AIO Engineer Technical Assessment

Shift Scheduling System

Rust

Position	AIO Engineer
Level	Mid / Senior
Duration	7 days
Format	Independent, remote
Submission	Git repository

1. Overview

Design and implement a shift scheduling system composed of two microservices:

Service	Responsibility	Storage
Data Service	Staff and group management	PostgreSQL
Scheduling Service	Asynchronous shift schedule generation	PostgreSQL

Each service should be independently deployable and communicate over HTTP.

2. Technology Stack

Component	Requirement
Language	Rust (edition 2021+)
Web Framework	Axum
Async Runtime	Tokio
Database	PostgreSQL with sqlx
Serialization	serde / serde_json
Containerization	Docker, docker-compose
API Documentation	utoipa (OpenAPI / Swagger)
Caching	Redis

3. Data Service

3.1 Entities

- **Staff** - id, name, email (unique), position, status (ACTIVE / INACTIVE), timestamps
- **Staff Group** - id, name, parent group reference (supports nesting)
- **Group Membership** - many-to-many relationship between staff and groups

3.2 Database

Table	Purpose
`staff`	Staff records
`staff_groups`	Groups with self-referencing parent for hierarchy
`group_memberships`	Staff-to-group associations

Column types, constraints, and indexes are left to the candidate.

3.3 API

Candidates are expected to design a complete set of RESTful APIs, including:

- Full CRUD operations for staff and groups
- Adding and removing staff from groups
- Batch import from JSON files for all resources
- Hierarchical group resolution - retrieving all members under a group, including members of nested subgroups

Reference endpoint:

```
GET /api/v1/groups/{id}/resolved-members
```

3.4 Caching

Read-heavy endpoints should be cached using Redis.

4. Scheduling Service

4.1 Responsibility

- Accept schedule generation requests and process them asynchronously
- Retrieve staff data from the Data Service over HTTP
- Persist job state and generated schedules in PostgreSQL

4.2 Shift Types

Type	Description
`MORNING`	Morning shift
`EVENING`	Evening shift
`DAY_OFF`	Day off

4.3 API

Submit a schedule generation request

```
POST /api/v1/schedules
```

Request:

```
{
  "staff_group_id": "group-123",
  "period_begin_date": "2025-05-19"
}
```

Response (`202 Accepted`):

```
{
  "schedule_id": "job-abc123",
  "status": "PENDING"
}
```

Check job status

```
GET /api/v1/schedules/{schedule_id}/status
```

Status values: `PENDING`, `PROCESSING`, `COMPLETED`, `FAILED`

Retrieve generated schedule

```
GET /api/v1/schedules/{schedule_id}/result
```

Response:

```
{
  "schedule_id": "job-abc123",
  "period_begin_date": "2025-05-19",
  "staff_group_id": "group-123",
  "assignments": [
```

```
    { "staff_id": "staff-1", "date": "2025-05-19", "shift": "MORNING" },
    { "staff_id": "staff-1", "date": "2025-05-20", "shift": "EVENING" },
    { "staff_id": "staff-1", "date": "2025-05-21", "shift": "DAY_OFF" }
  ]
}
```

4.4 Scheduling Rules

Input

- A staff group, fetched from Data Service via the `resolved-members` endpoint
- A period start date (`period_begin_date`), which must fall on a Monday
- Period length: 28 days (4 weeks)

Constraints

1. Each staff member is assigned exactly one shift per day: `MORNING`, `EVENING`, or `DAY_OFF`.
2. Each staff member must have at least 1 and at most 2 `DAY_OFF` entries per week (Monday through Sunday).
3. A `MORNING` shift must not immediately follow an `EVENING` shift. If day N is assigned `EVENING`, then day N+1 must not be `MORNING`.
4. Only staff with status `ACTIVE` should be included in scheduling.
5. On any given day, the difference between the number of staff assigned to `MORNING` and the number assigned to `EVENING` must not exceed a configurable threshold. This ensures balanced daily coverage across shifts.

Example - valid assignment (1 staff, 1 week)

```
Mon  MORNING
Tue  EVENING
Wed  EVENING       OK - EVENING may follow EVENING
Thu  DAY_OFF       OK - DAY_OFF may follow EVENING
Fri  MORNING
Sat  MORNING
Sun  EVENING

Total: 3 MORNING, 3 EVENING, 1 DAY_OFF
All constraints satisfied.
```

Example - valid daily coverage (4 staff, Monday)

```
Staff-1  MORNING
Staff-2  MORNING
Staff-3  EVENING
Staff-4  EVENING

MORNING: 2, EVENING: 2, difference: 0
Satisfies constraint 5.
```

Example - invalid daily coverage (4 staff, Monday, max_daily_shift_diff = 1)

```
Staff-1  MORNING
Staff-2  MORNING
```

```
Staff-3  MORNING
Staff-4  DAY_OFF

MORNING: 3, EVENING: 0, difference: 3
INVALID - violates constraint 5.
```

Example - invalid assignment

```
Mon  EVENING
Tue  MORNING       INVALID - violates constraint 3
```

Configuration

Each rule should be configurable (enable/disable and parameter adjustment) via a configuration file:

Rule	Key	Default
Minimum days off per week	`min_day_off_per_week`	1
Maximum days off per week	`max_day_off_per_week`	2
Disallow MORNING after EVENING	`no_morning_after_evening`	`true`
Maximum daily shift difference	`max_daily_shift_diff`	1

4.5 Database

Table	Purpose
`schedule_jobs`	Job metadata - group, period, status, timestamps
`shift_assignments`	Generated assignments - staff, date, shift type

Column types, constraints, and indexes are left to the candidate.

4.6 Error Handling

- Failed jobs must be logged and marked with status FAILED.
- Errors from the Data Service should be handled gracefully without crashing the process.

5. Architecture

- Apply **Clean Architecture** principles with clear separation between API, domain, and infrastructure layers.
- Use **traits** for dependency inversion between layers.
- Scheduling rules must be implemented as **configurable components** that can be enabled or disabled via configuration.
- Organize the project as a **Cargo workspace**:

```
shift-scheduler/
├── Cargo.toml                  # workspace root
├── docker-compose.yml
├── data-service/
│   ├── Cargo.toml
│   └── src/
├── scheduling-service/
│   ├── Cargo.toml
│   └── src/
├── shared/
│   ├── Cargo.toml
│   └── src/
└── sample-data/
    ├── staff.json
    └── groups.json
```

6. Testing

- **Unit tests** covering scheduling rules and domain logic.
- **Integration tests** covering API endpoints.
- Use **mockall** to mock Data Service dependencies within the Scheduling Service.
- All code must pass `cargo clippy` with no warnings and be formatted with `cargo fmt`.

7. Deliverables

7.1 Source Code

Two services organized within a single Cargo workspace, including a shared crate for common types.

7.2 Docker Compose

A `docker-compose.yml` that includes the following containers:

Container	Description
`data-service`	Data Service instance
`scheduling-service`	Scheduling Service instance

`postgresql`	PostgreSQL database
`redis`	Redis cache

The entire system must start with a single command: `docker-compose up`.

7.3 Documentation

A `README.md` covering:

- Build and run instructions
- API documentation (Swagger UI link or Postman collection)
- Database schema or ERD

7.4 Sample Data

JSON files suitable for batch import (staff, groups).

8. Evaluation Criteria

Criteria	Weight
Architecture and design - Clean Architecture, trait usage, layer separation	25%
Rust code quality - idiomatic patterns, error handling, effective use of the type system	25%
Feature completeness - CRUD, batch import, scheduling, async processing	25%
Testing and DevOps - test coverage, Docker setup, documentation	25%

Bonus

The following are not required but will be considered favorably:

- Automatic cache invalidation on data mutations
- Type-state pattern for job lifecycle management
- Graceful shutdown handling
- Distributed tracing via OpenTelemetry
- Compile-time query verification with sqlx
- Circuit breaker pattern for Data Service calls
- Distributed locking to prevent duplicate job processing