NYCPS Transportation Management System - AWS GovCloud Architecture (RFP R1804)

## **Status** : Draft

**Section Tracker**

| Section # | Section Name | Status |
| --- | --- | --- |
| 1 | Introduction | ReadyForReview |
| 2 | System Overview | Draft |
| 3 | System Architecture | Draft |
|  | Intro Section | ReadyForReview |
|  | 3.1 Presentation Layer | NotStarted |
|  | 3.2 Core Services Layer | NotStarted |
|  | 3.3 Event Processing Layer | ReadyForReview |
|  | 3.4 APIs & Communication Layer | NotStarted |
|  | 3.5 Data Services Layer | ReadyForReview |
|  | 3.6 Monitoring & Logging Layer | ReadyForReview |
|  | 3.7 Security Layer | ReadyForReview |
|  | 3.8 Full Architecture Diagram | ReadyForReview |
| 4 | GPS Tracking & Dynamic Routing | Draft |
| 6 | Data Model & Management | ReadyForReview |
| 6 | Throughput, Elasticity, & Scalability | ReadyForReview |
| 7 | Cost Efficiency | ReadyForReview |
| 8 | Network & Security | ReadyForReview |
| 9 | Appendix | Draft |
| - | Anything Else? | NotStarted |
| - | Anything that needs to be pulled back from the Archive Section? | NotStarted |

**TODO :** Ensure all the image links on this doc are accessible to the public before submitting. Currently, they are stored on the Sentry shared gDrive.

## 1. Introduction

This document presents the proposed technical architecture for Sentry SBMS (School Bus Management System), a next-generation Transportation Management System designed for NYCPS.

It is designed to meet the functional requirements specified in the Solution Design and the stringent non-functional requirements (performance, scalability, security, availability, compliance) outlined in RFP R1804. The architecture prioritizes deployment within AWS GovCloud (US) regions, leveraging managed services where possible to enhance reliability, scalability, and security while adhering to necessary compliance standards (e.g., FedRAMP High, ITAR, CJIS, HIPAA as applicable within GovCloud).

### 1.1 Document Organization

The current document is organized as follows:

| Section | Purpose |
| --- | --- |
| Section 1: Introduction | Provides a brief overview of the document and outlines its structure. |
| Section 2: System Overview | Describes the purpose of the system and outlines a high-level view of the technical components and their roles |
| Section 3: System Architecture | Presents the Sentry SBMS architecture through multiple perspectives, detailing how different layers and services interact |
| Section 4: GPS Tracking & Dynamic Routing | Explains how the platform handles real-time location data and route optimization |
| Section 5: Data Model & Management | Explains the rationale behind the selection of databases, structures, and schemas, and how they support system requirements |
| Section 6: Throughput, Elasticity, & Scalability | Covers how the system supports high-volume workloads with adaptable resource allocation |
| Section 7: Cost Efficiency | Highlights architectural and operational choices that optimize cloud spend |
| Section 8: Network & Security | Describes network layout, isolation, and security mechanisms across the platform |
| Section 9: Appendix | Includes references, definitions, and supporting materials |

### 1.2 Abbreviations & Acronyms

A selected list of commonly used abbreviations and acronyms is provided below for quick reference. The complete list is available in the Appendix.

| Abbreviation | Description |
| --- | --- |
| SMBS | School Bus Management System - Sentry’s proposed platform for NYCPS’s next-generation Transportation Management System |
| AWS | Amazon Web Services - A comprehensive cloud computing platform offering a wide range of services like computing, storage, and networking |
| EKS | Elastic Kubernetes Service - A managed Kubernetes service on AWS that simplifies the deployment and management of containerized applications |
| EC2 | Elastic Compute Cloud - A web-based service from AWS providing scalable virtual servers for on-demand computing capacity |
| ECS | Elastic Container Service - A fully managed AWS container orchestration service that enables developers to deploy, manage, and scale containerized applications efficiently in the cloud or on-premises environments |
| MSK | Managed Streaming for Apache Kafka - An AWS service that makes it easy to build and run applications using Apache Kafka for streaming data |
| SNS | Simple Notification Service - A fully managed AWS service for message delivery from publishers to subscribers using topics as communication channels |
| ETL | Extract, Transform, Load - A process used in data integration to extract data from sources, transform it into a usable format, and load it into a target system or database |
| VPC | Virtual Private Cloud - An isolated pool of resources within a public cloud environment, providing enhanced security and privacy through private IP subnets and virtual communication constructs |
| DNS | Domain Name System - A hierarchical naming system that translates domain names into IP addresses, enabling devices to locate and communicate over the internet |
| TTL | Time To Live - A concept used to define the lifespan or validity period of data. |

## 2. System Overview

**Web Portal**

Access to the Web portal is facilitated through Route 53, an Amazon Web Services (AWS) offering for domain registration and management. Authentication is conducted via the Department of Education (DOE) Identity Provider, resulting in the issuance of a token for the Sentry Web Portal. Network traffic is routed through AWS Shield and AWS Web Application Firewall, which provide Distributed Denial-of-Service (DDoS) protection and filtering based on IP addresses, HTTP headers and bodies, custom Uniform Resource Identifiers (URIs), Structured Query Language (SQL) injection patterns, and cross-site scripting vulnerabilities. AWS CloudWatch delivers real-time monitoring of the Web server, collecting and tracking metrics, logs, events, and performance data, which is then presented on custom dashboards. Automated alarms are triggered when predefined criteria are met.

Subsequent to authentication, users are routed through the Network Load Balancer to the Application Load Balancers, and subsequently to the Apache Web Service hosting the Web Portal application. The Web Service is deployed on a public subnet within an Amazon Elastic Container Service (ECS) Cluster composed of Elastic Compute Cloud (EC2) instances. Backend requests are routed to a parallel private subnet ECS Cluster, also composed of EC2 instances, where backend processes are executed. These processes involve the transfer of data to and from databases. Data is transmitted to an Elastic Kubernetes Service instance that hosts the Prometheus Database for logging and Grafana for reporting and dashboards, both of which are open-source tools implemented to minimize operational costs.

Data from the Web Backend is persisted to Amazon Aurora for Relational Database Service (RDS). When the backend retrieves data, it first queries the Redis ElastiCache for recent matching queries stored in the in-memory data store before accessing Aurora. This caching mechanism optimizes application performance by enabling rapid retrieval of frequently accessed data, thereby reducing database load and latency.

**Driver Mobile App**

Authentication for the driver application is conducted via the Department of Education Identity Provider (DOE IDP), with traffic routed through the Network Load Balancer (NLB) and Application Load Balancer (ALB) before reaching the public subnet Amplify Service. Amplify, a development framework and hosting service, facilitates a serverless architecture for applications deployed within AWS. On Amplify, traffic is processed through the Web Application Firewall (WAF) for DDoS mitigation and traffic filtering. Secure protocols direct user requests to the AWS API Gateway, which provides routing for REST, HTTP, and WebSocket APIs to the backend server located within the private subnet on an EC2 instance.

Dynamic data and event data are persisted to DynamoDB, while static data and metadata are persisted to Aurora. Redis ElastiCache is employed for retrieval requests; in the absence of cached queries, DynamoDB or Aurora is queried directly. Both databases implement an Extract, Transform, Load (ETL) process to archive historical data in S3 cold storage.

Drivers receive predefined routes from the backend system; however, a third-party routing API is utilized to provide real-time turn-by-turn navigation during routes. Scanners capture barcode and RFID data from student ID cards, transmitting this information to the backend servers to facilitate parental notifications regarding pickups and bus details.

**Parent Mobile App**

Parents will access the Parent/Student Mobile application with a username/password saved to , with traffic routed through the Network Load Balancer (NLB) and Application Load Balancer (ALB) to the public subnet Amplify Service. Amplify provides a serverless architecture for applications hosted on Amazon Web Services (AWS). Traffic undergoes filtering by the Web Application Firewall within Amplify. Subsequently, user requests are directed by Security to the AWS API Gateway for API routing to the backend server residing within an EC2 instance in the private subnet.

The Parent Application provides functionalities including bus route selection, real-time bus location streaming, and Estimated Time of Arrival (ETA) provision for pickup and drop-off locations. Additionally, the Parent Application allows adjustments to children's pickup and drop-off locations, which updates data within the Aurora database.

**GPS**

A dedicated GPS tracking device streams bus locations to the AWS IoT Core Service, which routes data to DynamoDB and AWS Location Services, a managed package for route tracking and geofencing. This allows real-time bus tracking and comparison of actual routes with predetermined DOE OPT routes. Data is passed to Amazon EventBridge to trigger Simple Notification Service if a route deviates or a bus leaves its geofenced location (NYC).

**Security Monitors**

VPC Flow Logs are collected at the VPC, subnet, and network interface level, capturing source/destination IP addresses, ports, protocols, packet and byte counts, time, and actions. CloudTrail logs AWS activity for each EC2, including API calls, console actions, data events, and insights events for unusual API activity. DNS Logging records domain/subdomain requests, date/time, DNS records, Route 53 edge locations, and DNS response codes. These tools store events in S3, integrate with Guard Duty for threat detection, and utilize Amazon Detective for ML statistical analysis. Information is passed to EventBridge and Simple Notification Service/Lambda based on the event/alert/activity. While Guard Duty detects threats, Amazon Detective is used by security operations for triage and investigation. Automated mitigation using Lambda is implemented for serious network compromises to isolate assets and facilitate forensic investigations.

**Reporting / Tracking**

All logging data is collected into Prometheus, an open-source systems monitoring and alerting toolkit. Grafana, an interactive web application, visualizes Prometheus data. This tech stack provides infrastructure monitoring for Sentry's platforms. Alerts will be generated from here for constant monitoring and updates on any items that require actions. For our frontend Web Portal, we will use Tremor and AG Grids to create custom made Dashboard, Reporting, and Grids ensuring we maximize flexibility to accomplish all current and future requirements. Tremor is a

## 3. AWS System Architecture

Sentry SBMS architecture is a cloud-native, event-driven microservices architecture deployed across multiple Availability Zones (AZs) within an AWS GovCloud (US) region. It is structured as a 7-layer application platform consisting of:

* **Presentation Layer:** Includes mobile and web applications that interact with backend services through APIs. These applications serve as the primary interface for end-users.
* **Core Services Layer:** Stateless compute services and containerized workloads handle business logic, event processing, and data transformation. A routing engine is included to manage complex decision flows and would leverage compute-optimized infrastructure where needed.
* **Event Processing Layer:** The system is designed for high-throughput ingestion, capable of handling millions of events per minute with low latency. Events are streamed and processed in near real-time through a horizontally scalable, partitioned messaging backbone that supports ordered, durable, and fault-tolerant delivery.
* **APIs & Communication Layer:** RESTful APIs and secure communication interfaces provide integration points for both external and internal consumers. Asynchronous messaging and notification services support decoupled communication between services.
* **Data Services Layer:** The platform uses a variety of database and storage solutions, chosen based on workload requirements such as high throughput, consistency, latency, and data structure.
* **Monitoring & Logging Layer:** A comprehensive observability stack is in place to capture application logs, metrics, and traces. Infrastructure and security logs are also collected to support audit, compliance, and operational visibility.
* **Security Layer:** The system uses a dedicated security monitoring subnet and integrates multiple AWS security services for multi-layered protection, threat detection, and secure secrets management. These tools work together to safeguard against DDoS attacks, malicious activity, and application-level threats while enabling automated event-driven responses and investigation workflows.

### 3.1 Presentation Layer

//Kenny to add

### 3.2 Core Services Layer

//HOLD until we get the Dynamic Routing Infra diagram from Kenny

### 

### 3.3 Event Processing Layer

The Event Processing Layer forms the real-time backbone of the system. It is responsible for ingesting, sequencing, transmitting, and storing high-volume, high-frequency events such as student tracking, vehicle telemetry, route status changes, and driver updates. This layer is optimized for throughput, reliability, and order-preservation in a highly distributed, horizontally scalable environment.

#### 

#### 3.3.1 Architecture Overview

This layer is composed of multiple independently scalable EC2-based microservices that act as Event Producers and Event Consumers, all communicating over Amazon MSK (Managed Streaming for Apache Kafka) as the central message bus. It is architected to seamlessly support millions of events per minute while maintaining the correct event sequence and delivery guarantees.

**Event Types & Rates**

| **Event Type** | **Producer** | **Kafka Topic** | **Expected Rates /Min (with Buffer) \*** |
| --- | --- | --- | --- |
| Student Boarding Event | EC2 Student Event Service | student.boarding.events | 310,000 |
| Student Disembarking Event | EC2 Student Event Service | student.disembarking.events | 310,000 |
| Vehicle Location Snapshot | EC2 Vehicle Tracker Service | vehicle.location.snapshot | 44000 |
| Route ETA Update | EC2 Route Management | route.eta.updates | 20000 |
| Route Status Update | EC2 Route Management | route.status.updates | 20000 |
| Driver Status Update | EC2 Driver Event Service | driver.status.updates | 22000 |
| Exception Event (e.g. Delay) | EC2 Exception Monitor | system.exception.events | 22000 |
| Parent Notification Trigger | EC2 Notification Service | parent.notify.events | 620,000 |
| System Audit Event | All Microservices | system.audit.events | 20,000 |

\* Buffer factor of 2x to account for unexpected spikes

\* Numbers are rounded up for simplicity. Students - 153000 to 155000, Buses - 10500 to 11000

#### 3.3.2 Key Components

Event producers, consumers, sequencers, and MSK work together to ensure reliable, ordered event delivery and decoupled communication within the system

##### 3.3.2.1 Event Producers (EC2 Microservices)

Each event producer is a standalone EC2 service responsible for capturing and publishing domain-specific events to MSK topics. These include:

* Student Events (e.g., boarding, disembarking, missed pickups)
* Vehicle Tracking Events (e.g., Location snapshots, route deviations)
* Route and ETA Updates
* Driver Activity Updates

To ensure event consistency and correct ordering in a multi-threaded and multi-process environment, each producer integrates with a sequencer. This sequencer is responsible for assigning sequence numbers to events per entity (e.g., per student or per vehicle) before publishing them to Kafka. This guarantees in-order delivery for consumers that are grouped by key (e.g., student\_id or vehicle\_id).

##### 3.3.2.2 Sequencer

In distributed systems, maintaining the order of event publication is non-trivial, especially when multiple threads or processes produce events concurrently. The sequencer component within each producer ensures:

* Entity-level ordering: Events for the same logical entity (e.g., a student or vehicle) are assigned incrementing sequence numbers.
* Idempotent and deterministic publishing: MSK is configured with partition keys aligned with the entity ID to ensure order preservation within each Kafka partition.
* Retry-safe mechanisms: In case of network failures or partial writes, the sequencer re-evaluates the last committed sequence number to avoid duplicate or out-of-order events.

##### 3.3.2.3 Amazon MSK (Kafka Cluster)

MSK serves as the central event bus, allowing decoupled communication between producers and consumers with the following benefits:

* Horizontal scalability via partitioned topics
* High throughput for handling millions of events/min
* Replayability of events for audit, reprocessing, or ETL
* Entity-based partitioning for deterministic consumer behavior

Each domain (e.g., vehicle, student, route) is mapped to a dedicated Kafka topic, ensuring clean separation of event flows and granular control over consumer processing.

##### 3.3.2.4 Event Consumers (EC2 Microservices)

Event consumers are EC2-based services that subscribe to Kafka topics, ingest ordered streams of events, and process them based on business rules.

* Consumers ensure idempotency by tracking the last processed sequence number per entity to prevent duplicate effects.
* Processing includes business logic, real-time alerts, and routing events to downstream systems (e.g., DynamoDB for persistent storage).
* Consumers would independently scale out to meet increased event loads, as each Kafka topic would have multiple partitions and consumer groups.

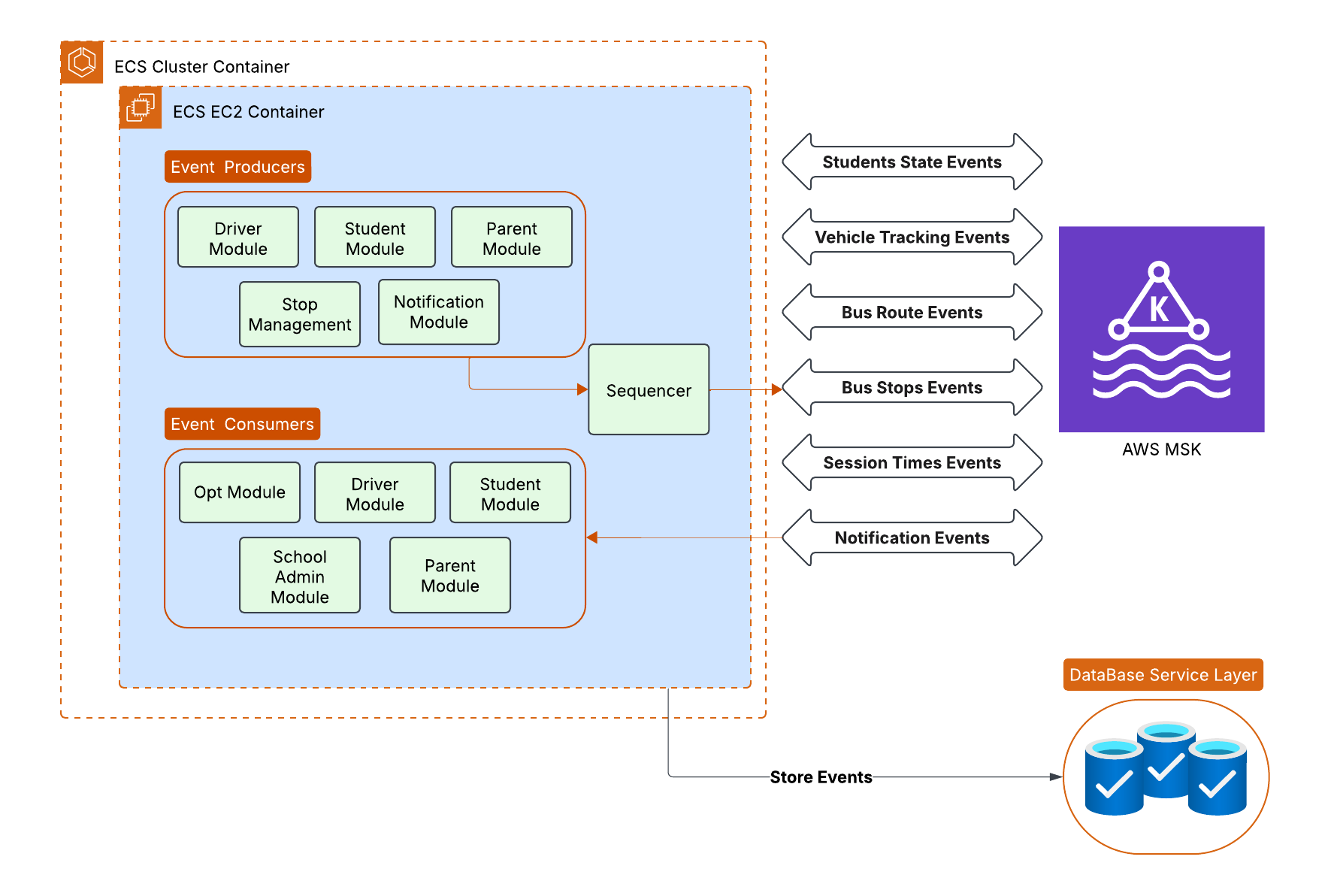
#### 3.3.3 Scalability and Resilience

* Elastic EC2 Scaling: Both producers and consumers run on EC2 Auto Scaling Groups that dynamically adjust based on CPU utilization, network I/O, or custom Kafka lag metrics.
* Kafka Partitioning ensures scalability and isolation of event streams. New partitions would be added as the event volume grows.
* At-Least-Once Semantics are enforced to ensure events are not lost, even in the face of service restarts or infrastructure failures.
* Sequence Guarantees ensure correctness in consumer state machines even under parallel processing.

#### 3.3.4 Summary

The Event Processing Layer is built for high throughput and scalability, capable of handling millions of real-time events per minute with low latency. Leveraging EC2-based producers and consumers and Amazon MSK (Kafka), it ensures reliable, ordered event delivery and decoupled communication. Kafka, used by industry leaders for processing millions of events per second, powers the system's high-volume event streams, providing both scalability and fault tolerance.

The architecture supports horizontal scaling, allowing the system to dynamically adjust to traffic spikes with auto-scaling EC2 instances and Kafka partitioning for load distribution. Event consumers ensure idempotency and sequence guarantees, while resilience is maintained through at-least-once semantics, preventing data loss and ensuring consistent state.



**Figure 3.3 Event Processing Diagram**

### 

### 3.4 APIs & Communication Layer

The APIs & Communication Layer acts as the interface between the frontend clients (web and mobile applications), internal services, and event-driven subsystems. It is responsible for enabling both synchronous and asynchronous communication across the system with high reliability, security, and scalability.

### 3.4.1 APIs (Synchronous, Asynchronous, Streaming)

The platform exposes a suite of APIs that support multiple interaction models , including traditional request-response flows and asynchronous operations and streaming responses.. These APIs power various frontend experiences and serve as the communication backbone for internal service interactions.

The request flow is designed with a layered security and routing model:

* AWS WAF (Web Application Firewall): Protects the APIs against common attack patterns, such as SQL injection, cross-site scripting (XSS), and bot-based threats.
* Application Load Balancer (ALB): Handles routing logic, SSL termination, and request forwarding.
* API Gateway: Acts as the central entry point for all external and internal API traffic, enabling rate limiting, caching, and request validation.
* EC2-based Services: The final destination of the request is the EC2-based application layer, which processes the request, triggers necessary events or database transactions, and generates the response.

The APIs are capable of:

* Standard synchronous operations (e.g., fetching student details, retrieving bus routes)
* Asynchronous workflows (e.g., requests for report generation or notification triggers)
* Streaming responses (e.g., live location tracking of buses or real-time student status updates), leveraging long-lived connections where appropriate (e.g., WebSockets, SSE, or other future iteration methods)

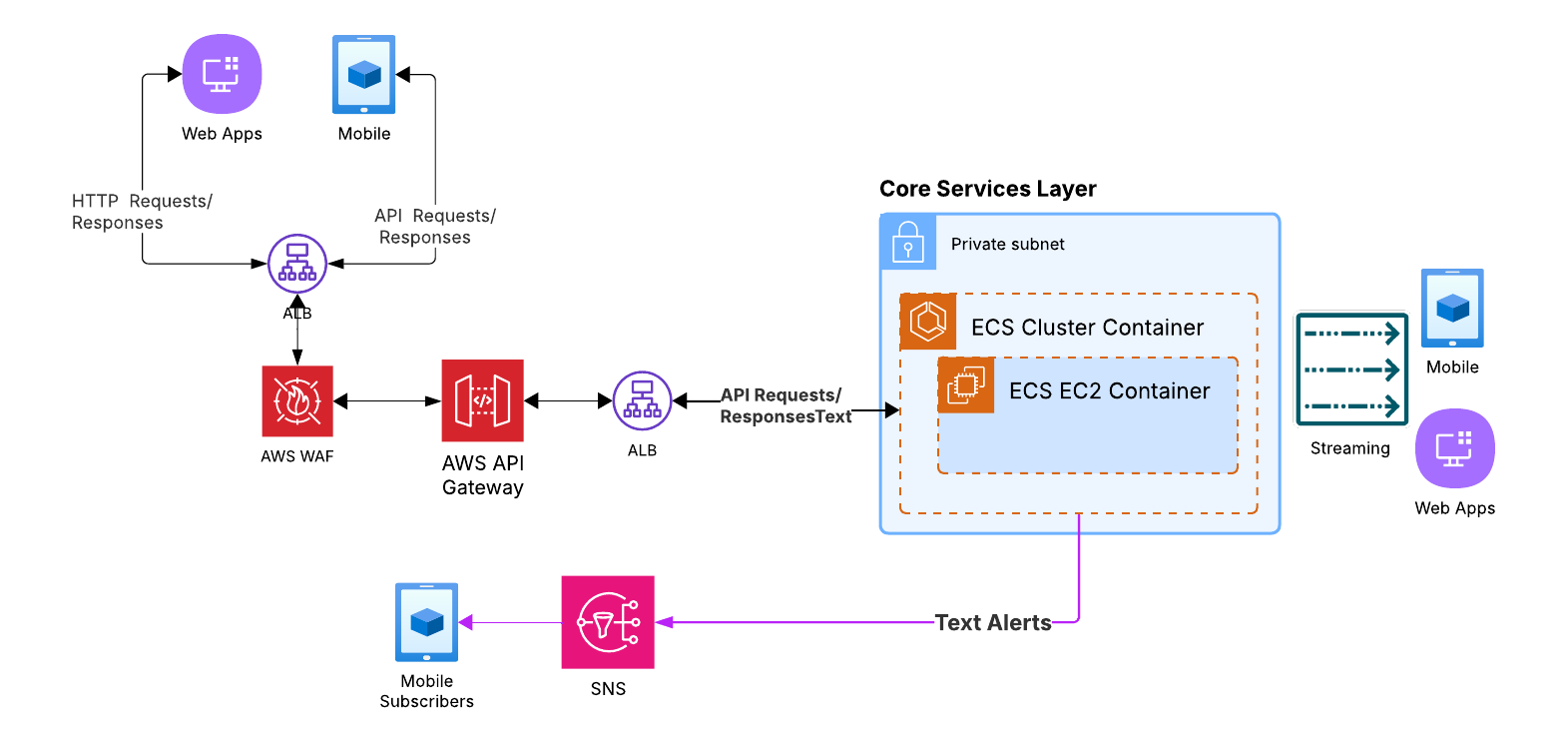
This API design ensures flexibility and future extensibility for hybrid interaction models.

### 3.4.2 SNS-Based Asynchronous Messaging

To support cross-component communication and decoupled workflows, the system uses Amazon SNS (Simple Notification Service) for asynchronous notifications and message fan-out.

SNS is primarily used for:

* Sending push notifications to mobile apps
* Triggering alerting mechanisms (e.g., SMS, email, or app banners)
* Broadcasting system-level events (e.g., bus delay notifications, school-level announcements)



**Figure 3.4 APIs & Communication Diagram**

### 3.5 Data Services Layer

The Data Services Layer is responsible for the storage, access, and high-speed retrieval of both real-time and static data across the system. This layer supports a mix of transactional, time-series, metadata, and cache-optimized data stores, each selected to meet the performance, scalability, and consistency requirements of their respective workloads.

#### 3.5.1 Amazon DynamoDB (Event Store)

DynamoDB serves as the primary storage engine for all real-time message events produced and consumed within the system. It is designed to handle extremely high write-throughput, with single-digit millisecond latency, ensuring that the system would efficiently capture and store large volumes of event data , such as vehicle tracking updates, student transit events, and route statuses , in real time.

DynamoDB’s built-in support for horizontal scaling allows it to automatically adjust capacity based on demand, enabling it to seamlessly handle spikes in traffic without manual intervention. This elasticity ensures that even during peak traffic periods, such as during heavy school commute times or sudden route updates, the system remains highly responsive and continues to process events without delay or degradation in performance.

To further optimize performance, DynamoDB tables are partitioned and indexed appropriately, ensuring efficient querying of both recent and historical events. The integration with Amazon MSK enables near real-time ingestion of event data, ensuring that data flows smoothly into the system with minimal lag. This combination of high throughput, low latency, and elastic scaling allows the system to consistently deliver real-time insights, regardless of fluctuating traffic volumes, while maintaining cost-efficiency by adjusting resources dynamically based on actual usage.

#### 3.5.2 Amazon AuroraDB(Relational Data Store)

Aurora (PostgreSQL-compatible) is used to manage structured, relational data representing static or slow-changing entities in the system. This includes buses, drivers, stops, routes, parents, students, and configuration metadata. Aurora is designed to deliver high throughput and low-latency performance compared to traditional relational databases like PostgreSQL and MySQL, making it suitable for handling large-scale operational workloads.

Aurora provides strong consistency, support for complex joins, and transactional integrity , essential for core service logic and API interactions that require accurate, up-to-date reference data. Aurora's distributed, fault-tolerant architecture ensures better scalability and resilience, particularly when managing significant event volumes or complex queries.

#### 3.5.3 Amazon S3(ETL, Auditing & Reporting Layer)

Amazon S3 is utilized for storing extracted and transformed data from both DynamoDB and Aurora, serving as a central repository for analytical, auditing, and reporting purposes. AWS Glue is used to extract and transform the data from these sources before storing it in S3.

As a highly scalable and cost-effective solution, S3 supports long-term data retention and batch processing of large datasets. It stores historical event data and metadata snapshots, allowing for efficient querying and analysis. This setup ensures that past events and system state changes would be easily accessed and utilized for various analytics, auditing, and reporting needs, helping to drive informed decision-making.

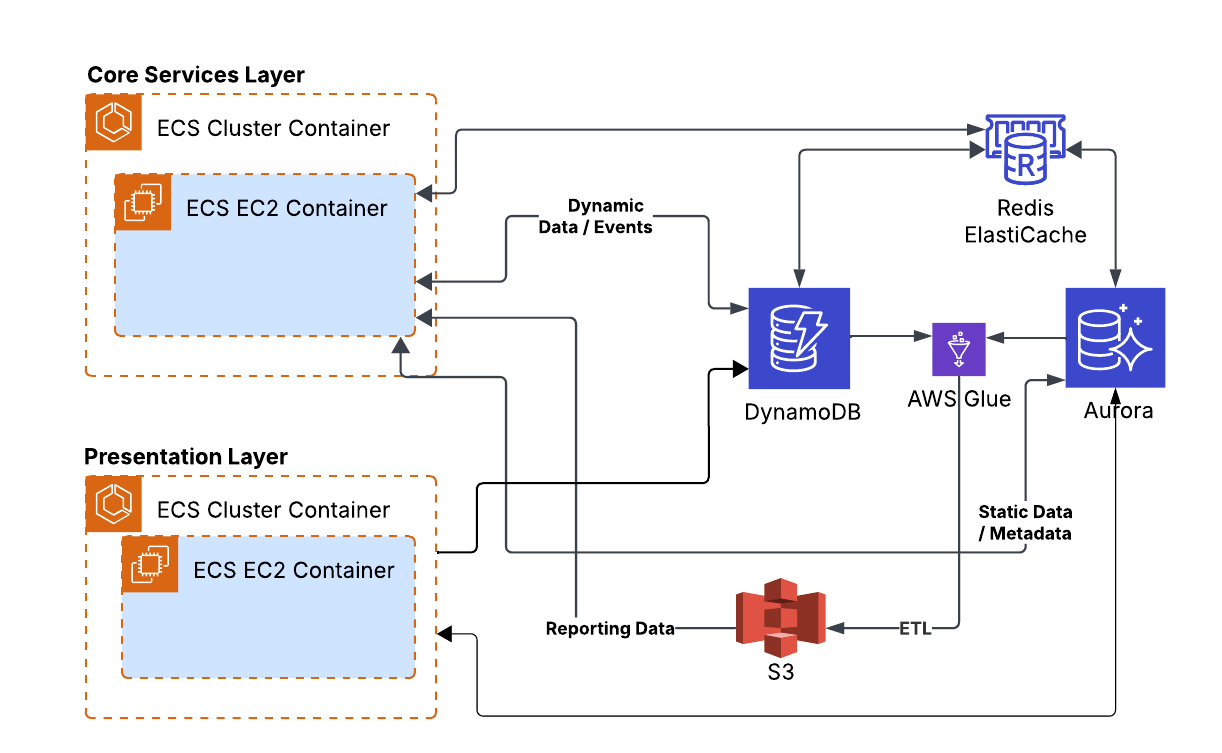
#### 3.5.4 Redis ElastiCache (Low-Latency Caching Layer)

Redis is deployed as a caching layer to improve read performance across frequently accessed datasets , both from Aurora (e.g., stop metadata, route lookup) and DynamoDB (e.g., recent event lookups, in-transit status). It is used to reduce read amplification and latency across the platform’s high-throughput APIs and internal services.

This cache layer is especially critical during peak hours, ensuring that response times remain consistently low even when the system scales over 10x.

This multi-tiered data architecture ensures:

* Real-time performance where it matters (Redis + DynamoDB)
* Relational consistency and strong querying for business-critical entities (Aurora)
* Scalable archival and reporting (S3)
* Seamless integration with the event-driven pipeline (via MSK and DynamoDB)



**Figure 3.5 Data Services Diagram**

### 

### 3.6 Monitoring & Logging Layer

The Monitoring & Logging Layer plays a critical role in ensuring the health, reliability, and observability of the overall system , especially given its high-throughput, event-driven nature. This layer is responsible for real-time collection of logs and metrics, root-cause diagnostics, anomaly detection, and automated alerting for both infrastructure and application events.

#### 3.6.1 Application Monitoring & Logging

The application layer is monitored using a cloud-native observability stack orchestrated on Amazon EKS:

* **Promtail** runs on the EC2 instances and is responsible for scraping logs from containerized microservices running on those instances. These logs include event traces, error logs, throughput statistics, and internal service state transitions.
* **Grafana Loki**, which is deployed on EKS, serves as the centralized log aggregation system, efficiently handling the scale and volume of logs. It stores and indexes logs based on various attributes like service name, route, event type, or other relevant metadata for fast querying and troubleshooting.
* **Prometheus**, also running on **EKS**, collects real-time metrics from the application layer, tracking key business and infrastructure performance indicators such as:
  + Bus route latency (e.g., time taken for buses to complete a specific route or segment)
  + On-time performance (e.g., percentage of buses arriving at stops within the expected timeframe)
  + Bus occupancy rate (e.g., average number of commuters on a bus at any given time)
  + Event processing volume (e.g., the number of real-time events processed per minute)
  + Average bus wait time (e.g., time spent by buses at each stop)
  + Kafka queue lag and throughput
  + API response times (e.g., time taken to process route or student-related requests)
  + Container resource usage (CPU, memory, I/O)
  + Redis cache hit/miss ratios
* Both metrics from Prometheus and logs from Loki are visualized in **Grafana Dashboards**, providing the monitoring teams with insights into system performance, potential bottlenecks, and service availability across multiple zones and use cases.

This observability pipeline is fully scalable and resilient, ensuring end-to-end monitoring even during peak load when the system is scaled over 10x.

#### 3.6.2 Infrastructure Logging

To maintain deep visibility into network, infrastructure, and security-level operations, the system integrates several AWS native services:

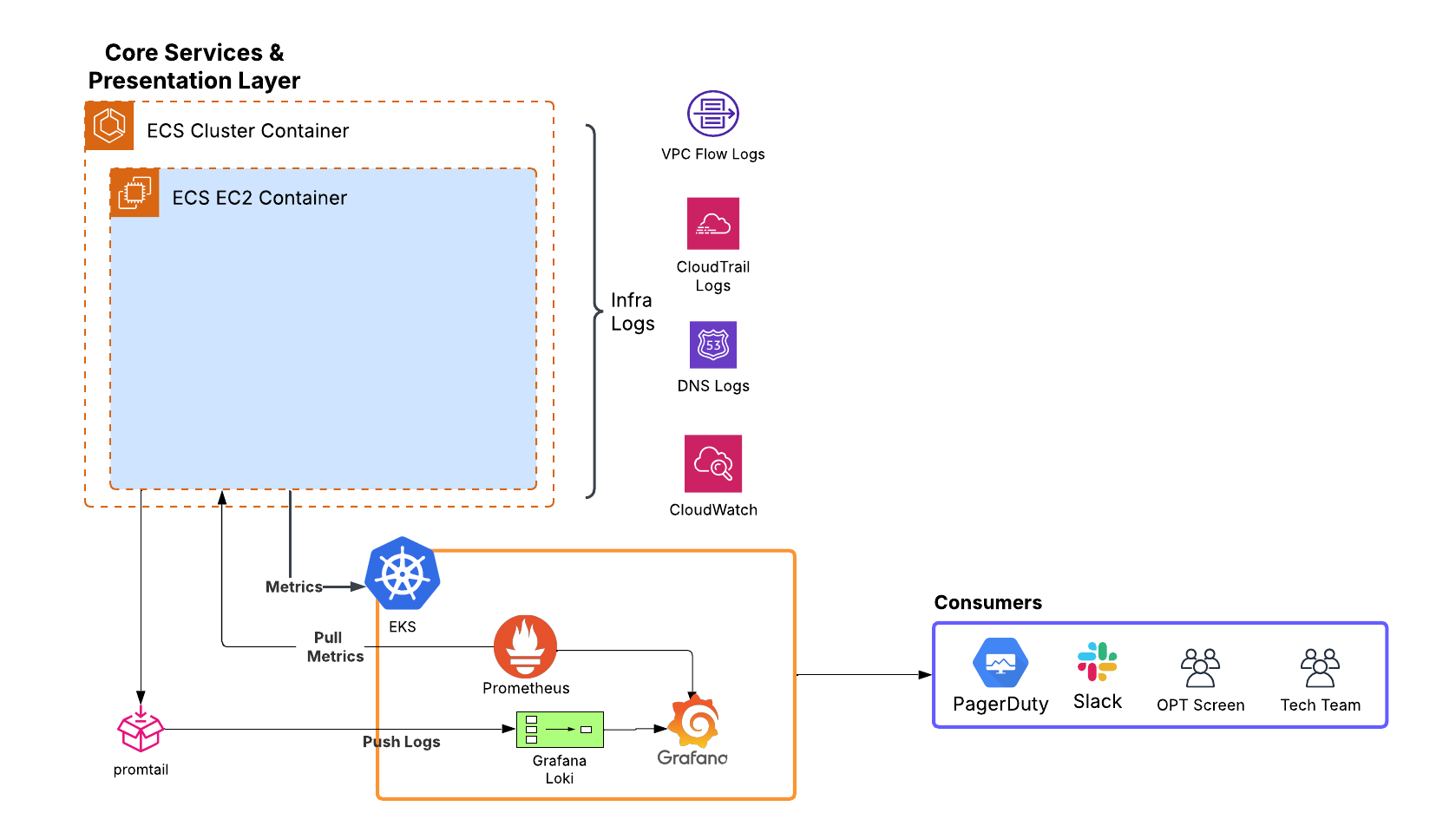
* **AWS CloudWatch Logs** capture logs from EC2 instances, autoscaling groups, and system-level operations across services.
* **CloudTrail Logs** provide a complete audit trail of all user/API activities across the AWS account, essential for compliance, debugging, and tracing configuration changes or access anomalies.
* **DNS Logs** (via Route 53) track DNS resolution activity to detect misuse, outages, or unexpected spikes in traffic.
* **VPC Flow Logs** monitor all ingress and egress traffic across subnets and interfaces, providing full visibility into network communication patterns , crucial for both security and performance diagnosis.

Together, these tools form the infrastructure backbone for logging and would be integrated into automated security monitoring and alerting pipelines.

#### 3.6.3 Alerting, Notifications Consumer Tools

Monitoring is only as effective as the actions taken in response. To that end, the system uses a combination of tools to ensure real-time incident awareness and resolution:

* **PagerDuty** is used for on-call alerting and incident escalation. Custom rules are defined based on critical metrics or failure patterns (e.g., service downtime, processing latency spikes, DynamoDB throttling).
* **Slack Integration** allows for real-time alerts and system health summaries to be pushed directly into dedicated operations and engineering channels, promoting rapid response.
* OPT Admin & Tech Team UI Dashboards provide real-time visibility into critical components of the system (e.g., active routes, stuck commuter events, failed message retries), with role-based access for operational users, customer support, and engineering



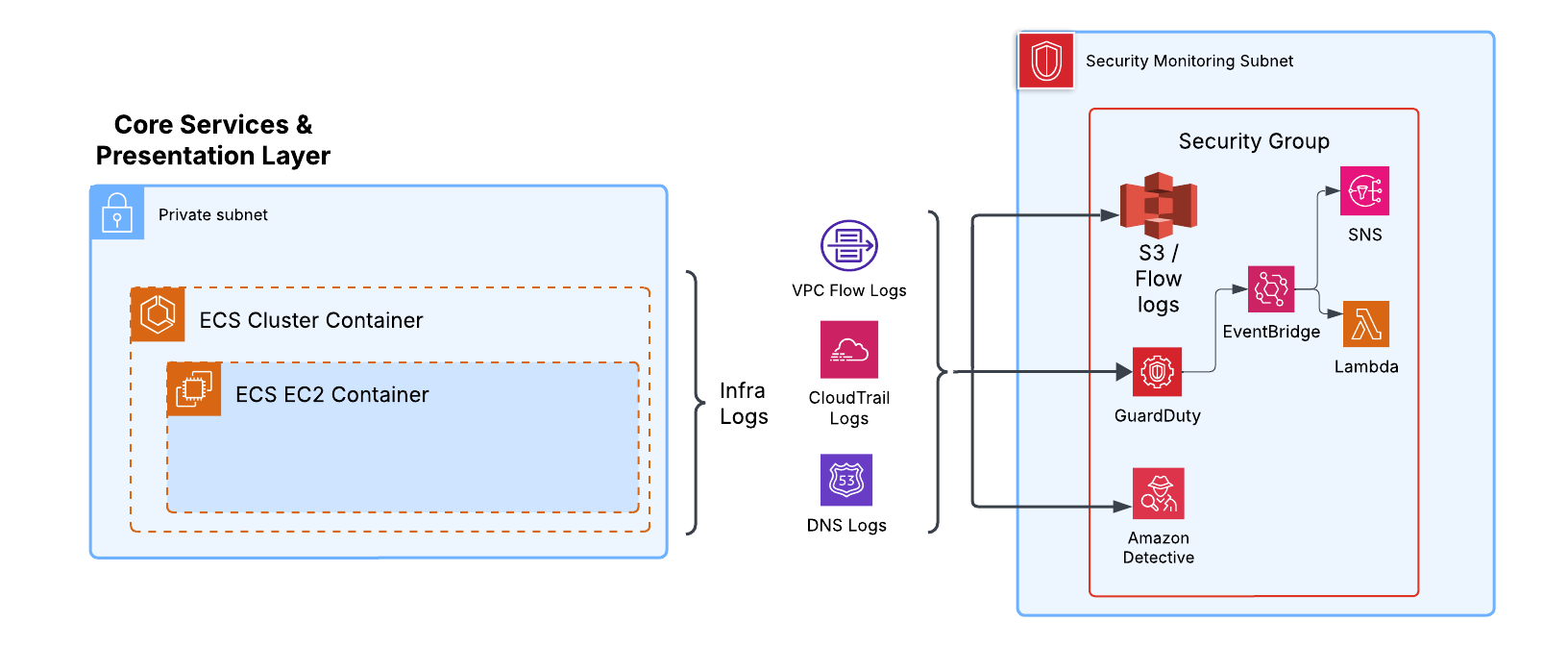
**Figure 3.6 Monitoring & Logging Diagram**

### 3.7 Security Layer

The security monitoring layer comprises systems that collect security-related events from throughout the VPC (both public and private subnets), including network, infrastructure, application, and management events. These events are analyzed to identify anomalies and detect known signs of adversarial activity within the network. The event sources include:

* **VPC Flow logs:** These logs capture the metadata from traffic flow within the VPC, including the source/destination IPs, port, protocol and connection status. Any anomaly in these flows indicate lateral movement by an adversary.
* **DNS logs:** They capture the queries and responses, along with the source IP, which is valuable in spotting anomalies, as well as data exfiltration attempts using DNS tunneling.
* **CloudTrail logs**: These logs capture API events, along with source IP, identity and parameters, which is helpful to detect unauthorized access, privilege escalation, service misuse and other suspicious activity.

These logs would be fed to **GuardDuty** for analysis and the findings would be routed to SNS for alerting. Certain high confidence, severe incidents detected by GuardDuty are fed to lambda for immediate mitigative action, such as network isolation of compromised services. Other events are triaged and investigated by the SOC team using **Amazon Detective**. We would also store VPC Flow logs on S3 for long term archival storage for use in forensic investigations.

****

**Figure 3.7 Security Layer Diagram**

### 3.8 Full Architecture Diagram

The following diagram provides a high-level view of the complete system architecture, illustrating how the various layers interact within the platform.

[**View Full Architecture Diagram**](http://drive.google.com/uc?id=1PTwLBljPVsrdClP9BaTyImQAwnLRcaD3)

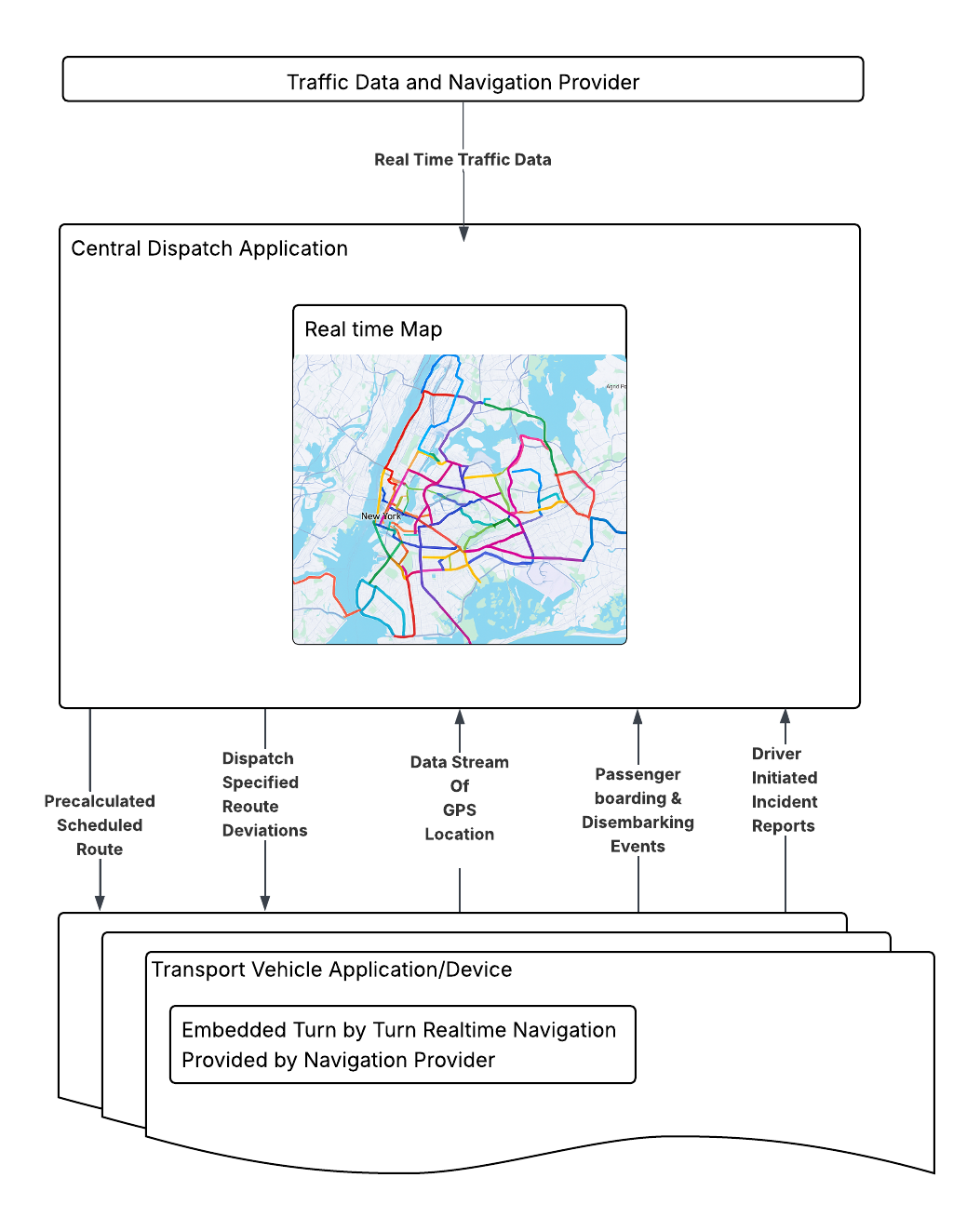
For detailed explanations of each layer, refer to the corresponding sections (3.1 to 3.7) above.

## 4. GPS Tracking & Dynamic Routing

### **4.1 Dynamic Dispatch overview**

Sentry SBMS offers a pragmatic balance between providing transport providers autonomy and maintaining a high level of flexibility, observability and accountability for the dispatch team. The system was designed to prioritize the following:

* Providing a birdseye view of active routes across the entire service area with real-time vehicle location, with passenger & driver information, expected time of arrival, and vehicle speed.
* Integrating live traffic data feeds provided by both government agencies, navigation providers, and the vehicle operators into the Live Map.
* Anticipating disruptions to current and upcoming vehicle routes and proactively providing alternate routes, while tracking updated ETAs through the alternate routes
* Tracking the status and location of all Drivers on their routes managed by the dispatch system, as well as all passenger embark and disembark events.
* Issuing warning and alerts for non-trivial route deviations, and
* Allowing for manual intervention by dispatch agents to modify both the existing trips as well as the pre-scheduled routes to react to unforeseen circumstances.



### **4.2 Transport Vehicle Application & Device**

Each driver managed by the SBMS will be utilizing the Driver Mobile Application to improve the simplicity and punctuality of their driving experience. The Application will provide turn-by-turn navigation in a couple of flavors. The embedded Map application will show 2 separate routes representing:

1. the fastest route available for their vehicle class (As presented by their Navigation Provider of choice. I.E. Google, Waze,Apple)
2. Dispatch Provided Route which is either the default route, or an alternate route provided via dispatch agent intervention.

The Routes provided will include all necessary stops, and for Busses that provide Door-to-School service, will intelligently skip stops that are not necessary on that particular date due to Planned Student Absences or Student having an alternate mode of transportation.

The onus is on the driver to choose between the Navigation Provider’s optimized route and their routine predetermined route. We want to give the flexibility to react to developing traffic and road conditions. Drivers will also be given the ability to relay current traffic and road statuses to SBMS cloud via quick buttons on their navigation application. These statuses will be highlighted in the Central Dispatch Application and can be actioned upon by dispatch agents.

Each vehicle managed by the SBMS will actively feed their GPS Location to the SBMS cloud application. This GPS data will be tracked and sent via the mobile application through mobile device’s 5g data plans as well as through Satellite GPS Tracking Devices which will work without cellular coverage. This ensures that GPS tracking is always on. There’ll also be onboard accelerometers and gyrometers to determine the speed of the vehicles. This information will be ingested into SMBS Cloud similar to the GPS data. This combination of data allows us to provide insight on the driver as well as the routes they are provided.

### **4.3 Central Dispatch Application**

The Central Dispatch Application (CDA) will provide dispatch agents with a near-Omniscient view of the active vehicle operators under their management. An interactive map will be the central focus of the application. Key Features of the Application are:

* View the Location and Status of all in progress trips.
* Drill down to individual Trips to see real time data on the Driver, Vehicle and Passengers
* View Real time Traffic and Road Conditions Road Data
* Create, view and edit predefined routes for Vehicles
* Update routes for enroute vehicles to suggest alternate routes based on information
* Notify users of trip anomalies such as non-trivial route deviations, vehicle speed outliers, major estimated time of arrival increases, route-affecting traffic conditions, etc.
* Allow users to set up new alerts based on the multiple real-time data streams the application is enriched with.

### **4.4 Dispatch Orchestration & Automation Services**

Navigation optimization is a fairly complex problem. When compounded with New York’s very dynamic Traffic and Road conditions, it presents an exponentially elevated dependency on compute resources. For this large SBMS deployment that spans thousands of intersecting routes, economies of scale can be used to lower the compute / External API Service Calls. Some resource-optimization strategies SBMS will utilize are:

* Combining A-Star and Sectioning Algorithms with an optimized data structure that represents a map as a graph to perform resource efficient shortest path algorithms.
* Caching common alternate paths to road closures/congestions that happens more frequently to save on External API Calls and internal pathing calculations
* Updating existing route templates based on historical route deviation and associated travel time data improvements.
* For vehicle routes that share common paths, use the first vehicle’s actual route traveled and travel time to update the optimal path for subsequent vehicles.

## 5. Data Model & Management

Sentry SBMS is designed to handle real-time tracking and state management for a large fleet of vehicles, numerous stops, and a significant number of passengers. Given the high velocity and volume of data, the data model adopts a polyglot persistence strategy, optimizing for scalability, performance, and real-time responsiveness to meet the demands of a high-throughput system

### 5.1 Data Storage Overview

To meet the diverse requirements of real-time ingestion, low-latency access, durable storage, and analytical processing, the system utilizes four primary data storage and caching layers:

| **Purpose** | **Technology** |
| --- | --- |
| Real-time event ingestion & state storage | **DynamoDB** |
| Relational metadata and static configuration | **AuroraDB (PostgreSQL)** |
| Audit logs, reporting, and ETL | **Amazon S3** |
| High-speed access and hot data caching | **Redis** |

### 5.2 Real-Time Event Data Store (DynamoDB)

DynamoDB is the primary store for high-velocity message events coming from the transportation network.

**Use Cases:**

* Bus location snapshots (every 30s per vehicle)
* Stop arrival/departure timestamps, ETA
* Student state transitions (boarding, in-transit, disembarking)
* Route progress (stop completion, ETA calculations)

**Data Characteristics:**

* Time-series nature
* High write throughput
* Primarily write-heavy with occasional queries for the latest state
* Requires TTL (Time to Live) for event data retention to manage cost

**Schema Highlights:**

| **Table Name** | **Partition Key** | **Sort Key** | **Attributes** | **Global Secondary Indexes (GSIs)** |
| --- | --- | --- | --- | --- |
| BusEventStream | bus\_id | event\_timestamp | location, speed, route\_id, stop\_id, status, event\_type, ETA | GSI 1: route\_id (Partition Key), event\_timestamp (Sort Key) |
| StopEventStream | stop\_id | event\_timestamp | bus\_id, status, route\_id, event\_type, actual\_arrival\_time, actual\_departure\_time | GSI 1: bus\_id (Partition Key), event\_timestamp (Sort Key) |
| StudentStateTransitions | student\_id | event\_timestamp | bus\_id, stop\_id, transition\_type (boarding, disembarking), status | GSI 1: bus\_id (Partition Key), event\_timestamp (Sort Key) |
| RouteTracker | route\_id | bus\_id#stop\_id | status, event\_timestamp, ETA, completion\_flag | GSI 1: stop\_id (Partition Key), event\_timestamp (Sort Key) |
| ExceptionEvents | event\_type | event\_timestamp | bus\_id, route\_id, description, severity, ack\_status | GSI 1: severity (Partition Key), event\_timestamp (Sort Key) |

**Rationale for DynamoDB:**

* Scalable and High-Throughput: DynamoDB handles high-velocity workloads with automatic scaling, making it ideal for ingesting millions of real-time events per minute without manual intervention.
* Flexible Data Modeling: Its schema-less design supports dynamic, high-velocity event data, such as location updates and status changes, without a rigid schema structure.
* Low Latency: Optimized for low-latency reads and writes, ensuring real-time access to the latest event data with minimal delay.
* High Availability: Offers multi-region replication and automatic data redundancy across AZs, ensuring fault tolerance and resilience.
* Cost-Effective for Write-Heavy Workloads: DynamoDB’s pricing model supports variable traffic patterns, providing a cost-effective solution for high write throughput.
* Optimized for Event-Driven Architectures: The schema design in DynamoDB efficiently partitions and scales event data, ensuring fast querying and processing of high-velocity, time-series event streams across various use cases.

### 5.3 Relational Metadata Store (AuroraDB)

Aurora serves as the system-of-record for all static and slowly changing data.

**Use Cases:**

* Bus information, capacity, assignments
* Driver profiles and schedules
* Stop locations, sequencing, service windows
* Route definitions and timetables
* Student profiles (e.g., students, their pickup/drop preferences)
* Parent-child relationships for notifications and visibility

**Schema Highlights:**

| **Table Name** | **Fields** | **Partition Key** | **Indexes** |
| --- | --- | --- | --- |
| buses | bus\_id, license\_plate, capacity, assigned\_driver\_id | bus\_id | license\_plate, assigned\_driver\_id |
| drivers | driver\_id, name, license\_info, availability | driver\_id | availability , name |
| stops | stop\_id, location, name, type | stop\_id | location , name |
| routes | route\_id, name, list\_of\_stop\_ids, service\_days | route\_id | name, service\_days |
| students | student\_id, name, type, associated\_parent\_id | student\_id | associated\_parent\_id , name |
| parents | parent\_id, name, contact\_info | parent\_id | name |

#### 

**Rationale for RDBMS:**

* Enforces strong consistency and relational integrity
* Complex joins needed for operational dashboards and internal tools
* Supports transactional updates and constraints

### 5.4. Data Lake for Audit & Analytics (Amazon S3)

Amazon S3 serves as a cold-storage and ETL sink for long-term storage, audit trails, and analytical reporting.

**Use Cases:**

* Archiving bus event streams after TTL expiration in DynamoDB
* Analysis of route performance, delays, driver behavior
* Auditable event history (e.g., when did a student board or exit a bus)

**ETL Strategy:**

* AWS Glue jobs to extract data from DynamoDB and Aurora
* Scheduled batch ETLs
* Partitioning by relevant attributes (e.g., date, route\_id, event\_type) to optimize query performance and access

### 5.5 High-Speed Cache Layer (Redis)

Redis is used as an in-memory cache to serve frequently accessed data with low latency.

**Caching Layers:**

* On top of DynamoDB: To serve real-time bus states, route progress, and active stop ETAs
* On top of AuroraDB: To serve stop metadata, route configurations, and student details during live trip processing

**Use Cases:**

* API layer performance (sub-ms access to current bus location or ETA)
* Dashboard real-time metrics
* Current bus load or occupancy per route
* Pre-fetched route metadata during trip initialization

**Caching Strategy:**

* TTL-based expiry (5–10 min for dynamic state)
* Write-through caching for frequently accessed metadata
* Pub/Sub or event-driven invalidation

### 5.6 Data Modeling Considerations

**Scalability:**

* DynamoDB partitions designed for high write throughput with predictable performance.
* Aurora horizontally scales for read replicas to serve dashboards and reporting tools.

**Durability and Retention:**

* Storage: Tiered storage using S3 Standard, S3 Intelligent-Tiering, S3 Glacier, and S3 Glacier Deep Archive to meet access needs and 7-year retention requirements cost-effectively
* Short-lived real-time state stored in Redis for rapid access.

**Cost Optimization:**

* DynamoDB TTL + S3 archiving for cost-effective retention
* Redis for hot data only; cold metadata queries hit Aurora directly
* Use of caching layers minimizes database load and ensures linear cost scalability

**Consistency & Synchronization:**

* Eventual consistency in DynamoDB is acceptable for tracking use cases in the bus tracking system. For **1million** event writes per minute, updates such as bus locations or event states typically experience a propagation delay of **less than 1 second**
* Aurora handles strongly consistent operations like updates to driver schedules
* Redis caches are invalidated based on changes via pub/sub or event triggers

### 5.7 Summary

The data model and management strategy is designed to ensure scalability, performance, and responsiveness, even under the highest data loads. By utilizing a polyglot persistence approach , combining DynamoDB for real-time event storage, AuroraDB for relational data management, and Amazon S3 for cold storage and batch analytics , the system is optimized to handle millions of events per minute with minimal latency. This flexible, distributed architecture enables the platform to scale seamlessly, handle traffic spikes without manual intervention, and support real-time decision-making and historical analysis. Through efficient data partitioning, indexing, and integration with Amazon MSK, the system ensures that both high-velocity event data and slow-changing reference data are stored and queried effectively, providing a robust foundation for operational performance and analytics.

## 6. Throughput, Elasticity, & Scalability

Sentry SBMS is engineered to handle massive event volumes and to elastically scale based on workload demands, ensuring seamless service under peak and off-peak conditions. The architecture is designed to achieve super high throughput, accommodate over 10x scaling during high traffic periods, and downscale efficiently during holidays or off-peak times. Below is an in-depth overview of how these principles are implemented, utilizing cutting-edge technologies and scaling strategies.

### 6.1 High Throughput Handling

Given the scale of the system, with a large number of buses, routes, students, and stops, the system generates 1 million+ real-time events every 2 minutes. For example:

* Approximately 10500 buses send periodic status updates and route data, each generating multiple events per minute, such as:
  + Position updates (latitude/longitude every 30 seconds)
  + Route status updates (completed stops, remaining stops, delays)
  + Driver updates (availability, status changes)
* Approximately 153,000 students generate events related to their boarding and disembarking (e.g., every time a commuter gets on or off a bus).
* Approximately 10,000 routes will have frequent updates related to their progress, including service interruptions, delays, and ETA adjustments.

With these data points, and a buffer factor of 2x for unexpected spikes, the system could easily generate up to **1 million+ events** per minute during peak traffic, which can be broken down as:

| **Event Type** | **Calculation** | **Expected Rates/Min**  **(with Buffer) \*** |
| --- | --- | --- |
| Student Boarding Event | 155,000 students \* 1 event/minute | 310,000 |
| Student Disembarking Event | 155,000 students \* 1 event/minute | 310,000 |
| Vehicle Location Snapshot | 11,000 buses \* 2 events/minute | 44000 |
| Route ETA Update | 10,000 routes \* 1 events/minute | 20000 |
| Route Status Update | 10,000 routes \* 1 events/minute | 20000 |
| Driver Status Update | 11,000 buses \* 1 events/minute | 22000 |
| Exception Event (e.g. Delay) | 11,000 buses \* 1 events/minute | 22000 |
| Parent Notification Trigger | 155,000 students \* 1 event/minute \* 2 consumers per student | 620,000 |
| System Audit Event | 10,000 routes \* 1 events/minute | 20,000 |
| Sum of the Above | | 1,388,000 |
|  | | |
| Total Number of Events/Min factoring in Student Boarding and  Disembarking Event would not occur within the same minute = **1,078,000** | | |

\* Buffer factor of 2x to account for unexpected spikes

\* Numbers are rounded up for readability and clarity in the architecture document. Students - 153000 to 155000, Buses - 10500 to 11000

**How the System Handles This Throughput:**

* Amazon MSK (Kafka) ensures high throughput by providing a distributed, partitioned event bus. With Kafka’s ability to horizontally scale, this enables the system to handle millions of events per minute without causing bottlenecks or performance degradation. Kafka is widely used by industry leaders for handling and processing millions of events per second. It is specifically designed for high-throughput, distributed event streaming, making it a popular choice for real-time data processing at massive scales.
* Auto-Scaling EC2 Instances for microservices, coupled with container orchestration through Amazon EKS, guarantees that the service layer would handle a surge in traffic by scaling horizontally. To handle any sudden traffic increases by say 2x during rush hours, additional EC2 instances are provisioned, ensuring that no processing delay occurs.
* The system leverages asynchronous event-driven architecture, where Kafka producers publish messages to Kafka topics, and microservices consume messages in parallel, processing them concurrently. This allows for low-latency and high-throughput processing.

### 6.2. Elasticity & Horizontal Scaling (10x)

During non-peak hours (such as late evenings, nights, or holidays), activity will significantly decrease, particularly for events related to student boarding, disembarking, and parent notifications, which primarily occur during school hours.

The system is designed to elastically scale to accommodate traffic surges (such as during rush hours or special events) and scale down during off-peak times (such as during holidays). This **elasticity** isn't limited to predictable peaks; the architecture is robust enough to handle **unexpected spikes or sudden shifts** in traffic patterns, allowing the system to adapt within minutes with **no manual intervention**. Whether it's a sharp increase in student numbers, a larger bus fleet, or unforeseen events, the system dynamically scales to maintain consistent performance.

Key components ensuring elasticity and horizontal scaling are:

* Auto Scaling Groups (ASG):
  + EC2 instances running microservices (e.g., route processing, commuter management, and bus tracking) are deployed in ASGs, ensuring automatic scaling based on system load.
  + During normal operation, 25 EC2 instances would be sufficient. However, during peak load (such as peak commuting hours), the system would scale out to over 250 instances (10x) automatically, maintaining consistent performance even under massive load.
* Amazon MSK:
  + Kafka’s partitioned architecture supports the dynamic addition of new brokers to accommodate increased message throughput. MSK would horizontally scale by increasing the number of partitions, allowing the system to handle significant event volume increases without overloading any single broker.
  + Kafka consumers would dynamically scale, adding more containers and instances to process events in parallel. When events increase, new consumers are spawned automatically to keep pace with the message rate.
* Redis ElastiCache:
  + Redis ElastiCache is leveraged to enhance performance by providing high-speed, in-memory data storage for frequently accessed data. It automatically scales by adding or removing Redis nodes based on system demands.
  + During peak periods, ElastiCache would scale horizontally by adding more nodes or clusters to distribute the load, ensuring rapid access to cache data such as commuter schedules, bus statuses, and route details. Redis ElastiCache’s ability to elastically scale ensures that the system manages fluctuating traffic volumes and maintains fast data retrieval during both high and low demand periods.
* Elastic Load Balancer (ELB):
  + An Elastic Load Balancer (ELB) is placed in front of microservices, distributing incoming traffic to the EC2 instances or containers within EKS clusters. The ELB automatically adjusts based on scaling events, ensuring optimal resource utilization during peak and off-peak hours.
* DynamoDB & Aurora Auto Scaling:
  + DynamoDB uses auto-scaling to manage throughput for real-time event storage, scaling read and write throughput based on traffic volume. This ensures low-latency access to event data, even under high load.
  + AuroraDB scales automatically by adding read replicas to distribute the load and improving the availability and fault tolerance of the database, while write scaling is managed through partitioning strategies.

### 6.3 Elasticity and High Throughput Log Processing & Monitoring

To ensure high-performance logging, real-time visibility, monitoring and reliable troubleshooting, the logging stack is designed with Promtail, Grafana Loki, and EKS orchestration. This combination allows the system to handle massive throughput, scale elastically under varying loads, and maintain optimal performance even during peak traffic periods.

**Promtail for Efficient Log Collection and High Throughput**

Promtail is deployed alongside EC2-based microservices in the application layer to gather logs in real time. Promtail scales horizontally, adjusting to the number of EC2 instances, ensuring that logs are captured from each instance without delay, even during high event volumes.

* Real-Time Log Collection: Promtail collects log data from all EC2 instances running the application, processing logs efficiently at high throughput, even during peak periods. This ensures that no event data is lost and all application logs are immediately available for analysis.
* Elastic Scaling with EC2: As the number of EC2 instances increases or decreases based on the load (using EC2 auto-scaling policies), Promtail dynamically adjusts to handle the change, collecting logs from new instances without requiring additional configuration.

**Grafana Loki for High-Efficiency Log Aggregation and Real-Time Querying**

Grafana Loki provides the backbone of the log aggregation system, designed to efficiently store and query logs under high load. Loki is purpose-built for scalability and performance, making it ideal for systems that experience high throughput and need to scale elastically.

* Scalable Storage and Indexing: Loki’s ability to efficiently index logs ensures that even with a high volume of logs - potentially reaching millions per minute during peak traffic - the system maintains fast query performance. The indexing mechanism is optimized for high-throughput scenarios, ensuring that logs are stored in a space-efficient way without sacrificing speed.
* Real-Time Log Querying: During peak traffic periods, the system would process massive numbers of logs. Loki would query large volumes of log data across distributed systems without degradation in performance. Engineers would quickly search through logs for troubleshooting, even as the system scales horizontally.
* Optimized for Large Event Streams: With event-based architectures like this one, where event volume would increase drastically, Loki’s architecture is designed to handle massive log streams and provide near-instant search results across distributed microservices.

**EKS Orchestration for Elastic Scalability and Robust Log Handling**

While the application layer uses EC2 auto-scaling to handle variable traffic, the logging infrastructure is managed through Amazon EKS. EKS orchestrates the log collection and processing microservices, enabling the system to scale efficiently under high log volumes.

* Elastic Log Processing: EKS manages the log processing layer, which is decoupled from the application layer. It ensures that log processing services scale automatically based on the incoming log volume. As the number of log events increases during peak periods (e.g., 10x surge), EKS would add more resources (pods) for handling log aggregation and querying, maintaining performance without impacting the application layer.
* Horizontal Scaling for Log Services: In response to high throughput, EKS automatically scales the logging services - such as Promtail and Loki - ensuring that the logging infrastructure would keep up with the increase in logs. This scaling happens independently of the EC2-based application layer, ensuring that the system would handle increased event traffic without delays or data loss.

**Elastic Log Handling During Traffic Surges**

During peak traffic, such as when scaling to over 10x, the system is designed to handle the increased load without degradation. This is possible due to the following features:

* Elastic Log Collection with Promtail: Promtail scales alongside the microservices it collects logs from. As new instances are added or removed, Promtail seamlessly adjusts to ensure no logs are missed.
* High-Performance Aggregation with Grafana Loki: Loki’s scalable indexing and efficient storage mechanisms ensure that even with millions of log entries per minute, log queries remain fast and responsive.
* Auto-Scaling with EKS: EKS would scale microservices over 10x, with automated scaling based on custom metrics. This horizontal scaling ensures that the infrastructure is always able to keep up with the demands of both the event processing and logging components.

### 6.4 Summary

Sentry SBMS is designed for ultra-high throughput and elasticity, capable of processing millions of events per minute. The system dynamically scales, capable of handling over 10x traffic surges during peak periods while efficiently downscaling during off-peak times. Powered by Amazon MSK (Kafka), EC2 Auto Scaling, and Amazon EKS, it ensures seamless performance even under extreme loads.

With over 1 million events generated every 2 minutes from buses, routes, students, and stops, Kafka's distributed architecture ensures high-throughput processing without bottlenecks. EC2 Auto Scaling and containerized microservices automatically adjust to fluctuating demand, maintaining consistent performance. Redis is leveraged for in-memory caching of frequently accessed data, reducing response times and offloading the database, which further enhances the overall system efficiency.

This architecture, trusted by industry leaders for real-time event streaming, delivers unmatched scalability, reliability, and performance.

## 7. Cost Efficiency

While the system is designed for high throughput and elastic scaling during peak periods, it is equally important to ensure cost efficiency during periods of low traffic. The system utilizes several strategies to optimize operational costs without compromising performance, ensuring that resources are allocated efficiently based on actual demand. Below are the key components of the architecture that contribute to cost savings during low traffic periods:

### 7.1 Auto-Scaling Infrastructure

The core design of the system is built on elastic scaling, which allows infrastructure to scale up during high demand and scale down during low traffic periods. This is achieved through several mechanisms:

* **EC2 Auto Scaling**: During periods of low traffic, the system will automatically downscale the number of EC2 instances based on real-time traffic demands. By reducing the number of active instances during low traffic periods (such as holidays, weekends or off-peak hours), the system ensures that you are only paying for the resources that are actually required.  
    
   Example:
  + During peak periods, the system would scale up to 250 EC2 instances.
  + During low traffic periods, the system would scale down to a minimum of 25 EC2 instances (or even fewer if traffic is extremely low).

This dynamic scaling ensures that unnecessary compute resources are not provisioned, thereby reducing costs without affecting system performance or availability.

* **Amazon MSK (Kafka) Auto Scaling:** Amazon MSK supports the automatic scaling of brokers and partitions based on traffic. When event throughput drops during low traffic periods, unnecessary partitions and brokers would be de-provisioned, leading to cost reductions. Similarly, during high traffic, MSK would scale out to accommodate higher event volumes.
* **Amazon EKS Autoscaling:** Elastic Kubernetes Service (EKS) is highly cost-efficient due to its ability to automatically scale the number of pods and nodes in the cluster based on the traffic load. During low traffic, Kubernetes would reduce the number of running pods and nodes in the cluster, thus optimizing infrastructure costs. This ensures that containerized microservices are only running when required and are shut down when not needed.

### 7.2 Serverless Components for Cost Savings

Several components of the system utilize serverless technologies to ensure cost efficiency during low traffic periods:

* **AWS Lambda:** For event processing or lightweight background tasks, AWS Lambda functions are employed to handle specific workloads without the need to keep idle EC2 instances running. Lambda is a pay-per-use service, meaning that you are only charged for the execution time of the functions. During low traffic periods, the number of Lambda executions will decrease, leading to significant cost savings.
  + Example: Tasks such as Data transformation (ETL operations to S3) , Alerts and notifications are handled through Lambda functions, which ensures that the costs scale directly with actual event volume.
* **Amazon S3 for Data Storage:** Data that is stored in Amazon S3 is billed based on usage (storage volume and retrieval operations). During low traffic periods, ETL operations would be scheduled to run less frequently or more efficiently (e.g., batch processing), reducing the frequency of data transfers and API calls to S3, which in turn reduces costs.

### 7.3 Database Cost Optimization

Both the DynamoDB and AuroraDB databases used in the system offer auto-scaling and provisioned capacity models, allowing the system to scale cost-effectively during low traffic periods:

* **DynamoDB:** DynamoDB supports on-demand scaling for read and write throughput. During low traffic, the system automatically scales down read/write throughput to match the lower event volume, avoiding over-provisioning and keeping costs down. Additionally, the system will leverage DynamoDB's on-demand mode where charges are only incurred based on the actual traffic, allowing for automatic adjustments with minimal manual intervention.
* **AuroraDB (PostgreSQL):** Aurora is designed to scale horizontally, and its Auto-Scaling Read Replicas adjust based on demand. During low traffic periods, fewer read replicas are needed, allowing Aurora to downscale accordingly and reduce costs. Additionally, the serverless Aurora feature allows the database to scale compute resources in real-time, adding or removing resources as required. This enables cost optimization by only provisioning the resources required for the given load.
  + In this architecture, write operations are managed by the primary instance, while read-heavy queries during off-peak times are served by a smaller number of read replicas, ensuring cost savings during idle periods.

### 7.4 Redis Cache Auto-Scaling

Redis is used as a cache layer to accelerate data access for both DynamoDB and AuroraDB. Redis is highly cost-effective when leveraged for frequent access to data that doesn't change often, such as Bus routes, Student schedules, and Stops metadata.

* **Elastic Cache:** During low traffic periods, the Redis cache cluster will automatically scale down by reducing the number of nodes in the cluster. This ensures that resources are not wasted on underutilized caches, leading to reduced operational costs.

### 7.5 Efficient Logging

The logging architecture is designed not only for scalability but also for cost optimization, using a lightweight, pull-based logging stack and efficient orchestration strategies.

* **Promtail + Loki = Lightweight, Index-Free Logging:** Unlike traditional log aggregation systems (e.g., ELK stack) that create heavy full-text indexes, Grafana Loki uses a label-based indexing model, which drastically reduces storage and compute overhead. Log generation naturally aligns with system traffic patterns. While this behavior applies across most logging systems, Loki stack’s minimal indexing and efficient storage model ensure that costs scale linearly, and more predictably , with actual log volume
* **EKS-Based Isolation:** Running the logging stack within its own EKS cluster allows for independent scaling. This prevents over-provisioning and allows aggressive right-sizing based on actual log ingestion needs,keeping infra costs lean.

### 7.6 Cost Monitoring & Management

* AWS Cost Explorer and AWS CloudWatch are used to monitor and optimize costs based on actual usage. These tools allow the system administrators to review resource utilization patterns and identify cost-saving opportunities during low traffic periods. For example:
  + Unused EC2 instances would be identified and terminated.
  + DynamoDB’s read and write capacity would be adjusted to match actual usage.
  + Aurora’s instance sizes would be optimized for lower traffic.

With these tools in place, the system would automatically or manually adjust resource utilization to ensure maximum cost efficiency during periods of low traffic while maintaining performance during peak times.

### 7.7 Summary

The system is designed with cost efficiency as a priority during low traffic periods, leveraging auto-scaling, serverless architectures, and resource optimization strategies to minimize unnecessary spending. By dynamically scaling infrastructure, optimizing database and cache utilization, and reducing the frequency of logging and ETL operations, the system ensures that costs are closely aligned with actual demand. This results in significant cost savings during off-peak periods without compromising performance or availability during peak times.

## 8. Networking & Security (AWS GovCloud)

### 8.1 VPC Structure

Multiple VPCs (Virtual Private Clouds) would be used (e.g., Prod, Dev, Test). Each VPC spans multiple AZs. Use of public subnets restricted to necessary components (e.g., Load Balancers, NAT Gateways). Private subnets host core application/database resources. VPC Endpoints (Gateway and Interface) used for private access to AWS services (S3, DynamoDB, Kinesis, etc.).

### 8.2 Segmentation

Security Groups (stateful) and Network ACLs (stateless) enforce strict ingress/egress rules to limit exposure to sensitive data and systems, prevent adversary lateral movement and aid with containment of breaches during security incidents.

### 8.3 Connectivity

AWS Direct Connect or Site-to-Site VPN established between AWS GovCloud VPC and NYCPS data centers for secure, private access to internal resources/databases.

### 8.4 Edge Security

CloudFront for caching and DDoS protection for web applications using AWS Shield. AWS WAF deployed with API Gateway and CloudFront to filter malicious requests (SQLi, XSS). AWS Shield Advanced for enhanced DDoS protection.

### 8.4 Encryption

* In Transit: TLS 1.2+ enforced for all external and internal API Gateway endpoints, CloudFront distributions, Load Balancers, and direct service communications.
* At Rest: Server-side encryption enabled for S3 (SSE-S3 or SSE-KMS), EBS volumes, RDS instances/snapshots, DynamoDB tables, SQS queues, SNS topics using AWS KMS with customer-managed keys (CMKs) where appropriate for enhanced control and auditability.

### 8.5 Identity & Access Management (IAM)

* Strict use of IAM Roles for EC2 instances, Lambda functions, ECS/EKS tasks (via IAM Roles for Service Accounts - IRSA - in EKS). Avoid long-lived access keys.
* Fine-grained IAM Policies adhering to principles of least privilege.
* Multi-Factor Authentication (MFA) is enforced for all human access to the AWS Management Console and API, especially for privileged accounts.
* Federated identity management via SAML 2.0 integration with NYCPS's Identity Provider (e.g., ADFS, Azure AD) for console/API access if feasible, otherwise stringent controls on IAM users.

### 8.6 Secrets Management

AWS Secrets Manager used to store and rotate database credentials, API keys, and other secrets securely. Applications retrieve secrets at runtime via IAM roles.

### 8.7 Monitoring & Logging

* CloudTrail enabled in all regions, logging to a central, secured S3 bucket with log file validation enabled.
* CloudWatch Logs collected from all services (Lambda, EC2, RDS, etc.). CloudWatch Alarms configured for critical metrics (CPU, memory, latency, error rates, queue depths).
* AWS Config used to track resource configurations and compliance.
* Security Hub aggregated findings from GuardDuty, Inspector, Macie (if used), Config, and partner integrations.
* GuardDuty enabled for intelligent threat detection.
* Inspector used for vulnerability assessments on EC2 instances (if used).
* AWS Detective used to investigate security findings and alerts, along with related context, and historical behaviour.

## 9. Appendix

### 9.1 Abbreviations, Acronyms and Definitions

| Abbreviation | Description |
| --- | --- |
| SMBS | School Bus Management System - Sentry’s proposed platform for NYCPS’s next-generation Transportation Management System |
| AWS | Amazon Web Services - A comprehensive cloud computing platform offering a wide range of services like computing, storage, and networking |
| EKS | Elastic Kubernetes Service - A managed Kubernetes service on AWS that simplifies the deployment and management of containerized applications |
| EC2 | Elastic Compute Cloud - A web-based service from AWS providing scalable virtual servers for on-demand computing capacity |
| RDS | Relational Database Service - A managed database service by AWS for setting up, operating, and scaling relational databases in the cloud |
| MSK | Managed Streaming for Apache Kafka - An AWS service that makes it easy to build and run applications using Apache Kafka for streaming data |
| ETL | Extract, Transform, Load - A process used in data integration to extract data from sources, transform it into a usable format, and load it into a target system or database |
| AZ | Availability Zones - Isolated data centers within specific regions used by cloud service providers like AWS and Microsoft Azure to ensure stable connections, redundancy, and failover support for cloud services |
| ECS | Elastic Container Service - A fully managed AWS container orchestration service that enables developers to deploy, manage, and scale containerized applications efficiently in the cloud or on-premises environments |
| SNS | Simple Notification Service - A fully managed AWS service for message delivery from publishers to subscribers using topics as communication channels |
| VPC | Virtual Private Cloud - An isolated pool of resources within a public cloud environment, providing enhanced security and privacy through private IP subnets and virtual communication constructs |
| DNS | Domain Name System - A hierarchical naming system that translates domain names into IP addresses, enabling devices to locate and communicate over the internet |
| TTL | Time To Live - A concept used to define the lifespan or validity period of data. |

### 9.2 Chosen Technologies/Services

| Technologies/Services | Purpose |
| --- | --- |
| AWS Gov Cloud | **AWS GovCloud (US)** is designed for US government agencies and regulated industries with specific features:  • Compliance:  ○ ITAR compliance \* FedRAMP certified  ○ DoD SRG Impact Levels 2, 4, and 5  ○ HIPAA, PCI-DSS compliant  • Key Characteristics:  ○ Physically located in US  ○ Operated by US citizens  ○ Accessible to US entities only  ○ Separate and isolated from standard AWS regions  ○ Supports most AWS services  • Use Cases:  ○ Government workloads  ○ Healthcare data  ○ Financial services  ○ Defense applications  ○ Sensitive data processing |
| AWS ECS Cluster Container | **An Amazon ECS cluster** is a logical grouping of tasks or services. Key components include:  • Infrastructure capacity combining:  ○ Amazon EC2 instances  ○ Serverless (AWS Fargate)  ○ On-premises VMs/servers  • Core elements:  ○ Tasks - Groups of containers that run together  ○ Services - Applications maintaining specified number of tasks  ○ Network (VPC and subnet)  ○ Optional namespace for service communication  ○ CloudWatch Container Insights for monitoring  ECS clusters are region-specific and help separate resources. They enable efficient container management, scheduling, and deployment of microservices architectures while providing seamless integration with other AWS services like ELB and Auto Scaling. |
| AWS ECS EC2 Container | **With ECS on EC2 launch** type, you:  • Have complete control over EC2 instances in your cluster  • Run containers on EC2 instances you manage  • Are responsible for provisioning, patching, and scaling servers  • Can choose specific EC2 instance types and customize the OS  • Use Auto Scaling Groups to manage EC2 capacity  • Need to install ECS container agent (included in ECS-optimized AMI)  Your containers are defined in task definitions that specify parameters like CPU, memory, ports, and IAM roles. ECS handles container scheduling and placement across your EC2 instances based on your resource requirements. |
| AWS Router 53 | **Amazon Route 53** is a highly available and scalable cloud DNS web service. Key features include:  • Domain registration and management  • DNS routing with multiple options:  ○ Latency-based routing  ○ Geolocation routing  ○ IP-based routing  ○ Weighted routing  • Health checking and DNS failover  • Route 53 Resolver for VPC DNS resolution  • Integration with other AWS services\* DNS Firewall for filtering outbound DNS traffic |
| Amazon API Gateway | Amazon API Gateway is a fully managed service for creating, publishing, and managing APIs. Key features:  • API Types:  ○ REST APIs  ○ HTTP APIs (faster and lower cost)  ○ WebSocket APIs  • Capabilities:  ○ Request/response transformation  ○ API versioning  ○ Security (IAM, Lambda authorizers)  ○ Throttling and caching  ○ SDK generation  ○ API documentation  • Benefits:  ○ Serverless architecture  ○ Pay-per-use pricing  ○ Automatic scaling  ○ Edge caching  ○ Monitoring and logging  ○ Global deployment options |
| Amazon Managed Streaming Kafka | Amazon Managed Streaming for Apache Kafka (Amazon MSK) is a fully managed service that helps you build and run applications using Apache Kafka for streaming data processing. Key benefits include:  • No server management required - AWS handles provisioning, configuration, and maintenance  • High availability with multi-AZ deployment  • Built-in security with IAM integration  • Easy scalability to support load changes  • Native integration with AWS services like Lambda, Glue Schema Registry, and Amazon Managed Service for Apache Flink\* Pay-for-what-you-use pricing model  Amazon MSK eliminates the operational overhead of managing Kafka infrastructure, letting you focus on building streaming applications. |
| EKS | **Amazon Elastic Kubernetes Service** (EKS) is a fully managed Kubernetes service provided by AWS. It simplifies the deployment, scaling, and management of containerized applications by automating the setup and operation of Kubernetes clusters. Key features include:  • Integration with AWS Services: EKS seamlessly integrates with services like EC2, S3, IAM, and VPC for networking, storage, and security.  • Flexible Deployment: Applications can run on AWS infrastructure (EC2 or Fargate) or on-premises using EKS Anywhere.  • Ease of Use: Reduces operational overhead by automating updates, scaling resources, and patching nodes |
| Promtail | **Promtail** is a lightweight log collection agent that:  • Reads logs from local files, systemd journals, or application containers.  • Enriches logs by attaching metadata (e.g., pod name, namespace, labels).  • Sends logs to a Grafana Loki instance for storage and analysis. |
| Grafana Loki | **Grafana Loki** is an open-source log aggregation system. Key features include:  • Log storage: Logs are compressed and stored in chunks, with only metadata (labels) indexed.  • Scalability: Loki scales horizontally by distributing logs across multiple ingesters, ensuring high availability and efficient handling of heavy log traffic. It prevents data loss during node failures and redirects logs to healthy nodes.  • Log Compression: Logs are compressed into chunks, reducing storage space requirements and improving retrieval times. Indexed logs allow for quick querying based on timestamps and labels  • Integration with Grafana: Loki seamlessly integrates with Grafana, allowing visualization of logs alongside metrics and traces in a unified dashboard |
| Prometheus | Prometheus is an open-source systems monitoring and alerting toolkit that collects and stores time-series data, offering a powerful and flexible solution for monitoring cloud-native applications and infrastructure. |
| Grafana | Grafana is an open-source, interactive web application used for monitoring and visualizing data, allowing users to create customizable dashboards, charts, and graphs from various data sources |
| Aurora | **Amazon Aurora** is a cloud-native relational database offering key features:  • Performance:  ○ 5x throughput of MySQL, 3x of PostgreSQL  ○ Compatible with MySQL and PostgreSQL  ○ Auto-scaling storage up to 128 TiB  • High Availability:  ○ Data replicated across 3 Availability Zones  ○ 6 copies of data  ○ Automated failover  ○ Up to 15 read replicas  • Cost-effective:  ○ 1/10th cost of commercial databases  ○ Pay for what you use  ○ Serverless option available  • Management:  ○ Fully managed by AWS  ○ Automated patching, backup, and recovery  ○ Built-in monitoring and security |
| NoSQL | **Amazon DynamoDB:**  ○ Serverless key-value and document database  ○ Single-digit millisecond latency  ○ Automatic scaling  ○ Global tables capability |
| Redis ElastiCache | **Amazon ElastiCache for Redis** is a fully managed, in-memory data store and cache service provided by AWS  • Provides microsecond latency for real-time applications  • Provides up to 80x faster read performance compared to disk-based databases  • Offers auto-scaling capabilities  • Reduce database query costs by >50% |
| S3 | **Amazon Simple Storage Service** (Amazon S3) is an object storage service that offers industry-leading scalability, data availability, security, and performance  • Low cost Data storage and backup  • High durability, availability and Security |
| AWS CloudWatch | **Amazon CloudWatch** is a monitoring and management service that provides:  • Real-time monitoring of AWS resources and applications  • Collection and tracking of:  ○ Metrics  ○ Logs  ○ Events  ○ Application performance data  • Key capabilities:  ○ Create custom dashboards  ○ Set automated alarms  ○ Collect data from on-premises, hybrid, and cloud infrastructure  ○ Take automated actions based on predefined thresholds  ○ Monitor complete stack (applications, infrastructure, network)  ○ Access via API, CLI, SDKs, or AWS Console  CloudWatch helps you gain system-wide visibility, optimize resource utilization, and reduce mean time to resolution (MTTR) for operational issues. |
| DNS Logs | **Route 53 DNS** query logging provides:  • Information logged:  ○ Domain/subdomain requested  ○ Date and time  ○ DNS record type (A, AAAA)  ○ Route 53 edge location  ○ DNS response codes  • Logs can be sent to:  ○ CloudWatch Logs (default)  ○ Amazon S3  ○ Kinesis Data Firehose  • Features:  ○ Logs available within minutes of queries  ○ Log group must be in US East (N. Virginia)  ○ Only logs queries forwarded to Route 53  Helps monitor DNS traffic and troubleshoot issues |
| AWS Cloudtrail Logs | **AWS CloudTrail provides:**  • Event logging of AWS account activity:  ○ Management events (API calls, console actions)  ○ Data events (S3 object-level activities)  ○ Network activity events  ○ Insights events for unusual API activities  • Key features:  ○ 90-day event history by default  ○ Logs delivered to S3 bucket  ○ Optional CloudWatch Logs integration  ○ Multi-region and multi-account logging  ○ Immutable audit trail  ○ Security analysis and compliance auditing  ○ Resource change tracking  CloudTrail helps answer "who did what, where, and when?" by recording user activity and API calls across AWS services. |
| AWS VPC Flow Logs | **Amazon Virtual Private Cloud (VPC)**.  1. Captured Information:  ○ Source and destination IP addresses  ○ Source and destination ports  ○ Protocol  ○ Number of packets and bytes transferred  ○ Time interval during which the flow was observed  ○ Action (accepted or rejected traffic)  2. Logging Options: VPC Flow Logs can be published to: [2]  **○ Amazon CloudWatch Logs**  ○ Amazon S3  ○ Amazon Kinesis Data Firehose  3. Scope: Flow logs can be created at different levels:  ○ VPC level  ○ Subnet level  ○ Network interface level  4. Use Cases:  ○ Diagnosing overly restrictive security group rules  ○ Monitoring traffic reaching your instances  ○ Determining traffic direction (inbound/outbound)  ○ Enhancing network security analysis  5. Performance Impact: VPC Flow Logs do not affect network performance or latency as they are collected outside the network traffic path. [3]  6. Limitations: Some types of traffic are not logged, such as:  ○ Traffic to Amazon DNS servers  ○ Windows instance license activation  ○ Instance metadata  ○ DHCP traffic  ○ Traffic to reserved IP addresses for the VPC router  7. Security and Compliance:  ○ Helps in monitoring and auditing network traffic for compliance requirements  ○ Assists in identifying potential security issues or unauthorized access attempts  8. Cost Considerations: There are charges associated with using VPC Flow Logs. For current pricing details, please refer to the AWS documentation.  9. Integration: VPC Flow Logs can be integrated with other AWS services for further analysis and visualization of network traffic patterns.  When implementing VPC Flow Logs, it's important to consider your specific security and monitoring needs, and to configure the logs according to the principle of least privilege. Always test your configurations in a non-production environment before deploying to production systems. |
| AWS VPC Flow Logs | **AWS Shield** is a managed DDoS protection service that safeguards applications running on AWS. It comes in two tiers:  • AWS Shield Standard:  ○ Free, automatic protection against common Layer 3 and 4 DDoS attacks  ○ Always-on detection and automatic inline mitigations  ○ Protects AWS services like CloudFront and Route 53  • AWS Shield Advanced:  ○ Enhanced protection against sophisticated Layer 3-7 attacks  ○ Protects EC2, ELB, CloudFront, Global Accelerator, and Route 53  ○ Includes 24/7 access to AWS Shield Response Team  ○ Provides cost protection against DDoS-related spikes  Offers real-time attack visibility through **CloudWatch** |
| AWS WAF | **AWS WAF (Web Application Firewall)** helps protect web applications from common exploits. Key features include:  • Web traffic filtering based on:  ○ IP addresses  ○ HTTP headers and body  ○ Custom URIs  ○ SQL injection patterns  ○ Cross-site scripting  • Can be deployed on:  ○ Application Load Balancer  ○ Amazon API Gateway  ○ CloudFront  ○ AppSync for GraphQL APIs  • Includes Bot Control to manage bot traffic  • Offers managed rules that are automatically updated  • Allows creation of custom security rules  • Enables centralized rule management across multiple websites |
| AWS GuardDuty | **Amazon GuardDuty** is a threat detection service that continuously monitors your AWS environment for:  • Malicious activity and unauthorized behavior  • Account compromise and credential theft attempts  • Data exfiltration and potential ransomware events  • Unauthorized cryptomining\* Malware in EC2 instances and container workloads  • Suspicious database login patterns  • Malicious files in S3 buckets  Key features:  • Uses AI/ML and threat intelligence  • Analyzes AWS CloudTrail, VPC Flow Logs, and DNS logs  • Integrates with AWS Organizations for multi-account monitoring  • No additional security software needed  • Continuous monitoring without performance impact  Accurate account-level threat detection |
| AWS Detective | **Amazon Detective** automatically collects log data from your AWS resources and uses machine learning (ML), statistical analysis, and graph theory to build a dataset that you can use to conduct more efficient security investigations.  Key features:  • Determine potential security issues through a unified view of user and resource interactions.  • Save time and effort with graph models that automatically summarize security-related relationships running on AWS.  • Investigate and respond to security findings with streamlined visualizations.  • Accelerate security investigations with generative AI insights to more quickly comprehend threats |
| AWS Event Bridge | **Amazon EventBridge** is a serverless event bus service that helps you build event-driven applications. Key features include:  • Integration with 200+ AWS services and SaaS providers  • EventBridge Pipes for point-to-point integrations  • EventBridge Scheduler for task automation  • Built-in filtering and transformation capabilities  • JSON-based event structure  • Multiple targets including Lambda, SQS, SNS, Kinesis  • Support for event-driven architecture  • SaaS provider integration over private AWS network\* Pay-per-use pricing model  • Real-time event routing and delivery  EventBridge extends beyond AWS services to bring external SaaS data into your AWS environment, eliminating the need for polling or custom webhooks. |
| Amazon Simple Notification Service | **Amazon SNS** is a fully managed pub/sub messaging service that enables:  • Application-to-application messaging  • Application-to-person notifications via:  ○ Mobile push notifications  ○ SMS  ○ Email  ○ HTTP/HTTPS endpoints  • Integration with AWS services like SQS, Lambda, and CloudWatch  • Standard and FIFO topic types  • Message filtering and data protection\* Pay-as-you-go pricing with no upfront costs  • Secure message delivery using access control and encryption  • High availability and durability across multiple AZs  SNS helps build distributed applications and microservices architectures while e |
| AWS Secret Manager Data Encryption | **AWS Secret Manager** encrypts data using:  • Encryption in transit:  ○ TLS-protected channels between AWS internal systems  ○ Secure data transfer from EC2 instances to AWS  ○ Protected telemetry data collection  • Encryption at rest options:  ○ AWS owned CMK (default)  ○ Customer managed CMK  ○ AWS KMS keys for customized encryption  Systems Manager integrates with AWS KMS for key management and uses envelope encryption to protect sensitive data. All communication between components is encrypted using TLS to ensure data security during transmission. |
| React utilizing JS | **React** is a JavaScript library for building user interfaces, particularly for web applications, that uses a component-based architecture to create interactive and dynamic UIs efficiently |
| Next.js | **Next.js** is a Web Framework built on React. It allows for seamless integration between the API End and the Frontend Webpage. NextJS is one of the most used React Web Frames, and provide optimizations and many useful Utilities for building Web Pages of all sizes and complexities  **Server Side Render** and **HTML streaming** provides substantial performance gains  **Static Site Generation** allows for Prebuilding of pages to make simple pages load near instantly |
| SpringBoot / Java | Java Backend Framework known for its ability to streamline the development of robust applications through simplified Configuration, Increased productivity, microservice support, large ecosystem and community |
| Apache HTTP Server | **Apache HTTP Server** is a Free open source Cross-platform web server  Can support Various Languages  Most Popular Web Servers in the world  Known for its Security Features and large community that work to improve security |
| React-Native / JS | **React-Native** isa JavaScript framework developed by Meta Platforms (formerly Facebook Inc.) that enables developers to build native mobile apps for Android and iOS using a single codebase, leveraging the React framework and native platform capabilities |
| Amplify / Javascript | **AWS Amplify** is a development framework and hosting service that offers:  • Development Features:  ○ Libraries and UI components  ○ CLI for backend setup  ○ Authentication via Cognito  ○ Storage with S3  ○ APIs (GraphQL/REST)  ○ Analytics and push notifications  • Hosting Capabilities:  ○ Full-stack web application hosting  ○ Built-in CI/CD workflows  ○ Support for React, Angular, Vue, Gatsby  ○ Serverless architecture  ○ Automatic HTTPS  • Key Benefits:  ○ Quick setup and deployment  ○ Scalable infrastructure  ○ Pay for what you use  ○ Multiple storage bucket support  Built-in security features |

# Archived Section

(Below sections were removed from the original Doc that was generated from the html file)

2.1 Design Principles

* GovCloud Native: Utilize AWS services available within the GovCloud (US) regions.
* Security by Design: Implement security controls at every layer (network, compute, data, application). Assume a Zero Trust posture.
* High Availability & Resilience: Design for fault tolerance using multi-AZ deployments and managed services with built-in HA. Implement robust Disaster Recovery (DR).
* Scalability & Elasticity: Employ auto-scaling, serverless components, and elastic managed services to handle variable loads (daily peaks, seasonal changes) efficiently.
* Modularity (Microservices): Utilize a microservices architecture to promote independent development, deployment, scaling, and fault isolation of components.
* Event-Driven: Leverage asynchronous messaging and event streams for decoupling components and enabling real-time processing.
* Infrastructure as Code (IaC): Manage infrastructure provisioning and configuration through code for consistency, repeatability, and auditability.
* Managed Services Preference: Utilize AWS managed services (AuroraDB, DynamoDB, MSK, SNS, Lambda, EKS etc.) to reduce operational overhead and leverage AWS expertise.

## 3. Overall Architecture Pattern

The proposed architecture is a cloud-native, event-driven microservices architecture deployed across multiple Availability Zones (AZs) within an AWS GovCloud (US) region.

* Data Ingestion: GPS devices communicate via MQTT/HTTPS to AWS IoT Core or directly to API Gateway endpoints. Data flows through streaming services (MSK).
* Processing: Lambda functions and containerized services (on ECS) process events and data streams. The Routing Engine may use dedicated EC2 instances for intensive computation.
* Data Storage: A mix of databases (AuroraDB, DynamoDB, ElastiCache) and object storage (S3) is used based on data type and access patterns.
* APIs & Communication: RESTful APIs exposed via API Gateway serve front-end applications and internal service communication. SNS/SQS/SES handle asynchronous messaging and notifications.
* Frontend: Mobile applications (iOS/Android) and Web applications (React/Angular/Vue hosted on S3/CloudFront) interact with backend APIs.

## 4. Component-to-AWS Service Mapping

| Functional Component | Primary AWS GovCloud Service(s) | Rationale & Notes |
| --- | --- | --- |
| GPS Data Ingestion & Processing | IoT Core (optional, for MQTT), API Gateway + Lambda/Fargate, Kinesis Data Streams or MSK, S3 (raw data), DynamoDB (real-time location), RDS PostgreSQL w/ PostGIS (processed tracks, geofences), Lambda/Fargate (ETA calc, geofence logic), CloudWatch (device status) | IoT Core provides scalable MQTT ingestion if devices support it. Kinesis/MSK handles high-throughput streaming data. Lambda/Fargate enables scalable, event-driven processing. DynamoDB offers low-latency reads/writes for real-time location. RDS/PostGIS handles complex spatial queries & history. |
| Ridership Tracking Module | API Gateway + Lambda/Fargate, DynamoDB or RDS PostgreSQL, S3 (for ID scan images if needed) | API endpoints handle scan/manual entry events. DynamoDB or RDS store ridership records, depending on query patterns. Lambda/Fargate process events, validate against rosters (from Student Mgmt). |
| Dynamic Routing Engine | EC2/ECS/EKS (potentially w/ GPU instances), RDS PostgreSQL w/ PostGIS, ElastiCache (Redis/Memcached), AWS Location Service (if available/suitable) or self-managed ESRI ArcGIS on EC2/RDS, SQS/Step Functions (async tasks), Lambda (rule triggers) | EC2/Containers needed for complex, potentially long-running routing algorithms. RDS/PostGIS stores road network, stops, routes, constraints. ElastiCache speeds up access to frequently used data (e.g., traffic, routes). AWS Location or ESRI for mapping/geocoding/traffic. SQS/Step Functions for orchestration. |
| Notification & Communication | SNS (Push, SMS), SES (Email), Amazon Connect (optional, for Robocalls/IVR), Lambda/Fargate (notification logic), API Gateway (for internal communication APIs) | SNS/SES provide scalable, managed notification delivery. Connect offers programmable voice capabilities. Lambda/Fargate implement business logic for triggering and formatting notifications based on events from other systems. |
| User Modules (Frontend) | S3 + CloudFront (Web Apps), Native iOS/Android SDKs, Amplify (optional, for mobile/web dev acceleration) | S3/CloudFront provide scalable, secure, low-latency hosting for web frontends. Native SDKs for mobile apps. Amplify can streamline frontend development and backend integration. |
| User Modules (Backend APIs) | API Gateway, Lambda / Fargate (ECS/EKS), Cognito or SAML integration via backend, IAM | API Gateway provides secure, scalable API endpoints. Lambda/Fargate host backend microservices logic. Cognito handles user authentication/pools if direct DOE integration isn't used. IAM secures service-to-service communication. |
| Student Management Backend | API Gateway + Lambda/Fargate, RDS PostgreSQL or DynamoDB, Glue (for ETL from NYCPS systems if needed), SFTP/DataSync (for file-based integration if needed) | Standard API-driven microservice pattern. Choice of DB depends on data model complexity and query patterns. Glue/SFTP/DataSync facilitate integration with potentially legacy NYCPS systems. |
| Reporting & Analytics | S3 (Data Lake), Glue (ETL/Catalog), Athena, Redshift, QuickSight, Kinesis Data Analytics (optional, real-time KPIs), Lambda (scheduled report generation) | S3 provides a scalable data lake. Glue handles ETL and cataloging. Athena for ad-hoc queries on S3. Redshift for performant data warehousing. QuickSight for dashboards/visualizations. Kinesis Data Analytics for streaming analytics. Lambda for automated report generation. |
| Device Management & Inventory | DynamoDB or RDS PostgreSQL, IoT Core (Device Shadow/Registry), Systems Manager (potentially for config), API integration with MDM & Ticketing | DynamoDB/RDS stores inventory data. IoT Core can track device state/metadata if devices connect via MQTT. Systems Manager might assist with certain configurations. Requires integration points. |
| Ticketing System Integration | API Gateway + Lambda/Fargate | Acts as a secure facade/adapter layer to mediate API calls between AWS environment and the NYCPS/Vendor ticketing systems. |

## 4. Data Model

* Databases: Use appropriate databases per component needs (RDS for relational/spatial, DynamoDB for key-value/high-throughput, ElastiCache for caching). Regular backups and point-in-time recovery configured.
* Data Privacy: Implement controls aligned with Solution Design, including encryption, RBAC, audit logging, and data minimization principles.
* Data Ownership & Export: Mechanisms (e.g., APIs, data exports to S3) provided for NYCPS to access and extract their data on demand, reaffirming DOE data ownership.

## 7. Deployment & Operations (DevSecOps)

* Infrastructure as Code (IaC): AWS CloudFormation, AWS CDK, or Terraform used to define and provision all infrastructure resources. Templates stored in version control (AWS CodeCommit or GitHub/GitLab).
* CI/CD Pipeline: AWS CodePipeline orchestrating CodeCommit (source), CodeBuild (build/test, including SAST/DAST scans), and CodeDeploy or ECS/EKS deployment strategies (blue/green, canary) for automated, secure deployments across environments (Dev, Test, Staging, Prod).
* Monitoring: Leverage CloudWatch dashboards, alarms, and logs. Integrate with centralized logging/monitoring solutions if used by NYCPS. Potentially use APM tools (e.g., Datadog, Dynatrace, configured appropriately for GovCloud) for deeper application insights.

## 8. High Availability & Disaster Recovery (DR)

* High Availability (HA):
  + Deploy critical components (API Gateway, Lambda, Fargate/ECS/EKS clusters, RDS, ElastiCache, DynamoDB) across multiple AZs (typically 3) within the primary GovCloud region.
  + Use Elastic Load Balancing (ALB/NLB) to distribute traffic across AZs.
  + Configure RDS Multi-AZ deployments for automatic failover.
  + Leverage DynamoDB global tables (if cross-region active-active is needed and available) or rely on inherent multi-AZ replication.
* Disaster Recovery (DR):
  + Establish a DR strategy (e.g., Pilot Light, Warm Standby) in a second AWS GovCloud (US) region.
  + Regularly back up data (using AWS Backup, RDS snapshots, DynamoDB backups) and replicate backups/snapshots to the DR region.
  + Use IaC to provision infrastructure in the DR region quickly.
  + Utilize Route 53 health checks and DNS failover mechanisms.
  + Regularly test the DR plan (at least annually).
  + Design to meet RFP RPO/RTO targets (RPO=0 for GPS, RPO<=1hr for Routing/Notifications; RTO=0 for GPS, RTO<=15min for Routing/Notifications). Achieving RTO=0/RPO=0 for GPS likely requires an active-active or hot standby approach across AZs/regions for critical ingestion/processing components.

## 9. Compliance

* The architecture leverages services within AWS GovCloud (US), designed to host sensitive data and regulated workloads, meeting standards like FedRAMP High, ITAR, CJIS, DoD SRG IL4/5.
* Specific configurations (encryption, logging, IAM, network controls) align with security requirements from NYCPS, NYC3, OTI, DIIT, FERPA, CIPA, HIPAA (as applicable within GovCloud).
* Regular audits and security testing (as required by RFP) will validate ongoing compliance.