The Liftinator High-Level Design

Version: 1.0

Updated: 3/3/2025

Author: Chris Harper

Table Of Contents

[1. Overview 3](#_Toc191891180)

[2. Requirements 3](#_Toc191891181)

[2.1. Basic Floor Operation 3](#_Toc191891182)

[2.2. Monitoring 3](#_Toc191891183)

[3. High-Level Design 4](#_Toc191891184)

[4. Elevator Scheduling 5](#_Toc191891185)

[4.1. SCAN Algorithm Overview 5](#_Toc191891186)

[4.2. Capacity-Aware Coordinated Elevators 5](#_Toc191891187)

[5. Deployment/Fail-over 6](#_Toc191891188)

[6. Issues/Risks 6](#_Toc191891189)

[7. Appendix: 6](#_Toc191891190)

# Overview

This document presents a high-level software design for the Liftinator, a multi-elevator management system designed for large high-rise buildings. It outlines the core operations, decision-making logic, and processing involved in efficiently coordinating multiple elevators to optimise travel time, reduce wait times, and enhance overall system efficiency.

Unlike a typical design document that focuses solely on system architecture and implementation details, this document takes a more comprehensive approach. In addition to covering the structural and functional aspects of the Liftinator, I will also explain the thought process behind its design choices, including algorithm selection, efficiency considerations, and the reasoning behind key implementation decisions.

# Requirements

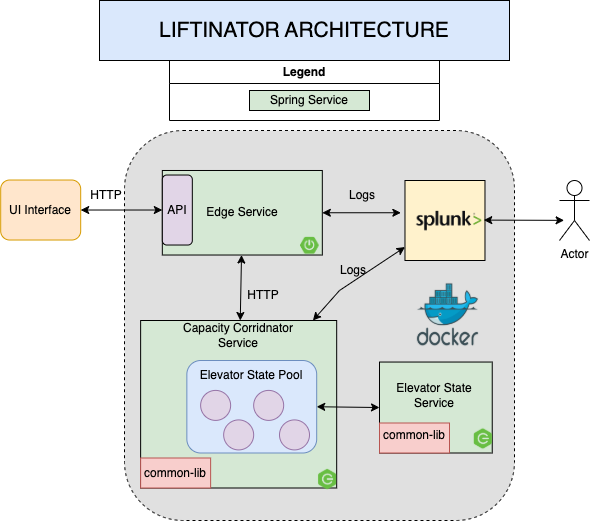
## Basic Floor Operation

1. Ability to queue multiple floor selections in a single trip.
2. Dispatch elevators to floors for waiting occupants if none are already on the same floor.
3. SCAN scheduling algorithm to determine the most efficient route based on active requests.
4. Capacity awareness and coordination between elevators.

## Monitoring

1. Send logs from all services to a common location (Splunk) for operator monitoring.

# [High-Level Design](https://softwaredominos.com/home/software-design-development-articles/high-level-solution-design-documents-what-is-it-and-when-do-you-need-one/)



The **Liftinator Architecture** is a **distributed, micro-services based** system for managing multiple elevators efficiently. It includes an **API gateway (Edge Service), a coordination service (Capacity Coordinator Service), and a state management service (Elevator State Service).** Logging and monitoring are integrated via **Splunk**, and **Docker** is used for deployment

Normally, within a Spring architecture, we would have a Eureka Server, which acts as a service registry, playing a central role in the automatic detection of devices and services on a network. Additionally, a Spring Cloud Config Server is typically used for storing and serving distributed configurations. However, for this exercise, that level of complexity seems excessive.

Key Components:

1. UI Interface:
   * Since this exercise is for back-end only there is no front-end UI.
   * When testing I utilised Postman to test my Interface Service API endpoints.
2. Edge Service
   * Acts as the 'front door' for the service architecture.
   * Handles API requests for all services and routes appropriately.
   * Provides a security layer for token validation for all inbound requests.
3. Capacity Coordinator Service
   * Manages the Elevator State Pool, which keeps track of the status of each elevator.
   * Coordinates the assignment of occupants to elevators based on their capacity and location.
   * Communicates with the Edge Service to send and receive commands and statuses for elevator operations.
4. Elevator State Service
   * Maintains the current state of an elevator, including its location (which floor), direction (up/down), occupant weight, and occupant capacity.
   * Maintains state based on the SCAN algorithm.
5. Common-Lib
   * Maintains common libraries used across services, including models, security, logging, and communication classes.
6. Logging and Monitoring
   * Splunk collects logs from the Edge Service and Capacity Coordinator Service.
   * Docker is used for containerized deployment and service orchestration.

# Elevator Scheduling

Elevators have been around for centuries, so finding the most effective algorithm for elevator operations doesn’t require reinventing the wheel. Through research, it became clear that the elevator algorithm, also known as SCAN (a disk-scheduling algorithm), is the standard approach.

However, I later came across an article on Medium (see the appendix below) that discussed capacity-aware coordinated elevators, which manage multiple elevators based on their location and current capacity. This concept had not initially occurred to me when defining my requirements, but it made complete sense as a real-world issue in large, busy high-rise buildings. As a result, I decided to incorporate this functionality into my design. Additionally, figuring out how to distribute occupants evenly among multiple elevators seemed like a fun design challenge.

## SCAN Algorithm Overview

The Elevator Algorithm, also known as SCAN, is a scheduling method used for optimising movement by picking up passengers going in its current direction before reversing, the SCAN algorithm services all requests in one direction before switching.

* Group requests by direction – Serve all "Up" requests first before switching to "Down" (or vice versa).
* Minimise travel distance – Prioritise the closest floor in the current travel direction.
* Account for new requests dynamically – Allow new requests to be added while processing existing ones.
* Handle priority requests – Emergency requests or priority floors override normal operations.
* Avoid starvation – If no more requests exist in the current direction, switch directions.

## Capacity-Aware Coordinated Elevators

**Capacity-Aware Coordinated Elevators** optimise the movement of multiple elevators by considering both **location** and **current capacity** when handling requests. Unlike traditional algorithms that only focus on the closest elevator, this approach ensures a more efficient and balanced distribution of passengers.

* Each elevator tracks its current load capacity and position in real time.
* When a request is made, the system assigns the most suitable elevator based on:
  + Proximity to the request.
  + Current passenger load.
  + Direction of travel.
* This prevents overloading a single elevator while others remain underutilised.

# Deployment/Fail-over

In case of failures such as a power outage, to minimise the risk of downtime resulting in people being stuck in elevators, I would suggest deploying a multi-region failover in AWS. If one data centre goes down, I would utilise AWS disaster recovery strategies such as Route 53 for DNS failover, automate fail-over processes with Cloud Watch and Lambda, and deploy infrastructure across multiple regions.

# Issues/Risks

* Small children, fascinated by the glowing buttons, eagerly pressing every single floor selection as if playing a game.
* Intoxicated individuals, having overindulged in their night out, ending up passed out in the elevator doorway, unknowingly preventing the doors from closing.
* Widespread building power failures occurred as Godzilla—or any other colossal creature—rampaging through the city.
* Secret agents either dangling from the bottom or riding on top of elevators to avoid detection (James Bond: Skyfall, Mission Impossible: I, James Bond: Diamonds Are Forever). We strongly discourage this type of activity, as elevators are not designed for such use and can pose serious safety risks to both the individuals involved and others. Unfortunately, we continue to see these type of behaviours from the secret agent community.

# Appendix:

|  |  |
| --- | --- |
| **Description** | **Link** |
| Capacity-Aware Coordinated Elevators Article | <https://medium.com/@mumbaiyachori/stuck-between-floors-how-elevator-algorithms-make-us-wait-and-wait-and-wait-b29e7e4bf728> |