

SOUTHEAST EUROPEAN UNIVERSITY

FACULTY OF CONTEMPORARY SCIENCE AND TECHNOLOGY

STUDY PROGRAM: “COMPUTER ENGINEERING”

SEMINAR PAPER FROM THE SUBJECT: SOFTWARE DESIGN AND ARCHITECTURE

Theme: “Real-Time price drop alert platform”

SUBJECT LECTURE: STUDENT:

Asst. Prof. Dr. Nuhi Besimi Irfan Uruchi

TETOV, MAY 2025

# Introduction

## Problem statement

Stock prices can swing several percent in seconds. Large trading firms handle that speed with premium data feeds and algorithmic orders, but most individual investors rely on web dashboards or delayed e-mail alerts from their broker. Because those tools refresh slowly and cannot check user-defined conditions in real time, many traders learn about a favorable price only after the market has already reversed.

## Example

A student trader would like to buy Samsung if the price drops below **$38**. During a two-hour lab the share dips briefly to **$34**, rebounds to nearly **$40**, and closes higher. When the student finally reloads the trading app, the opportunity has passed—not because the analysis was wrong, but because the alert system reacted too slowly.

## Why the issue deserves attention

It has a financial impact where missing a 1% intraday swing even a few times a semester can add up to a meaningful sum over the several years of study and early employment. Another reason is cognitive load, where reliable, real-time alerts let users focus on classes or work instead of repeatedly checking price screens. And also, another benefit is better market data, where aggregated and anonymous alert thresholds reveal where many small investors consider a share “fair value,” information that can improve liquidity planning for brokers.

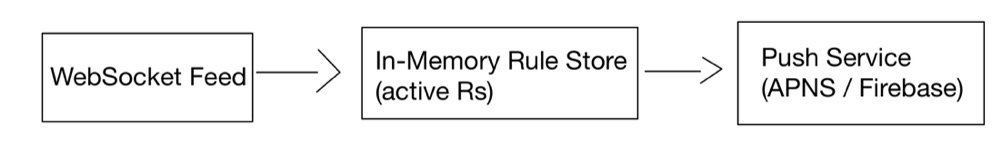
For pulling this off the system must keep an eye on live market tricks, run thousands of “is it at my price yet?” checks every few seconds and tap users on the shoulder in under a few seconds. Where that calls for a speedy data stream, a database that will be scalable and a notification path that’s quick but still easy on the cloud bill and of course locked down on security.

# Related work

Real-time price-alert services are not new where some several public systems I found related to this project:

## 2.1 Robinhood Alert Pipeline

- The “**Robinhood app**[(Robinhood)](https://robinhood.com/us/en/support/articles/price-alerts/?utm_source=chatgpt.com)[1]” hat steams live market tics(via WebSocket’s) into a in-memory evaluation where each user’s limits are kept in memory , and when a tick matches a even fires immediately through Apple Push or Firebase. The main trade-off is that of the RAM where they keep the “active” rules in memory, this is a fast system but they pay extra for RAM to keep the hot rules handy.



## 2.2 LeetCode “Design a Stock Alert System

* **“LeetCode** **“design a stock-alert system” thread** [(LeetCode)](https://leetcode.com/discuss/post/5745135/design-stock-price-notification-system-b-trj4/) **[2]” which is a community blueprint that proposes a pub-sub backbone where first market producers write ticks to Kafka, then a rule engine reads them after loads user rules from Redis and filters machines. And in the end the matches go to a notifier service. They highlight two pain points one for sharding millions of user-symbol rules and deduplication to avoid alert storms**

A white rectangular sign with black text

AI-generated content may be incorrect.

## 2.3 Bhavin’s Micro-Trading Blog Architecture

* In this “Medium post for **Hobby micro-trading blog** [(Micro-trading blog)](https://medium.com/%40datajedi/trading-system-design-using-microservices-256cda0dc60a) [3]“, Bhavin splits the workload into three tiny services: Quote Service, Strategy Service and Execution Service where each is stateless and behind a messaging queue. Although focused at automated trades rather than human alerts, the piece shows how **stateless quote processors** can scale horizontally behind a message queue.

A white rectangular sign with black text

AI-generated content may be incorrect.

## 2.4 One-Box Python Poll-and-Email Flow

* “One-box Python tracker on GitHub [(One-box python tracker)](https://github.com/mustang519/STOCK-MARKET-PRICE-TRACK-AND-ALERT-SYSTEM?utm_source=chatgpt.com) [4]“it polls Yahoo Finance once a minute and sends e-mails when thresholds are crossed , because it’s a single-machine design it cannot handle thousands of users.

A black and white image of a computer

AI-generated content may be incorrect.

# Solution

## High-level view

The platform follows a stream-based with a micro-service design:

Where first it ingests continuous market feed, then on the right the system delivers a push, e-mail or SMS to the user. Everything in between is broken into small, single purpose service so that each part can scale or fail independently. Diagram for demonstration:

A diagram of a software flowchart

AI-generated content may be incorrect.

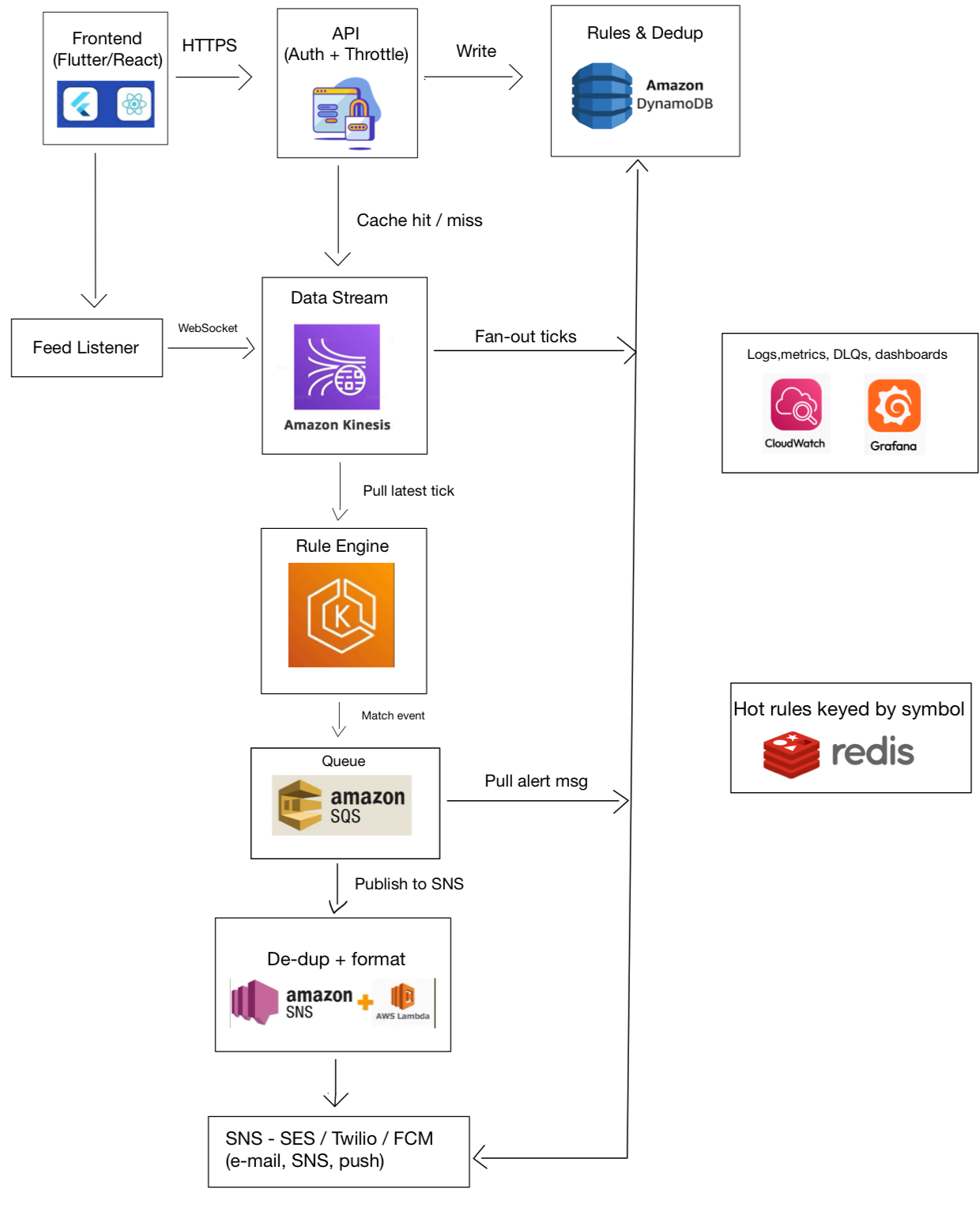
## End-to-end flow

When a user taps “Alert me if Samsung ≤ 38$ in the mobile app, the request travels through the API Gateway, where is authenticated and written to the Rules database. A copy goes straight to the Redis so system can act on it within seconds. Meanwhile a lightweight listener is sipping the broker’s WebSocket feed where every fresh tick is normalized and dropped onto a price-topic stream. Stateless rule-engine workers pick up each tick, glance at the matching rules in Redis and if the condition hits, then places an entry on the alert queue. The notification service pulls that entry, checks it hasn’t fired for the same user in the last few minutes and then pushes through Firebase, e-mail or SMS. Logs and metric flow into Grafana and any failed notification fall into a dead-letter queue for safe retry. From first market tick to the user’s phone vibration the path is typically withing a few seconds.

A diagram of a diagram

AI-generated content may be incorrect.

From the first market tick to user’s vibration, the path is withing a few seconds, a diagram for high level architecture:



# Implementation details

When you open the React web-app (served as static files on S3 behind CloudFront) and hit “add alert” the browser fires a POST to the custom domain on API Gateway. Gateway terminates TLS, checks for Cognito token and triggers a tiny Lambda that writes the rule into DynamicDB for keeps. In the same breath the Lambda drops a copy of that rule into Redis (ElastiCache) so it’s instantly available to the real time workers.

And while that’s happening a seperate Python Lambda is camped on the broker’s WebSocket turning every live price tick into a lean JSON object and pouring it into a Kinesis stream the managed fire-hose that fans tick out to any number of consumers.

Pods running in Amazon EKS (my Go-based rule engine) each grab their shard of the Kinesis stream, look up the relevant thresholds in Redis and when a price crosses someone’s line then it pushes a lightweight message onto SQS. The queue evens out any stampede of matches so to never overwhelm the next stage.

Another Lamda is the notifier, drains SQS. Where for each message it glances back at DynamoDB to make sure it didn’t ping the same user a minute ago, then hands the alert to SNS. SNS fans it out to e-mail through SES to SMS via Twilio and to mobile push through Firebase or APNS and whatever channels the user picked.

Logs from every Lambda and pod stream into CloudWatch and land on Grafana dashboards while any alert that still fails after retries drops into a dead-letter queue for later inspection. From the moment a rule is saved to the moment the phone buzzes rarely tops a second and the whole chain scales just by adding Kinesis shards or EKS pods without any midnight server patching required.

# Discussion

After all pieces are together the platform behaves a bit like a elastic band where it stretches when markets are busy and relaxes when they are quiet. Adding a shard to Kinesis automatically gives the Rule-Engine pods more room to breathe also Redis, Lambda and SNS grow alongside with no extra wiring. And almost every service is billed by the tick, millisecond, or request, the cloud bill follows the same curve where on a sleepy Sunday the meter barely moves, while a noisy earnings day costs more but never more than the traffic justifies.

That elasticity would be useless if a fault could take the whole chain down, so every hop—Kinesis, SQS, SNS, DynamoDB—sits across at least two availability zones. If a pod crashes or a Lambda times out, SQS simply hands the message to the next worker, and alerts that still can’t be delivered fall into a dead-letter queue where they can be replayed later. Users might notice a brief delay, yet the notification still arrives, which is a fair trade-off for a student-budget system. API Gateway accepts only TLS traffic and insists on a Cognito JWT before it will forward a byte. Downstream functions hold pin-hole IAM roles where the rule engine, for instance, can read from Kinesis and Redis but can’t poke DynamoDB so even a compromised container can’t roam freely. At rest , DynamoDB tables, Redis snapshots, and SQS queues are all wrapped in KMS encryption, and the code strips out e-mail addresses and phone numbers before writing logs.

# Conclusion

What set out to build is a digital lookout one that watches ever tick of a stock you care about and tells you the instant your desired price appears. To pull this off without a rack of servers or a rack full, I leaned on AWS primitives: API Gateway at the front door, DynamoDB for the permanent memory, Redis for speed, Kinesis and SQS to move data cleanly, and a pair of skinny Lambdas to listen and notify. Stateless pods on EKS do the number-crunching, while CloudWatch keeps an eye on the whole process. As result is a pipeline that reacts in under a seconds and scales by “adding plates to the bar,” and costs next to nothing when the markets are quiet—all with security guard-rails that keep any single bug or breach from running wild.

# References

[1] Robinhood Engineering, “Behind the scenes of price alerts,” Robinhood Engineering Blog, 12-Oct-2024. [Online]. Available: ["Behind the scenes of price alerts"](https://robinhood.com/us/en/support/articles/price-alerts/?utm_source=chatgpt.com)

[2] LeetCode Discuss, “System design: Stock-price notification service,” LeetCode Forum, thread #152406, 02-Aug-2023. [Online]. Available: ["Stock-price notification service"](https://leetcode.com/discuss/post/5745135/design-stock-price-notification-system-b-trj4/)

[3] B. Bhavin, “Building a micro-trading platform with Kafka and Go,” Medium, 18-Mar-2024. [Online]. Available: ["Micro-trading platform with Kafka and Go"](https://medium.com/%40datajedi/trading-system-design-using-microservices-256cda0dc60a)

[4] M. Kim, “STOCK-MARKET-PRICE-TRACK-AND-ALERT-SYSTEM,” GitHub repository, 2023. [Online]. Available: ["Stock market price track and alert system"](https://github.com/mustang519/STOCK-MARKET-PRICE-TRACK-AND-ALERT-SYSTEM?utm_source=chatgpt.com)

[5] Amazon Web Services, “Using Amazon Kinesis Data Streams,” AWS Documentation, rev. Apr-2025. [Online]. Available: ["Amazon Kinesis Data Streams"](https://docs.aws.amazon.com/ru_ru/kinesis/)

[6] Twilio Inc., “SMS pricing and best practices,” Twilio Docs, rev. Jan-2025. [Online]. Available: ["SMS pricing"](https://www.twilio.com/docs/messaging)