Oulunliikenne.fi

SERVICE DOCUMENTATION

Table of Contents

Introduction

The oulunliikenne.fi service is built on top of the Digitransit journey planner platform, provided by HSL, the Finnish Transport Agency and TVV LMJ Oy. The Digitransit platform provides the runtime environment that serves as the base for the oulunliikenne.fi service. It is hosted and maintained by Digitransit on Microsoft Azure. The oulunliikenne.fi service extends the base functionality of this environment by providing additional information and services for travellers in the Oulu region.

The oulunliikenne.fi service uses its own version of the open-source Digitransit UI, which is a React-based user interface for communicating with a collection of APIs that provide the journey planner functionality. These APIs constitutes the interface towards the Digitransit runtime environment that the oulunliikenne.fi service extends and adds functionality to. All oulunliikenne.fi services are hosted on Amazon Web Services.

Architecture

The high-level architecture illustrated in Figure 1 shows the difference between the base Digitransit environment on left and the oulunliikenne.fi service on the right. In Digitransit, the user interface component communicates directly with the API layers of the backed services, while in oulunliikenne.fi there is a gateway component between the user interface and the API layers that handles routing the API calls to the correct services.

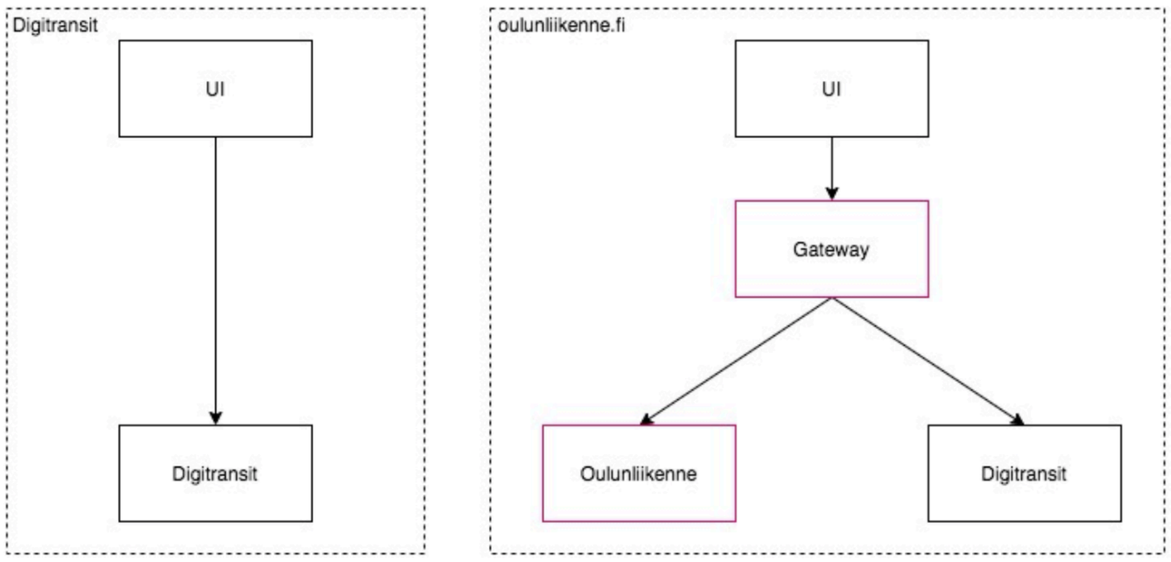


Figure 1: High-level architecture diagram

While this oversimplification of the architecture does not provide any implementation details, it shows one of the key traits between the environments that allows the oulunliikenne.fi service to extend the base functionality of the Digitransit platform.

The oulunliikenne.fi service can thus be divided into 4 main component groups:

1. User interface
2. API Gateway
3. Digitransit APIs
4. Oulunliikenne APIs

User interface

The user interface is based on the open-source Digitransit UI component, modified for the needs of oulunliikenne.fi. It is a React JS user interface that is configurable to some extent for different cities using configuration files. However, the modifications and additions to the user interface for the oulunliikenne.fi service are so broad and, in some cases, complicated, that the simple extendibility built into the software was not enough. As a result, the Digitransit UI project on GitHub has been forked into Oulunliikenne Digitransit UI and is being developed independently.

The initial software architecture largely dictates how new features are added and existing ones modified. All changes are created as separate React-components, where possible, to keep the updateability with the parent repository, in order to bring in new features developed in the Digitransit UI.

API Gateway

The API gateway acts as a proxy between the UI, Digitransit Routing API and the oulunliikenne.fi microservices. The gateway itself is a GraphQL API that combines the other APIs into a single interface for the UI and any 3rd. party that wishes to use the data provided through the API. That is to say, the API is also an open data provider for data distributed by Oulunliikenne.

The illustration in Figure 1 showed that all the communication between the UI and the APIs goes through the gateway, but this is not actually the case. There are APIs that the UI uses directly, namely:

- Geocoding API - Provides a way to perform address searches and address lookups (also known as geocoding and reverse geocoding)
- Map API - Provides raster map images (background map tiles) as well as vector map tiles for stops and other points of interests like ticket sales positions, city bike stations and park and ride areas
- Service Alerts API - Provides HSL's disruption information in the GTFS-RT Service Alerts format and cancelled trips as Trip Updates
- Trip Updates API - Provides real-time trip progress and schedule deviations (predictions) in the GTFS-RT Trip Updates format
- High Frequency Positioning API - Provides real-time vehicle locations in a JSON format over MQTT (not in use by oulunliikenne.fi as it has an alternative implementation)

These APIs are single purpose APIs and use different technologies for communication and data transfer than the Routing API, which is a GraphQL API that provides a way to plan itineraries and query public transport related information about routes, stops and timetables.

The UI architecture is designed around the use of the Routing API, from where it fetches most of the data, while the other APIs are more complementary data sources. This design choice led to the implementation where the API gateway serves only the Routing API data from Digitransit and complements it with data from other sources provided by Oulunliikenne, as it was the best design to implement without disrupting the logical workflows of the UI component.

Likewise, all oulunliikenne.fi microservices are not used only through the API gateway, some also provide additional APIs that are used directly by the UI. More on this below in the Oulunliikenne APIs section.

The API gateway is built using Apollo and uses schema stitching to combine the numerous GraphQL APIs into one single API for the UI to use. This allows all microservices to be developed and run independently and then configured into the gateway, which handles routing the API requests to the underlaying APIs based on the combined schema definition of all the attached APIs.

Digitransit APIs

For more information about the Digitransit APIs, please read the official documentation.

Oulunliikenne APIs

The microservice architecture of the oulunliikenne.fi APIs illustrated in Figure 2 shows all the services and how they are connected to each other as well as to 3rd party services.

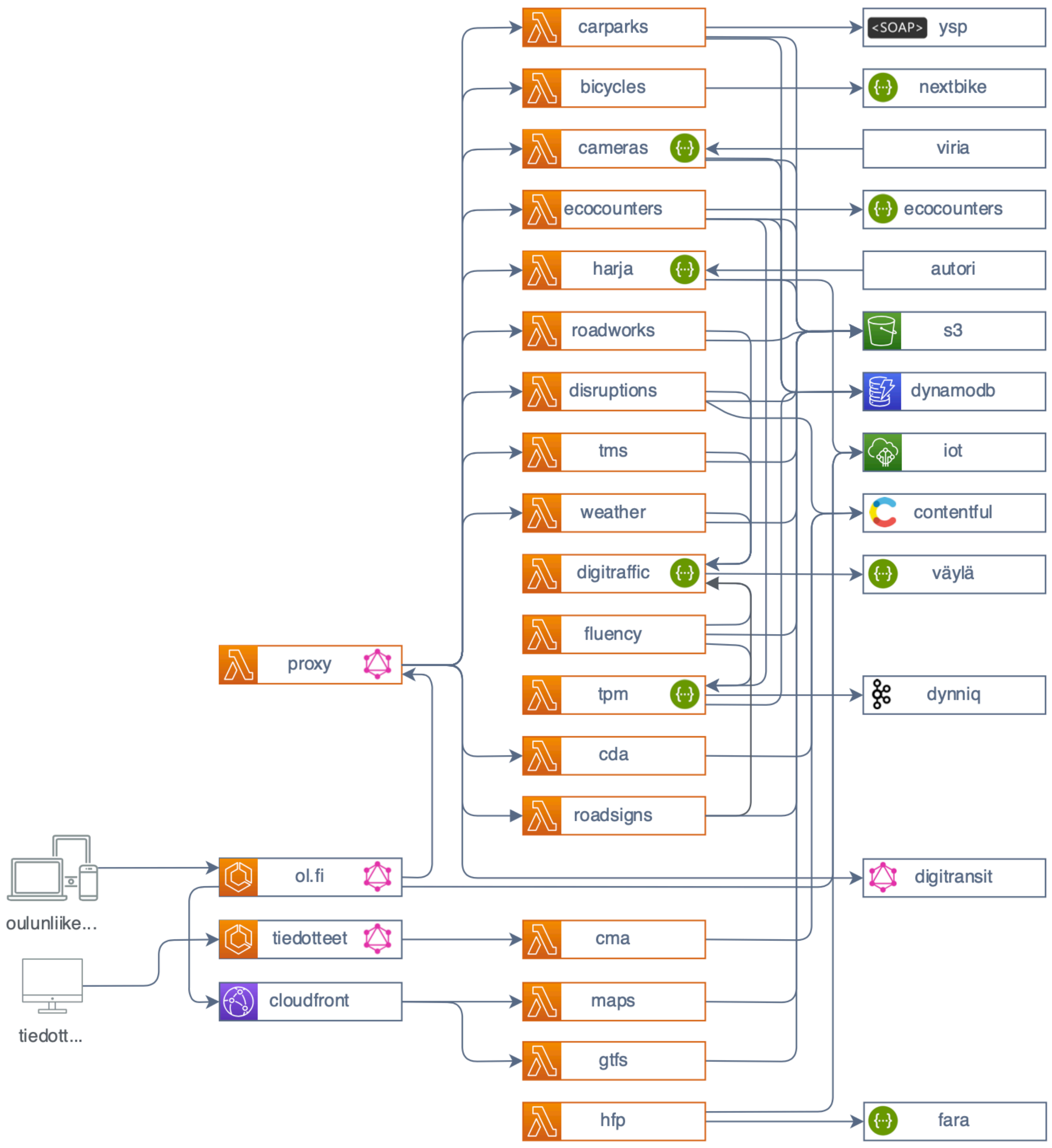


Figure 2: Microservice architecture of oulunliikenne.fi

Most microservices are built using the Serverless framework with AWS as the provider, but some services, or part of the services, are built using Docker containers due to technical limitations with the serverless architecture. It is possible, with relatively small effort, to convert any or all microservices to use Docker instead of Serverless, if it is preferred.

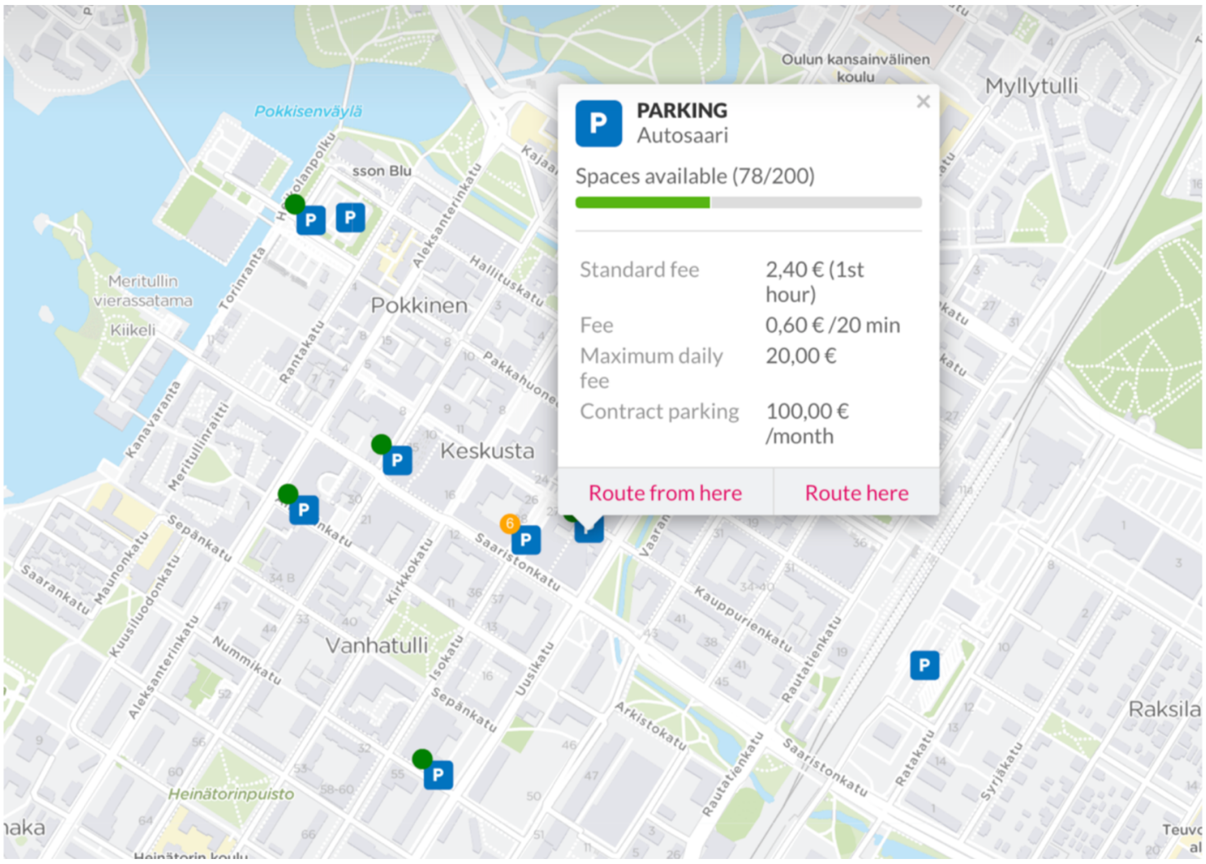
The microservices utilize many other AWS services for e.g., databases, event queues and file storage, which ties them to the AWS ecosystem, but the services are, for the most part, built to work independently which makes it possible to migrate them to use other technologies as well. One benefit of the Serverless framework is that is supports the other major cloud platforms too, i.e. Microsoft Azure and Google Cloud. While it requires some changes to configurations and permissions schemes, it is possible to re-deploy the entire system to another cloud providers ecosystem. Some technical solutions might not be directly transferrable to another provider, due to implementation differences between the providers, but for the most part it is a straightforward process.

The main API architecture used in all services is GraphQL, due to its dynamic nature and abilities to merge multiple APIs into one for the API Gateway. Some services are using REST APIs, especially for integrations with 3rd. party services, and they are usually not exposed directly to the API consumer, but instead work as data sources for the GraphQL APIs, allowing them to expose the data, or parts of the data, in a more structured way across all services.

The main programming language used is NodeJS, as it is supported natively by AWS Lambda and works seamlessly with Docker as well. Using the same language and software architecture across all microservices makes understanding and adopting of the services much easier and efficient.

Carparks API

The Carparks API provides information about carparks in the city of Oulu, their location, real-time space availability and optional pricing information as seen in Figure 3.



PARKING
Autosaari
Spaces available (78/200)

Standard fee	2,40 € (1st hour)
Fee	0,60 € /20 min
Maximum daily fee	20,00 €
Contract parking	100,00 € /month

Figure 3: Carparks in the city of Oulu

At the time of writing, the carparks are shown independently and are not used in any journey planning queries that would allow the user to plan a route by using the carparks as a part of the journey.

Most carparks are fetched from a 3rd. party API managed by YSP and these will have real-time space availability shown and updated every minute. Other carparks are managed locally by the service and do not have any space information. Any pricing information shown for carparks is not fetched from any source, but locally managed in the service.

All carparks are stored by the service in a NoSQL database.

The Carparks API is built using the Serverless framework and consists of the following:

- GraphQL API
    - Public API used by the API Gateway
    - Queries for fetching all available carparks as well as individual carparks ○ Schema and data models can be viewed here
- Scheduled events

- Update space availability
    - Timed event that runs every minute
    - Fetches current space availability for all carparks from the YSP API
- Synchronize carparks
    - Not run automatically at the moment, can be run manually
        - Synchronizes the carparks from the YSP API
    - Only needed when new carparks are added or removed
- Create GeoJSON of carpark locations
    - Not run automatically at the moment, can be run manually
    - Generates a GeoJSON structure for the carpark locations to be used by the Maps API when UI renders them on the map
    - Only needed when new carparks are added or removed

The Carparks API architecture is illustrated in Figure 4.

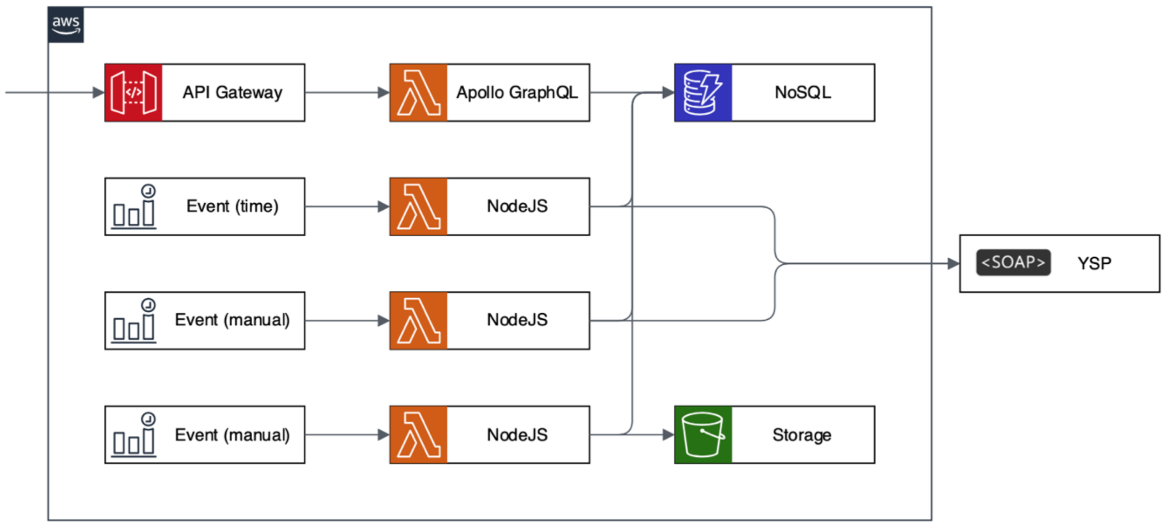


Figure 4: Carparks API architecture diagram

Technologies:

- AWS API Gateway
- AWS Lambda
- AWS CloudWatch
- AWS DynamoDB
- AWS S3
- GraphQL
- SOAP
- NodeJS

Bicycles API

The Bicycles API provides information about city bike stations in the city of Oulu, their location and real-time availability as seen in **[FIGURE]**.

City bike stations are fetches real-time from the Nextbike API when queried through the Bicycles API and each station has the real-time availability of how many bikes are currently at the station as well as the empty spaces available to leave a bike at.

The bicycle station can be used as a part of the journey planning when searching for routes. This feature is available through Digitransit has it also has a direct integration with Nextbike APIs.

The Bicycles API is built using the Serverless framework and consists of the following:

- GraphQL API
  - Public API used by the API Gateway
  - Queries for fetching all available bicycle stations as well as individual ones
  - Schema and data models can be viewed here
- Scheduled events
  - Create GeoJSON of bicycle station locations
    - Timed event run nightly
    - Generates a GeoJSON structure for the bicycle station locations to be used by the Maps API when UI renders them on the map

The Bicycles API architecture is illustrated in Figure 5.

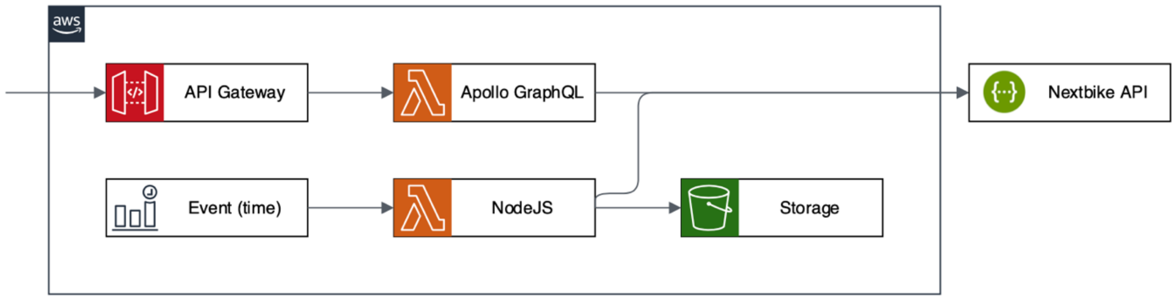


Figure 5: Bicycles API architecture diagram

Technologies:

- AWS API Gateway
- AWS Lambda
- AWS CloudWatch
- AWS S3
- GraphQL
- REST
- NodeJS

Additionally, the Bicycles API includes static resources in GeoJSON format for displaying different kinds if bicycle routes in the Oulu area as seen in Figure 6. These are generated from shapefiles provided by Oulunliikenne and used by the Maps API when UI renders them on the map.

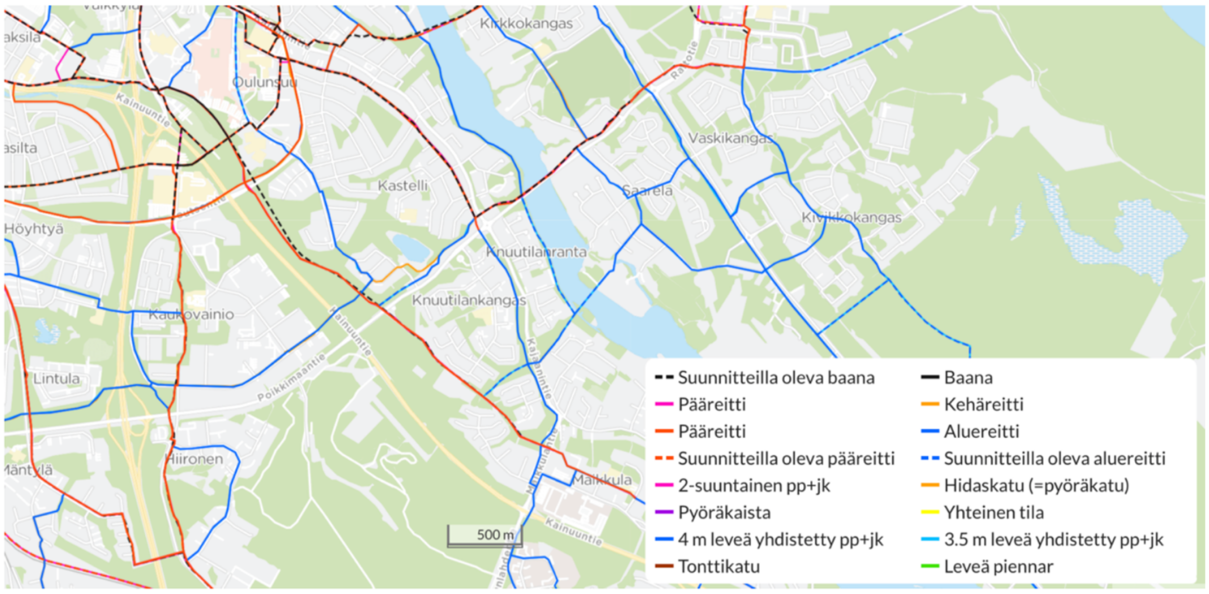


Figure 6: Bicycle routes in the Oulu region

Cameras API

The Cameras API provides near real-time traffic camera captures and locations of cameras in the Oulu region as seen in Figure 7.

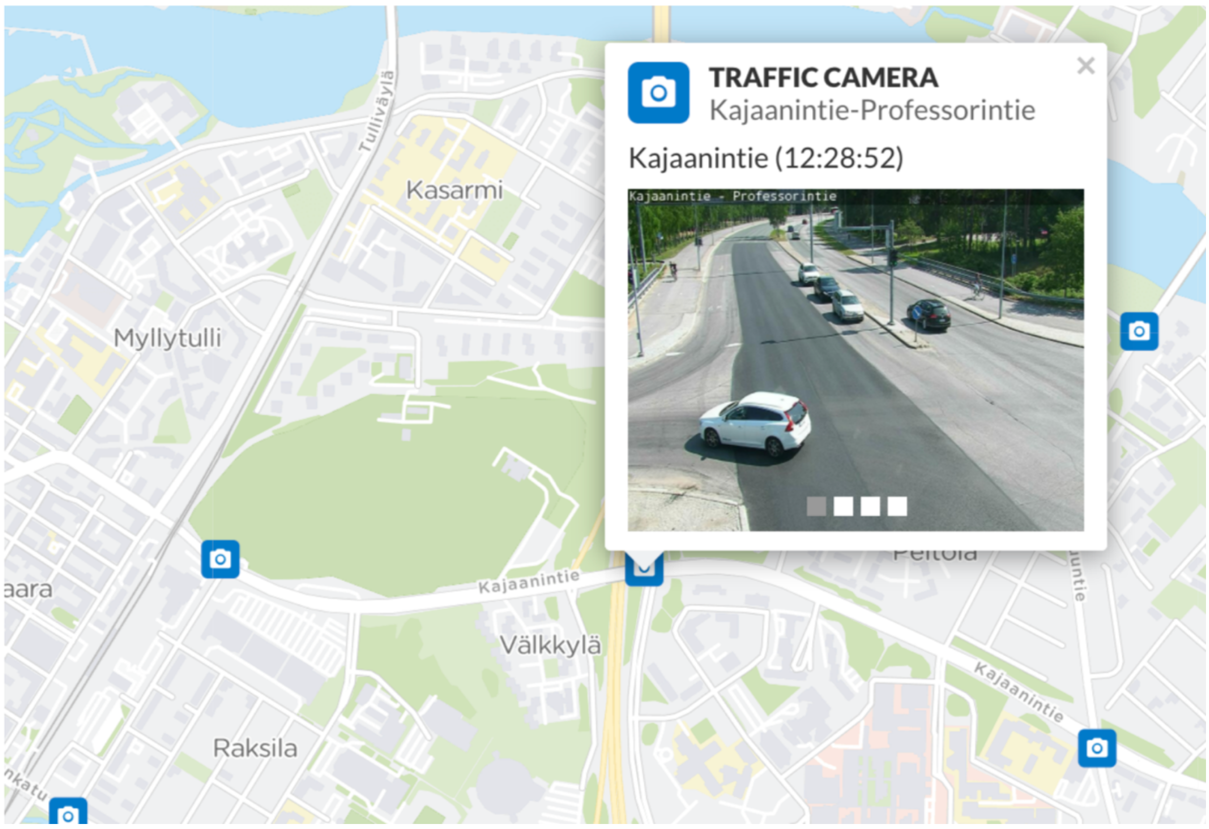


Figure 7: Traffic cameras in the Oulu region

The traffic camera images come from two different sources; the Digitraffic service operated by Traffic Management Finland, and from cameras operated by Viria Oy. The oulunliikenne.fi service fetches new camera images from Digitraffic every minute, but Digitraffic determines the final refresh interval of the images which is around 15 minutes. Viria sends their camera images to the oulunliikenne.fi service and the refresh interval is thus completely based on that interval. The image files for Digitraffic sourced images are hosted by Digitraffic, while the image files sent by Viria are hosted by oulunliikenne.fi.

The traffic camera locations for Digitraffic sourced camera images comes from the Digitraffic API and they are updated in the oulunliikenne.fi service on a nightly basis, which means either newly added cameras or removed ones will update once a day. Camera locations for Viria sourced camera images is managed locally in the oulunliikenne.fi service. Viria provides the information in an Excel file which is imported into a NoSQL database that hold the camera information.

The Cameras API is built using the Serverless framework and consists of the following:

- GraphQL API
    - Public API used by the API Gateway

        - Queries for fetching all available cameras including current images as well as individual ones, also supports querying cameras by source
        - Schema and data models can be viewed here
- REST API
    - Used by Viria to send traffic camera images to the service
- Queue
    - Used to process camera images received by Viria to offload the work from the REST API into an asynchronous service
- Scheduled events
    - Create GeoJSON of bicycle station locations
        - Timed event run nightly
        - Generates a GeoJSON structure for the traffic camera locations to be used by the Maps API when UI renders them on the map

The Cameras API architecture is illustrated in Figure 8.

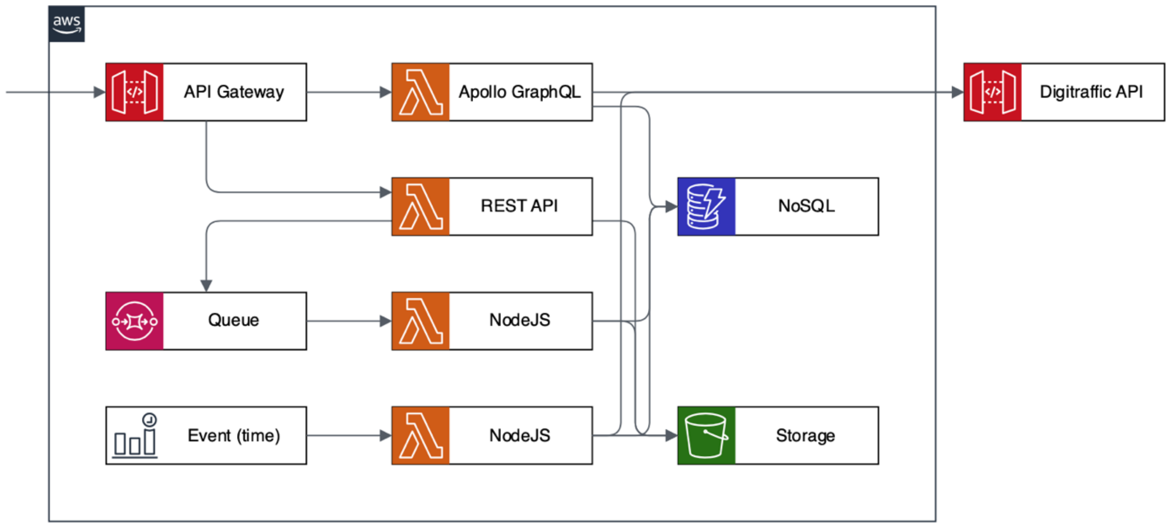


Figure 8: Cameras API architecture diagram

Technologies:

- AWS API Gateway
- AWS Lambda
- AWS CloudWatch
- AWS SQS
- AWS DynamoDB
- AWS S3
- GraphQL
- REST
- NodeJS

Eco-counters API

The Eco-counters API provides information about pedestrian and bicycle path usage for defined locations in the Oulu region as seen in Figure 9.

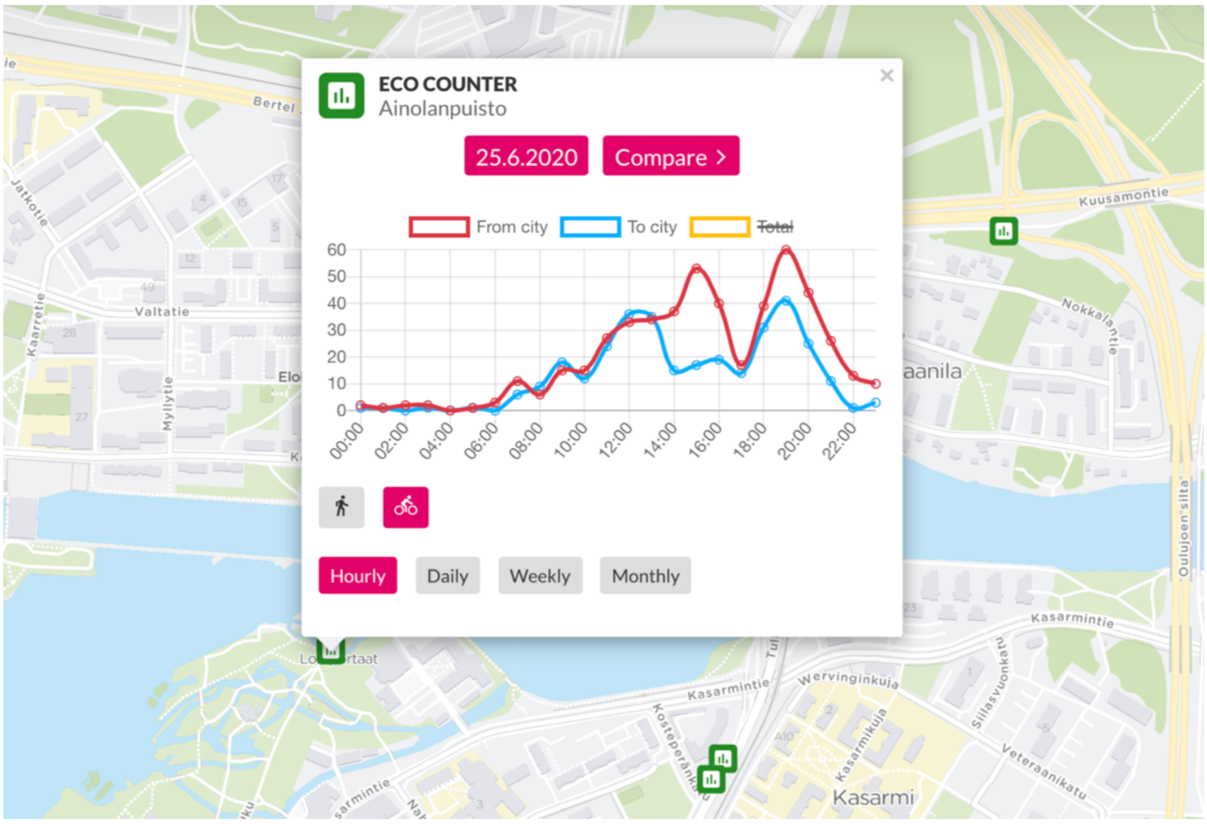


Figure 9: Eco-counters in the Oulu region

The API provides eco-counters from three different sources:

1. The Eco Counter company that provides counter hardware on location and APIs to retrieve information
2. Local counters maintained by Oulunliikenne which report usage to this API individually
3. The Traffic Performance Monitoring API, also maintained by Oulunliikenne, which reports usage based on counters in traffic lights in the Oulu region

The Eco-counters API is built using the Serverless framework and consists of the following:

- GraphQL API
  - Public API used by the API Gateway
  - Queries for fetching all available eco-counter locations as well as statistics for a selected time interval
  - Schema and data models can be viewed here
- REST API
  - Used by local eco-counters to send usage data to the service
- Scheduled events
  - Create GeoJSON of eco-counter locations

        - Timed event run nightly
        - Generates a GeoJSON structure for the traffic camera locations to be used by the Maps API when UI renders them on the map
    - Update usage data from TPM API
        - Timed event run every minute
        - Fetches new data for all registered eco-counter devices in the NoSQL database and stores the usage data
- Stream events
    - Aggregate usage data for defined time intervals
        - Both local and TPM based eco-counter usage data is stored in a NoSQL database and all changes to the data is streamed to a Lambda function that calculates aggregations of the data and stores it into different database tables
        - The aggregation tables are used by the GraphQL API to fetch the selected time interval without having to query the entire data set

The Eco-counters API architecture is illustrated in Figure 10.

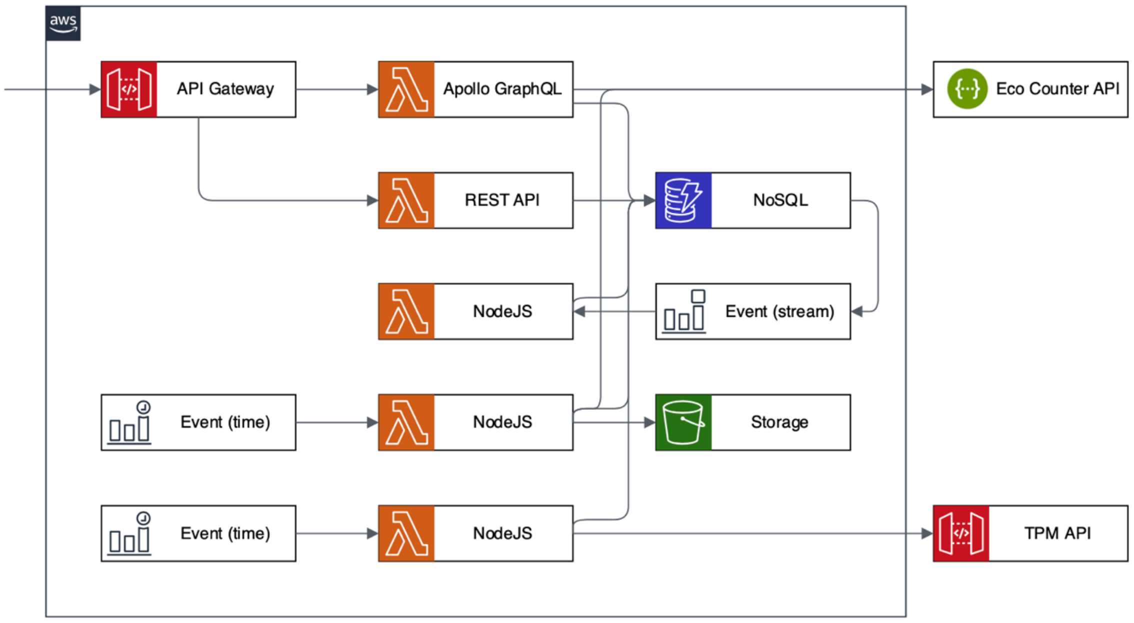


Figure 10: Eco-counters API architecture diagram

Technologies:

- AWS API Gateway
- AWS Lambda
- AWS CloudWatch
- AWS S3
- GraphQL
- REST
- NodeJS

Harja API

The Harja API provides information about performed maintenance tasks and the real-time location of maintenance vehicles in the Oulu region. The completed tasks available in the oulunliikenne.fi service as coloured geometry lines drawn on top of the map following the road network. The different colours represent the last completed task as seen in Figure 11.

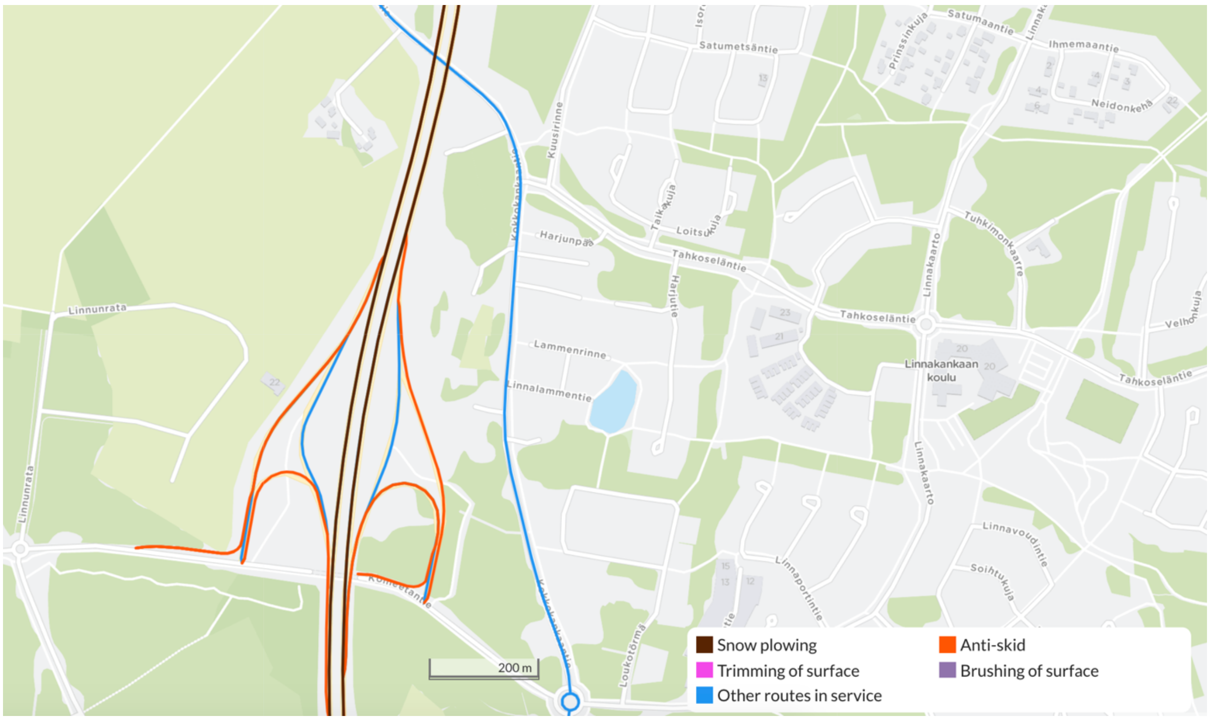


Figure 11: Road maintenance tasks in the Oulu region

At the time of writing, the maintenance tasks recorded are for the winter months and consists of:

- Snow ploughing
- Anti-skid
- Trimming of surface
- Brushing of surface

All information about completed maintenance tasks and vehicle locations are provided by Autori through a REST API in the service. The API follows the Finnish Transport Infrastructure Agency's Harja API definition and the service is also named after that API. The Harja API uses only two of the endpoints defined; one for receiving real-time vehicle locations, and the other for completed maintenance tasks.

Real-time vehicle positions are written both to a NoSQL database and to the AWS IoT service. The database is used for retrieving past positions of the vehicles as statistical data through the GraphQL API, while the IoT service is used for real-time communication of the current vehicle positions to any and all users currently using the oulunliikenne.fi service. The IoT service provides a message broker implementation based on the MQTT connectivity protocol, for lightweight publish/subscribe functionality, that is able to serve all users in real-time. This is the same technology used for the real-time positions of public transportation vehicles in the HFP API.

Completed maintenance tasks are written into a NoSQL database table from where they are streamed to an AWS Lambda function that writes them to an AWS Kinesis stream, which in turn uses a fan-out pattern to provide the tasks to any and all listeners for processing. There are two listeners for the Kinesis stream; the first replicates the task into the development environment, as Autori can only send data to one environment, the other maps the task geometry to the road geometry so that the task can be shown on the map as a section of the road that has been maintained.

The mapping of task geometries is based on finding the closest road geometry for the maintenance task geometry points that Autori sends. Every task has only got 2 geometry points when data is received from Autori, and to be able to draw a line string on the map for the section of the road that the maintenance task is for, we need more points that follow along the road. Tasks are divided into 2 types, the main bicycle routes in the Oulu area and routes maintained by the ELY Centre of North Ostrobothnia. Both task types have their own geometries, the first maintained by Oulunliikenne and the second by the Finnish Transport Infrastructure Agency. The mapping algorithm chooses the correct geometry by the task type and then finds the closest geometry by measuring the distance between the start and end points for each of the geometries included. The algorithm also compares the points within a longer geometry line, in order to also find partial matches if the task has only been performed for a small section of the road. The algorithm allows for a 50-meter radius around both the start and end points as the two points received for the task are not always very accurate as they are based on GPS coordinates from the maintenance vehicles.

The road geometries are pre-processed and formatted into GeoJSON LineString features to make it easier for the mapping algorithm to handle them consistently.

The Harja API is built using the Serverless framework and consists of the following:

- GraphQL API
    - Public API used by the API Gateway
    - Queries for fetching completed maintenance tasks and maintenance vehicle positions for a time interval
    - Schema and data models can be viewed here
- REST API
    - Used by Autori to send data to the service
- Scheduled events
    - Create GeoJSON of completed maintenance tasks for past 3 days
        - Timed event run every minute
        - Generates a GeoJSON structure for the maintenance task geometry to be used by the Maps API when UI renders them on the map
- Stream events
    - Streams maintenance tasks from AWS DynamoDB to AWS Kinesis

        - All maintenance tasks written to AWS DynamoDB are streamed to an AWS Lambda function that writes them into an AWS Kinesis stream which uses a fan-out pattern to process the data for different purposes
        - AWS DynamoDB only allows a small amount of stream handlers, which is why AWS Kinesis is used for fan-out.
    - Maps maintenance task geometries from AWS Kinesis stream
        - All maintenance tasks received from Autori need to be mapped against a predefined geometry that can be used to draw the completed tasks on the map
        - Autori sends only 2 geometry points for each completed task, which need to be mapped to a complete geometry line string that follows the section of the road that the task was for
        - The mapped geometry is written back into the AWS DynamoDB table to be used both by the GraphQL API and the GeoJSON event handler
    - Replicates maintenance tasks from AWS Kinesis stream into development environment
        - Autori can only send data to the production environment, so the incoming maintenance tasks in production are replicated in AWS DynamoDB for the development environment by streaming them from AWS Kinesis and writing them into the development AWS DynamoDB table using an AWS Lambda function
- MQTT message broker
    - Delivers real-time maintenance vehicle positions to users

The Harja API architecture is illustrated in Figure 12.

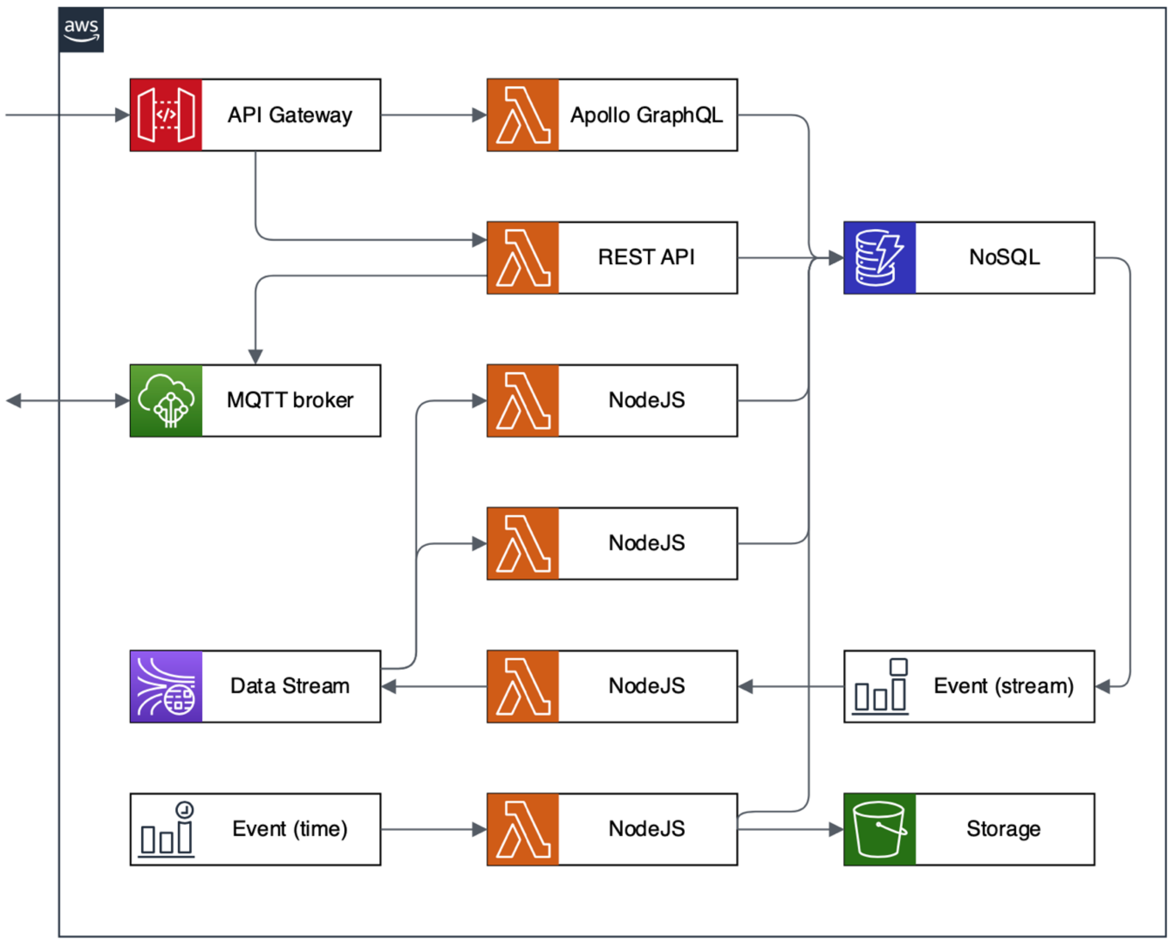


Figure 12: Harja API architecture diagram

Technologies:

- AWS API Gateway
- AWS Lambda
- AWS CloudWatch
- AWS S3
- AWS DynamoDB
- AWS Kinesis
- AWS IoT
- GraphQL
- REST
- MQTT
- NodeJS

Roadworks API

The Roadworks API provides information about ongoing roadworks in the Oulu region as seen in Figure 13.

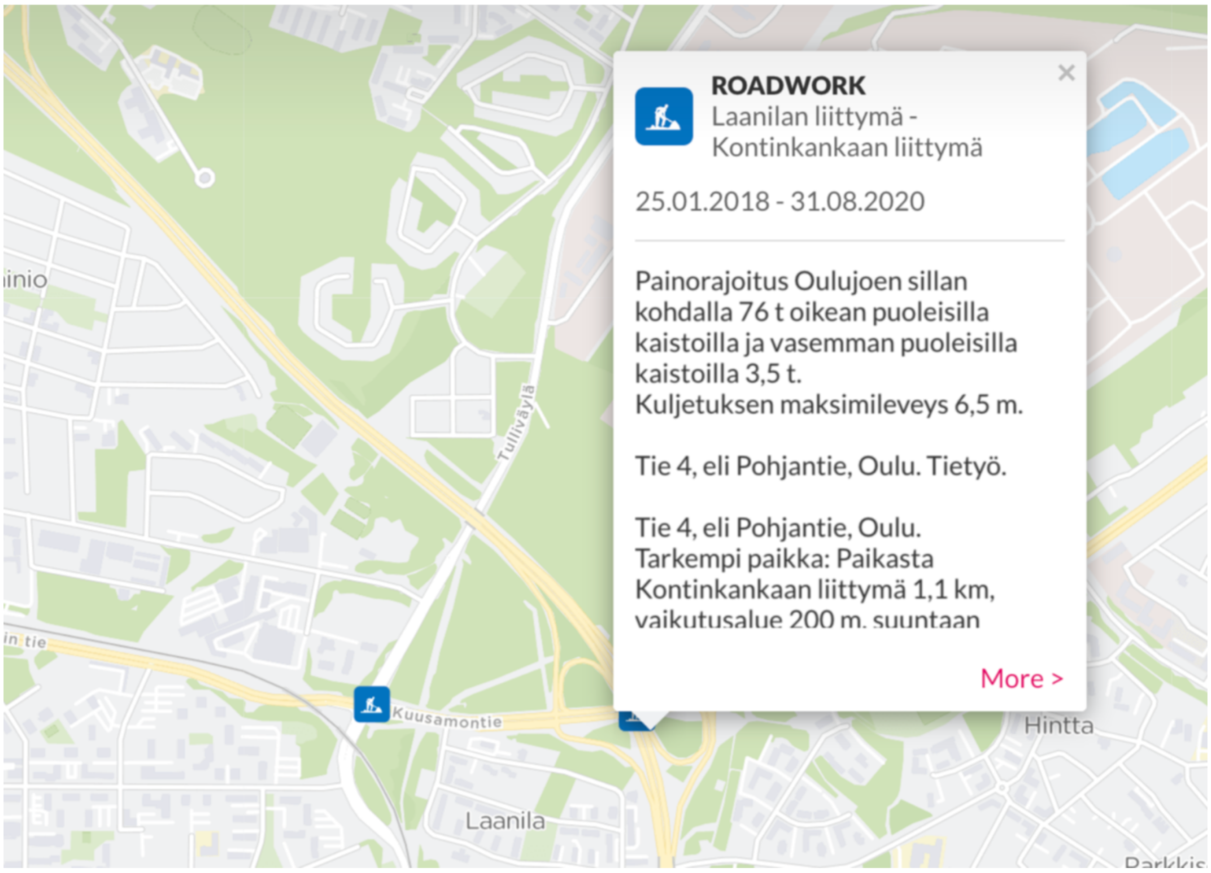

ROADWORK
Laanilan liittymä -
Kontinkankaan liittymä
25.01.2018 - 31.08.2020
Painorajoitus Oulujoen sillan kohdalla 76 t oikean puoleisilla kaistoilla ja vasemman puoleisilla kaistoilla 3,5 t.
Kuljetuksen maksimileveys 6,5 m.
Tie 4, eli Pohjantie, Oulu. Tietyö.
Tie 4, eli Pohjantie, Oulu.
Tarkempi paikka: Paikasta Kontinkankaan liittymä 1,1 km, vaikutusalue 200 m, suuntaan
More >
Kuusamontie
Tulliväylä
Laanila
Hintta

Figure 13: Roadworks in the Oulu region

The roadwork details come from the national Digitraffic service operated by Traffic Management Finland. The Roadworks API fetches the roadwork details through the oulunliikenne.fi Digitraffic API which filters the roadworks found in the national Digitraffic service to include only ones in the Oulu region.

The Roadworks API updates a GeoJSON file for the roadwork locations once every minute from the Digitraffic API, which in turn caches the roadworks found in the Oulu region for the same amount of time. This is done both to prevent high traffic peaks to the national Digitraffic service and to improve the performance of loading the roadwork details in the service, as Digitraffic always returns all roadworks nationwide.

Roadworks are not currently used as a part of the journey planning, e.g., to find alternative routes where roadworks affect the travel time significantly.

The Roadworks API is built using the Serverless framework and consists of the following:

- GraphQL API
    - Public API used by the API Gateway

    - Queries for fetching all ongoing roadworks as well as individually from the Digitraffic REST API
    - Schema and data models can be viewed here
- Scheduled events
    - Create GeoJSON of ongoing roadworks
        - Timed event run every minute
        - Generates a GeoJSON structure for the locations to be used by the Maps API when UI renders them on the map

The Roadworks API architecture is illustrated in Figure 14.

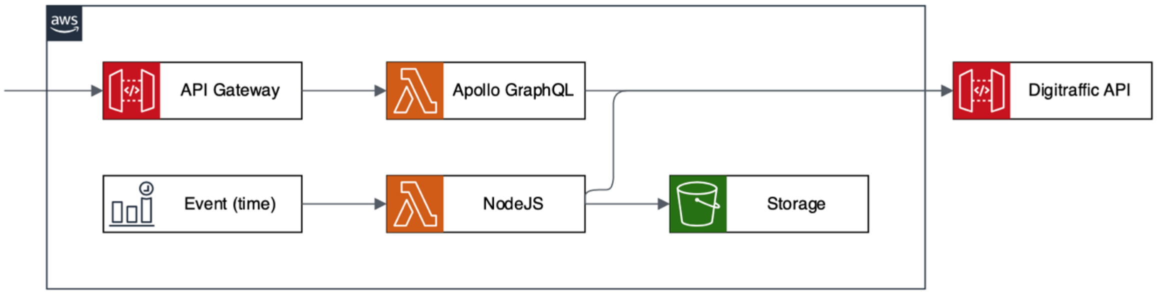


Figure 14: Roadworks API architecture diagram

Technologies:

- AWS API Gateway
- AWS Lambda
- AWS CloudWatch
- AWS S3
- GraphQL
- REST
- NodeJS

Disruptions API

The Disruptions API provides information about any disruptive maintenance, construction work or events going on in the Oulu region that may hinder or slow down public transportation as well as car-, bicycle- and pedestrian traffic as seen in Figure 15.

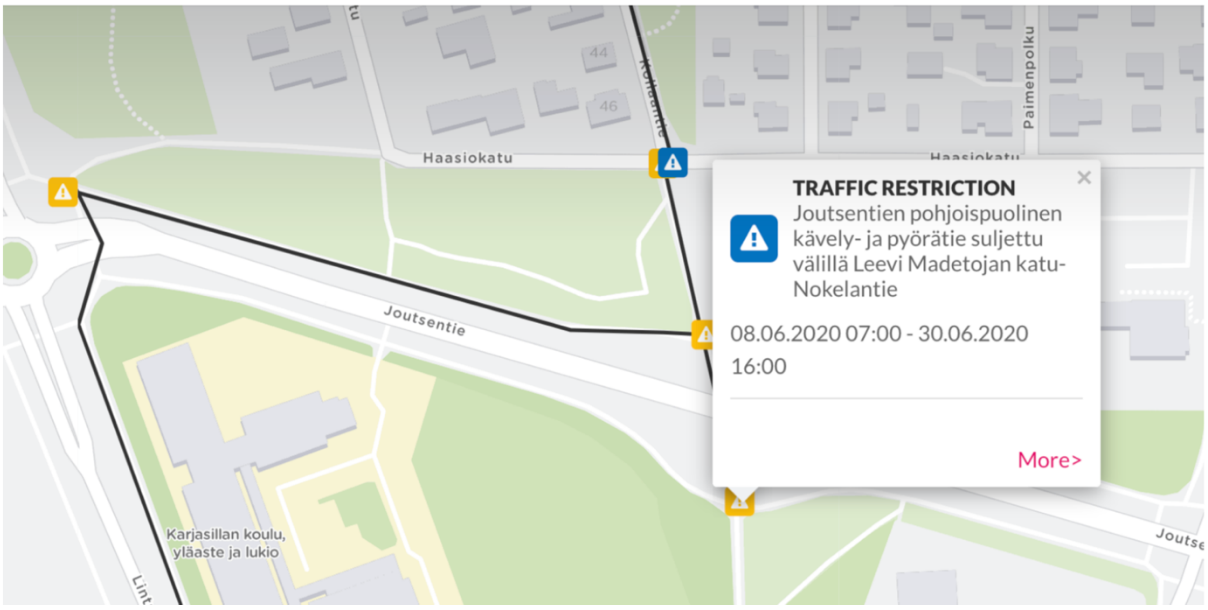

Haasiokatu
Haasiokatu
Paimenpolku
Nokelantie
Joutsentie
Joutse
Karjasillan koulu, yläaste ja lukio
Lint
44
46
TRAFFIC RESTRICTION
Joutsentien pohjoispuolinen kävely- ja pyörätie suljettu välillä Leevi Madetojan katu-Nokelantie
08.06.2020 07:00 - 30.06.2020 16:00
More>

Figure 15: Disruptions in the Oulu region

Disruption data comes from two different sources; the national Digitraffic service operated by Traffic Management Finland, and disruptions managed by Oulunliikenne.

The Disruptions API fetches the disruption details from the national Digitraffic service through the oulunliikenne.fi Digitraffic API which filters the results to include only ones in the Oulu region. These are cached for 1 minute in the API. The Digitraffic disruptions are usually related to maintenance work where roads and/or lanes are closed.

Disruptions managed by Oulunliikenne are fetched from Contentful and they created and updated from a separate tool developed for purpose. The disruptions are saved to Contentful from this tool and then fetched from there real-time when accessed in the oulunliikenne.fi service.

The Disruptions API creates a GeoJSON file for all locations once every minute, to be used by the Maps API when UI renders them on the map. The Digitraffic API caches the disruptions also for one minute to both to prevent high traffic peaks to the national Digitraffic service and to improve the performance of loading the disruption details in the service, as Digitraffic always returns all disruptions nationwide. Disruptions fetched from Contentful are always the latest data available when generating the GeoJSON.

Disruptions are not currently used as a part of the journey planning, e.g., to find alternative routes where any disruption affect the travel time significantly.

The Disruptions API is built using the Serverless framework and consists of the following:

- GraphQL API
    - Public API used by the API Gateway
    - Queries for fetching all current disruptions as well as individually and separated by the source type
    - Schema and data models can be viewed here
- Scheduled events
    - Create GeoJSON of current disruptions
        - Timed event run every minute
        - Generates a GeoJSON structure for the locations to be used by the Maps API when UI renders them on the map

The Disruptions API architecture is illustrated in Figure 16.

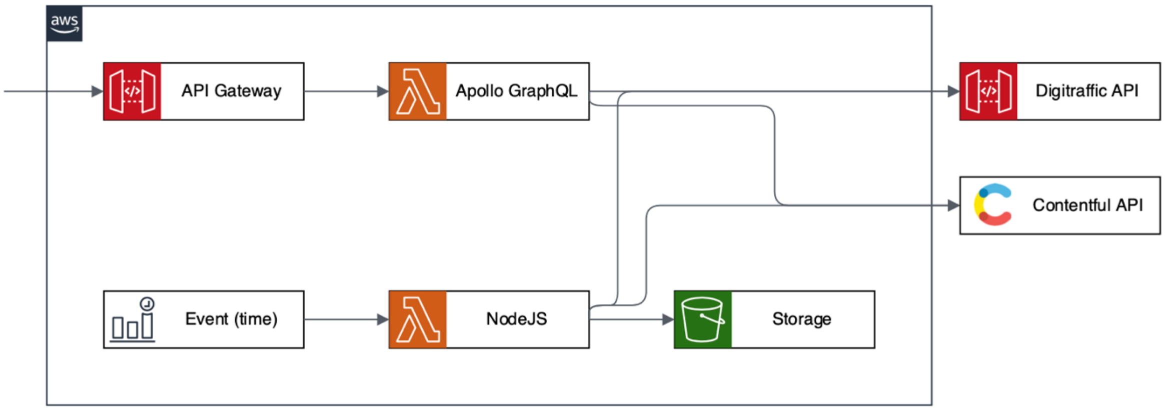


Figure 16: Disruptions API architecture diagram

Technologies:

- AWS API Gateway
- AWS Lambda
- AWS CloudWatch
- AWS S3
- GraphQL
- REST
- NodeJS

TMS API

The Traffic Monitoring System API (TMS) provides information about TMS/LAM stations in the Oulu region as seen in Figure 17.

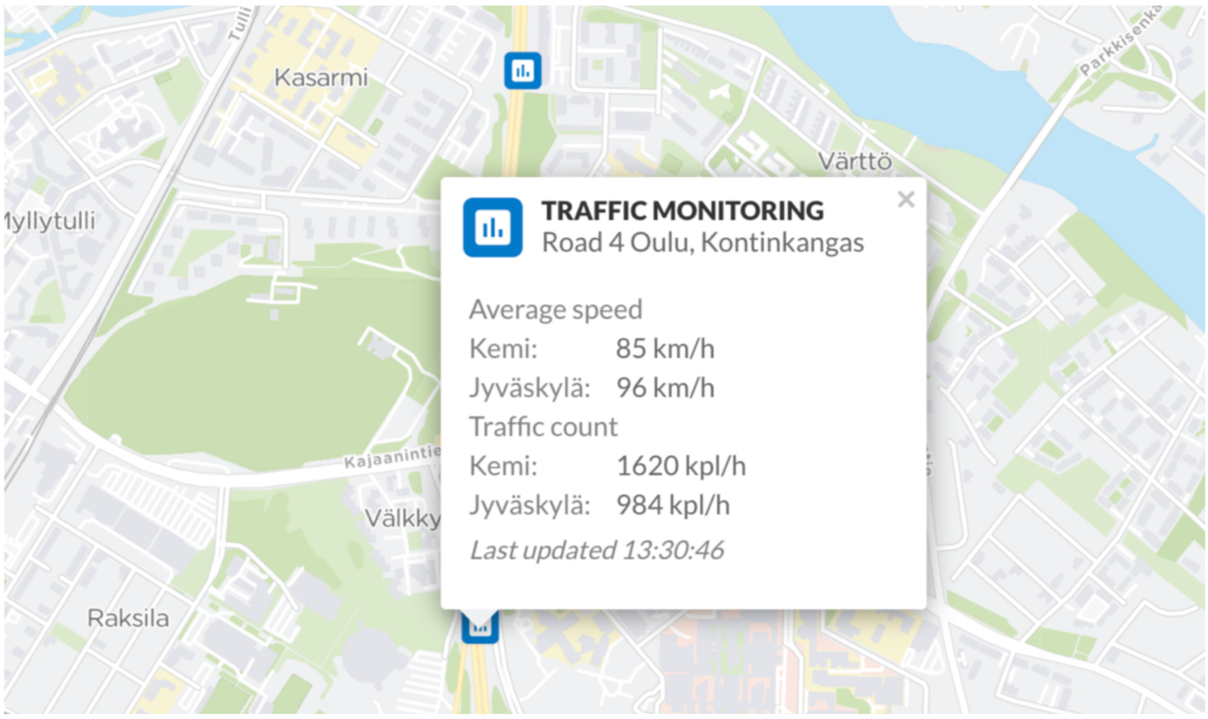


Figure 17: TMS stations in the Oulu region

The TMS API fetches the information from the national Digitraffic service operated by Traffic Management Finland. The details are fetched through the oulunliikenne.fi Digitraffic API which filters the results found in the national Digitraffic service to include only ones in the Oulu region.

The TMS API updates a GeoJSON file for the TMS-station locations once every day from the Digitraffic API, which in turn caches them for the same amount of time. This is done both to prevent high traffic peaks to the national Digitraffic service and to improve the performance of loading the TMS-station details in the service, as Digitraffic always returns all TMS-stations nationwide. The data provided for each individual TMS-station is fetched in near real-time and cached for 1 minute in the Digitraffic API. The national Digitraffic service dictates when the individual sensor values for each TMS-station is actually updated.

TMS-stations are not currently used as a part of the journey planning, e.g., to find alternative routes where traffic congestion affect the travel time significantly.

The TMS API is built using the Serverless framework and consists of the following:

- GraphQL API
  - Public API used by the API Gateway
  - Queries for fetching all TMS-stations and their data as well as individually
  - Schema and data models can be viewed here
- Scheduled events
  - Create GeoJSON of all TMS-stations
    - Timed event run every day
    - Generates a GeoJSON structure for the locations to be used by the Maps API when UI renders them on the map

The TMS API architecture is illustrated in Figure 18.

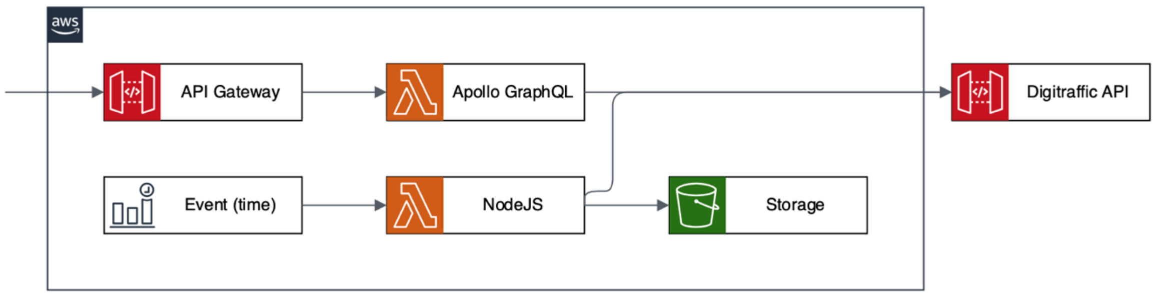


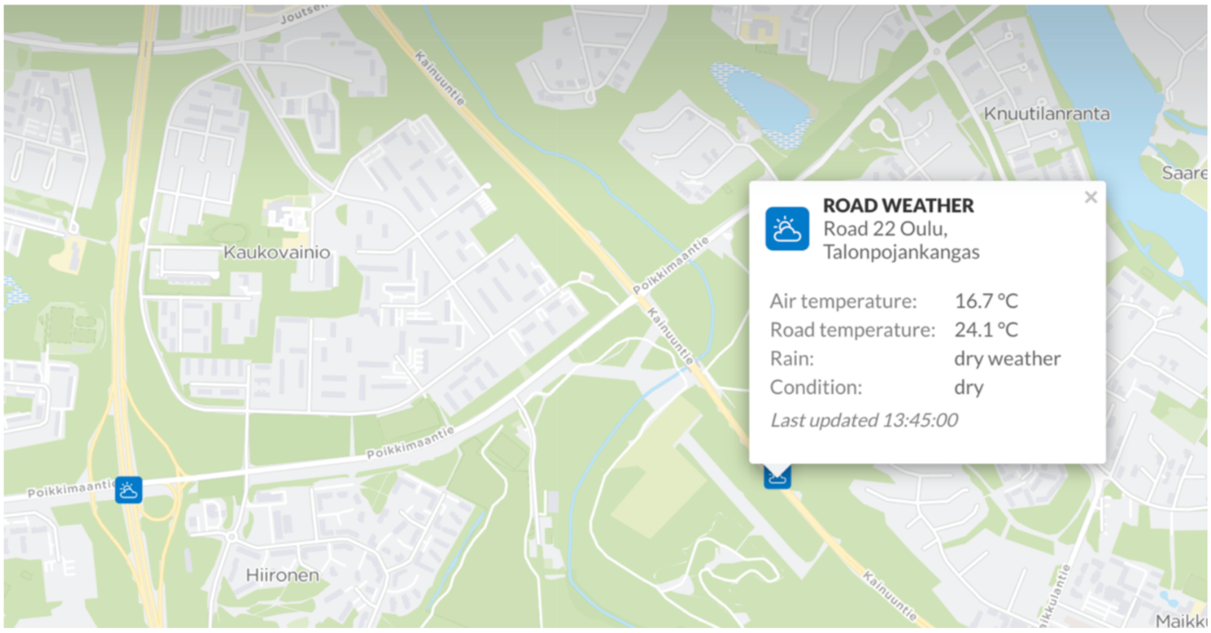
Figure 18: TMS API architecture diagram

Technologies:

- AWS API Gateway
- AWS Lambda
- AWS CloudWatch
- AWS S3
- GraphQL
- REST
- NodeJS

Weather API

The Weather API provides information about road weather conditions in the Oulu region as seen in Figure 19 and Figure 20.

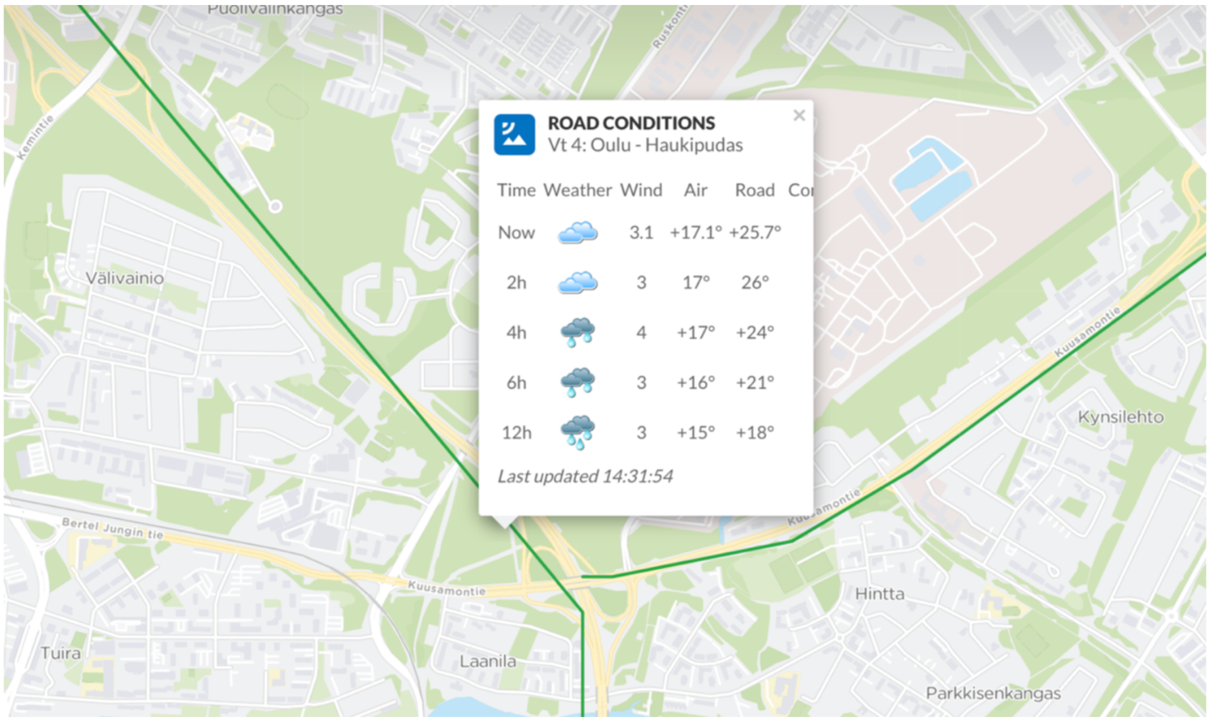


ROAD WEATHER
Road 22 Oulu,
Talonpojankangas

Air temperature:	16.7 °C
Road temperature:	24.1 °C
Rain:	dry weather
Condition:	dry

Last updated 13:45:00

Figure 19: Road weather stations in the Oulu region



ROAD CONDITIONS
Vt 4: Oulu - Haukipudas

Time	Weather	Wind	Air	Road	Co
Now	☁	3.1	+17.1°	+25.7°	
2h	☁	3	17°	26°	
4h	🌧	4	+17°	+24°	
6h	🌧	3	+16°	+21°	
12h	🌧	3	+15°	+18°	

Last updated 14:31:54

Figure 20: Road weather conditions in the Oulu region

The Weather API fetches the information from the national Digitraffic service operated by Traffic Management Finland. The details are fetched through the oulunliikenne.fi Digitraffic API which filters the results found in the national Digitraffic service to include only ones in the Oulu region.

The Weather API updates two GeoJSON files for the Maps API, one for the road weather-station locations once every day, and another for the road weather conditions once every minute, to be used by the Maps API when UI renders them on the map. The source data is cached in the Digitraffic API both to prevent high traffic peaks to the national Digitraffic service and to improve the performance of loading the details in the service, as Digitraffic always returns all stations and conditions nationwide. The data provided for each individual weather-station or road condition is fetched in near real-time and cached for 1 minute in the Digitraffic API. The national Digitraffic service dictates when the individual sensor values for are actually updated.

The Weather API is built using the Serverless framework and consists of the following:

- GraphQL API
  - Public API used by the API Gateway
  - Queries for fetching all weather-stations and road conditions including their data as well as individually
  - Schema and data models can be viewed here
- Scheduled events
  - Create GeoJSON for weather-stations
    - Timed event run every day
    - Generates a GeoJSON structure for the locations to be used by the Maps API when UI renders them on the map
  - Create GeoJSON for road conditions
    - Time event run every minute
    - Generates A GeoJSON structure for the road condition geometry to be used by the Maps API when UI renders them on the map

The Weather API architecture is illustrated in Figure 21.

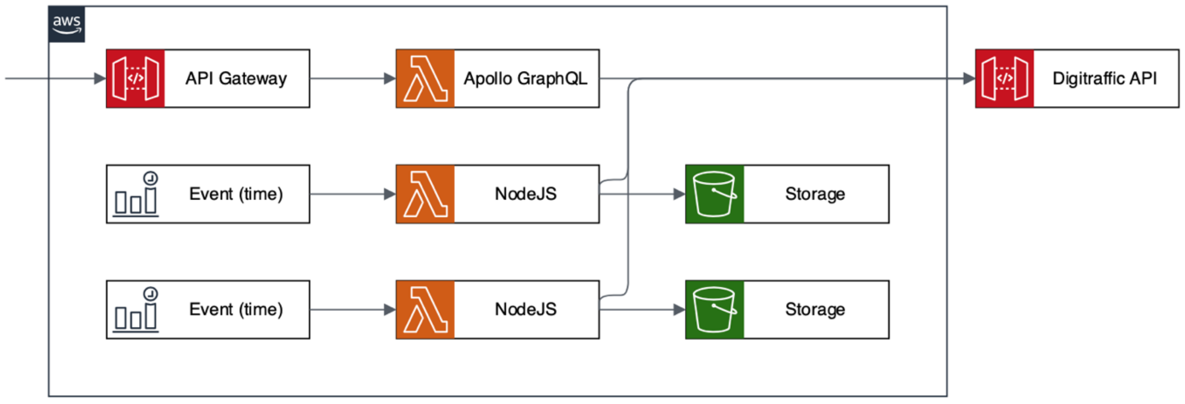


Figure 21: Weather API architecture diagram

Technologies:

- AWS API Gateway
- AWS Lambda
- AWS CloudWatch

- AWS S3
- GraphQL
- REST
- NodeJS

Digitraffic API

The Digitraffic API works a proxy API to the national Digitraffic service operated by Traffic Management Finland. The purpose of this proxy is to both be able to filter data only relevant to oulunliikenne.fi and to cache the data for faster retrieval and at the same time reduce traffic load to the Digitraffic service caused by oulunliikenne.fi users.

The Digitraffic API only implements those API endpoints that are needed by oulunliikenne.fi and it keeps the endpoint URLs and data models of the source intact, which means that the documentation provided by Digitraffic is accurate for the endpoints this API implements.

The Digitraffic API is a REST API implemented using the Serverless framework and it consists of the following:

- API Gateway
  - Public API used by many of other the Oulunliikenne APIs
- Lambda functions for each of the APIs
  - Metadata APIs
    - Get TMC locations as GeoJSON
    - Get traffic camera stations as GeoJSON
    - Get TMS stations as GeoJSON
    - Get TMS sensors
    - Get road weather stations as GeoJSON
    - Get road weather forecast sections as GeoJSON
    - Get variable road sign codes
  - Data APIs
    - Get roadworks as Datex2
    - Get roadwork by situation ID as Datex2
    - Get traffic disorders as Datex2
    - Get traffic disorder by situation ID as Datex2
    - Get traffic camera data
    - Get traffic camera data by ID
    - Get TMS data
    - Get TMS data by ID
    - Get road weather station data
    - Get road weather station data by ID
    - Get road condition data
    - Get variable road sign data
    - Get variable road sign data by ID
- Scheduled events
  - Update locations
    - Time event run every night
    - Keeps an updated version of TMC locations from the Finnish Transport Infrastructure Agency on S3 with only Oulu specific locations
  - Update forecast sections
    - Time event run every night

        - Keeps an updated version of forecast sections from the Finnish Transport Infrastructure Agency on S3 with only Oulu specific sections
    - Update camera stations
        - Time event run every night
        - Keeps an updated version of camera stations from the Finnish Transport Infrastructure Agency on S3 with only Oulu specific locations
    - Update TMS stations
        - Time event run every night
        - Keeps an updated version of TMS stations from the Finnish Transport Infrastructure Agency on S3 with only Oulu specific locations
    - Update TMS sensors
        - Time event run every night
        - Keeps an updated version of TMS sensors from the Finnish Transport Infrastructure Agency on S3
    - Update weather stations
        - Time event run every night
        - Keeps an updated version of road weather stations from the Finnish Transport Infrastructure Agency on S3 with only Oulu specific locations
    - Update roadworks
        - Time event run every minute
        - Keeps an updated version of ongoing/planned roadwork data from the Finnish Transport Infrastructure Agency on S3 with only Oulu specific locations
    - Update traffic disorders
        - Time event run every minute
        - Keeps an updated version of active traffic disorder data from the Finnish Transport Infrastructure Agency on S3 with only Oulu specific locations
    - Update camera data
        - Time event run every minute
        - Keeps an updated version of traffic camera data from the Finnish Transport Infrastructure Agency on S3 with only Oulu specific locations
    - Update TMS data
        - Time event run every minute
        - Keeps an updated version of TMS data from the Finnish Transport Infrastructure Agency on S3 with only Oulu specific locations
    - Update weather data
        - Time event run every minute
        - Keeps an updated version of road weather data from the Finnish Transport Infrastructure Agency on S3 with only Oulu specific locations
    - Update road condition data
        - Time event run every minute

        - Keeps an updated version of road condition data from the Finnish Transport Infrastructure Agency on S3 with only Oulu specific locations
    - Update variable signs
        - Time event run every minute
        - Keeps an updated version of variable road signs from the Finnish Transport Infrastructure Agency on S3 with only Oulu specific locations

The Digitraffic API architecture is illustrated in Figure 22.

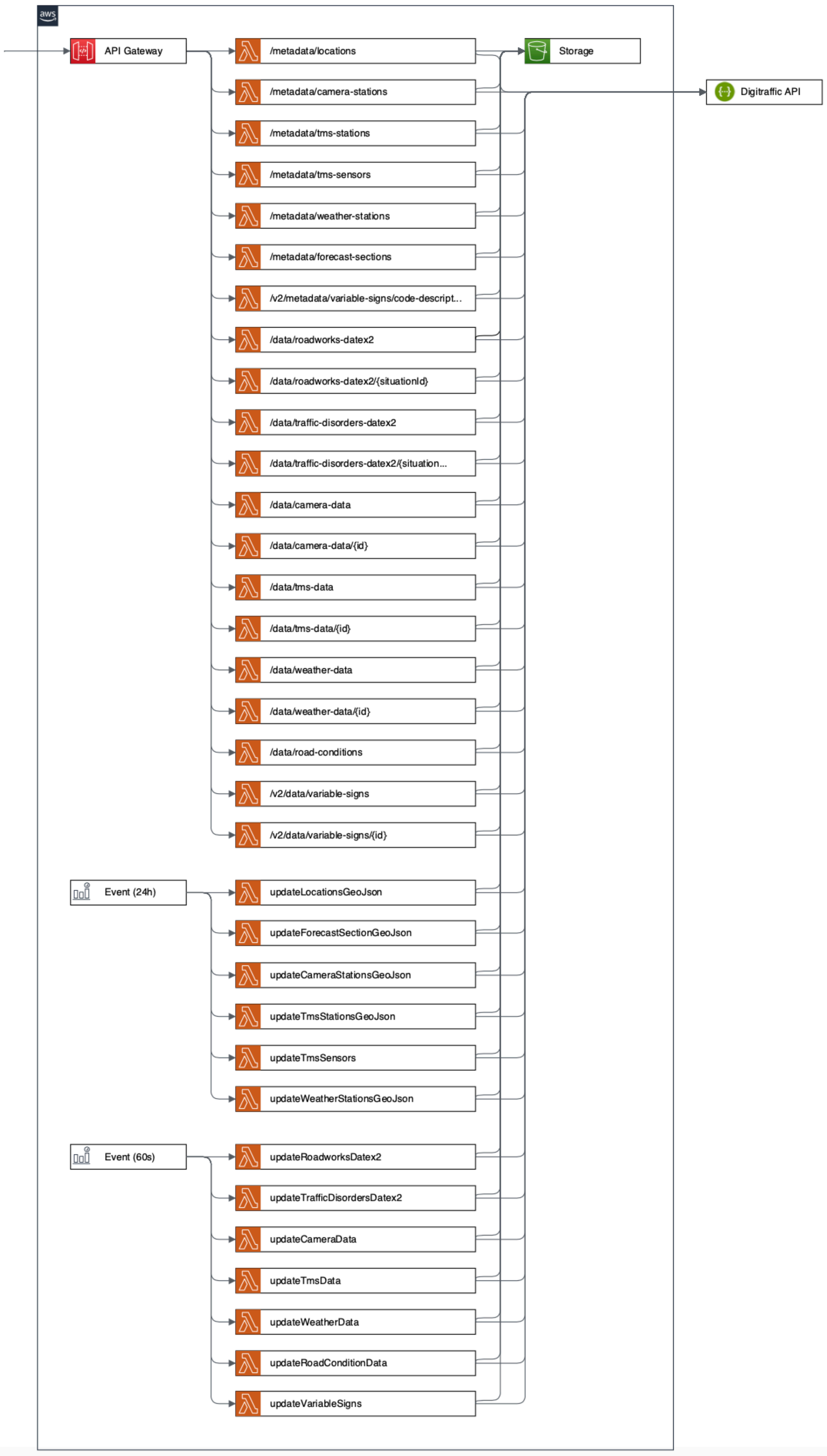


Figure 22: Digitraffic API architecture diagram

Technologies:

- AWS API Gateway
- AWS Lambda
- AWS CloudWatch
- AWS S3
- REST
- NodeJS

Fluency API

The Fluency API provides real-time traffic fluency analysis, i.e., traffic congestion information, for the major road and street network in the Oulu region. The information is available on the oulunliikenne.fi site as coloured geometry lines drawn on top of a map following the road network. The different colours represent the state of the traffic fluency at any given time as the illustration in Figure 23.

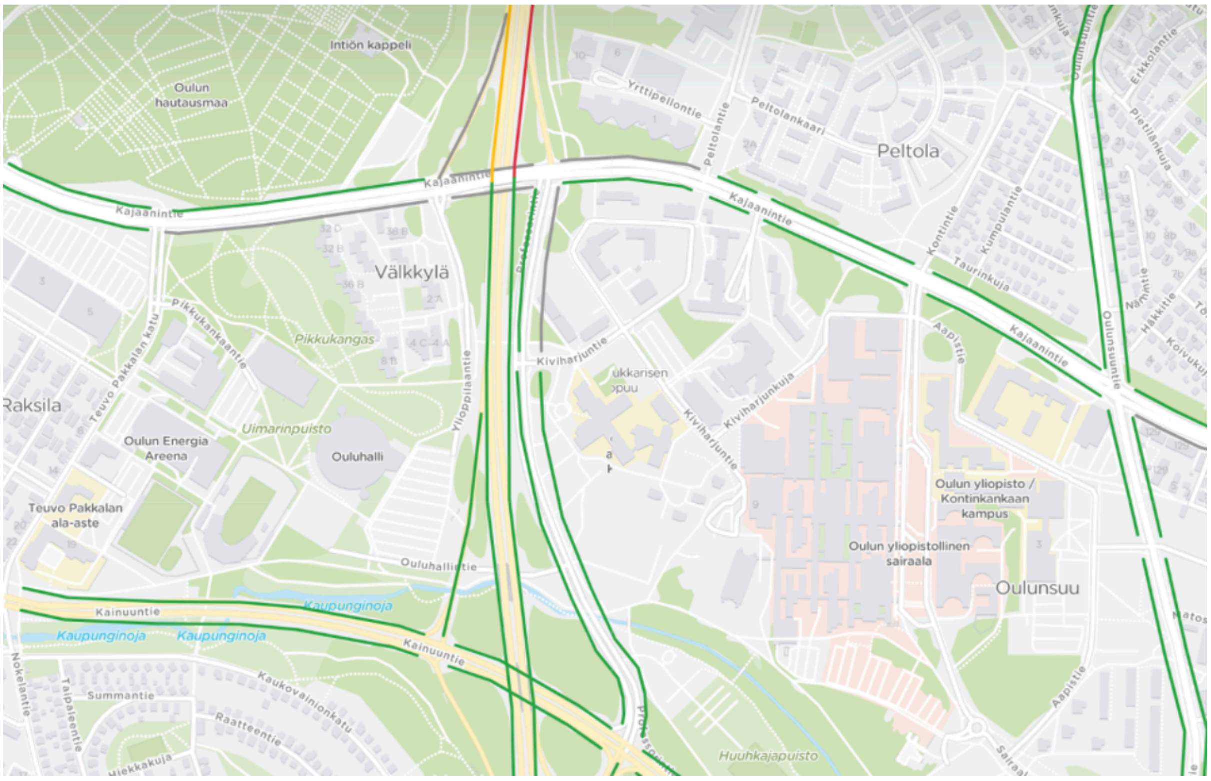


Figure 23: Traffic fluency lines in the Oulu region

The analysis is based on data from both the nationwide Traffic Monitoring System (TMS) maintained by Traffic Management Finland and the regional Traffic Performance Monitoring (TPM) system maintained by City of Oulu and ELY Centre of North Ostrobothnia.

The TMS data is made available for public use by Traffic Management Finland through their Digitraffic Road API. This API provides real-time information about average speed of the vehicles passing the different measurement points installed on the road network.

The TPM data is gathered by vehicle detectors installed in the traffic signals throughout the major road and street network in the Oulu region. This information includes Key Performance Indicator (KPI) values for control delay and maximum wait cycles in the queue of vehicles by the traffic signals. The TPM data is made publicly available on oulunliikenne.fi site by City of Oulu and ELY Centre of North Ostrobothnia through their TPM API.

The traffic fluency analysis, illustrated in Figure 24, is done based on mean values collected in 5-minute intervals from both data sources. The result of the analysis is a GeoJSON data structure including all the road network geometries with the current traffic fluency state.

This GeoJSON is then utilized by a specially designed map server to create map tile-based content for the oulunliikenne.fi service to display the information to the end user.

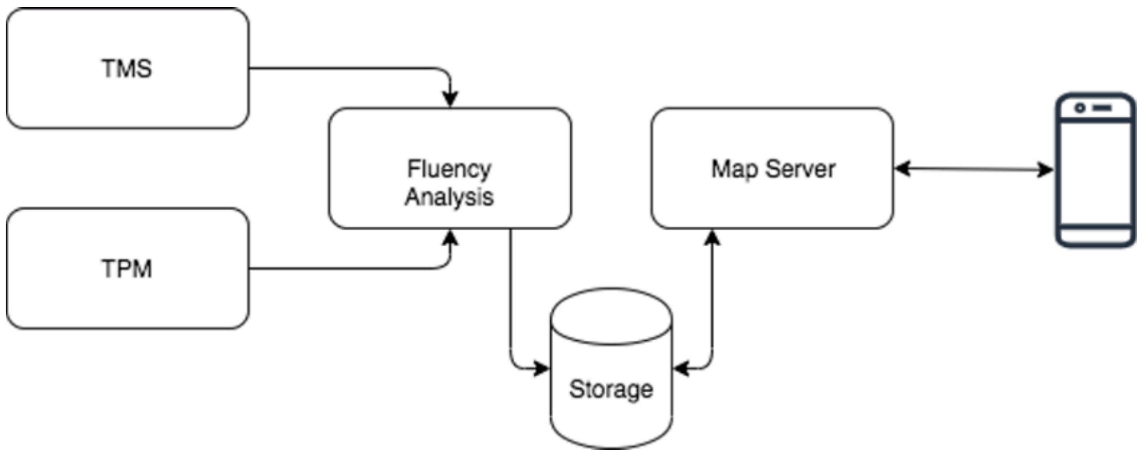


Figure 24: Traffic fluency calculation architecture diagram

Traffic fluency is analysed for a pre-defined set of road geometries in the Oulu region which are maintained by City of Oulu and ELY Centre of North Ostrobothnia. These geometries have additional meta data attached to them, defining how and what data source to use for the specific geometries. Those geometries that use TMS data for the fluency calculations have the TMS station id, name, direction and the speed limit of the road it represents. Those geometries that use TPM data for the fluency calculations have the TPM intersection and lane information attached for the primary lanes, including main turning lanes like off ramps where applicable.

TMS based traffic fluency calculation is based on the average speed that the TMS station for that part of the road is measuring and the speed limit. The traffic fluency has 4 states:

- Green - normal conditions - average speed is less or equal to 9km/h below speed limit
- Yellow - light traffic congestion - average speed is above or equal to 10km/h and less or equal to 24km/h below speed limit
- Red - heavy traffic congestion - average speed is above or equal to 25km/h below speed limit
- Grey - unknown conditions - average speed could not be determined

TPM based traffic fluency calculation is based on 2 Key Performance Indicator (KPI) values, control delay and maximum wait cycles, measured by the traffic signal devices. The traffic fluency has 4 states which are determined by the weaker of the two KPI values as defined here:

- Green - normal conditions - control delay is less than 60 seconds or max wait cycles is zero
- Yellow - light traffic congestion - control delay is larger than or equal to 60 seconds and smaller than or equal to 120 seconds or max wait cycles is 1
- Red - heavy traffic congestion - control delay is larger than 120 seconds or max wait cycles larger than 1

- Grey - unknown conditions - neither KPI could be determined

Technologies:

- AWS API Gateway
- AWS Lambda
- AWS CloudWatch
- AWS S3
- GraphQL
- REST
- NodeJS

TPM API

The Traffic Performance Monitoring API (TPM) provides a REST API for Key Performance Indicator (KPI) values gathered by traffic light detector devices in the Oulu region. The KPI values are maintained by Dynniq who provide an Apache Kafka server that the TPM API connects to.

KPIs are grouped by device and detectors, where a device represents an intersection and the detector a specific data gathering point. The device name "devName" usually starts with the city name followed by the intersection number, e.g., "OULU108". If one device is controlling multiple intersections, there may be multiple numbers. The detector name "detName" usually includes the direction of arrival, the distance in meters from the stop line and lane identification. These values are separated by an underscore "_", e.g., "D4_60_1M". Some KPIs, so called signal group KPIs, do not have a detector and are identified by the signal group name "sgName" instead.

The TPM Kafka consumer subscribes to a predefined list of KPI topics and receives values for these KPIs whenever they are published by the server. KPIs have different update cycles and all values will therefore not be updated as the same time. More information about the available KPI values and their update cycle can be found in the official documentation provided by Dynniq.

KPI values are written into an AWS Kinesis stream when received by the TPM Kafka consumer. Kinesis is used to control the throughput of the KPI values written into AWS DynamoDB, as the Kafka server can dump a large amount of KPI values at the same time to the subscribed topics which would cause a high write peak of values to the database in return. The Kinesis stream will flatten this peak and by doing so save on cost, as the DynamoDB writes cost quite a lot when doing tens of thousands of writes simultaneously. The write throughput is configured to be able to handle writing of all KPI values before the next batch of values come in.

The TPM Kafka consumer is implemented using Docker as it needs to keep running 24/7 listening to incoming KPI values. The consumer consists of a single Docker container running on AWS ECS Fargate.

The TPM REST API provides endpoints for fetching values by KPI as well as filtered by the KPI device and/or detector for more granularity. The API is documented using OpenAPI and can be viewed here.

The TPM REST API is implemented using the Serverless framework and it consists of the following:

- API Gateway
  - Public API also used by other Oulunliikenne APIs, e.g., the Eco-counters API and the Fluency API
- RESTful API endpoints using AWS Lambda
  - Get values by KPI name

    - Get values by KPI name and device name
    - Get values by KPI name, device name and detector name
- AWS Kinesis stream handler using AWS Lambda
    - Reads the Kinesis stream and write KPI values into AWS DynamoDB

The TPM API architecture is illustrated in Figure 25.

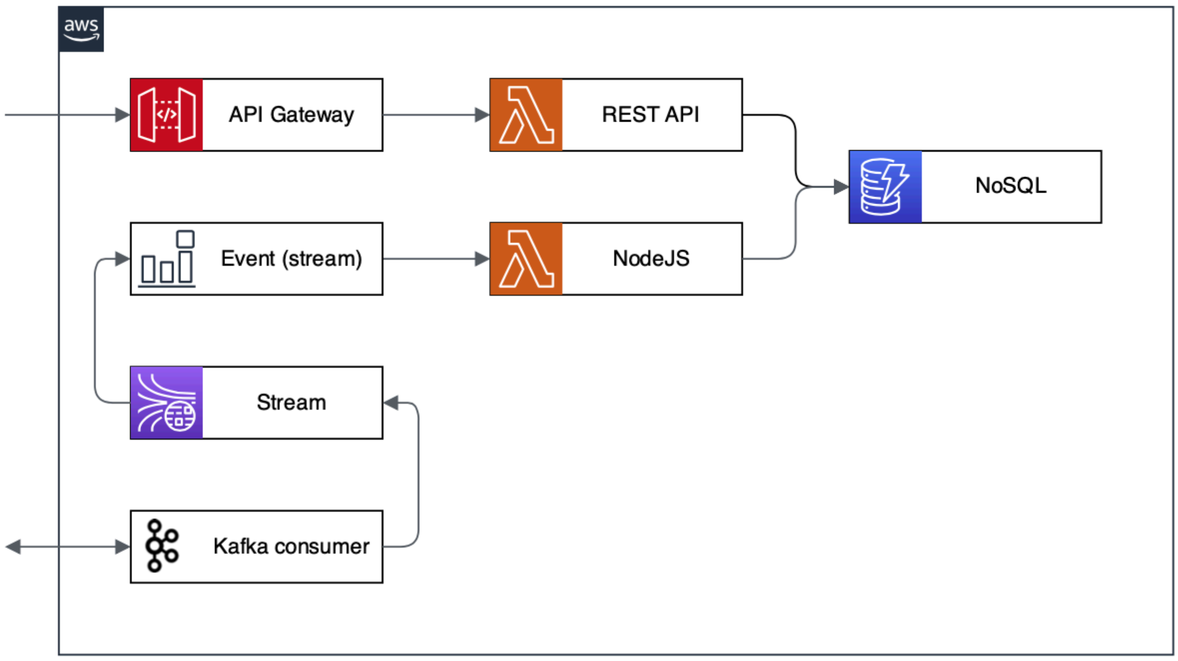


Figure 25: TPM API architecture diagram

Technologies:

- AWS API Gateway
- AWS Lambda
- AWS CloudWatch
- AWS Kinesis
- AWS DynamoDB
- AWS ECS
- Docker
- REST
- NodeJS

CDA API

The Content Delivery API (CDA) provides data from the Contentful Content Management System (CMS), and is utilized for navigation menu items in the oulunliikenne.fi service UI as seen in Figure 26 and Figure 27.

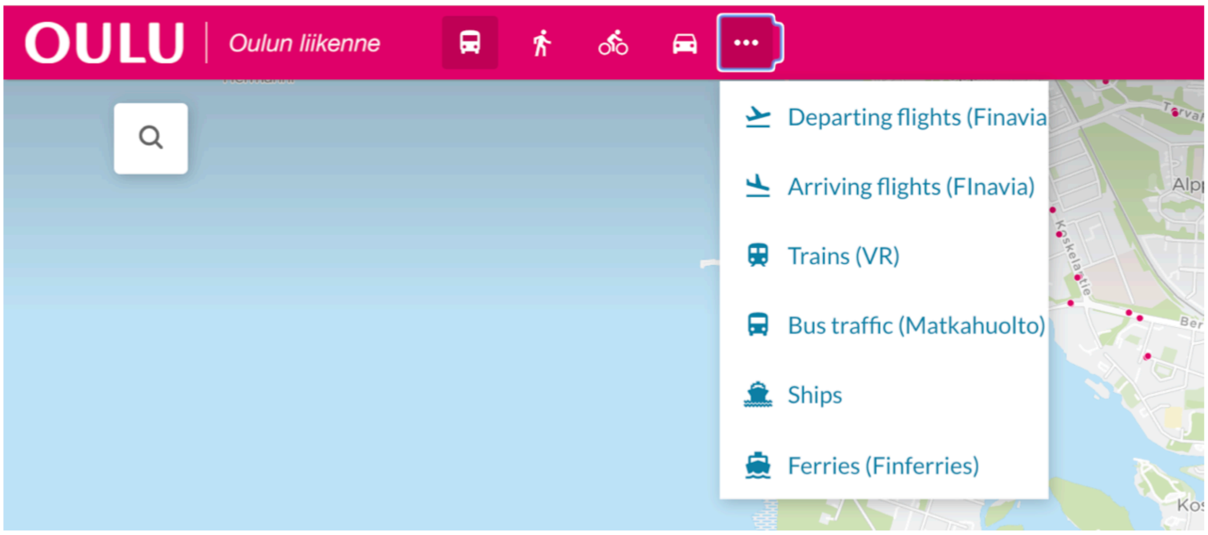

OULU
Oulun liikenne
Departing flights (Finavia
Arriving flights (FInavia)
Trains (VR)
Bus traffic (Matkahuolto)
Ships
Ferries (Finferries)

Figure 26: Menu items provided by the CDA API

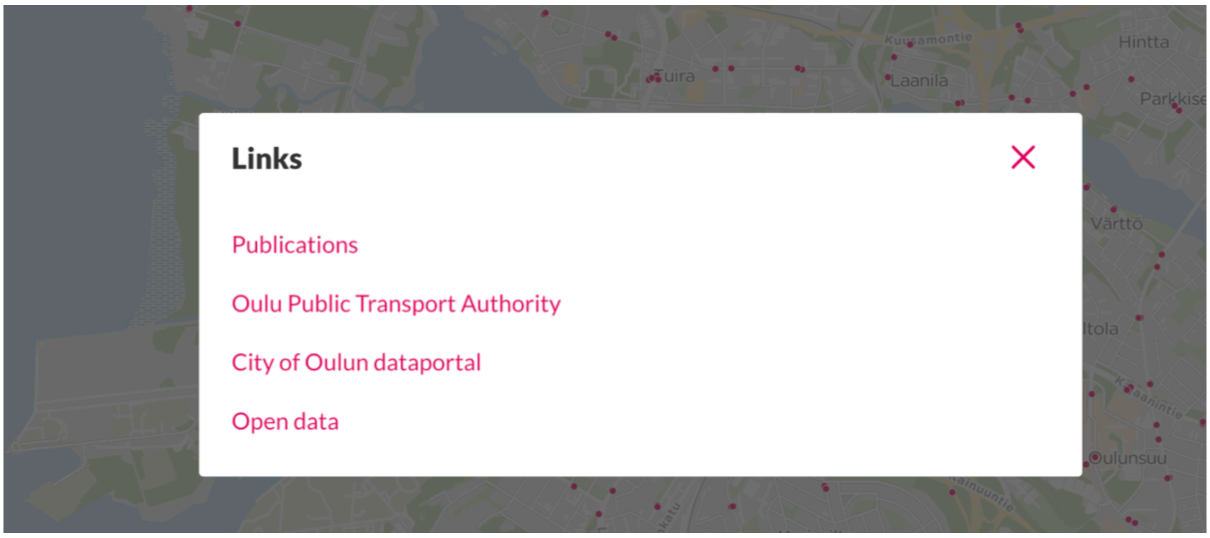


Figure 27: External links provided by the CDA API

The CDA API works as a proxy for the Contentful CDA API and converts the Contentful data models into a representation better suited for use in the oulunliikenne.fi service.

The menu items and link are managed in the Contentful admin interface and are accessed in real-time by the oulunliikenne.fi site, which means that any updates to the content will reflect to the site almost immediately.

The CDA API is built using the Serverless framework and consists of the following:

- GraphQL API

- Public API used by the API Gateway
- Queries for fetching menu items and links for the UI
- Schema and data models can be viewed here

The CDA API architecture is illustrated in Figure 28.

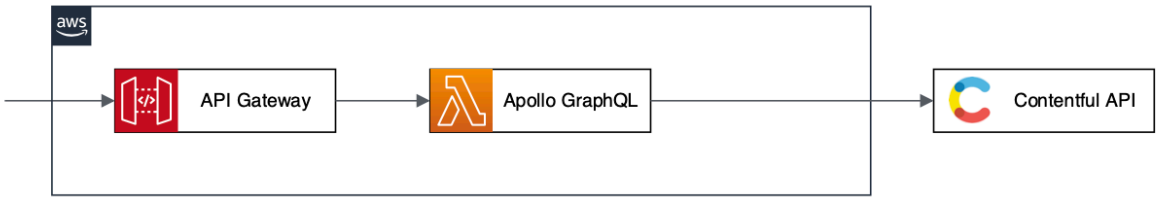


Figure 28: CDA API architecture diagram

Technologies:

- AWS API Gateway
- AWS Lambda
- AWS CloudWatch
- GraphQL
- REST
- NodeJS

CMA API

The Content Management API (CMA) provides a data management layer for the Contentful Content Management System (CMS), and is utilized by the traffic disruption/announcement tool used by Oulunliikenne.

The CMA API works as a proxy for the Contentful CMA API and also acts as a mapper between the Contentful data format and the format used by the oulunliikenne.fi tool. User authentication for the tool is implemented using the Contentful OAuth2 service which means that every user has to have an Contentful account in order to use the tool. The authentication request is made directly to Contentful and any request to the CMA API is required to include the authorization header which is verified by Contentful when trying to access and/or modify data.

The CMA API is built using the Serverless framework and consists of the following:

- GraphQL API
  - Protected API used by the Oulunliikenne tool
  - Queries for managing traffic disruptions/announcements

The CMA API architecture is illustrated in Figure 29.

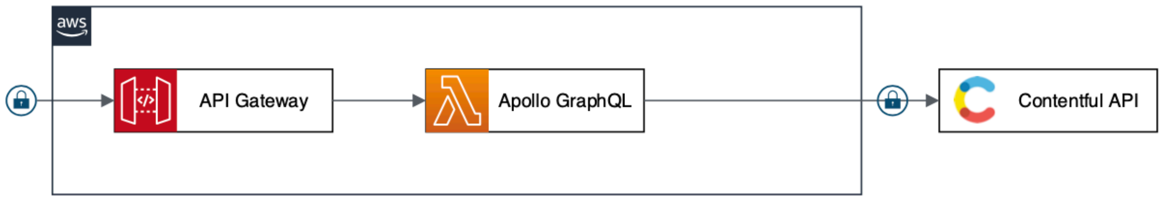


Figure 29: CMA API architecture diagram

Technologies:

- AWS API Gateway
- AWS Lambda
- AWS CloudWatch
- GraphQL
- REST
- NodeJS
- OAuth2

Maps API

The Maps API provides a unified way to produce vector tile protocol buffers of GeoJSON based data created and managed by other Oulunliikenne APIs. Vector tiles is a technique also utilized by OSM that allows packaging geographic data into smaller chunks, i.e., tiles. Digitransit utilizes this in the UI for drawing the map component. Protocol buffers is a data format developed by Google that optimizes data serialization for smaller and faster data transfer.

The Maps API uses a GeoJSON Vector Tile library to slice the source GeoJSON data provided by the Oulunliikenne APIs into vector tiles, then serializes the vector tiles into protocol buffers and servers the binary files through a AWS CloudFront CDN that caches the files for a configurable amount of time.

These vector tile protocol buffers (VTPBF) are used by the oulunliikenne.fi user interface to draw all kinds of elements, e.g., icons and geometry lines following the road network. All Oulunliikenne APIs that provide content to be drawn on the map in the user interface maintains a GeoJSON file in an AWS S3 bucket that the Maps API can read and convert into VTPBF files when requested.

The Maps API is built using the Serverless framework and consists of the following:

- AWS CloudFront CDN
    - Entry point for the API
    - Caches VTBPF files at the edges of the AWS network
    - If requested file not found in cache, it delegates the request to a Lambda at Edge function
- AWS Lambda at Edge function
    - Generates the VTPBF files
    - Stores them in an S3 bucket for caching
- AWS S3 bucket
    - Storage for generated VTPBF files
    - Acts as a second cache layer if the file is no longer cached in the CDN but the source data hasn’t changed

The Maps API architecture is illustrated in Figure 30.

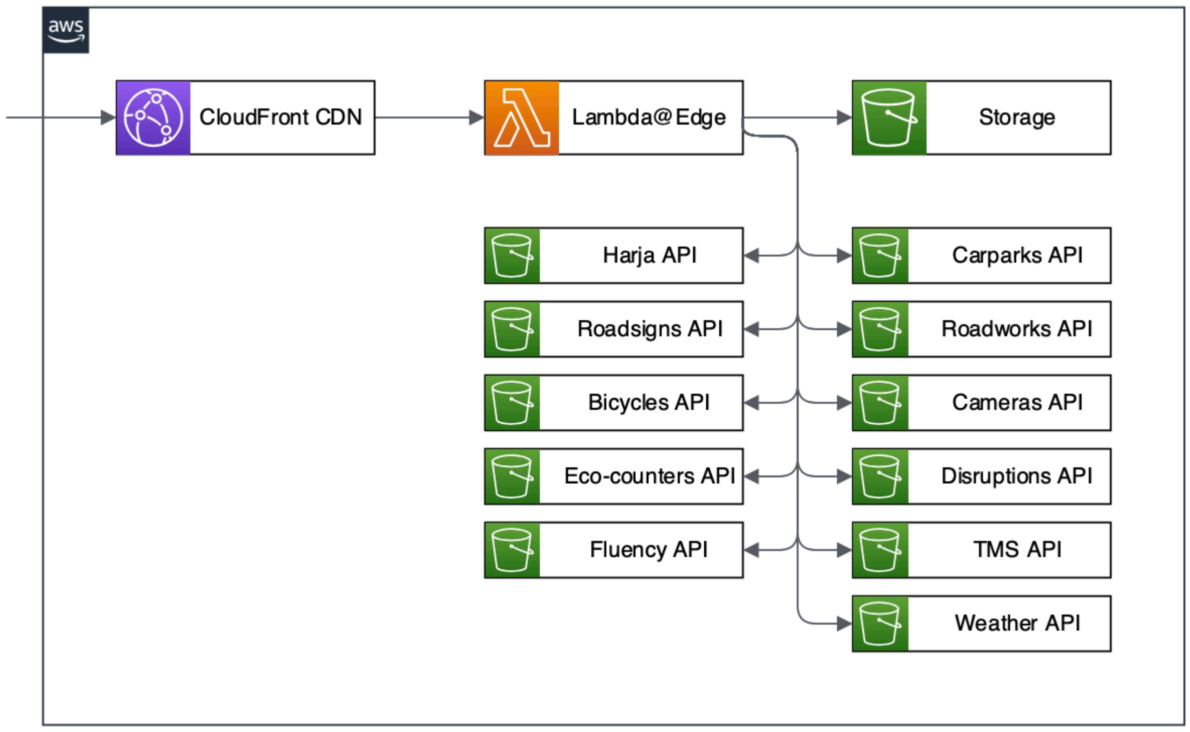


Figure 30: Maps API architecture diagram

Technologies:

- AWS CloudFront
- AWS Lambda at Edge
- AWS CloudWatch
- AWS S3
- NodeJS

GTFS API

The General Transit Feed Specification API (GTFS) provides downloadable assets for public transportation schedules and associated geographic information.

The GTFS assets are updated by uploading a zip-file to an AWS S3 bucket which triggers an event handled by an AWS Lambda function that splits the data into 2 separate new zip-files that are stored in another S3 bucket which serves as the backend for an AWS CloudFront CDN.

The GTFS API is built using the Serverless framework and consists of the following:

- AWS CloudFront CDN
    - Entry point for downloading the assets
    - Caches them until the cache is cleared by a new version of the assets being uploaded
- AWS Lambda function
    - Triggered by a new GTFS zip-file upload to S3 o Splits the file into 2 separate assets
    - Uploads them to another S3 bucket
    - Clears the CDN cache
- AWS S3 buckets
    - One for uploading new GTFS zip-files (private)
    - One for storing the downloadable asset zip-files through the CDN (public)

The GTFS API architecture is illustrated in Figure 31.

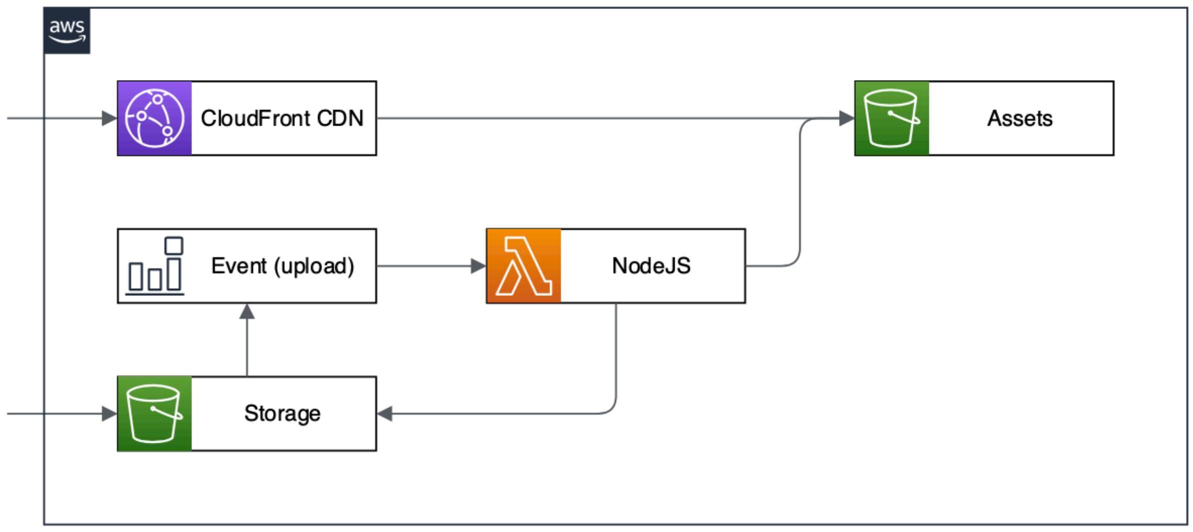


Figure 31: GTFS API architecture diagram

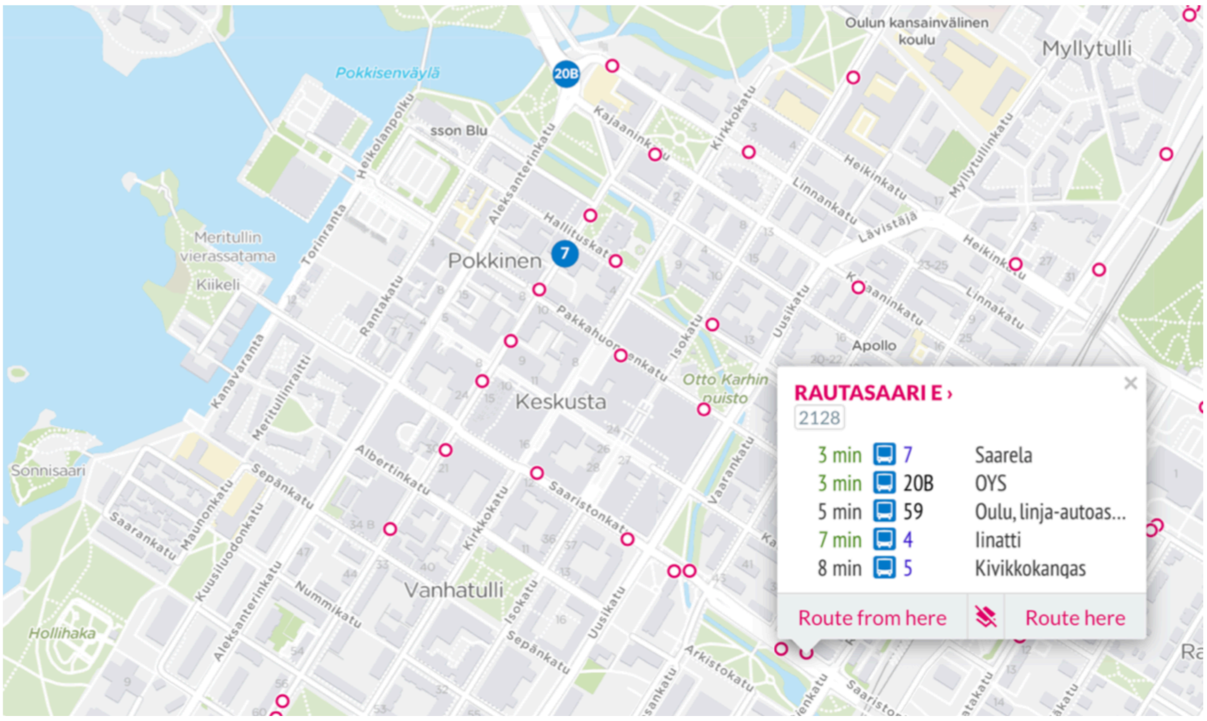
Technologies:

- AWS CloudFront
- AWS Lambda

- AWS CloudWatch
- AWS S3
- NodeJS

HFP API

The High Frequency Positioning API (HFP) provides real-time location data of public transportation vehicles in the Oulu region as seen in Figure 32.



RAUTASAARI E ›
2128

3 min	7	Saarela
3 min	20B	OYS
5 min	59	Oulu, linja-autoas...
7 min	4	Iinatti
8 min	5	Kivikkokangas

Figure 32: Real-time positions of public transportation vehicles

The HFP API gets its information from FARA by polling their APIs at an interval of once every 2 seconds. The data is then pushed to an AWS IoT MQTT broker on a vehicle number and route ID basis, providing the option for any client to listen to only specific vehicles or routes if needed. This implementation is similar to that of Digitransit, which only provides location data for the Helsinki region at the moment.

The AWS IoT service provides a message broker implementation based on the MQTT connectivity protocol, for lightweight publish/subscribe functionality, that is able to serve all users in real-time. The client can subscribe to all vehicle locations, specific vehicle numbers or routes.

The user interface communicates directly with the IoT service and does not go through the proxy API. The IoT service requires the client to be authenticated and have the necessary access to subscribe to and receive location events. As all oulunliikenne.fi users are anonymous, we utilize the AWS Cognito service behind the scenes to create an anonymous user with the required permissions for the IoT services. This authentication is necessary to prevent anyone from also publishing new events to the IoT service, which is something only the backend services are allowed to do.

The HFP API is built using the Serverless framework and consists of the following:

- AWS Lambda function
    - Triggered every minute
    - Polls FARA APIs for location data and pushes it AWS IoT
    - Runs for approximately 1 minute and does as many as 30 polling to FARA, after which it shuts down and a new Lambda function is triggered by CloudWatch. This is done as Lambda functions have a maximum runtime of 5 minutes, after which they are automatically shut down. This function should be converted into a Docker container and be run on AWS ECS instead as it would be better suited for handling this kind of continuous polling jobs.
- AWS IoT MQTT broker
    - Delivers real-time public transportation vehicle locations to users
- AWS Cognito
    - Provides anonymous user authentication for AWS IoT

The HFP API architecture is illustrated in Figure 33.

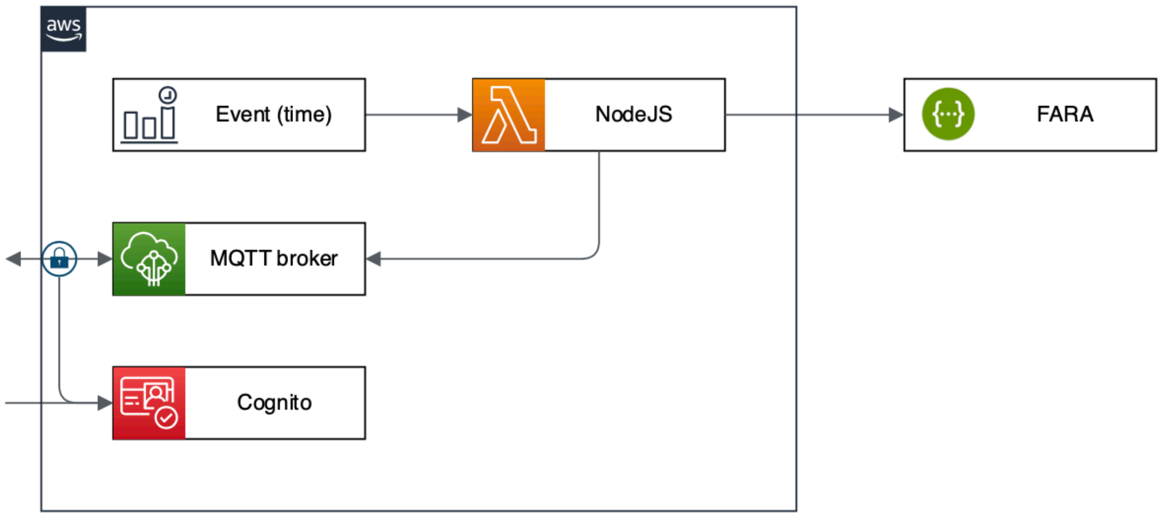


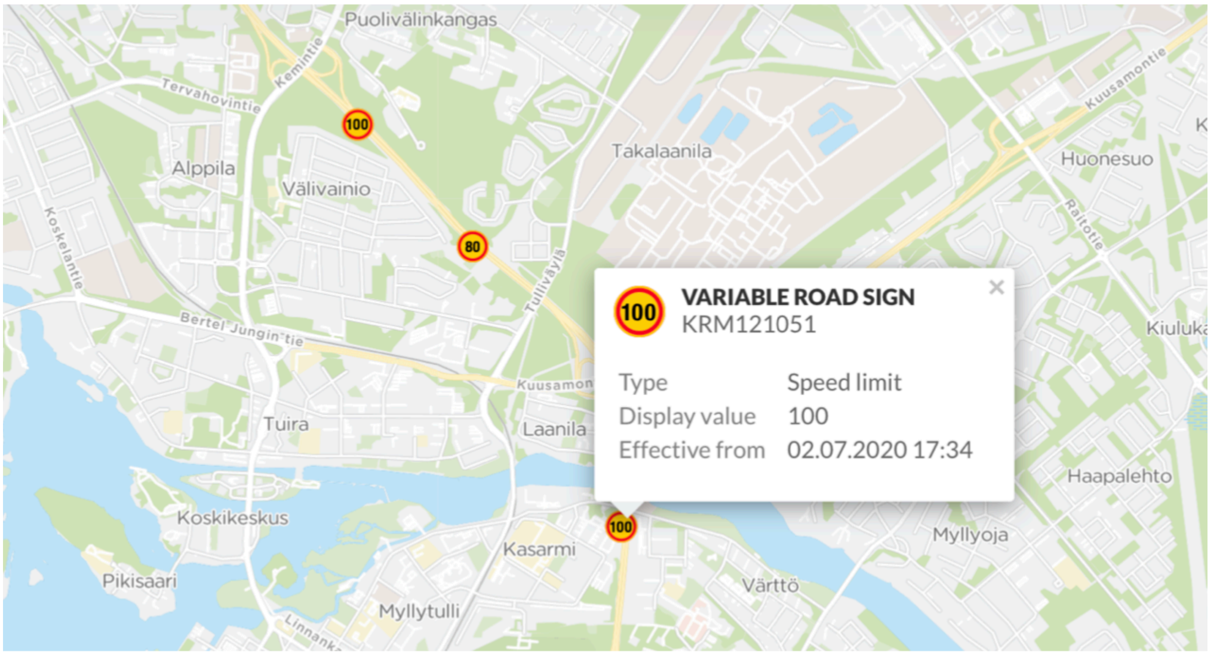
Figure 33: HFP API architecture diagram

Technologies:

- AWS Lambda
- AWS CloudWatch
- AWS IoT
- MQTT
- REST
- NodeJS

Road-signs API

The Road-signs API provides information about variable road signs found in the Oulu regions as seen in Figure 34.



VARIABLE ROAD SIGN
KRM121051

Type	Speed limit
Display value	100
Effective from	02.07.2020 17:34

Figure 34: Variable road signs in the Oulu region

The Road-signs API fetches the information from national Digitraffic service operated by Traffic Management Finland. The Roads-signs API fetches the details through the oulunliikenne.fi Digitraffic API which filters the found data in the national Digitraffic service to include only ones in the Oulu region.

There are two types of variables road signs: speed limits and warning sign. All signs have a unique icon displayed on the map that indicates either the speed limit at that section of the road, or what kind of warning is found ahead.

The Road-signs API updates a GeoJSON file for the sign locations once every minute from the Digitraffic API, which in turn caches the data found in the Oulu region for the same amount of time. This is done both to prevent high traffic peaks to the national Digitraffic service and to improve the performance of loading the data in the service, as Digitraffic always returns all variable road signs nationwide.

Warning signs are not currently used as a part of the journey planning, e.g., to find alternative routes where some hazard would affect the travel time significantly.

The Road-signs API is built using the Serverless framework and consists of the following:

- GraphQL API
  - Public API used by the API Gateway

        - Queries for fetching all road signs as well as individually from the Digitraffic REST API
        - Schema and data models can be viewed here
- Scheduled events
    - Create GeoJSON of variable road signs
        - Timed event run every minute
        - Generates a GeoJSON structure for the locations to be used by the Maps API when UI renders them on the map

The Road-signs API architecture is illustrated in Figure 35.

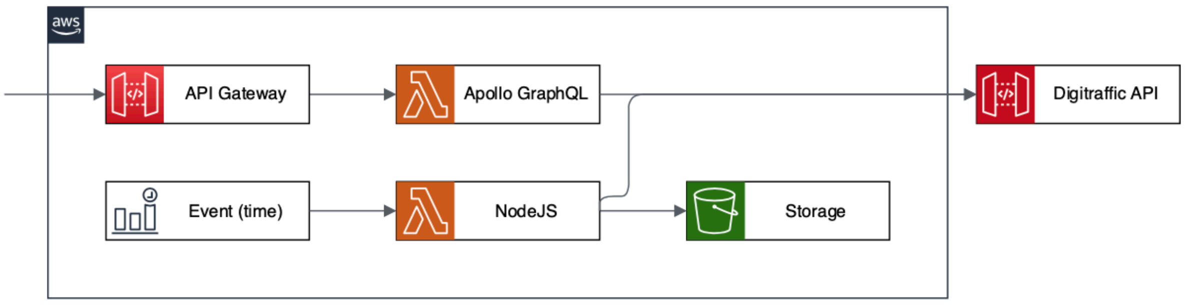


Figure 35: Road-signs API architecture diagram

Technologies:

- AWS API Gateway
- AWS Lambda
- AWS CloudWatch
- AWS S3
- GraphQL
- REST
- NodeJS

Infrastructure

The entire oulunliikenne.fi service is hosted on AWS using many of the different services that AWS provides. The main architectural principle of the service is to have separate independent microservices for all the feature areas identified in the service. This architecture allows for each component in the service to be developed and maintained independently which reduces costs, minimizes breaking changes and simplifies updating the service reducing the time it takes to bring new features to the service.

Another architectural design choice is to favour so called serverless or fully managed services that do not require managing any servers, physical or virtual ones, which greatly reduces the amount of maintenance tasks that would otherwise have to be performed in order to keep the servers up to date with recent security patches etc. AWS provides many services that follow this principle e.g., Lambda, DynamoDB, ECS Fargate.

The service components also use the same technologies and services as much as possible to keep a unified architecture design while maintaining the flexibility of using something purpose built when necessary e.g., AWS IoT Core for MQTT brokers and AWS Kinesis for data streams.

Infrastructure changes are managed on two (2) levels, globally and individually for each service component. The global level is for service wide infrastructure components like S3 buckets, VPC network configurations, CloudWatch dashboards etc. While the service component level is used to manage most of the infrastructure needed to each component. Both levels use the infrastructure as code (IaC) principle, meaning almost everything in AWS has been created and is managed using either Terraform (global level) or CloudFormation (service level). Very few resources have been created manually and they are documented in the infrastructure repository. The IaC principle also helps keeping infrastructure changes automatically documented and version controlled in the repositories.

Terraform is used for the global infrastructure components because it’s easier to understand and use than CloudFormation which is used on the service level only because of the Serverless framework which uses it under the hood. It also helps separating the two levels by using different tools, however, the idea of them both is very similar; to define resources and their configuration as logical blocks of code that are read by command line tools which sync the state to AWS via the AWS APIs.

Environments

There are two (2) separate environments of the service, one for testing new features and performing quality assurance tasks and one for production. Both environments are hosted on the same AWS account but separated in terms of resources and networking.

The testing environment user interface can be accessed using the following link https://next-dev.oulunliikenne.fi/. There are currently no restrictions on who can access this page as it contains only publicly available data.

The production environment user interface can be accessed using the following link https://oulunliikenne.fi/.

Both environments have the same features, although the testing environment can be a bit ahead in the development cycle at times, but some features are disabled in the testing environment or in some cases not collecting or receiving data.
Disabled features currently only include parts the TPM API as the Kafka consumer is quite resource heavy and would end up costing too much to keep it running at all times. The feature can be enabled when needed. Naturally all other components depending on data provided by the TPM API will not have up to date data and may display old or no information in the user interface.
There are also features that collect or receive no data in the testing environment. These currently only include parts of the Cameras API and Harja API. The traffic cameras that get their images from Viria Oy are currently not receiving data as Viria only has the capability to send the images to one location. The same is true for the Harja API which currently does not receive data in the testing environment, but the maintenance tasks are still available because they are streamed from the production database into the testing database to be able to test new features related to the maintenance tasks.

The environments share two (2) things, location information of both public transport vehicles and maintenance vehicles. Both of these are managed through the AWS IoT Core service two which both environments connect. Technically it is possible to separate the functionality for these as well, but for the moment the features for updating the vehicle locations are enabled only in production and the testing environment is configured to connect to production when displaying the data in the user interface.

Other differences between the environments are related to the infrastructure resources and update intervals for features that fetches new data periodically. Resources are configured to use less CPU and RAM in the testing environment and the periodic data retrieval interval is 15 minutes instead of the 1 minute that is configured for production.

In addition, the testing environment is partly shut down during nights and weekends in order to save on running costs when nobody is using it. The user interface is completely shut down and not accessible during this time and no new data is fetched 3rd party sources. All APIs are functional but may provide old and outdated information.

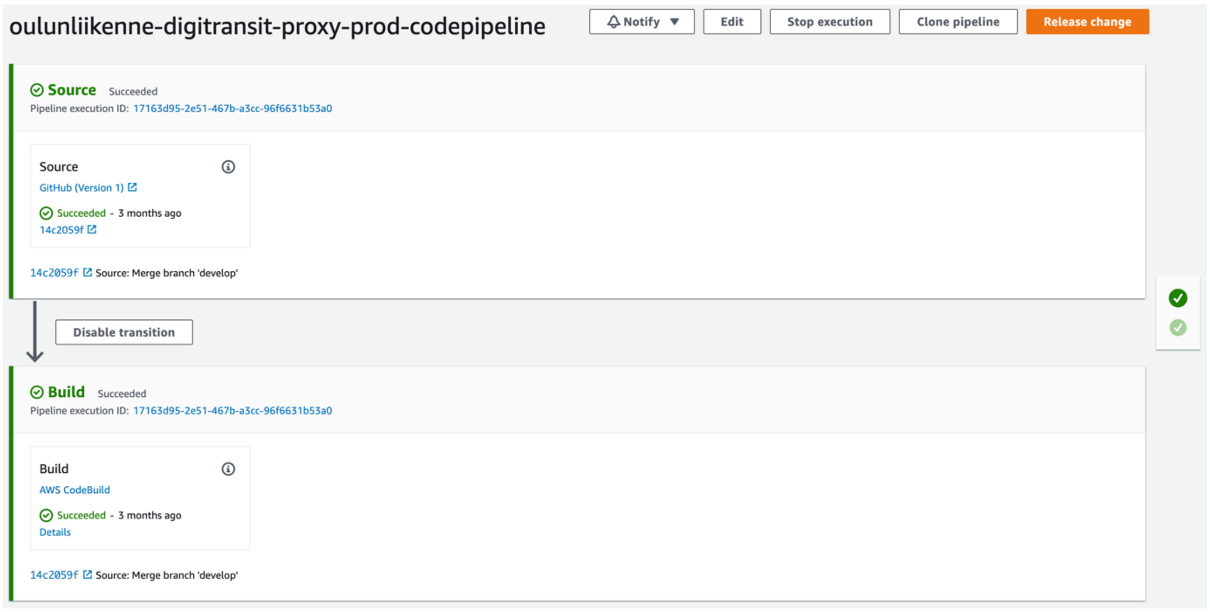
Deployment

Deployment of the service is handled by automated Continuous Integration / Deployment (CI/CD) pipelines that are connected to the Github repositories. As the service is split up into many smaller components, each of these components has its own CI/CD pipeline that operates on its own and is also connected to the components own Github repository. This way, making updates to only 1 of the components doesn't require the entire service to be redeployed saving time and money while minimizing the risk of breaking changes.

While all components have their own CI/CD pipeline, they are very similar due to the uniform technology choices of the individual components. There are effectively two (2) types of pipelines: 1) Serverless pipelines that deploy AWS Lambda functions behind API Gateways 2) Docker pipelines that deploy containerized components in AWS ECS Fargate clusters.

Most components use the Serverless scheme as they consist of APIs and event triggered tasks, which fits well into the AWS Lambda architecture. Docker is used only for specific use cases where some technical requirements exist that fits better into the AWS ECS architecture. These limitations are e.g., long running tasks that AWS Lambda does not support.

All CI/CD pipelines are built using the AWS CodePipeline service, which is a fully managed service for automating build and deployment workflows. These pipelines are configured per component in Terraform and deployed separately for environments in AWS. The pipelines follow the same branching scheme for the Github repositories, namely the "develop" branch is deployed to the testing environment and the "master" branch to the production environment. All commits to these branches are automatically processed by the pipelines.


oulunliikenne-digitransit-proxy-prod-codepipeline

Source Succeeded
Pipeline execution ID: 17163d95-2e51-467b-a3cc-96f6631b53a0

Source
GitHub (Version 1)
Succeeded - 3 months ago
14c2059f

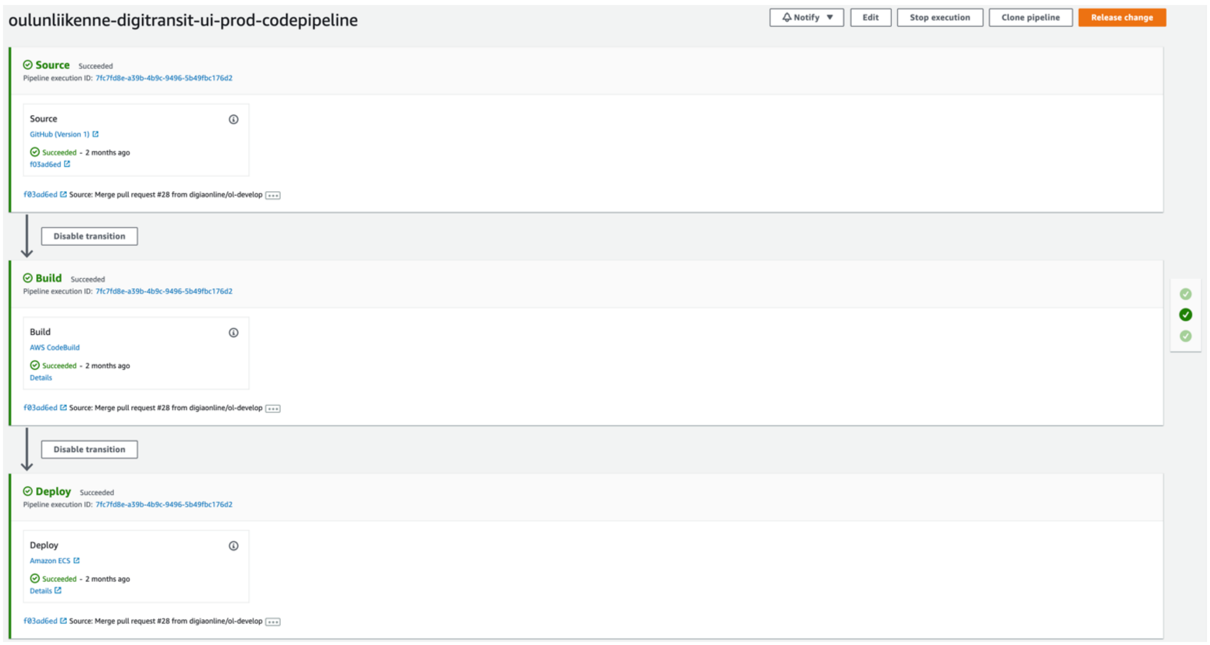
14c2059f Source: Merge branch 'develop'

Build Succeeded
Pipeline execution ID: 17163d95-2e51-467b-a3cc-96f6631b53a0

Build
AWS CodeBuild
Succeeded - 3 months ago
Details

14c2059f Source: Merge branch 'develop'

Figure 36: Serverless CI/CD pipeline visualisation


oulunliikenne-digitransit-ui-prod-codepipeline

Source Succeeded
Pipeline execution ID: 7fc7fd8e-a39b-4b9c-9496-5b49fbc176d2

Source
GitHub (Version 1)
Succeeded - 2 months ago
f03ad6ed

f03ad6ed Source: Merge pull request #28 from digiaonline/ol-develop

Build Succeeded
Pipeline execution ID: 7fc7fd8e-a39b-4b9c-9496-5b49fbc176d2

Build
AWS CodeBuild
Succeeded - 2 months ago
Details

f03ad6ed Source: Merge pull request #28 from digiaonline/ol-develop

Deploy Succeeded
Pipeline execution ID: 7fc7fd8e-a39b-4b9c-9496-5b49fbc176d2

Deploy
Amazon ECS
Succeeded - 2 months ago
Details

f03ad6ed Source: Merge pull request #28 from digiaonline/ol-develop

Figure 37: Docker CI/CD pipeline visualisation