## Preface

This project is an integration challenge, not a programming challenge. The focus of this project is on how the numerous 3rd party modules we selected such as OSRM, Spark and MongoDB interact with each other; the code we wrote is mostly just glue to hold them together. Ultimately, our system’s architecture was a higher priority than the actual programming.

This report analyzes the following modules:

* Routetagger
* Web application
* Mobile application
* Spark jobs
* Other scripts

## Routetagger

Routetagger is an application that allows users to assign street segments to ITSOS sensors. It is for internal use only and development has been suspended as the team is using a different telemetry dataset.

**Formatting**: The project does not have a linter enforcing code style in the build script. However, standard JavaScript and TypeScript formatting styles are followed.

**Architecture**: Routetagger follows the ideal architecture conventions of an Electron application. Code is divided into common, main, and renderer directories representing the different electron processes, with the majority of the code in the renderer process. This process is further divided into a react/redux architecture with a good separation of view, storage, and redux middleware layers.

**Best Practices**: Aside from a hard-coded OSRM address, this program avoids hard coding values. The need for comments is minimized by self-documenting code. When appropriate, framework features (such as the Electron file picker) are used. No magic numbers.

**Maintainability**: The code is readable and testable; however no logging is done for debugging and it is not very configurable. The address of the OSRM server it depends on is defined in code and must be changed to change OSRM providers. There are no unit or integration tests.

**Reusability**: Good reusability; little to no code repetition.

**Security:** Lacking. The program has two vulnerabilities:

1. Routetagger opens a file selected by a user, but there is no validation to ensure the file is in the correct format when it is loaded. An invalid file causes undefined behaviour.
2. Routetagger depends on an OSRM instance, but does not validate that HTTP responses are in the correct format. Making a request to an incompatible version of OSRM (if such a thing existed) that worked on the surface would cause undefined behaviour.

**Scalability:** The program is not very scalable. It would be very slow opening a sensor file of millions of sensors. It loads the entire file into memory and attempts to display all sensors on the screen regardless of the viewport position.

**Usability**: The program does not have a script to publish it; the source code must be downloaded and it can only be run in debug mode. The interface was also not designed for use by anyone other than the developer who wrote it; it was not designed for the public.

## Web Application

**Security:** There is a known denial of service vulnerability in one of the web application’s dependencies. However, at the time of writing this report no patches are available. The website is secured with HTTPS and signed with letsencrypt. There is no login or user account functionality on the site. Geocoding is provided through Bing and thus security of that is dependant on Bing. Specific location data is handled in browser and is not collected. Routing data may be stored in OSRM log files as well as IP traffic to and from the servers. In the future this information should either be encrypted or dumped, the current implementation simply stores this data in plain text files.

**Usability**: The website is in a usable state, aside from the mobile version of the site it functions properly in chromium based browsers as well as firefox. The browser that seems to exhibit the most problems is Microsoft Edge. The site loaded on a mobile phone exhibits multiple interface issues but is still useable. It is recommended to use the app if on mobile.

## Mobile Applications

**SCOTT FILL THIS IN**

**Usability**: SCOTT PLEASE SAY IT IS USABLE

## Spark jobs

There are three spark jobs: the OSRM loader, stream job, and batch job.

**Formatting:** A linter is not used to enforce code style; however standard Scala formatting is used in the code.

**Architecture**: Mediocre. The OSRM loader and batch job each have only a single Scala source file; this file contains startup and data transformation code. The stream job main file is the same, though the live speeds database update component is split out into separate files in an object-oriented manner. One file is very badly named: the main file of the stream job is called SparkPI.

**Best Practices**: Hard coding is entirely avoided; everything is configured through the program’s command line arguments. Self-documenting variable names minimize the need for comments. Spark, Kafka, and MongoDB frameworks are used appropriately. No magic numbers.

**Maintainability**: Somewhat maintainable. No logging other than basic Spark logging (which the team does not have enough experience in Spark to understand) is done. In the stream job, the database update module is thoroughly covered by integration tests.

**Scalability**: Extremely scalable by using Apache Spark. This was the priority of these programs.

**Reusability**: Some code is duplicated between the spark and batch jobs. However, the team does not have enough experience with Scala to create a ‘common’ Scala package and a build pipeline that compiles the package as a build artifact consumed by the spark and batch jobs.

**Security**: Lacking. Two vulnerabilities:

1. All of the spark jobs are configured with command line arguments, however there is no validation on command line argument order or format. A command line application framework should be used to allow more robust configuration with named arguments, such as ‘-argument value’.
2. The stream and batch jobs assume the Kafka topic they read from contains only valid data and does not check for invalid data. As the Kafka topic is intended to contain raw telemetry, consumers of this topic should have to check for bad data before proceeding.

## Other Scripts

Various other shell, python, and node scripts have been developed. This includes several one-time data extraction scripts, one-time data processing scripts, a Kafka mock data producer, and OSRM refresh and reload script.

**Formatting**: Python and Node scripts follow the language formatting guidelines; shell scripts do not follow any particular formatting style but are still readable.

**Architecture**: As scripts, most do everything in one file; however the node script separates code that integrates with reading shapefiles or making requests to OSRM into separate files. The OSRM loader script is very poorly architected; it calls a shell script on another computer over SSH and copies a file back using SCP in a loop to update OSRM, among other things. This script should be replaced with a properly developed OSRM loading program, as described in the after action review report.

**Best Practices**: Mixed. Some values are hard-coded; some values are configured by command-line arguments. Some scripts were developed following a UNIX pipe mentality reading from standard in and printing to standard out; however some scripts simply manipulate files with file IO interfaces. Additionally, in many cases CSVs are written manually without a CSV library. No magic numbers.

**Reusability**: Good reusability; little to no code repetition.

**Security**: Most user input is not validated.