Overview

* I will describe an AWS + ServiceNow architecture and provide:
  + A clear sequence of operations for both Lambda functions.
  + A recommended MEMBER\_DATA schema (Aurora/MySQL) with data types and sample rows.
  + A modular, transparent assignment algorithm (weights explained mathematically).
  + Complete Python code for both Lambda functions (poller and processor) that you can deploy to AWS Lambda (with dependencies and environment variables).
  + Operational notes: concurrency, idempotency, monitoring, IAM, and tuning suggestions.

Design summary

* Components:
  + ServiceNow (source of incidents and user/group info via REST APIs).
  + Lambda A (Poller): scheduled (CloudWatch Event / EventBridge) — queries ServiceNow for new/unassigned incidents in specified assignment groups and pushes messages to SQS.
  + SQS queue (FIFO recommended if ordering/strict dedupe matters; standard otherwise).
  + Lambda B (Processor): triggered by SQS messages — gets group members, filters by MEMBER\_DATA (Aurora MySQL), fetches each member’s active incidents from ServiceNow, computes weights, chooses a member, updates the ServiceNow incident assignment, and logs/records assignment history to DB.
  + Aurora (MySQL/MariaDB/Aurora MySQL) with MEMBER\_DATA and ASSIGNMENT\_HISTORY tables.
  + CloudWatch logs/metrics, AWS X-Ray tracing optional.

High-level flow

1. EventBridge triggers Poller Lambda periodically (e.g., every minute or 5 minutes).
2. Poller queries ServiceNow for incidents matching: assignment groups of interest, unassigned (or specific states), created or updated since last poll.
3. Poller pushes each incident as an SQS message (incident number, sys\_id, assignment\_group\_sys\_id, priority, created\_on, etc).
4. Processor Lambda consumes SQS messages. For each:
   * Confirm incident still unassigned (re-check ServiceNow). If already assigned, skip.
   * Retrieve group members from ServiceNow (sys\_user\_grmember) (member sys\_id list).
   * Query MEMBER\_DATA in Aurora to filter members by active status and shift roster using incident timestamp.
   * For remaining members, fetch their currently assigned incidents from ServiceNow and calculate each member’s workload score.
   * Apply role experience multipliers and other factors to compute final weights.
   * Pick member with best (lowest) final weight, break ties deterministically + jitter.
   * Assign incident via ServiceNow update API; on success store assignment in ASSIGNMENT\_HISTORY and log any metrics.
   * If assignment fails because someone else took it concurrently, optionally requeue or log.

MEMBER\_DATA table: schema, reasoning and sample data

* Table name: MEMBER\_DATA
* Recommended columns (MySQL / Aurora):
  + id: BIGINT AUTO\_INCREMENT PRIMARY KEY
  + assignment\_group\_sys\_id: VARCHAR(64) NOT NULL -- ServiceNow group sys\_id
  + member\_sys\_id: VARCHAR(64) NOT NULL -- ServiceNow user sys\_id
  + member\_name: VARCHAR(200)
  + role: ENUM('L1','L2','L3','SME') NOT NULL -- map to experience modifiers
  + shift\_start\_time: TIME NOT NULL -- in group's timezone or UTC (choose one)
  + shift\_end\_time: TIME NOT NULL
  + shift\_days: SET('Mon','Tue','Wed','Thu','Fri','Sat','Sun') NOT NULL -- which days this shift applies
  + weekend\_shift\_flag: BOOLEAN NOT NULL DEFAULT 0
  + active: BOOLEAN NOT NULL DEFAULT 1
  + weight\_modifier: DECIMAL(5,3) DEFAULT 1.000 -- optional manual multiplier for business override
  + last\_manual\_update\_by: VARCHAR(200)
  + last\_manual\_update\_at: DATETIME DEFAULT CURRENT\_TIMESTAMP ON UPDATE CURRENT\_TIMESTAMP
* ASSIGNMENT\_HISTORY (optional, for traceability and fairness):
  + id: BIGINT AUTO\_INCREMENT PRIMARY KEY
  + incident\_sys\_id: VARCHAR(64)
  + incident\_number: VARCHAR(50)
  + assigned\_to\_member\_sys\_id: VARCHAR(64)
  + assignment\_timestamp: DATETIME
  + algorithm\_snapshot: JSON -- weights and reason for auditing
  + created\_by: VARCHAR(64) -- lambda function id or user
  + success: BOOLEAN

Sample MEMBER\_DATA rows (values illustrative)

* Row 1:
  + assignment\_group\_sys\_id = '46d3c1a0a9a0c0101a2b3c4d5e6f7a8'
  + member\_sys\_id = 'user\_sys\_001'
  + member\_name = 'Alice Kumar'
  + role = 'L2'
  + shift\_start\_time = '09:00:00'
  + shift\_end\_time = '17:30:00'
  + shift\_days = 'Mon,Tue,Wed,Thu,Fri'
  + weekend\_shift\_flag = 0
  + active = 1
  + weight\_modifier = 1.0
* Row 2:
  + member\_sys\_id = 'user\_sys\_002'
  + member\_name = 'Bob Chen'
  + role = 'L1'
  + shift\_start\_time = '16:00:00'
  + shift\_end\_time = '00:30:00' (overnight)
  + shift\_days = 'Mon,Tue,Wed,Thu,Fri,Sat,Sun'
  + weekend\_shift\_flag = 1
  + active = 1
  + weight\_modifier = 1.05
* Row 3:
  + member\_sys\_id = 'user\_sys\_003'
  + member\_name = 'Carla Diaz'
  + role = 'SME'
  + shift\_start\_time = '08:00:00'
  + shift\_end\_time = '20:00:00'
  + shift\_days = 'Mon,Tue,Wed,Thu,Fri'
  + weekend\_shift\_flag = 0
  + active = 1
  + weight\_modifier = 0.95

Assignment algorithm — modular & transparent

* Goals:
  + Favor members with lower current workload.
  + Factor priority and age (older incidents are more costly).
  + Factor role experience (SME less penalized).
  + Respect shift availability.
  + Avoid starvation and encourage fairness.
* Inputs:
  + For each candidate member M:
    - Current assigned incidents I\_j, each with priority P\_j, severity S\_j, created\_at t\_j
    - Role R\_M and weight modifier W\_mod\_M
    - Number of assignments in recent window (optional)
* Priority weight mapping (example; tune to your org):
  + Priority 1 (P1/Critical): 6
  + Priority 2 (P2/High): 3
  + Priority 3 (P3/Medium): 1
  + Priority 4/5 (Low): 0.5
* Severity multiplier mapping (optional):
  + severity 1 (Critical): 1.3
  + severity 2: 1.1
  + severity 3: 1.0
* Age factor:
  + For incident j: age\_hours = now - t\_j (in hours)
  + age\_factor\_j = 1 + (age\_hours / 24) (so each 24h increases weight by 1)
  + or use: age\_factor\_j = 1 + log(1 + age\_hours)/k if you prefer sublinear growth.
* Member base workload score:
  + base\_workload\_M = sum over assigned incidents j: (priority\_weight(P\_j) \* severity\_multiplier(S\_j) \* age\_factor\_j)
* Role multiplier:
  + role\_multiplier = mapping (L1: 1.2, L2: 1.0, L3: 0.9, SME: 0.85)
    - Lower multiplier => more preferred (reduces effective workload).
* Manual weight modifier:
  + multiply by weight\_modifier column from MEMBER\_DATA.
* Final computed weight:
  + final\_weight\_M = (base\_workload\_M \* role\_multiplier \* weight\_modifier) + fairness\_factor + epsilon
  + fairness\_factor: add +0.1 \* recent\_assignments\_count to slightly penalize those who had many assignments recently (to balance load). Store counts in ASSIGNMENT\_HISTORY to compute.
  + epsilon: small random jitter in [0, 0.01] to break ties deterministically as needed.
* Selection:
  + Select member with minimum final\_weight\_M.
  + Log algorithm\_snapshot JSON with per-member scores and reason.

Algorithm example (numbers)

* Member A has two incidents:
  + I1: P2 (3), severity 2 (1.1), age 6h => age\_factor=1+6/24=1.25 -> contribution: 3*1.1*1.25=4.125
  + I2: P3 (1), severity 3 (1.0), age 2h => age\_factor=1.083 -> contribution ≈ 1.083
  + base=5.208
  + role=L2 => role\_multiplier=1.0, weight\_modifier=1.0, fairness=0.1\*2=0.2
  + final=5.208\*1.0 + 0.2 = 5.408
* Member B has one incident:
  + I1: P1 (6), severity 1 (1.3), age 12h => age\_factor=1.5 -> cont = 6*1.3*1.5 = 11.7
  + role=L3 => 0.9, fairness=0.1\*1=0.1
  + final = 11.7\*0.9 + 0.1 = 10.63
  + Therefore pick Member A.

Code: Lambda Poller (poller\_lambda.py)

* Dependencies: requests, boto3
* Environment variables to configure:
  + SERVICENOW\_BASE\_URL, SERVICENOW\_USER, SERVICENOW\_PASS
  + ASSIGNMENT\_GROUPS (comma-separated list of ServiceNow group sys\_ids or names)
  + SQS\_QUEUE\_URL
  + POLL\_LOOKBACK\_MINUTES (optional)
  + INCIDENT\_STATES\_FILTER (optional e.g., "1,2,3" or use assigned\_toISEMPTY)

Full code (drop into Lambda with requests and boto3 packaged):

//// Poller Lambda Code

Notes:

* Adjust sysparm\_query as needed (ServiceNow date queries can be crafted differently).
* You may want to maintain a cursor (timestamp of last poll) in DynamoDB to avoid overlap.

Code: Lambda Processor (processor\_lambda.py)

* Dependencies: requests, boto3, pymysql (or use AWS RDS Data API if Aurora Serverless and prefer boto3 rds-data)
* Environment variables:
  + SERVICENOW\_BASE\_URL, SERVICENOW\_USER, SERVICENOW\_PASS
  + DB\_HOST, DB\_PORT, DB\_USER, DB\_PASS, DB\_NAME
  + SQS\_QUEUE\_URL (for requeueing if needed)
  + ASSIGNMENT\_HISTORY\_TABLE, MEMBER\_DATA\_TABLE
  + MAX\_CANDIDATES (optional)
* Behavior:
  + For each SQS message, process as described. Will use PyMySQL to query MEMBER\_DATA and insert into ASSIGNMENT\_HISTORY.

Full code (packaged with dependencies):

/// Processor Lambda

Notes about the processor

* The code uses PyMySQL. For Lambda packaging, include PyMySQL in the deployment package or use a Lambda Layer.
* Timezone: pick a consistent timezone (recommended: store shift times in UTC or include timezone in DB and convert incident time accordingly). The code above assumes times comparable to incident timestamp — adapt for your environment.
* Races: the code re-checks assigned\_to before assignment; still race conditions possible. Stronger protection:
  + Use ServiceNow business logic (e.g., check\_and\_set with sys\_updated\_on) or
  + use a distributed lock (DynamoDB conditional write) keyed by incident\_sys\_id before assignment.
* For high throughput, parallelize per SQS batch but be mindful of API rate limits.

Database DDL (MySQL / Aurora)

Example DDL for MEMBER\_DATA and ASSIGNMENT\_HISTORY:

/// Table Creation SQL

Monitoring, observability and security

* Logging: CloudWatch Logs for both Lambdas. Log algorithm\_snapshot for auditing (but redact sensitive data).
* Metrics: push custom CloudWatch metrics for assignments per minute, failures, average candidate count, latency.
* Tracing: enable AWS X-Ray on Lambdas to trace downstream HTTP calls.
* Secrets: use AWS Secrets Manager or Parameter Store (SSM) for DB and ServiceNow credentials, not plain env vars.
* IAM:
  + Lambda Poller: read-only to SQS (SendMessage).
  + Lambda Processor: SQS ReceiveMessage permission (if used with SQS trigger Lambda this is managed), plus access to Secrets Manager and RDS network access (via VPC). Use least privilege.
* Rate limiting: respect ServiceNow API limits; implement backoff and retries for transient 429/5xx responses.

Operational considerations & tuning

* Tuning priority weights and role multipliers requires historical data. Backtest using logs: simulate the algorithm using past incidents and compare with manual assignments.
* Fairness: add a bigger fairness factor if some members are starved or consider a rotating bias to prefer members who had fewer assignments overall in last 7 days.
* Shifts: for complex shift rosters (rotations, on-call rotations), store a JSON schedule or integrate with an on-call provider (PagerDuty) or ServiceNow schedule table and query that system instead of a simple shift\_start/shift\_end.
* Concurrency and consistency: if assignment correctness is critical, consider: (a) using a DB-backed claim (DynamoDB conditional write) to claim incident\_sys\_id before trying to assign; (b) atomically check assigned\_to when updating and retry on failure.

How to test locally and in staging

* Use a staging ServiceNow instance and staging DB.
* Seed MEMBER\_DATA with several sample members and create synthetic assigned incidents to test weight ordering.
* Run the selection code locally to verify ranking and inspect algorithm\_snapshot logs.
* Run load tests to observe ServiceNow API limits and tune Poller interval and batch sizes.

Wrap-up

* The provided code is a complete starting implementation: scheduled poller pushes SQS items, processor computes weights and assigns incidents via ServiceNow API while recording assignment history.
* Key next steps:
  + Adapt shift/time handling to your timezone and roster model.
  + Secure credentials in Secrets Manager and wire into Lambda.
  + Add robust retry/backoff handling for ServiceNow and DB calls (the samples have minimal retry).
  + Backtest/tune weights and multipliers with historical incident data.
  + Add unit tests for the algorithm module (separate compute functions from I/O for easy testability).

If you want, I can:

* Provide a packaged Lambda layer and deployment artifacts (SAM/CloudFormation or Terraform) for both Lambdas and RDS setup.
* Convert the shift storage to a richer JSON schedule or integrate reading ServiceNow on-call/schedule tables.
* Produce unit tests for the assignment algorithm with sample incident sets and expected ordering.