipconfig /flushdns`).

    ✅ Solved if local issue; escalated if site down globally.

Complete Example Use Case (Word-by-Word Flow)

Step	Detailed Explanation
1. User Problem	John, a remote employee, tries to connect to office VPN — gets error: “Authentication failed.”
2. Ticket Creation	John sends email to ithelpdesk@company.com. ITSM tool (ServiceNow) automatically creates INC0023411.
3. Ticket Categorization	Tool categorizes under “Network → VPN.” Assigned to L1 Network Team.
4. L1 Receives Ticket	L1 engineer reads ticket: checks logs and user’s last login time in VPN dashboard.
5. Identify Source	L1 sees that user credentials expired yesterday → cause identified.
6. Action	L1 resets John’s credentials and shares new password securely.
7. Verification	John con firms VPN connected successfully.
8. Closure	L1 updates resolution notes: “Reset VPN credentials, user verified connection.” Ticket marked Resolved.

Category	Example	L1 Action	Tools Used	Escalate If

Access	Password Reset	Unlock/reset	AD / Okta	Persistent lockout
Network	VPN issue	Credential reset	FortiClient, Cisco AnyConnect	Gateway down
Email	Outlook issue	Reconfigure profile	Outlook console	Exchange down
Hardware	Printer issue	Restart service	Printer admin	Hardware failure
Access	App access	Grant via AD	AD, IAM	Role issue
Software	Install request	Install via SCCM	SCCM, Intune	License issue
System	Disk full	Clean temp data	RDP, remote tools	Hardware issue
Web	Website down	Flush DNS, clear cache	Browser, CMD	Server issue

1 knew “from where the issue came” by:

- Error message (“Authentication failed”)
- VPN tool showing “Account expired”
- Comparing logs of other users (only one affected)
- **What Is Level 2 (L2) Support? :-** Level 2 (L2) support is the second line of technical defense in IT or production support.
  They are experienced technical specialists who handle incidents that L1 cannot resolve due to complexity, configuration depth, or system access limitations.

⚙️ **Theory – How L2 Fits in Support Hierarchy**

Level	Who They Are	Role Summary
L1 (Level 1)	Service desk / frontline	Handles basic, repetitive issues and triage
L2 (Level 2)	Technical support engineers	Handles moderately complex technical issues, configuration, and system-level troubleshooting
L3 (Level 3)	Developers / system architects	Fix deep code issues, bugs, or design problems

So:

- **L1 = “First responder”**
- **L2 = “Technical fixer”**
- **L3 = “Product expert / developer”**

Example Use Case: Web Application Slow

Step	What Happens	Who Handles
1	User reports: “Application very slow”	L1 receives ticket
2	L1 checks — user’s network OK, browser cache cleared	L1 tries basic fix
3	Issue persists — escalates to L2 (Application Support)	Escalation
4	L2 checks server logs → finds high CPU usage on backend server	L2 analysis

5	L2 restarts one service and optimizes thread pool size	L2 fix
6	Confirms performance restored	Verification
7	Adds RCA: “CPU spike due to heavy query load — optimized configuration”	RCA

Daily Activities of L2 (in Theory)

1. **Review escalated tickets from L1.**
2. **Check system dashboards and error logs.**
3. **Participate in change deployments.**
   Resolve moderate to complex incidents.
4. **Update documentation and knowledge base.**
5. **Escalate unresolved code issues to L3.**
6. **Perform preventive checks and tuning.**
7. **Coordinate with infrastructure, network, or DB teams if needed.**

In Simple Words

L1 = “User-facing quick fixers”
L2 = “Technical detectives and fixers”
L3 = “Engineers who change the product itself”

L2 = *Finds root cause and fixes operational/configuration issues*

- **L3 = *Finds product or code-level root cause and fixes the actual system/software itself***

In simple terms:

- **L2 fixes the *environment***
- **L3 fixes the *product or code base***

Area	Level 2 (L2)	Level 3 (L3)
Main Role	Technical troubleshooting & root cause analysis of known systems	Deep engineering — analyze software code or system design
Access Level	Application / server configuration access	Source code, build systems, architecture
Skill Type	System admin, middleware, database, networking, app support	Developers, architects, or product engineers
Fix Type	Fixes environment, restart services, tune configs, database queries	Changes actual code, logic, API, or data model
Example Root Cause	“Server memory leak due to high JVM heap usage.”	“Bug in Java method causing memory leak — fix in code.”
Tools Used	Splunk, Kibana, Grafana, SQL tools, Jenkins	IntelliJ, Eclipse, GitHub, Jira, IDEs
Documentation	Writes KB articles, support runbooks	Writes code fixes, release notes, patches
When Involved	When L1 cannot fix (technical)	When L2 cannot fix (development needed)
Response Focus	Quick technical recovery (restore service)	Permanent fix in product code
Example Team Name	“Application Support” / “Ops Support”	“Engineering” / “Development” / “R&D”

Imagine a company uses a banking app.Even though both perform "root cause analysis,"the depth is very different.

Role	Example Action
L1	User says "App not loading." L1 asks user to clear cache → issue still persists.
L2	L2 logs into app server → sees repeated "500 Internal Server Error." Checks logs → finds database query timing out → applies temporary fix (restart DB connection pool).
L3	Developers investigate the code and discover that the SQL query has a logic bug causing infinite loops. They modify the code, test it, and release a new version of the application.

L2 finds *technical root cause* (external factors).
L3 finds *code root cause* (internal logic errors)

Email Application Crashing

Step	What Happens	Who Handles
1	Users report that sending attachments crashes email app.	L1 logs ticket.
2	L2 checks logs → finds crash happens when file size > 5 MB.	L2 identifies pattern.
3	L2 tries configuration fix (increase upload limit to 10 MB). Still crashes.	Temporary workaround fails.
4	L2 escalates to L3 with logs and test case.	Escalation.
5	L3 developer checks code → finds bug in attachment upload function (buffer overflow).	Code-level RCA.
6	L3 updates code, builds new version, deploys via release team.	Permanent fix.
7	L2 validates fix in production and closes incident.	Verification.

L2 – knows *why* something failed, L3 – knows *how* to make sure it never fails again (by fixing the product itself).

Aspect	Level 2	Level 3
Nature of Work	Operational troubleshooting	Development / Engineering
RCA Depth	Surface to configuration level	Code and architecture level
Resolution Scope	Environment, configuration, tuning	Source code fix, product upgrade
Example	Fix database pool settings	Fix SQL query in Java code
End Deliverable	Stable service	Permanent software fix

Example 1 — Application Service Down (OutOfMemory Error)

1. Application Service Down (OutOfMemoryError)

Issue Title Meaning: "Application service down" means the main running process (Java/Python/Node app) on the server has stopped responding or crashed — users or other systems calling its API get HTTP 503 Service Unavailable.

Behind the scenes:
Every application runs on a JVM or process with a fixed memory allocation (heap).
When too many requests or memory leaks occur, the service exceeds heap limit, JVM crashes automatically to prevent server hang.
Monitoring tools (like AppDynamics, Grafana, or Splunk) detect this and raise an alert.
Incident Summary: PaymentService API down — users unable to make UPI payments.
Ticket Source: Escalated from L1 after monitoring alert “HTTP 503 Service Unavailable”.
Symptoms:Service down; unable to restart via GUI.
How L2 Recognizes It:
Alert from monitoring: “Service unavailable (503)” or “Port not responding”.
Logs show OutOfMemoryError, confirming memory exhaustion.
Why It’s Critical:
Core business service unavailable = transaction loss.
Needs immediate restart and preventive memory tuning.
Actions Taken (L2):
1. Logged into server via SSH.
2. Checked service status:
systemctl status paymentservice
→ shows “failed”.
3. Checked application logs:
java.lang.OutOfMemoryError: Java heap space
4. Cleared cache, increased Java heap size, restarted service.
RCA: Application consumed more memory than allocated (2GB limit). During peak load, heap memory exhausted → JVM crashed.
Resolution:
Increased heap size from 2GB → 4GB.
Cleared temporary cache to free memory.
Restarted the service successfully.
Verification & Closure:
Post-restart, API health restored, monitoring showed green.
Ticket closed after 2 hours observation.
Why L3 Not Involved:
No code change needed, only configuration tuning.

One day during the peak business hour, we received multiple alerts from our monitoring tool AppDynamics stating — ‘PaymentService API is down — HTTP 503 Service Unavailable’.At the same time, L1 team also confirmed that users were unable to make UPI transactions, and the issue was escalated to L2.The first thing I did was connect to the application server through SSH and check the

service status using:
systemctl status paymentservice.The status showed ‘failed’, so I immediately checked the application logs located at /opt/app/paymentservice/logs/.
From the logs, I found repeated errors:
java.lang.OutOfMemoryError: Java heap space
That clearly indicated that the JVM heap memory had been exhausted.
I verified the heap allocation in the startup script and noticed it was still set to 2GB, which was insufficient considering the current transaction volume.

So, I performed these steps:
1. Cleared temporary cache files from /opt/app/paymentservice/cache/.
2. Updated the startup configuration by increasing the heap size to 4GB:
export JAVA_OPTS="-Xms2048m -Xmx4096m"
3.Restarted the service:
systemctl start paymentservice

4. Checked the logs again to ensure there were no errors.
5. Verified API health through Postman and the load balancer endpoint — received 200 OK.
After restart, the API was stable, and monitoring turned green.We observed the service for 2 hours to confirm stability before closing the ticket.The Root Cause was memory exhaustion due to higher-than-expected concurrent transactions and unoptimized cache usage.As a preventive action, we increased the heap permanently in configuration and suggested the application team (L3) to implement cache cleanup logic in the next release.So in summary, it was an L2-resolved incident — no code change required, but log analysis, heap tuning, and quick service restart restored functionality and avoided downtime.

🔹 **Example 2 — Batch Job Failure (DB Lock Error)**

Batch Job Failure (DB Lock Error)
Issue Title Meaning:
A batch job is a scheduled background process (run via Control-M, Autosys, etc.) that executes SQL or application scripts for daily settlements, reconciliation, etc.
“DB lock” means one database session is holding exclusive access to a table/row — so the job trying to update it can’t proceed and fails.
Behind the Scenes:
Oracle and other DBs use row-level locks to ensure data consistency.
When one session is still holding a lock, another can’t modify same row.
The batch job fails with:
ORA-00054: resource busy and acquire with NOWAIT specified.
How L2 Identifies:
From job logs → finds DB error code.
Runs SQL queries to identify blocking session.
Terminates blocking session safely.
Impact:
Batch job failure = incomplete day-end processing.
Must be fixed before next cycle.
Incident Summary:
Nightly job EOD_TXN_LOAD failed in Control-M.
Ticket Source:
Auto-generated Control-M alert.
Symptoms:
Job failed with: ORA-00054: resource busy and acquire with NOWAIT specified
Actions Taken (L2):
1. Logged into DB and identified blocking session:
SELECT sid, serial#, blocking_session FROM v$session WHERE blocking_session IS NOT NULL;
2. Found blocking session from report query.
3. Coordinated with reporting team and killed blocking session:
ALTER SYSTEM KILL SESSION '1234,5678';
4. Re-ran job; completed successfully.
RCA: Another user report query locked the same table → EOD job couldn’t update records → job failed.
Resolution: Terminated blocking session, reran the job.
Verification & Closure: Batch ran successfully and reconciled data.
Why L3 Not Involved: No application defect — pure DB lock issue.

Example 2: Batch Job Failure (Database Lock)
You can explain it like this:
“Another example I’d like to share is about a batch job failure incident I handled.”
During our end-of-day processing, a Control-M job named EOD_TXN_LOAD failed with an Oracle error:
ORA-00054: resource busy and acquire with NOWAIT specified

The Control-M alert was automatically assigned to the L2 batch support queue.As an L2 engineer, I started by checking the job logs in Control-M and confirmed that the job had failed while trying to update the TRANSACTION_SUMMARY table.To diagnose further, I logged into the Oracle database using SQL*Plus and ran the below query:
SELECT sid, serial#, blocking_session FROM v$session WHERE blocking_session IS NOT NULL;
This showed that another session, belonging to a reporting application, was holding a lock on the same table.I coordinated with the reporting team and confirmed that their ad-hoc report was still running.
Once their report completed, I executed:
ALTER SYSTEM KILL SESSION '1234,5678';
Then, I manually re-ran the Control-M job using:
ctmorder <jobname>
This time it completed successfully.
After that, I validated the batch output and downstream data reconciliation — everything matched as expected.The Root Cause was a table-level lock caused by a long-running report query overlapping with the EOD batch schedule.As a preventive step, we adjusted the report job schedule to run after the batch window and requested a proper NOWAIT retry logic to be implemented by the L3 developer team.This issue was handled completely by L2 — no code fix was required, only a database session cleanup and coordination between teams.

🔹 **Example 3 — API Timeout (Slow DB Query)**

Issue Title Meaning:
An API timeout means when an application sends a request to a backend service or DB, the response takes longer than expected (beyond timeout threshold like 5s).
So user sees failure or slowness in payment/transaction.
What Happens Inside:
Each API internally executes multiple SQL queries or external service calls.
If any of these take too long (due to missing index, large data, or network latency), the entire API call crosses timeout.
Monitoring tools like New Relic, Datadog show spikes in latency.
How L2 Identifies:
Looks at logs → “TimeoutException” or “Query took 10 sec”.
Uses DB tools to check top slow queries (v$sql view in Oracle).
Finds missing index or inefficient query.
Business Impact: User-facing delay, payment API failing.
Critical SLA breach if not resolved quickly.
Incident Summary: High latency observed in UPI payment API.
Ticket Source: Escalated by L1 from monitoring tool (response time > 5 sec).
Symptoms: Timeouts from payment gateway integration.
Actions Taken (L2):
1. Checked API logs → found slowness in “FetchTransaction” DB query.
2. Checked DB:
SELECT sql_id, elapsed_time FROM v$sql ORDER BY elapsed_time DESC;
→ Query took >10 sec.
3. Found missing index on txn_id column.
4. Created temporary index:
CREATE INDEX idx_txnid_temp ON transaction(txn_id);
5. Latency reduced to <1 sec.
RCA: New large data load increased table size; missing index caused full table scan → performance degradation.
Resolution:Added temporary index to optimize query.
Verification & Closure:Response time normalized.Suggested permanent fix to L3 via problem/change request.
Why L3 Not Involved: No code change — only DB optimization by L2.

Example 3: API Timeout Issue (Database Slowness)

How to say in interview:

“One of the incidents I resolved as an L2 engineer was an API timeout issue during UPI payments.”We received a ticket from L1 after multiple user complaints — “Transaction taking too long, sometimes failing with 504 Gateway Timeout.”The monitoring tool also showed API latency above the SLA threshold.I started by checking the API gateway logs and backend service logs.In the logs, the request was hitting a query fetching customer transaction details.The execution time in logs was showing around 18 seconds, whereas the timeout limit was 10 seconds. So I connected to the Oracle DB and executed the same query manually:
SELECT COUNT(*) FROM txn_table WHERE cust_id='12345';

It took around 17 seconds. Then I ran EXPLAIN PLAN and saw that the query was doing a full table scan because there was no index on cust_id. Since it was peak business time, I didn’t make schema changes (that’s L3 responsibility).
As a temporary L2 fix, I:
Cleared the DB buffer cache.Restarted the connection pool to free up stale sessions.Added a hint in the config for shorter fetch size (reducing round trips).After restarting the service, response time improved to 3–4 seconds.Then I raised a Problem ticket for L3 suggesting creation of an index on cust_id column.
🧠 Root Cause: Poor query performance due to missing index.
🛠️ Resolution: Cleared cache, optimized connection pool.
📋 Preventive: L3 created proper index and optimized query.

🔹 **Example 4 — Disk Space Full (Log Rotation Failed)**
Disk Space Full (Log Rotation Failure)

I*ssue Title Meaning:*
Servers have limited disk storage for logs and temp files.
When “Disk space full” alert appears, it means:Application or OS can’t write new data/logs.Services may crash due to lack of space.
What Happens Internally: Application generates log files daily.A log rotation utility (like logrotate) compresses & archives old logs.If that utility fails or permission issues occur, logs keep growing endlessly.Finally, disk reaches 100%.
How L2 Recognizes: Monitoring alert from Nagios/Zabbix → “Filesystem 95% full”.Checks which file or directory consumed most space (du -sh * | sort -rh).Identifies old logs, misconfigured rotation.
Impact:Application can stop due to “No space left on device”.Immediate cleanup required to prevent outage.
Incident Summary: Server disk usage reached 95%.
Ticket Source: Monitoring tool alert (Nagios).
Symptoms: Services started throwing “No space left on device” errors.
Actions Taken (L2):
1. Checked:
du -sh * | sort -rh | head -10
Found 50GB old_transaction.log.
2. Verified logrotate.conf misconfigured (rotation disabled).
3. Compressed & archived old logs:
gzip old_transaction.log
mv old_transaction.log.gz /opt/app/archive/
4. Fixed logrotate config and restarted logrotate.
RCA: Log rotation cron job failed due to incorrect permission → logs not rotated → disk full.
Resolution:Manually archived old logs, corrected permissions, restarted logrotate service.
Verification & Closure: Disk usage reduced from 95% → 45%, alert cleared.
Why L3 Not Involved:Infra/configuration issue, not application code.
Example 4: Disk Space Full on Application Server
How to say in interview:
“One incident was related to disk full issue on our production server.”
Monitoring tool Nagios raised an alert — ‘Disk usage 95% on /opt/app’.
Ticket was auto-assigned to L2 support queue.

First, I connected to the server using SSH and ran:
df -h
It showed /opt/app partition was 97% full.
Then I checked the largest directories:
du -sh /opt/app/* | sort -h
Found that /opt/app/logs was consuming ~15GB.Inside /opt/app/logs, there were old rotated logs not purged automatically because the logrotate script had failed earlier.
So I:
1. Took backup of the last 7 days logs.
2. Removed old .gz files older than 15 days:
find /opt/app/logs -name "*.gz" -mtime +15 -exec rm -f {} \;
3. Re-ran logrotate manually and confirmed it succeeded.4. Freed around 12GB space.
Finally, updated monitoring dashboard — disk usage dropped to 45%.
Service stability verified successfully.
🧠 Root Cause: Logrotate cron job failure → accumulation of old logs.
🛠️ Resolution: Manual cleanup + fixed cron permission issue.
📋 Preventive: Added monitoring for logrotate success alerts

Example 5 — MQ Queue Stuck (Listener Failure)

MQ Queue Stuck (Listener Failure)
Issue Title Meaning:
In enterprise apps, MQ (Message Queue) systems like IBM MQ or RabbitMQ carry messages between applications asynchronously.A “Queue stuck” issue means messages are piling up but not consumed by the backend listener service.
Internally: One app (producer) sends messages to MQ.Another app (consumer/listener) reads and processes them.If listener goes down or loses connection, queue keeps filling → transaction delays.
How L2 Identifies:
Monitoring tool shows “Queue depth high”.
Checks MQ depth using commands like:DISPLAY QLOCAL('PAYMENT.IN.QUEUE') CURDEPTHReviews listener logs — finds “Connection reset” or “Unable to connect”.
Impact:Messages delayed → payment status not updated → business delay.
Incident Summary: Messages stuck in queue “PAYMENT.IN.QUEUE”.
Ticket Source: Monitoring alert from MQ tool (queue depth > threshold).
Symptoms: Queue depth increasing; no messages processed.
ACtions Taken (L2):
1. Checked queue depth:
echo "DISPLAY QLOCAL('PAYMENT.IN.QUEUE') CURDEPTH" | runmqsc QM1
CURDEPTH = 5000.
2. Checked listener logs → JMS Connection reset.
3. Restarted listener service:
systemctl restart payment-listener
4. Monitored queue depth decreasing.
RCA: Listener process lost connection to MQ manager due to network glitch.
Resolution: Restarted listener to re-establish connection.
Verification & Closure: All pending messages processed, depth 0.
Why L3 Not Involved: No defect — runtime listener restart resolved it.

🔹 **Example 6 — Incorrect Transaction Status (Data Sync Issue)**

Wrong Data in Table (Transaction Pending)
Issue Title Meaning: Sometimes a user transaction completes successfully (money debited), but the status in DB remains “Pending” or “Failed”.This happens due to data synchronization issues.

Inside Story: The transaction flow involves multiple systems: app → gateway → core banking.When a callback/response is delayed due to network lag, the app doesn't receive the success signal in time.Hence DB record stays "Pending".
How L2 Identifies: User complaint comes via L1.L2 checks logs, DB record, and confirms callback received late.Cross-verifies payment gateway or core system logs.
Impact: User confusion, financial discrepancy risk.Needs immediate data correction.
Incident Summary: User reported successful transaction showing "Pending" on UI.
Ticket Source: Escalated from L1 (user ticket).
Symptoms: Status mismatch between core DB and payment gateway.
Actions Taken (L2):
1. Checked txn record:
SELECT * FROM txn_table WHERE txn_id='12345';
Status = 'P'.
2. Checked gateway logs → callback success received late.
3. Verified transaction completed successfully.
4. Updated record manually:
UPDATE txn_table SET status='S' WHERE txn_id='12345';
COMMIT;
5. Informed user, revalidated on UI.
RCA: Callback API delayed due to temporary network lag → DB record not updated to "Success".
Resolution: Manually updated DB to correct state.
Verification & Closure: Transaction displayed correctly after fix.
Why L3 Not Involved: No bug — one-time network delay issue.

🔹 **Example 7 — Jenkins Deployment Failure (Config Missing)**

Application Deployment Failure (Config Missing)
Issue Title Meaning: In CI/CD pipeline (Jenkins, Azure DevOps), when deploying a new build to UAT or Production, a deployment failure means some step in pipeline failed.Here — "Config Missing" means one of the required environment variables or property keys is absent.
Internally: Jenkins picks up application code + configuration.If any property (like DB_USER, API_KEY) is missing → build or startup fails.Error example: Could not resolve property DB_USER.
How L2 Identifies: Checks Jenkins console logs.Compares application.properties with last successful version.Finds missing variable or syntax issue.
Impact: Deployment blocked → UAT testing or production release delayed.
Incident Summary:UAT deployment pipeline failed.
Ticket Source: Directly assigned to L2 (deployment responsibility).
Symptoms: Error: Missing property DB_USER in application.properties.
Actions Taken (L2):
1. Checked Jenkins build logs.
2. Compared configs with last successful build:
diff config/application.properties config_backup/application.properties
3. Found DB_USER entry missing.
4. Added correct config and re-triggered Jenkins job → build successful.
RCA: Config file updated manually by developer; DB_USER property accidentally removed.
Resolution: Added correct parameter and secured credentials.
Verification & Closure: Deployment succeeded, app accessible in UAT.
Why L3 Not Involved: No code defect, just config fix.

🔹 **Example 8 — SSL Certificate Expiry**

Issue Title Meaning: Web applications use SSL/TLS certificates to encrypt user connections (HTTPS).
Each certificate has an expiry date.If not renewed before expiry, users get browser errors ("Connection not secure").

Internally: Certificates are stored on servers (e.g., /etc/pki/tls/certs/). Monitoring tools check expiry date & raise alert 3–7 days before.Renewal involves replacing .crt and .key files.
How L2 Identifies: Alert "SSL certificate expires in 3 days".Runs: openssl x509 -in certfile.crt -noout -enddateConfirms expiry date.
Impact: If expired — users can't access site via HTTPS.Must renew before expiry.
Incident Summary:SSL certificate for api.bank.com expiring soon.
Ticket Source: Monitoring alert (auto-generated).
Symptoms: None yet — proactive alert 3 days before expiry.
Actions Taken (L2):
1. Verified expiry:
openssl x509 -in /etc/pki/tls/certs/api.bank.com.crt -noout -enddate
2. Coordinated with Infra/Cert team to get renewed certificate.
3. Replaced new cert:
cp new_api.crt /etc/pki/tls/certs/
systemctl restart nginx
4. Verified via browser and SSL checker.
RCA: Certificate renewal process not automated → manual replacement needed periodically.
Resolution: Replaced SSL cert with renewed one, restarted service.
Verification & Closure: Certificate valid for next 1 year.
Why L3 Not Involved: Infra-level maintenance; no code changes.

Example 1: Payment API Timeout

- **Inside Story:** High DB load caused connection pool exhaustion.
- **How L2 Identified:** Dynatrace alert on 500 errors.
- **Impact:** 20% of users couldn't make payments.
- **Actions Taken:** Restarted payment service, increased DB pool size.
- **RCA:** Pool misconfiguration.
- **Why L3 Not Involved:** No code issue, only config fix.

  Payment Service – API Timeout – Transactions Delayed

  Meaning:

This title indicates that the **Payment Service module** of the application was **facing API timeout errors**, which resulted in **delayed or failed payment transactions** for users. It tells us immediately which component is affected and what type of issue it is.

Inside Story (Detailed Technical Background):

During high transaction volume around 11:45 AM, the Payment microservice started showing increased response times.
The internal API that connects to the **Payment Database** began throwing **connection timeout exceptions**.
Investigation showed that the **database connection pool** had reached its **maximum limit (50 connections)**.
Some old connections were not being released back to the pool due to a **stale session** issue. This caused new requests to wait until timeout, resulting in **HTTP 500 and 504 errors** being sent to end users.

How L2 Identified:

- An **alert** was triggered in **Dynatrace** for **spike in API failure rate (>10%)**.
- L2 verified in **Kibana logs** and saw repeated errors:
  `java.sql.SQLTransientConnectionException: HikariPool-1 - Connection is not available`.
  L2 correlated with database monitoring (via Grafana) and confirmed max pool utilization at 100%.

So L2 identified it **proactively from monitoring tools**, before multiple tickets came from users — a key L2 responsibility.

Impact:

- Around **25% of transactions** failed between **11:45 AM and 12:15 PM**.Approx. **500 users** faced delays or retries.**Business Impact:** Temporary delay in payment processing, but no financial loss since retries succeeded post-resolution.

Incident Summary:

At 11:45 AM, the Payment API started failing with timeout errors due to connection pool saturation in the Payment microservice. L2 Support identified the issue through Dynatrace alerts and confirmed via application logs and DB metrics. To restore services quickly, L2 increased the pool size temporarily and restarted the affected pod.
After restart, the service stabilized and transactions resumed normally by 12:20 PM.

Ticket Source:

- Automatically generated from **Dynatrace alert**, integrated with **ServiceNow** incident queue.No manual user ticket initially; detected through proactive monitoring.

- Symptoms:

  Users reported **"Payment failed, please try again"** messages.Dynatrace dashboard showed **API latency > 20s** and **error rate > 30%**.Logs contained frequent **SQLTransientConnectionException** errors.Queue messages (Kafka) for transaction updates started backing up.

Actions Taken (L2 Engineer):

Acknowledged alert in ServiceNow and started triage.Checked Dynatrace and confirmed API timeouts.**Reviewed Kibana logs** → connection pool exceptions observed.Verified **DB connection pool metrics** → reached 100% usage.Increased connection pool limit temporarily from 50 → 75 in config.Restarted Payment microservice pod from Kubernetes dashboard.Cleared pending Kafka queue backlog using script.Monitored API metrics post-fix for 1 hour — error rate dropped to normal (<1%).

RCA (Root Cause Analysis):

The **root cause** was **connection pool saturation** due to **stale connections not being released** in certain threads during high load.
This exhausted all available connections, causing timeouts.
The pool size (50) was insufficient for the current transaction volume.
It was a **configuration issue**, not a code defect.

Resolution (Manually Applied by L2):

- Increased DB connection pool size from 50 → 75 in runtime configuration file.Restarted Payment microservice pod. Cleared pending Kafka messages. Verified that DB connections were released properly after fix.

Verification & Closure:

Post-restart, API latency normalized to <500ms.Dynatrace showed no error spikes for 2 continuous hours.Verified 10 sample payment transactions successfully completed.Business confirmed no pending customer complaints.Incident closed in ServiceNow with RCA attached.

- **Why L3 Not Involved:** Issue was purely **configuration-based**, not code-related. No code debugging or patch deployment required.L2 was able to restore service using runtime parameter tuning and restart. Permanent fix (connection pool size adjustment in config repo) planned in next release, validated by L2, and reviewed by DevOps.

Final Statement for Interview: If the interviewer asks you to **explain your RCA process**, you can say this **word-for-word** confidently: "In one incident, our Payment API started timing out due to a database connection pool limit. As L2, we identified the issue from Dynatrace alerts, analyzed logs in Kibana, and found all connections were stuck. We temporarily increased the pool size and restarted the microservice, restoring service within 30 minutes. The RCA confirmed a configuration issue — no code defect — so L3 involvement wasn't required. Post-fix monitoring confirmed stability, and we documented RCA with business sign-off in ServiceNow."

Example **1: Application Server Crash**

- Scenario: Web application down, users cannot login.
- L2 Action:
    1. Check server logs → JVM crash detected.
    2. Restart application server.
    3. Tune JVM memory settings.
    4. RCA: Memory leak causing crash under heavy load.
- Flow: Ticket assigned directly or escalated by L1. L2 restores service and documents RCA.

Example **2: Database Timeout / Connection Pool Exhaustion**

- Scenario: Application slow or failing due to DB timeout.
- L2 Action:
    1. Check DB logs and connection pool metrics.
    2. Increase pool size or restart DB service.
    3. RCA: Heavy query load caused connection exhaustion.
- Flow: L1 escalates after users report slowness. L2 identifies backend DB issue.

Example **3: Payment Gateway Integration Fails**

- Scenario: Transactions failing in production.
- L2 Action:
    1. Check server logs, API response codes.
    2. Restart payment service or refresh API keys.
    3. RCA: Third-party API token expired.
- Flow: Ticket may come directly from monitoring or L1 escalation.

Example **4: Middleware / API Service Not Responding**

- Scenario: REST API returns 500 errors.
- L2 Action:
    1. Check API logs, microservice status.
    2. Restart affected service.
    3. RCA: Thread pool exhausted due to high traffic.
- Flow: Monitoring alerts generate ticket → L2 fixes and updates documentation.

Example **5: Disk Space Full on Server**

- Scenario: Production server disk > 90% full → services failing.
- L2 Action:
    1. SSH into server → check disk usage.
    2. Clean temp files, rotate logs.

3. RCA: Old logs and backups not purged automatically.
- Flow: L1 may escalate after user complains, or direct alert triggers ticket.

Example **6: Application Configuration Error**
- Scenario: New deployment causes app errors due to misconfigured environment variables.
- L2 Action:
  1. Check configuration files and deployment logs.
  2. Correct variable or rollback deployment.
  3. RCA: Incorrect environment settings after deployment.
- Flow: Directly assigned to L2 if detected during deployment monitoring.

Example **7: Network Latency Affecting Application**
- Scenario: Users report slowness; internal ping/traceroute shows high latency.
- L2 Action:
  1. Check network monitoring dashboards (Cisco, SolarWinds).
  2. Apply routing adjustments, restart network service.
  3. RCA: Switch misconfiguration causing packet loss.
- Flow: L1 escalates after basic network checks fail.

Example **8: Batch Job / Scheduled Task Failure**
- Scenario: Nightly data import job fails → reports incomplete data.
- L2 Action:
  1. Check batch job logs and system status.
  2. Re-run job manually or fix script.
  3. RCA: Script timeout due to large data volume.
- Flow: Ticket escalated by L1 or directly assigned if monitoring detects failure.

LEVEL 3 SUPPORT

2 PROBLEM MANAGEMENT — find & fix root cause

What is it?

Process to investigate and eliminate the root causes of incidents (reactive) and to identify weaknesses proactively.

Goals

- Find the root cause (RCA).
- Implement permanent fix or workaround.
- Reduce number and impact of incidents.
- Maintain a **Known Error Database** (KEDB).

Types

- **Reactive problem management** — triggered after repeated/serious incidents.
- **Proactive problem management** — trend analysis, capacity planning, risk identification.

When to open a Problem ticket (decision prompts)

- The same incident repeats multiple times (e.g., >3 occurrences in X days).
- Single major incident with no known root cause.
- Trend analysis shows increasing errors of same class.
- When a workaround is implemented but not a permanent fix.

3) CHANGE MANAGEMENT — change safely & auditable

What is it?

A controlled process to plan, approve, schedule, implement and review changes to production (and other environments).

Why do we need it?

- Prevents unplanned outages caused by poorly tested changes.
- Ensures appropriate approvals, backout plans, and communication.
- Provides audit trail and scheduling visibility.

Types of change

1. **Standard change**
    - Low-risk, pre-authorized repeatable change (e.g., regular certificate renewal, routine patch with known steps).
    - Pre-approved by Change Manager; no full CAB required.

2. **Normal change**
    - Any change that requires assessment and approval via RFC and CAB. Varies by risk level (minor → major).
    - Planned, has change window, testing and backout plan.

3. **Emergency change**
    - High-priority, must be applied immediately to fix a critical incident (e.g., security patch to stop data leak).
    - Follow Emergency CAB (ECAB) for approval, expedite documentation afterward.

RFC (Request for Change) — what it contains

- Summary & description
- CI(s) impacted
- Business justification
- Risk & impact assessment
- Implementation steps & timeline
- Backout/rollback plan
- Test plan / validation criteria
- Scheduled window & communication plan
- CAB approvals / sign-offs

CAB (Change Advisory Board)

- Reviews normal changes, advises on risk and scheduling.
- ECAB deals with emergency changes.
- CAB members: Change Manager, Infrastructure/DB/Network leads, Security, Business reps.

Lifecycle

1. **Request** — RFC opened.
2. **Assess** — Change manager and technical reviewers evaluate risk.
3. **Approve** — CAB approves & schedules.
4. **Implement** — Execute change during window.
5. **Validate** — Verify success via tests/monitoring.
   Close & Review — PIR for major changes; update CMDB.

When to create a Change ticket

- To make permanent fixes that alter system behavior (code deploy, DB schema, infra config).
- To schedule routine work that affects production (patching, migration).
- Even for small changes if they impact production or business-critical components (if not covered by Standard change model).

When to raise Incident vs Problem vs Change — decision guide

1. **If service is degraded or users are impacted right now → RAISE AN INCIDENT**
    - Example triggers: monitoring alert (Splunk/CloudWatch), customer outage.
    - Goal: restore service fast (apply workaround if needed).

2. **If multiple incidents with similar symptoms or a single major incident with unknown root cause → RAISE A PROBLEM**
    - Link all related incidents under the Problem ticket.
    - Goal: do RCA and find permanent fix.

3. **If you need to implement a permanent fix / deploy code / change infra / schema → RAISE A CHANGE (RFC)**
    - If fix is urgent and production-impacting, consider **Emergency Change**.
    - If fix is a routine known task, see if **Standard Change** model applies.

Q: "What's the difference between an incident and a problem?"

"An incident is a one-off service interruption we must fix immediately to restore service. A problem is the underlying cause of one or many incidents — we open a problem ticket to do a root-cause analysis and plan a permanent fix. For example, if Redis crashes and we restart it to restore service, that's an incident. If Redis crashes several times for the same reason, we raise a problem and plan a change to fix it permanently."

Q: "When would you raise a change?"

> "We raise a change when we need to make permanent modifications — code deploys, DB schema changes, config edits. If it's routine and low-risk, we use a Standard change. If it's urgent to prevent customer impact, we use an Emergency RFC and notify ECAB."

Q: "How do you ensure changes don't break production?"

> "By requiring RFCs with rollback plans, CAB approval, testing in staging, scheduled maintenance windows, gradual rollout (canary/blue-green), and monitoring post-deploy with tools like Splunk and CloudWatch."

yes i wan both points ok and also explain me simple l1 incident thatthey can resolve level 1 engineers and if there is query then always remeber use case of query why we use and line by line word by word explainationation i want

give me 15 example if incident that level 1 solve only and how they are solve as i have to explain it to interviewer explain me how solve , how they are identifying issue, in detail everything explain me so that i can explain in detail how level 1 engineers solve and tell me priority as well so that i can explain it to interviewer

Now we can move to another topic explain me what is problem ticket, incident management, change management-types of change .also explain me why we are raising this above listed , and in which case they are raises, when to raise incident , when to problem tickets, and when to raise change management. explain everything in detail