SYSTEM ARCHITECTURE FOR AI-ENABLED CORRIDOR MANAGEMENT

By

Caleb Michael Van Geffen

Dissertation

Submitted to the Faculty of the

School of Engineering of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

MASTER OF SCIENCE

in

Civil and Infrastructure Systems Engineering

December 17, 2022

Nashville, Tennessee

Approved:

Daniel Work, Ph.D.

Jonathan Sprinkle, Ph.D.

I would like to dedicate this thesis to one of my lifelong friends, Gunnar Schultz, who sadly passed away on November 20, 2021. Gunnar was a graduate of the United States Air Force Academy with a degree in Physics. He was one of the main reasons I decided to pursue education at Vanderbilt University and always pushed me to be a better version of myself.

ACKNOWLEDGMENTS

I would like to thank Dan Work and Will Barbour for their constant mentorship and guidance throughout my career as a student at Vanderbilt I would also like to acknowledge that this work is supported from a grant from the U.S. Department of Transportation Grant Number 693JJ22140000Z44ATNREG3202.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

1.1 Motivation

Intelligent Transportation Systems (ITS) have been around for decades. Literature has implored the need for integrating novel systems with established systems specifically in the field of ITS seen as early as Meier et al. (2005). Recent investment from the U.S. government with the Infrastructure Investment and Jobs Act (IIJA) has allowed for nationwide upgrades of ITS systems. NHTSA has reported in NHTSA (2022) that roadway deaths halfway through 2022 have increased to 20,175 from 20,070 halfway through 2021. Not only do incidents impact safety, but also congestion. Congestion on United States roadways are caused by incidents in roughly 30% of cases via INRIX (2022).

Integrated Corridor Management (ICM) aims to boost safety and mobility. The FHWA (Federal Highway Administration) defines ICM as "the coordination of individual network operations between adjacent facilities that creates an interconnected system capable of cross-network travel management" in FHWA (2020). This strategy can leverage new control strategies in active traffic management (ATM) such as Lane Control System and Variable Speed Limit technologies to achieve its goals. LCS and VSL demonstrate the potential for improvements in mobility and safety by means of dynamically changing signaling based on roadway conditions. An example showing LCS and VSL is depicted in Figure 1.1. LCS provides current lane blockage information and VSL provides current speed limit values. These technologies provide a means by which artificial intelligence algorithms can take action by changing board configurations. There is plenty of literature displaying advanced traffic management on simulation in Yang et al. (2000), Jayakrishnan et al. (2001), Hawas (2002), de Souza and Villas (2016), but it is rarely demonstrated in actual real-time systems.

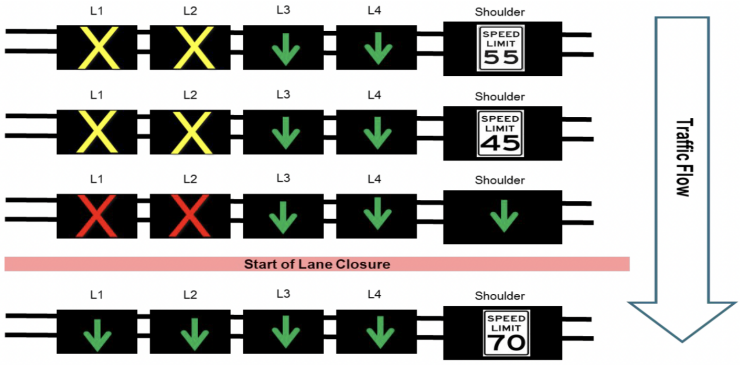


Figure 1.1: Example of LCS and VSL deployed on the freeway.

Traffic Management Centers (TMCs) have existing systems to manage incident response on the freeway through the use of human operators. Zhang et al. (2008) describes the desire of the FHWA for decision support systems to reside in TMCs. These systems are very well-established and introducing artificial intelligence components in the systems themselves could prove difficult with the amount of offline learning necessary. However, building a companion system to communicate with the existing system would be promising for the use of AI algorithms for incident response.

1.2 Problem statement

The primary concern of this work is to design a system architecture capable of supporting artificial intelligence enabled corridor management which will be referred to as an AI-DSS (Artificial Intelligence Decision Support System). In this context, the research is not concerned with the creation of the AI algorithm but rather the overall system to support its function.

1.3 Contribution statement

To my knowledge, this is the first study to propose a system architecture for AI-enabled corridor management, while also considering real freeway data and not that of simulation. It also provides a structure that existing systems being used by Traffic Management Centers can integrate into their software stack considering a human-in-the-loop (HITL). A summary of the contributions of this thesis are as follows:

1. A system architecture is designed for AI-enabled corridor management with a HITL.
2. This system architecture can be integrated with existing systems being used by TMCs through use of an API for TCP/IP connections.
3. An actual implementation of the architecture is detailed on Interstate 24 Smart Corridor in Tennessee as part of the TDOT ATCMTD project. The system passed User Acceptance Testing in November 2022.

1.4 Thesis organization

The remainder of this thesis is organized as follows. In Chapter 2, literature related to this research is discussed. Chapter 3 details an overview of the system architecture. In Chapter 4, a real-life implementation of the architecture on Interstate 24 in Nashville, Tennessee is shown. Chapter 5 concludes the thesis along with an overview of possibilities for future areas of research.

CHAPTER 2

Related Work

2.1 VSL and LCS algorithms

Literature review upon VSL and LCS algorithms was important to motivate architecture decisions for an ICM AI-DSS. The feasibility of these systems to improve safety and mobility have been evaluated in multiple studies. LCS is comparatively much more unexplored, but has still shown signs of potential for safety improvements.

Variable speed limit has been proven to reduce speed variability in studies such as Grumert et al. (2018) which evaluated four control algorithms using SUMO simulation. The four control algorithms seen in Van Toorenburg and De Kok (1999), Lee et al. (2006), Hegyi et al. (2008), and Müller et al. (2015) were chosen. The MTFC algorithm detailed in Carlson et al. (2011) showed the biggest improvement in Coefficient of Variation of Speed, a key indicator of safety. These results all predicated on the assumption that drivers comply with the set speed limits since it was done in simulation. Another study in Chang et al. (2011) tested a VSL algorithm in response to recurrent congestion with an 8 week trial on an actual corridor. The algorithm showed promising results, yielding higher throughput and more stable conditions of traffic. However, our algorithm will only be activated in response to incident-induced congestion.

Lane control systems are typically implemented in conjunction with VSL, but in Chang et al. (1999) solely LCS was implemented in simulation but found decreased throughput with efficient lane-changing. In other studies where both LCS and VSL algorithms are used such as Zhang and Ioannou (2016), Guo et al. (2020), Vrbanic et al. (2021), and Gregurić et al. (2022), it was seen that LCS systems have the potential to offset the negative throughput of VSL systems whilst maintaining safety impacts.

The algorithms used in the initial implementation of our system are rule-based controls where algorithms are reactive to traffic conditions such as lane blockage for LCS and speed, volume, and occupancy for VSL. These algorithms can be seen in Zhang et al. (2022) where the algorithms display high safety improvements based upon simulation data in TransModeler software.

2.2 Software architectures

Decision support systems for ITS have existed for decades, yet none have ingested real-time data and used artificial intelligence in order to produce LCS and VSL signals displayed on the freeway. However, there are many similarities between systems utilizing some, but not all, of the highlighted features.

A typical VSL decision support system has an architecture similar to that outlined in YAN (2013), Akhtar

Ali Shah et al. (2008), and Cheng and Zheng (2016), where emphasis is placed solely upon gathering detector data and producing an optimal speed limit output in a simulated environment. In some of these systems, data collection and generation of the new speed limit occur in a linear fashion YAN (2013), while in others, they are done simultaneously Akhtar Ali Shah et al. (2008). Our proposed architecture takes the latter approach, utilizing multiprocessing to continuously gather new data while also generating optimal VSL and LCS configurations.

A significant flaw consistently seen in the development of architectures revolving around decision support systems for ITS is "the lack of flexibility and scalability in supporting incremental growth". Much like my multiprocessing design, Osaba et al. (2016) proposes diverging from the traditional vertical management model in favor of a horizontal/distributed one. In order to avoid the tedious nature of restructuring the devices that the ITS is constructed from for each iterative addition, the network graph is automated such that no oversight is necessary. Additionally, the increased latency resulting from adding devices that must communicate with the management center is minimized, as VSL computation can occur simultaneously along data collection through multiprocessing. As a result, we are not waiting for the data from the expanded network to come across in full prior to crafting a solution.

The Texas Department of Transportation (TxDOT) Kuhn et al. (2015), working alongside the Southwest Research Institute and Texas A&M Transportation Institute, created a VSL decision support system closely related to our proposed architecture. Within this architecture, a VSL module determines the optimal value linked to an event based on data received through various input modules. Events may be created as a result of construction, weather, or traffic congestion. Prior to being displayed on the interstate, an operator must approve the use of the variable speed limit. Our architecture matches all of these aspects, except it currently only determines AI evaluation outputs in response to congestion from incidents.

CHAPTER 3

Methodology

3.1 Inter-system integration

The system typically used at TMCs has an operator interact with a software which manages responses to incidents on the freeway. This system would need to be integrated with an AI-DSS in order to provide the benefits of AI-enabled corridor management support. In order to bridge the gap between the TMC software and an AI-DSS, an API is created. This API allows for bidirectional communication between the AI-DSS and TMC software. A diagram displaying this communication structure is shown in Figure 3.1. The TMC operator is included in the figure to show how a HITL is considered in the design of this architecture. This adds an extra layer of complexity, but offers a verification method on which the algorithms can learn upon through operator expertise/feedback.

3.2 Inter-system communication

API GET requests are utilized for initial population of AI-DSS caches on startup of a process. API websockets are used for real-time updates to caches, requests for response plans from the TMC system, and communication of LCS/VSL evaluations to the TMC system. Websockets are becoming widely adopted for systems requiring bidirectional communication as seen in Murley et al. (2021). The GET request can be utilized again if the websocket is dropped to populate the data missed during downtime. A dropped websocket can be detected with pinging protocol over a set time interval.

3.3 Requirements

Requirements for the AI-DSS are shown in Table 3.1. These requirements motivated the architecture described in the next section.

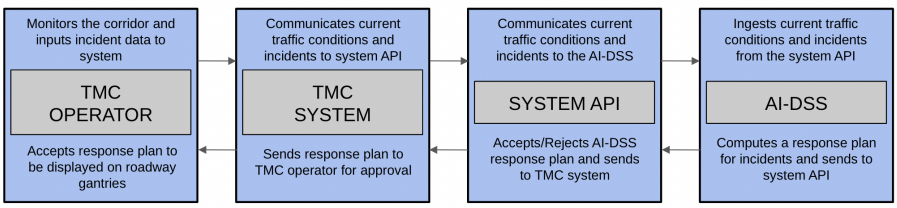


Figure 3.1: Communication diagram of TMC system with ICM AI-DSS.

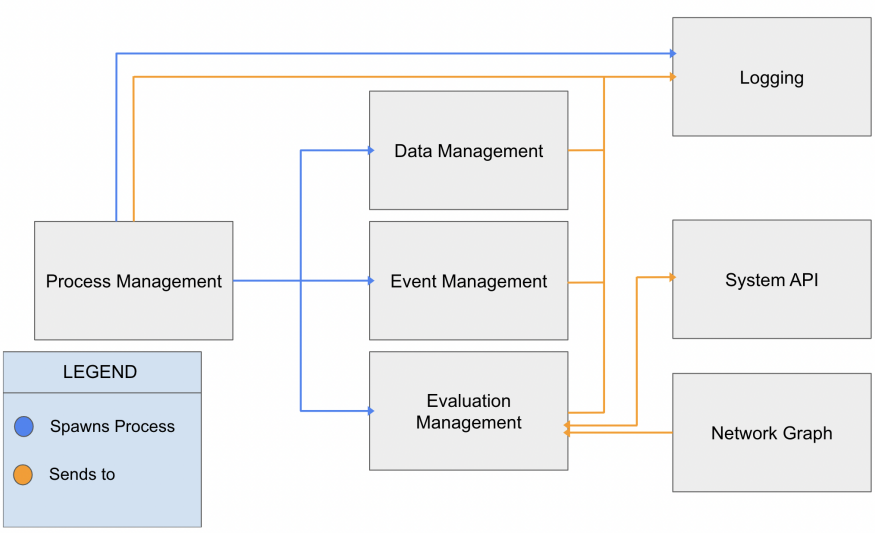


Figure 3.2: General architecture of an ICM AI-DSS.

#	Description
1	The AI-DSS shall have the ability to spawn and manage processes for communication/evaluation.
2	The AI-DSS shall have the ability to have bidirectional communication with the TMC system API.
3	The AI-DSS shall have the ability to ingest real-time traffic/incident data.
4	The AI-DSS shall keep a shared short-term data storage necessary for evaluations.
5	The AI-DSS shall send data to long-term storage necessary for offline learning.
6	The AI-DSS response plans for incidents shall be recommendations; must be enacted by operators.
7	The AI-DSS shall store response plan communication to be fed back into offline AI learning.
8	The AI-DSS shall have a relative understanding of the necessary roadway and device locations.
9	The AI-DSS shall log relevant information for debugging purposes.
10	The AI-DSS shall log relevant diagnostics/statistics for visualization of system state.

Table 3.1: Requirements defined for an ICM AI-DSS.

3.4 AI-DSS Architecture

The architecture that can be used for an AI-DSS can be divided into six main components motivated from the system requirements: process management, data management, event management, evaluation management, network graph, and logging. A visual depiction of this high-level organization is shown in Figure 4.3. The System API is also included to show how communication occurs to outside the system.

3.4.1 Process management

An AI-DSS requires many moving parts in its function. An over-arching process manager is necessary to spawn all of the processes in the system and respawn them if they have died. Multiprocessing also provides an essential component of shared memory. Process management creates data structures to be shared amongst all the processes. The relevant data structures being created are caches for data, event, and evaluation management.

3.4.2 Data management

A data management component is necessary for ingesting traffic data across the API and organizing it into the shared data cache. This component provides necessary data for evaluation functions. This also will dump data to long-term storage in a database once it is no longer relevant for current evaluations. The data management component will ingest channels such as TSS (traffic surveillance system) data, DMS data, VSL data, LCS data, and other necessary data determined to be useful for system function.

3.4.3 Event management

An event management component is necessary for ingesting incident/event data across the API and organizing it into the shared event cache. This component provides details about incidents for evaluation functions like locations and lane blockage information. It is also responsible for deleting resolved incidents from the cache and placing them into long-term storage in a database.

3.4.4 Evaluation management

An evaluation management component is necessary for storing the AI evaluation functions and sending LCS/VSL evaluations over the API. This component will have direct communication with the API system to provide recommendations on request. It is also responsible for storing response plan communication into a database for offline learning. LCS and VSL evaluations will differ slightly in implementation.

VSL evaluation will be run periodically as long as there are events/congestion on the corridor. Thus, the VSL evaluation functions should be called periodically for sending suggested board speeds for the corridor based upon speed, volume, and occupancy data. LCS functions, however, should be called on request from the API. Lane blockage details and traffic information in relation to a specific event can be used to calculate an appropriate board configuration to send for operator approval.

3.4.5 Network graph

The network graph is essential for device linking on the necessary roadways and roadway geometry. The system is able to understand how devices are located in relation to one another for evaluation management.

The graph will need to be maintained to the most current configuration of the roadway and status of devices associated with it. A network graph is important for diversion rerouting within the corridor as well.

3.4.6 Logging

The logging component is important for two reasons: debugging the system and visualization. The system logs messages at the standard logging levels. Logging also allows for visualization integration with intelligent log querying. Dashboards can be created in order to monitor the system as well as create informational visualizations like traffic speeds on the corridor. The Elasticsearch-Logstash-Kibana stack is an example of these log-based visualization systems - refer to Elasticsearch (2022).

CHAPTER 4

Results

4.1 Implementation

The proposed system architecture from Chapter 3 is implemented within the Advanced Transportation and Congestion Management Technologies Deployment (ATCMTD) project in Tennessee along the I-24 Smart Corridor. This project is a collaboration between TDOT, Stantec, Southwest Research Institute and Vanderbilt. The I-24 Smart Corridor runs from Nashville to Murfreesboro which includes Interstate 24 and State Route 1 (shown in Figure 4.1). The AI-DSS is integrated alongside the broader system used by the TDOT TMC named SmartwayCS in order to support operators in decision-making. The AI-DSS API links the two systems together where the systems can communicate over TCP/IP protocol using websockets and GET requests. This integration can be seen in Figure 4.2 from Stantec (2021).

4.2 Software details

The system was implemented using Python due to its well-maintained packages for multiprocessing, logging, API integration, and reinforcement learning for the evaluator algorithms. The system is run on a Linux VM (RedHat at TDOT, Ubuntu at Vanderbilt) for ease of integration with running the system as a system service.

The AI-DSS is run in four separate environments based on function and location currently: development at Vanderbilt, production mirror at Vanderbilt, demo at TDOT, and production at TDOT. The development systems are utilized for adding features/bug fixes and system testing. Once the systems have been thoroughly tested, they are pushed to production environments. The environment being utilized is specified to choose the correct configuration file to select in the config folder.

A diagram of the implemented system architecture for the AI-DSS is shown in Figure 4.3. The AI-DSS manager is in charge of spawning processes, creating shared data structures, and monitoring processes for respawn. The messaging subsystem takes log messages placed on the message queue and sends them to their correct mediums (console, elasticsearch, file log) for debugging/visualization purposes. The event subsystem ingests event (incident) information from the API and manages a cache. The data subsystem ingests traffic and sensor data from the API and manages a cache. The recommendation subsystem provides LCS board configurations for response plans when requested for an event based on lane blockage data from the event cache. The VSL subsystem provides suggestions for speed limits along the highway in response to events based on speed, volume, and occupancy data in the data cache. The evaluator is a file of functions necessary for making evaluations for VSL suggestions and LCS response plans. The configuration files make it easy to

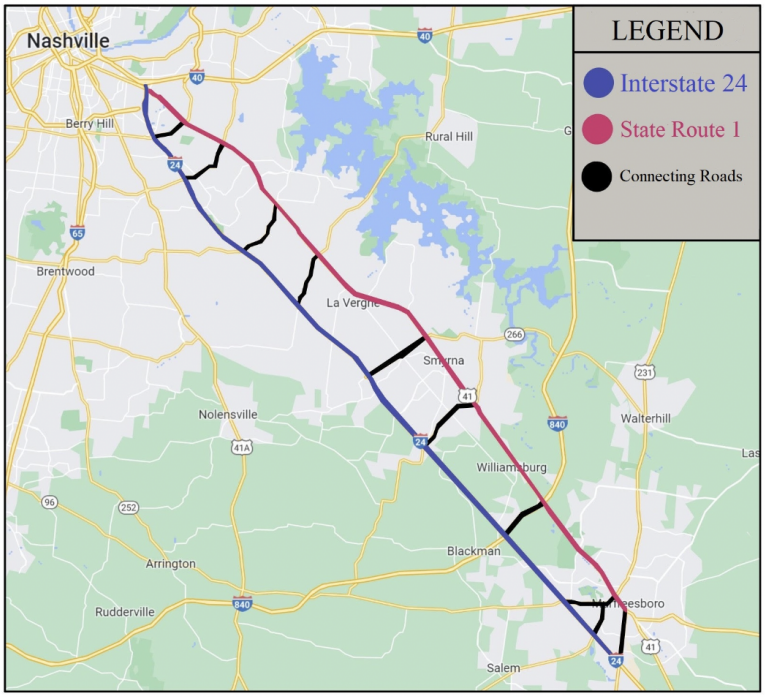


Figure 4.1: Map of the I-24 Smart Corridor.

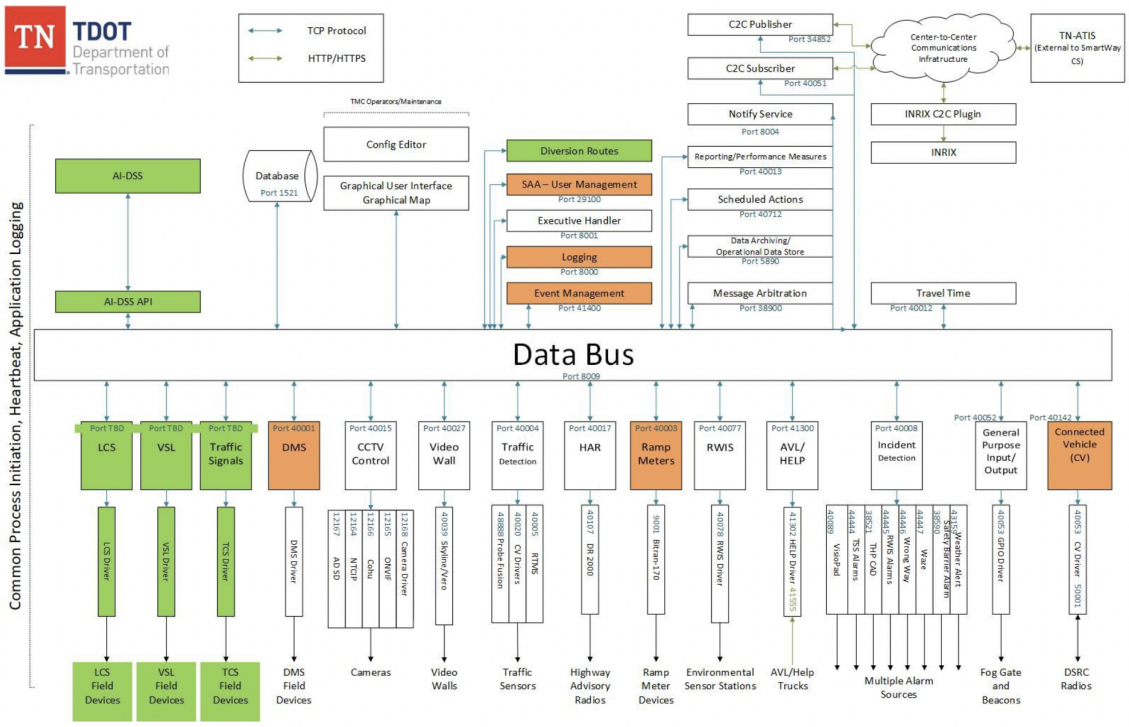


Figure 4.2: Integration of AI-DSS with existing TDOT system, the AI-DSS can be seen in the top left.

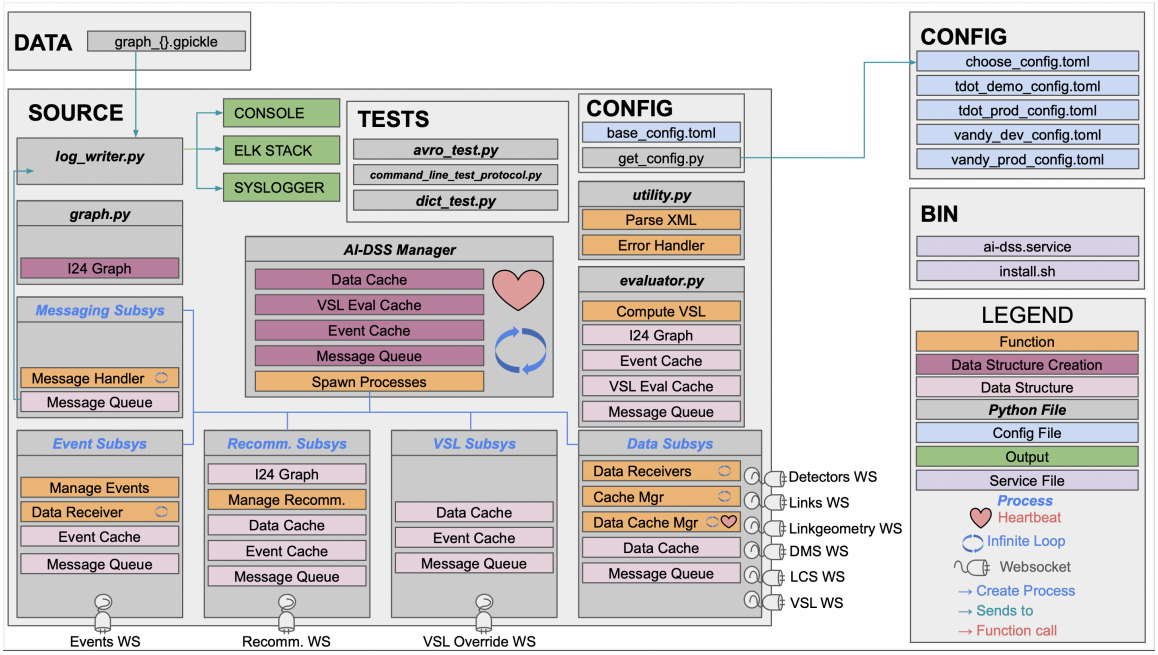


Figure 4.3: Architecture diagram of the AI-DSS for the ATCMTD project.

switch between development at production environments housed at Vanderbilt and TDOT.

4.3 User Acceptance Testing

The system was tested with TDOT utilizing User Acceptance Testing on a demo instance of SmartwayCS in order to confirm our ability to communicate across the API for evaluations. The AI-DSS passed UAT displaying abilities to spawn/manage all processes, ingest data, and communicate evaluations to SmartwayCS at the TMC for VSL/LCS suggestions. The tests were passed on November 10, 2022 and the entire UAT document can be found in Appendix C. This test is the major milestone for having the system ready to be run in production.

4.4 Code repository

The code documentation for the AI-DSS can be found in Appendix A and the actual code can be found in Appendix B. This release has not yet been deployed in production at TDOT, but has passed UAT. The project is held in a private repository on GitHub.

CHAPTER 5

Conclusion

5.1 Remarks

In this thesis, I addressed how to solve the challenge of integrating existing architectures with new infrastructure for AI-enabled corridor management considering a human-in-the-loop. This is helpful for future ITS implementations with TMCs wanting to integrate AI ICM to improve mobility and safety. The project is the first to establish an AI-DSS which leverages LCS and VSL algorithms to make these improvements on real-time data. The architecture was implemented and demonstrated to work with the TMC system, passing User Acceptance Testing.

5.2 Future work

In future work, we will consider arterial signal integration and diversion rerouting in the implementation of the architecture for the ATCMTD project. Diversion rerouting and arterial signal integration will be essential to future use of this architecture to respond to incidents. Utilizing these two additions can optimize use of arterial roads in corridors for rerouting off of interstates with congestion. Several studies have provided architectures that could be easily incorporated into the current system for diversion rerouting. One such study determines which road segments cannot be driven over and then iterates over potential candidate routes until the best one is selected. Since data gathering and evaluation occur exclusively from each other, we could easily expand the current architecture to make a second evaluation call to toggle routing recommendations if certain conditions are met, such as all lanes being blocked. DMS message boards would need to be integrated within response plan suggestions as well in order to specify rerouting to drivers on the freeway.

Appendix A

AI-DSS code documentation

This repository will serve as the working code base for the I-24 ATCMTD project AI-DSS. This system will be used to interact with SmartwayCS (a SwRI product) to receive traffic data, establish AI to learn upon that data, and construct response plans to incidents.

A.1 System requirements

- Ubuntu 20.04 or higher
- Python 3.9 or higher

A.2 Code install

The code is installed using the **install.sh** bash script. Before running this script, there are a few requirements of the system:

1. Your system should have python3.9 or higher installed and pip installed as well.
2. Make sure you have a user specified in your system with sudo and root privileges named **aidss** in the terminal.
3. There is a directory created at root named **AI-DSS/**; if not, create it: `sudo mkdir /AI-DSS`.
4. Run the command `chmod -R 777 AI-DSS/` in the terminal.
5. Now you can run the **install.sh** script in the terminal if the script is in your current working directory. Simply run `sudo bash install.sh` in the terminal.
6. Once that is run, type the command `pip3 install -r AI-DSS/I24-AI-DSS/requirements.txt` to install all necessary packages.
7. Test the AI-DSS from the terminal before starting it as a system service. You can do so by typing `python3 /AI-DSS/I24-AI-DSS/AIDSS_manager.py` in the terminal and make sure no errors occur.
8. If the terminal test runs smoothly, start the AI-DSS as a system service by typing `systemctl start ai-dss`.

```
I24-AI-DSS
+-- README.md
+-- .gitignore
+-- requirements.txt
+-- bin
|   +-- ai-dss.service
|   +-- install.sh
+-- config
|   +-- choose_config.toml
|   +-- tdot_demo_config.toml
|   +-- tdot_prod_config.toml
|   +-- vandy_demo_config.toml
|   +-- vandy_prod_config.toml
+-- data
|   +-- graph_{}.gpkl
+-- source
|   +-- AIDSS_manager.py
|   +-- evaluator.py
|   +-- graph.py
|   +-- log_writer.py
|   +-- subsys_data.py
|   +-- subsys_events.py
|   +-- subsys_messageing.py
|   +-- subsys_recommendations.py
|   +-- subsys_vsl.py
|   +-- utility.py
|   +-- I24customwebsocket
|   +-- config
|   |   +-- base_config.toml
|   |   +-- get_config.py
|   +-- tests
```

Figure A.1: File organization outline in the AI-DSS repository.

A.3 Configuration file selection

You can specify the configuration for each environment by using the **/config/choose_config.toml**. Set the value to `true` based on whether you are running the AI-DSS in the TDOT Demo, TDOT Production, Vandy Development, or Vandy Production environment.

A.4 File overview

- **ai-dss.service** - System service file to run the AI-DSS.
- **install.sh** - Bash script to install the AI-DSS on a new destination.
- **choose_config.toml** - Indicates which config file to use among tdot_prod, tdot_demo, vandy_prod, vandy_dev.
- **tdot_demo_config.toml** - Config file used when running on TDOT demo.
- **tdot_prod_config.toml** - Config file used when running on TDOT production.

- **vandy_dev_config.toml** - Config file used when running development at Vanderbilt.
- **vandy_prod_config.toml** - Config file used when running production mirror at Vanderbilt.
- **graph_.gpkl** - Current graph gpickle file that will be used by the AI-DSS.
- **AIDSS_manager.py** - Top level process for live AI-DSS. Spawns and manages child processes for sub-systems and owns data structures
- **evaluator.py** - Routine for ingesting an event and relevant data, then determining a response plan / VSL override.
- **graph.py** - Contains the class I24Graph, which holds a graph object loaded from file and has calculation functions for running queries against the graph.
- **log_writer.py** - Contains the logging backend for the AI-DSS.
- **subsys_data.py** - Contains the data subsystem that manages data processes for the AI-DSS.
- **subsys_events.py** - Contains the event subsystem that manages event processes for the AI-DSS.
- **subsys_messaging.py** - Contains the messaging subsystem that takes messages from other subsystems ; handles their logging / distribution.
- **subsys_recommendations** - Contains the recommendation subsystem that manages response plan processes for the AI-DSS.
- **subsys_vsl** - Contains VSL evaluation subsystem that watches for active events and computes VSL evaluation until event closures / congestion dissipation.
- **utility.py** - Contains utilities to be used throughout the AI-DSS repository.
- **./I24customwebsocket** - A modification of the websocket package to override default ping/pong behavior with custom message ping to fix issue with detecting ”silent” connection drops.
- **base_config.toml** - Declares paths for set-up of the AI-DSS.
- **get_config.py** - Converts appropriate .toml config file into python dictionary.

A.5 Websocket-client package edits

We have edited the websocket-client package in order to fit the needs of our system. The changes we made are reflected in changing the pinging protocol to actually send a message at the application level (instead of at the protocol level) and wait for a reply since SwCS replies with an ”OK” message every time bytes are sent across a websocket. Our code edits can be found within the **/source/I24customwebsocket** folder in this repository.

Appendix B

AI-DSS code

B.1 ai-dss.service

```
[Unit]
Description=System service using systemd on Linux for running the AI-DSS
After=multi-user.target

[Service]
Type=simple
Restart=always
ExecStart=python3.9 /AI-DSS/I24-AI-DSS/source/AIDSS_manager.py --serve-in-foreground

[Install]
WantedBy=multi-user.target
```

B.2 install.sh

```
#!/bin/bash

# AI-DSS service exists
if service --status-all | grep -Fq 'ai-dss'; then
  # Stop the service and delete
  systemctl stop ai-dss
  systemctl disable ai-dss
  rm /etc/systemd/system/ai-dss
  rm /usr/lib/systemd/system/ai-dss
fi

# Update the AI-DSS repository, forcing overwrite if applicable
unzip -o *.zip

# Create system service for AI-DSS (not existing at this point)
cp ./I24-AI-DSS/bin/ai-dss.service /etc/systemd/system

# Make outputs directory if it doesn't exist
if [ ! -d "outputs" ]; then
  mkdir outputs/
  mkdir outputs/responses
  mkdir outputs/logs
fi

# Reload the systemd daemon and delete the zip file
systemctl daemon-reload
rm *.zip
```

B.3 choose_config.toml

```
tdot_prod = false
tdot_demo = false
vandy_prod = false
vandy_dev = true
```

B.4 tdot_demo_config.toml

```
[LOGGING]
log_name = 'AI-DSS'
sl_address = "10.224.24.234, 8000"
processing_environment = 'general'
connect_sl = true
connect_console = true
connect_logstash = false
connect_file = true
file_log_level = "INFO"
sl_log_level = 'INFO'
console_log_level = 'INFO'
logstash_log_level = 'INFO'
logstash_address = ''

[GRAPH]
directory = 'data/'
blacklist_reload_interval = 5

[DATABASE]
activate = false
host = ''
port = 27017
username = ''
password = ''

[STORAGE]
activate = true

[PROCESSES]
subsys_events = true
subsys_data = true
subsys_recommendations = true
subsys_vsl = true
subsys_messaging = true
cache_manager = true
links_aggregate = false
data_detectors = true
data_links = true
data_linkgeometry = true
data_dms = true
data_lcs = true
data_vsl = true

[CONNECTIONS]
detectors_request_url = "http://10.224.24.234/decisionsupportapi/retrieve/tss/detectors"
detectors_websocket_url = "ws://10.224.24.234:80/decisionsupportapi/subscribe/tss/detectors"
event_request_url = "http://10.224.24.234/decisionsupportapi/retrieve/em/events"
event_websocket_url = "ws://10.224.24.234:80/decisionsupportapi/subscribe/em/events"
links_request_url = "http://10.224.24.234/decisionsupportapi/retrieve/tss/links"
links_websocket_url = "ws://10.224.24.234:80/decisionsupportapi/subscribe/tss/links"
linkgeometry_request_url = "http://10.224.24.234/decisionsupportapi/retrieve/tss/linkgeometry"
linkgeometry_websocket_url = "ws://10.224.24.234:80/decisionsupportapi/subscribe/tss/linkgeometry"
dms_request_url = "http://10.224.24.234/decisionsupportapi/retrieve/dms/dmses"
dms_websocket_url = "ws://10.224.24.234:80/decisionsupportapi/subscribe/dms/dmses"
lcs_request_url = "http://10.224.24.234/decisionsupportapi/retrieve/lcs/lcses"
lcs_websocket_url = "ws://10.224.24.234:80/decisionsupportapi/subscribe/lcs/lcses"
vsl_request_url = "http://10.224.24.234/decisionsupportapi/retrieve/vsl/segments"
vsl_websocket_url = "ws://10.224.24.234:80/decisionsupportapi/subscribe/vsl/segments"
rp_websocket_url = "ws://10.224.24.234:80/decisionsupportapi/subscribe/em/responseplansuggestions"
vsl_override_websocket_url = "ws://10.224.24.234:80/decisionsupportapi/subscribe/vsl/override"

[DATASIZES]
MSG_QUEUE_SIZE = 10000
```

```
DATA_QUEUE_SIZE = 10000
LINKS_QUEUE_SIZE = 10000
DATA_SIZE = 120

[TIMEOUTS]
RESPONSE_EVAL_TIMEOUT = 10
VSL_OVERRIDE_TIMEOUT = 5

[HEARTBEATS]
heartbeat_interval = 30
send_manager_heartbeat = true
send_events_heartbeat = true

[TOGGLES]
add_gantry = true
```

B.5 tdot_prod_config.toml

```
[LOGGING]
log_name = ''
sl_address = ''
processing_environment = ''
connect_sl = false
connect_console = false
connect_logstash = false
sl_log_level = ''
console_log_level = ''
logstash_log_level = ''
logstash_address = ''

[GRAPH]
directory = 'data/'
blacklist_reload_interval = 5

[DATABASE]
activate = false
host = ''
port = 27017
username = ''
password = ''

[STORAGE]
activate = false

[PROCESSES]
subsys_events = false
subsys_data = false
subsys_recommendations = false
subsys_vsl = false
subsys_messaging = false
cache_manager = false
links_aggregate = false
data_detectors = false
data_links = false
data_linkgeometry = false
data_dms = false
data_lcs = false
data_vsl = false

[CONNECTIONS]
detectors_request_url = ''
detectors_websocket_url = ''
event_request_url = ''
```

```
event_websocket_url = ''
links_request_url = ''
links_websocket_url = ''
linkgeometry_request_url = ''
linkgeometry_websocket_url = ''
dms_request_url = ''
dms_websocket_url = ''
lcs_request_url = ''
lcs_websocket_url = ''
vsl_request_url = ''
vsl_websocket_url = ''
rp_websocket_url = ''
vsl_override_websocket_url = ''

[DATASIZES]
MSG_QUEUE_SIZE = 10000
DATA_QUEUE_SIZE = 10000
LINKS_QUEUE_SIZE = 10000
DATA_SIZE = 120

[TIMEOUTS]
RESPONSE_EVAL_TIMEOUT = 10
VSL_OVERRIDE_TIMEOUT = 5

[HEARTBEATS]
heartbeat_interval = 30
send_manager_heartbeat = false
send_events_heartbeat = false

[TOGGLES]
add_gantry = false
```

B.6 vandy_dev_config.toml

```
[LOGGING]
log_name = "AI-DSS"
sl_address = "atcmtd-scs.isis.vanderbilt.edu, 8000"
processing_environment = "general"
connect_sl = false
connect_console = true
connect_logstash = false
connect_file = false
file_log_level = "INFO"
sl_log_level = "INFO"
console_log_level = "INFO"
logstash_log_level = "INFO"
logstash_address = "10.80.4.91, 5000"

[GRAPH]
directory = 'data/'
blacklist_reload_interval = 5

[DATABASE]
activate = false
host = "10.80.4.91"
port = 27017
username = "mongo-admin"
password = "i24-data-access"

[STORAGE]
activate = false

[PROCESSES]
```

```
subsys_events = true
subsys_data = true
subsys_recommendations = true
subsys_vsl = true
subsys_messaging = true
cache_manager = true
links_aggregate = true
data_detectors = true
data_links = true
data_linkgeometry = true
data_dms = true
data_lcs = true
data_vsl = true

[CONNECTIONS]
detectors_request_url = "http://atcmtd-scs.isis.vanderbilt.edu:8010/retrieve/tss/detectors"
detectors_websocket_url = "ws://atcmtd-scs.isis.vanderbilt.edu:8010/subscribe/tss/detectors"
event_request_url = "http://atcmtd-scs.isis.vanderbilt.edu:8010/retrieve/em/events"
event_websocket_url = "ws://atcmtd-scs.isis.vanderbilt.edu:8010/subscribe/em/events"
links_request_url = "http://atcmtd-scs.isis.vanderbilt.edu:8010/retrieve/tss/links"
links_websocket_url = "ws://atcmtd-scs.isis.vanderbilt.edu:8010/subscribe/tss/links"
linkgeometry_request_url = "http://atcmtd-scs.isis.vanderbilt.edu:8010/retrieve/tss/linkgeometry"
linkgeometry_websocket_url = "ws://atcmtd-scs.isis.vanderbilt.edu:8010/subscribe/tss/linkgeometry"
dms_request_url = "http://atcmtd-scs.isis.vanderbilt.edu:8010/retrieve/dms/dmses"
dms_websocket_url = "ws://atcmtd-scs.isis.vanderbilt.edu:8010/subscribe/dms/dmses"
lcs_request_url = "http://atcmtd-scs.isis.vanderbilt.edu:8010/retrieve/lcs/lcses"
lcs_websocket_url = "ws://atcmtd-scs.isis.vanderbilt.edu:8010/subscribe/lcs/lcses"
vsl_request_url = "http://atcmtd-scs.isis.vanderbilt.edu:8010/retrieve/vsl/segments"
vsl_websocket_url = "ws://atcmtd-scs.isis.vanderbilt.edu:8010/subscribe/vsl/segments"
rp_websocket_url = "ws://atcmtd-scs.isis.vanderbilt.edu:8010/subscribe/em/responseplansuggestions"
vsl_override_websocket_url = "ws://atcmtd-scs.isis.vanderbilt.edu:8010/subscribe/vsl/override"

[DATASIZES]
MSG_QUEUE_SIZE = 10000
DATA_QUEUE_SIZE = 10000
LINKS_QUEUE_SIZE = 10000
DATA_SIZE = 120

[TIMEOUTS]
RESPONSE_EVAL_TIMEOUT = 10
VSL_OVERRIDE_TIMEOUT = 5

[HEARTBEATS]
heartbeat_interval = 30
send_manager_heartbeat = true
send_events_heartbeat = true

[TOGGLES]
add_gantry = true
```

B.7 vandy_prod_config.toml

```
[LOGGING]
log_name = "AI-DSS"
sl_address = ''
processing_environment = "general"
connect_sl = false
connect_console = true
connect_logstash = true
connect_file = false
sl_log_level = "INFO"
console_log_level = "INFO"
logstash_log_level = "INFO"
```

```
logstash_address = "10.80.4.91, 5000"

[GRAPH]
directory = "data/"
blacklist_reload_interval = 5

[DATABASE]
activate = true
host = "10.80.4.91"
port = 27017
username = "mongo-admin"
password = "i24-data-access"

[STORAGE]
activate = false

[PROCESSES]
subsys_events = true
subsys_data = true
subsys_recommendations = false
subsys_vsl = false
subsys_messaging = true
cache_manager = true
links_aggregate = false
data_detectors = true
data_links = true
data_linkgeometry = true
data_dms = false
data_lcs = false
data_vsl = false

[CONNECTIONS]
detectors_request_url = 'https://c2c.tdot.tn.gov/decisionsupportapi/retrieve/tss/detectors'
detectors_websocket_url = 'wss://c2c.tdot.tn.gov:443/decisionsupportapi/subscribe/tss/detectors'
event_request_url = 'https://c2c.tdot.tn.gov/decisionsupportapi/retrieve/em/events'
event_websocket_url = 'wss://c2c.tdot.tn.gov:443/decisionsupportapi/subscribe/em/events'
links_request_url = 'https://c2c.tdot.tn.gov/decisionsupportapi/retrieve/tss/links'
links_websocket_url = 'wss://c2c.tdot.tn.gov:443/decisionsupportapi/subscribe/tss/links'
linkgeometry_request_url = 'https://c2c.tdot.tn.gov/decisionsupportapi/retrieve/tss/linkgeometry'
linkgeometry_websocket_url = 'wss://c2c.tdot.tn.gov:443/decisionsupportapi/subscribe/tss/linkgeometry'
dms_request_url = ''
dms_websocket_url = ''
lcs_request_url = ''
lcs_websocket_url = ''
vsl_request_url = ''
vsl_websocket_url = ''
rp_websocket_url = ''
vsl_override_websocket_url = ''

[DATASIZES]
MSG_QUEUE_SIZE = 10000
DATA_QUEUE_SIZE = 10000
LINKS_QUEUE_SIZE = 10000
DATA_SIZE = 120

[TIMEOUTS]
RESPONSE_EVAL_TIMEOUT = 10
VSL_OVERRIDE_TIMEOUT = 5

[HEARTBEATS]
heartbeat_interval = 30
send_manager_heartbeat = true
send_events_heartbeat = true

[TOGGLES]
```

```
add_gantry = false
```

B.8 AIDSS_manager.py

```
# ----------------------------
"""
Top level process for live AI-DSS. Spawns and manages child processes for sub-systems and owns data structures.
"""
__file__ = 'AIDSS_manager.py'
# ----------------------------

import multiprocessing as mp
import os
import sys
import signal
import time
import psutil

import subsys_events
import subsys_data
import subsys_recommendations
import subsys_messaging
import subsys_vsl

from config.get_config import config

# Change the path to absolute and load the config file
# os.chdir(os.path.dirname(os.path.abspath(__file__)))


def main():
    """
    Normal AI-DSS Manager behavior.
    """

    # MANAGER CREATION
    # ---------------------------------

    # Get manager process ID
    manager_pid = os.getpid()

    # Kill any other python processes present on system at startup
    process_iterator = psutil.process_iter()
    for proc in process_iterator:
        try:
            # Get process name & pid from process object
            processName = proc.name()
            processID = proc.pid
            if processID != manager_pid and ('python' in processName or 'PYTHON' in processName or 'Python' in processName):
                psutil.Process(pid=processID).kill()
        except (psutil.NoSuchProcess, psutil.AccessDenied, psutil.ZombieProcess):
            pass

    # Assign manager process to variable
    mp_manager = mp.Manager()

    # SHARED DATA STRUCTURES
    # ---------------------------------

    # Event cache is a single dictionary of format {eventID: timestamp, ...}
    # - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    event_cache = mp_manager.dict()

```

```
# VSL eval cache is a dictionary of, currently, one other item: a list of events that we're evaluating VSL on.
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
vsl_eval_cache = mp_manager.dict()
vsl_eval_cache['eval_events'] = mp_manager.dict()

# Data cache is a nested dictionary of format {dataName: {dataKey: dataValue}, ...}
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# We have to nest the multiprocessing data structures because nesting with simple types causes issues with
# mutability when changed in child processes.
# https://stackoverflow.com/questions/8640367/manager-dict-in-multiprocessing
data_cache = mp_manager.dict()
# Caches for data types are dictionaries organized by ID, each containing a list of dictionaries over time
data_cache['detectors'] = mp_manager.dict()
data_cache['links'] = mp_manager.dict()
data_cache['linkgeometry'] = mp_manager.dict()
data_cache['dms'] = mp_manager.dict()
data_cache['lcs'] = mp_manager.dict()
data_cache['vsl'] = mp_manager.dict()

# Message queue assumes that all messages are strings
# - - - - - - - - - - - - - - - - - - - - - - - - -
message_queue = mp_manager.Queue(config['DATASIZES']['MSG_QUEUE_SIZE'])
# These messages won't get logged until `subsys_messaging` starts.
message_queue.put(('INFO', "STARTUP: AI-DSS manager starting up."))
message_queue.put(('INFO', "STARTUP: AI-DSS manager has PID={}.".format(manager_pid)))

# PID tracker is a single dictionary of format {processName: PID}
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
pid_tracker = mp_manager.dict()

# SIGNAL HANDLING
# -----------------------------------

def handle_debug_elevate_signal(signal_number, frame):
    """
    Handle the receipt of the SIGSUR1 signal to set log level to DEBUG.

    :param signal_number: number associated with SIGSUR1
    :param frame: ?
    :return: None
    """
    message_queue.put(('INFO', "Received SIGUSR1 (num. {}) in AIDSS_manager.".format(signal_number)))
    message_queue.put('SIGUSR1')

def handle_debug_revert_signal(signal_number, frame):
    """
    Handle the receipt of the SIGSUR2 signal to undo the DEBUG log levels (set to config level)

    :param signal_number: number associated with SIGSUR2
    :param frame: ?
    :return: None
    """
    message_queue.put(('INFO', "Received SIGUSR2 (num. {}) in AIDSS_manager.".format(signal_number)))
    message_queue.put('SIGUSR2')

signal.signal(signal.SIGUSR1, handle_debug_elevate_signal)
signal.signal(signal.SIGUSR2, handle_debug_revert_signal)

# ASSISTANT/CHILD PROCESSES
# -----------------------------------
# References to subsystem processes that will get spawned so that these can be recalled
# upon any failure. Each list item contains the name of the process, its function handle, and
# its function arguments for call.
processes_to_spawn = {'subsys_events': (subsys_events.manage_events, (event_cache, message_queue)),
                      'subsys_data': (subsys_data.manage_data, (data_cache, message_queue, pid_tracker)),
```

```
                'subsys_recommendations': (subsys_recommendations.manage_recommendations,
                                           (event_cache, data_cache, message_queue, pid_tracker)),
                'subsys_vsl': (subsys_vsl.manage_vsl_eval, (event_cache, data_cache, message_queue)),
                'subsys_messaging': (subsys_messaging.message_handler,
                                     (message_queue,)),
                }

# Here we set which processes are set to ON or OFF based on boolean values specified in the config
manager_process_control = config['PROCESSES']
for name in list(processes_to_spawn):
    # If we specify the manager process to be false in config, we delete it and do not spawn it
    if not manager_process_control[name]:
        del processes_to_spawn[name]

# Stores the actual mp.Process objects so they can be controlled directly.
# PIDs are also tracked for now, but not used for management.
subsystem_process_objects = {}
message_queue.put(('INFO', "STARTUP: AI-DSS manager beginning to spawn processes."))

for process_name, (process_function, process_args) in processes_to_spawn.items():
    message_queue.put(('INFO', "STARTUP: AI-DSS manager starting {}.".format(process_name)))
    # Start up each process.
    # Can't make these subsystems daemon processes because they will have their own children; we'll use a
    # different method of cleaning up child processes on exit.
    subsys_process = mp.Process(target=process_function, args=process_args, name=process_name, daemon=False)
    subsys_process.start()
    message_queue.put(('INFO', "STARTUP: {} started.".format(process_name)))
    # Put the process object in the dictionary, keyed by the process name.
    subsystem_process_objects[process_name] = subsys_process
    # Each process is responsible for putting its own children's PIDs in the tracker upon creation.
    pid_tracker[process_name] = subsys_process.pid

message_queue.put(('INFO', "STARTUP: AI-DSS manager started all processes."))

try:
    while True:
        time.sleep(config['HEARTBEATS']['heartbeat_interval'])
        # for each process that is being managed at this level, check if it's still running
        running = []
        dead = []
        usage = {'manager_mem_percent': psutil.Process(manager_pid).memory_percent()}
        for child_key in subsystem_process_objects.keys():
            child_process = subsystem_process_objects[child_key]
            if child_process.is_alive():
                # Process is running; do nothing.
                message_queue.put(('DEBUG', 'HEARTBEAT: {} process is running.'.format(child_process.name)))
                running.append(True)
                # Get current memory usage
                usage['{}_mem_percent'.format(child_process.name)] = psutil.Process(
                    child_process.pid).memory_percent()
            else:
                # Process has died. Let's restart it.
                running.append(False)
                # Copy its name out of the existing process object for lookup and restart.
                process_name = child_process.name
                message_queue.put(('ERROR', "AI-DSS process manager restarting process: {}.".format(process_name)))
                # Add to list of restarted processes
                dead.append(process_name)
                # Get the function handle and function arguments to spawn this process again.
                process_function, process_args = processes_to_spawn[process_name]
                # Restart the process the same way we did originally.
                subsys_process = mp.Process(target=process_function, args=process_args, name=process_name,
                                            daemon=False)
                subsys_process.start()
                message_queue.put(('DEBUG', 'HEARTBEAT: {} process has restarted with PID {}'
```

```
                            .format(subsys_process.name, subsys_process.pid)))
                # Re-write the process object in the dictionary and update its PID.
                subsystem_process_objects[child_key] = subsys_process
                pid_tracker[process_name] = subsys_process.pid
        if all(running):
            # Log heartbeats and system utilization statistics
            system_usage = psutil.cpu_percent()
            manager_heartbeat_dict = {'message_type': 'manager_heartbeat', 'restarted_processes': 'none'}
            extra_dict = {**manager_heartbeat_dict, **usage, 'system_CPU_usage': system_usage}
            if config['HEARTBEATS']['send_manager_heartbeat']:
                message_queue.put(('INFO', "MANAGER HEARTBEAT: All processes are still running. Total system CPU "
                                   "percent usage is {}%".format(system_usage), extra_dict))
            if config['HEARTBEATS']['send_events_heartbeat']:
                event_dict = {'active_events': list(event_cache.keys())}
                message_queue.put(
                    ('INFO', "EVENTS HEARTBEAT: {} Active Events.".format(len(event_cache)), event_dict))
        else:
            message_queue.put(('ERROR', "Found a AI-DSS manager process that wasn't running.",
                               {'message_type': 'manager_heartbeat', 'restarted_processes': dead}))
    except KeyboardInterrupt:
        # Catch KeyboardInterrupt, which is the same thing as a SIGINT
        # The command `kill -INT [PID]` with the AIDSS_manager PID, executed on the command line, will gracefully
        # shut down the whole AI-DSS with its child processes.
        for pid_name, pid_val in pid_tracker.items():
            os.kill(pid_val, signal.SIGKILL)
            message_queue.put(('WARNING', "Sent SIGKILL to PID={} ({})".format(pid_val, pid_name)))


if __name__ == '__main__':

    opts = [opt for opt in sys.argv[1:] if opt.startswith("-")]
    args = [arg for arg in sys.argv[1:] if not arg.startswith("-")]

    if "-v" in opts:
        from _metadata import __version__, __status__, __maintainer__, __email__
        print("AI-DSS version = {}".format(__version__))
        print("Code status: {}".format(__status__))
        print("Maintainer: {} ({})".format(__maintainer__, __email__))
        sys.exit(0)
    elif "-t" in opts:
        print("Activating test mode for AI-DSS installation.")
        os.chdir(os.path.join(os.path.dirname(os.path.abspath(__file__)), '../'))
        from tests.command_line_test_protocol import command_line_test
        total_pass = command_line_test()
        if total_pass is True:
            print("All tests: PASS")
            sys.exit(0)
        else:
            print("All test: NOT passed")
            sys.exit(1)
    else:
        main()
```

B.9 evaluator.py

```
# ------------------------------
"""
Routine for ingesting an event and relevant data, then determining a response plan / VSL override.
"""
__file__ = 'evaluator.py'
# ------------------------------

import math
```

```python
from copy import deepcopy
from config.get_config import config


def compute_vsl_override(vsl_eval_cache, rds_cache, vsl_snapshot, message_queue) -> dict:
    """
    Determine VSL override to be sent to SwCS.

    :param vsl_eval_cache: Dictionary cache for values related to VSL evaluation; see subsys_vsl.py for structure detail
    :param rds_cache: Non-shared snapshot of RDS portion of data cache; regular Python dict of {id: [data, data, ...]}
    :param vsl_snapshot: Snapshot of data from the VSL data cache
    :param message_queue: Shared message queue that takes messages to log in the messaging subsystem
    :return: List of VSL overrides; empty list or None if no overrides desired
    """

    def get_target_speed(vsl_segments_info, direction) -> dict:
        """
        Get the latest target speed limit for all vsl segments according to "vsl_segments_info".

        :param vsl_segments_info: The dictionary containing all vsl segments' information, e.g., i24e_vsl_segments_info
        :param direction: Eastbound or Westbound
        :return: A dictionary with vsl_id as key and corresponding target speed limit as value.
        """
        target_speed = {}
        if direction == "Eastbound":
            # re-sort eastbound segments_info in an increasing order of vsl id since it has been reversed before
            vsl_segments_info = dict(reversed(vsl_segments_info.items()))
        for mm in vsl_segments_info:
            vsl_id = vsl_segments_info[mm]["VSL_ID"]
            target_speed[vsl_id] = vsl_segments_info[mm]["TARGET_SPEED"]
        message_queue.put(('DEBUG', "EVALUATE: Target speed limits set for all VSL segments.", target_speed))
        return target_speed

    def reset_vsl_target_speed(vsl_segments_info) -> None:
        """
        Initialize vsl speed limit as maximum at the beginning of each evaluation round.

        :param vsl_segments_info: The dictionary containing all vsl segments' information, e.g., i24e_vsl_segments_info
        :return: None
        """
        for mm in vsl_segments_info:
            vsl_segments_info[mm]["TARGET_SPEED"] = vsl_segments_info[mm]["MAX_SPEED"]
        message_queue.put(('DEBUG', "EVALUATE: Initialized VSL speed limits to max speeds.", vsl_segments_info))

    def vsl_activation_condition(vsl_segments_info, mile_marker, vsl_snapshot) -> bool:
        """
        Determine whether a specific vsl should be activated or not.

        :param vsl_segments_info: The dictionary containing all vsl segments' information, e.g., i24e_vsl_segments_info
        :param mile_marker: The key of vsl_segments_info, we can get access to the detailed thresholds through this
        :param vsl_snapshot: Snapshot of data from the VSL data cache
        :return: True if activated, False otherwise
        """
        volume = get_real_volume(vsl_segments_info, mile_marker, vsl_snapshot)
        message_queue.put(('DEBUG', "EVALUATE: MM {} assigned volume {}.".format(mile_marker, volume)))
        speed = get_real_speed(vsl_segments_info, mile_marker, vsl_snapshot)
        message_queue.put(('DEBUG', "EVALUATE: MM {} assigned speed {}.".format(mile_marker, speed)))
        occupancy = get_real_occupancy(vsl_segments_info, mile_marker, vsl_snapshot)
        message_queue.put(('DEBUG', "EVALUATE: MM {} assigned occupancy {}.".format(mile_marker, occupancy)))
        v_min = vsl_segments_info[mile_marker]["MIN_VOLUME"]
        s_threshold = vsl_segments_info[mile_marker]["SPEED_THRESHOLD"]
        o_threshold = vsl_segments_info[mile_marker]["OCCUPANCY_THRESHOLD"]

        # Determine if vsl override needs to be activated based on conditions
        if volume > v_min and 0 < speed < s_threshold and occupancy > o_threshold:
```

```
            message_queue.put(('DEBUG', "EVALUATE: Determined VSL must be activated at MM {}".format(mile_marker)))
            return True
        else:
            message_queue.put(('DEBUG', "EVALUATE: Determined VSL does not need to be activated at MM {}"
                               .format(mile_marker)))
            return False

    def get_real_volume(vsl_segments_info, mile_marker, vsl_snapshot) -> int:
        """
        Get the real traffic volume at a specific vsl segment.

        :param vsl_segments_info: The dictionary containing all vsl segments' information, e.g., i24e_vsl_segments_info
        :param mile_marker: The key of vsl_segments_info, we can get access to the detailed thresholds through this
        :param vsl_snapshot: Snapshot of data from the VSL data cache
        :return: Integer corresponding to the volume data
        """
        vsl_id = vsl_segments_info[mile_marker]["VSL_ID"]
        link_id = vsl_snapshot[vsl_id][-1]['link_id']
        volume = rds_cache[link_id][-1]['vol']
        return volume

    def get_real_speed(vsl_segments_info, mile_marker, vsl_snapshot) -> int:
        """
        Get the real traffic speed at a specific vsl segment.

        :param vsl_segments_info: The dictionary containing all vsl segments' information, e.g., i24e_vsl_segments_info
        :param mile_marker: The key of vsl_segments_info, we can get access to the detailed thresholds through this
        :param vsl_snapshot: Snapshot of data from the VSL data cache
        :return: Integer corresponding to the speed data
        """
        vsl_id = vsl_segments_info[mile_marker]["VSL_ID"]
        link_id = vsl_snapshot[vsl_id][-1]['link_id']
        speed = rds_cache[link_id][-1]['speed']
        return speed

    def get_real_occupancy(vsl_segments_info, mile_marker, vsl_snapshot) -> int:
        """
        Get the real traffic occupancy at a specific vsl segment.

        :param vsl_segments_info: The dictionary containing all vsl segments' information, e.g., i24e_vsl_segments_info
        :param mile_marker: The key of vsl_segments_info, we can get access to the detailed thresholds through this
        :param vsl_snapshot: Snapshot of data from the VSL data cache
        :return: Integer corresponding to the occupancy data
        """
        vsl_id = vsl_segments_info[mile_marker]["VSL_ID"]
        link_id = vsl_snapshot[vsl_id][-1]['link_id']
        occupancy = rds_cache[link_id][-1]['occ']
        return occupancy

    def round_up_five(value, base=5) -> int:
        """
        Round up the input value to the nearest multiple of base.

        :return: Integer of rounded value
        """
        rounded_val = base * math.ceil(value / base)
        message_queue.put(('DEBUG', "EVALUATE: Value {} rounded up by factor of {} to {}"
                           .format(value, base, rounded_val)))
        return rounded_val

    def swri_vsl_response(direction) -> dict:
        """
        Determine response plan based on SwRI algorithm rules.

        :param direction: Eastbound or Westbound
```

```
    :return: Dictionary of target speeds associated with vsl gantries for our override
    """
    # Define the number of bounce segments
    num_bounce_segments = 3
    # Define the step_down value
    step_down = 10
    # Downstream is the larger mile marker
    i24e_vsl_segments_info = {
       53.2: {"MIN_VOLUME": 20, "OCCUPANCY_THRESHOLD": 20, "SPEED_THRESHOLD": 60, "MIN_SPEED": 30, "MAX_SPEED": 70,
            "DISPLAY_DEVICE_ID": 2, "TARGET_SPEED": 70, "VSL_ID": 28},
       54.0: {"MIN_VOLUME": 20, "OCCUPANCY_THRESHOLD": 20, "SPEED_THRESHOLD": 60, "MIN_SPEED": 30, "MAX_SPEED": 70,
            "DISPLAY_DEVICE_ID": 4, "TARGET_SPEED": 70, "VSL_ID": 16},
       54.7: {"MIN_VOLUME": 20, "OCCUPANCY_THRESHOLD": 20, "SPEED_THRESHOLD": 60, "MIN_SPEED": 30, "MAX_SPEED": 70,
            "DISPLAY_DEVICE_ID": 7, "TARGET_SPEED": 70, "VSL_ID": 46},
       55.2: {"MIN_VOLUME": 20, "OCCUPANCY_THRESHOLD": 20, "SPEED_THRESHOLD": 60, "MIN_SPEED": 30, "MAX_SPEED": 70,
            "DISPLAY_DEVICE_ID": 9, "TARGET_SPEED": 70, "VSL_ID": 49},
       55.7: {"MIN_VOLUME": 20, "OCCUPANCY_THRESHOLD": 20, "SPEED_THRESHOLD": 60, "MIN_SPEED": 30, "MAX_SPEED": 70,
            "DISPLAY_DEVICE_ID": 11, "TARGET_SPEED": 70, "VSL_ID": 52},
       56.2: {"MIN_VOLUME": 20, "OCCUPANCY_THRESHOLD": 20, "SPEED_THRESHOLD": 60, "MIN_SPEED": 30, "MAX_SPEED": 70,
            "DISPLAY_DEVICE_ID": 14, "TARGET_SPEED": 70, "VSL_ID": 7},
       56.5: {"MIN_VOLUME": 20, "OCCUPANCY_THRESHOLD": 20, "SPEED_THRESHOLD": 60, "MIN_SPEED": 30, "MAX_SPEED": 70,
            "DISPLAY_DEVICE_ID": 16, "TARGET_SPEED": 70, "VSL_ID": 59},
       57.3: {"MIN_VOLUME": 20, "OCCUPANCY_THRESHOLD": 20, "SPEED_THRESHOLD": 60, "MIN_SPEED": 30, "MAX_SPEED": 70,
            "DISPLAY_DEVICE_ID": 18, "TARGET_SPEED": 70, "VSL_ID": 19},
       58.0: {"MIN_VOLUME": 20, "OCCUPANCY_THRESHOLD": 20, "SPEED_THRESHOLD": 60, "MIN_SPEED": 30, "MAX_SPEED": 70,
            "DISPLAY_DEVICE_ID": 20, "TARGET_SPEED": 70, "VSL_ID": 65},
       58.3: {"MIN_VOLUME": 20, "OCCUPANCY_THRESHOLD": 20, "SPEED_THRESHOLD": 60, "MIN_SPEED": 30, "MAX_SPEED": 70,
            "DISPLAY_DEVICE_ID": 22, "TARGET_SPEED": 70, "VSL_ID": 30},
       58.8: {"MIN_VOLUME": 20, "OCCUPANCY_THRESHOLD": 20, "SPEED_THRESHOLD": 60, "MIN_SPEED": 30, "MAX_SPEED": 70,
            "DISPLAY_DEVICE_ID": 23, "TARGET_SPEED": 70, "VSL_ID": 57},
       59.3: {"MIN_VOLUME": 20, "OCCUPANCY_THRESHOLD": 20, "SPEED_THRESHOLD": 60, "MIN_SPEED": 30, "MAX_SPEED": 70,
            "DISPLAY_DEVICE_ID": 25, "TARGET_SPEED": 70, "VSL_ID": 12},
       59.7: {"MIN_VOLUME": 20, "OCCUPANCY_THRESHOLD": 20, "SPEED_THRESHOLD": 60, "MIN_SPEED": 30, "MAX_SPEED": 70,
            "DISPLAY_DEVICE_ID": 27, "TARGET_SPEED": 70, "VSL_ID": 54},
       60.3: {"MIN_VOLUME": 20, "OCCUPANCY_THRESHOLD": 20, "SPEED_THRESHOLD": 60, "MIN_SPEED": 30, "MAX_SPEED": 70,
            "DISPLAY_DEVICE_ID": 29, "TARGET_SPEED": 70, "VSL_ID": 38},
       60.8: {"MIN_VOLUME": 20, "OCCUPANCY_THRESHOLD": 20, "SPEED_THRESHOLD": 60, "MIN_SPEED": 30, "MAX_SPEED": 70,
            "DISPLAY_DEVICE_ID": 31, "TARGET_SPEED": 70, "VSL_ID": 39},
       61.3: {"MIN_VOLUME": 20, "OCCUPANCY_THRESHOLD": 20, "SPEED_THRESHOLD": 60, "MIN_SPEED": 30, "MAX_SPEED": 70,
            "DISPLAY_DEVICE_ID": 32, "TARGET_SPEED": 70, "VSL_ID": 6},
       61.8: {"MIN_VOLUME": 20, "OCCUPANCY_THRESHOLD": 20, "SPEED_THRESHOLD": 60, "MIN_SPEED": 30, "MAX_SPEED": 70,
            "DISPLAY_DEVICE_ID": 34, "TARGET_SPEED": 70, "VSL_ID": 4},
       62.4: {"MIN_VOLUME": 20, "OCCUPANCY_THRESHOLD": 20, "SPEED_THRESHOLD": 60, "MIN_SPEED": 30, "MAX_SPEED": 70,
            "DISPLAY_DEVICE_ID": 36, "TARGET_SPEED": 70, "VSL_ID": 2},
       62.8: {"MIN_VOLUME": 20, "OCCUPANCY_THRESHOLD": 20, "SPEED_THRESHOLD": 60, "MIN_SPEED": 30, "MAX_SPEED": 70,
            "DISPLAY_DEVICE_ID": 38, "TARGET_SPEED": 70, "VSL_ID": 17},
       63.4: {"MIN_VOLUME": 20, "OCCUPANCY_THRESHOLD": 20, "SPEED_THRESHOLD": 60, "MIN_SPEED": 30, "MAX_SPEED": 70,
            "DISPLAY_DEVICE_ID": 40, "TARGET_SPEED": 70, "VSL_ID": 18},
       63.8: {"MIN_VOLUME": 20, "OCCUPANCY_THRESHOLD": 20, "SPEED_THRESHOLD": 60, "MIN_SPEED": 30, "MAX_SPEED": 70,
            "DISPLAY_DEVICE_ID": 42, "TARGET_SPEED": 70, "VSL_ID": 35},
       64.3: {"MIN_VOLUME": 20, "OCCUPANCY_THRESHOLD": 20, "SPEED_THRESHOLD": 60, "MIN_SPEED": 30, "MAX_SPEED": 70,
            "DISPLAY_DEVICE_ID": 44, "TARGET_SPEED": 70, "VSL_ID": 64},
       64.8: {"MIN_VOLUME": 20, "OCCUPANCY_THRESHOLD": 20, "SPEED_THRESHOLD": 60, "MIN_SPEED": 30, "MAX_SPEED": 70,
            "DISPLAY_DEVICE_ID": 46, "TARGET_SPEED": 70, "VSL_ID": 36},
       65.1: {"MIN_VOLUME": 20, "OCCUPANCY_THRESHOLD": 20, "SPEED_THRESHOLD": 60, "MIN_SPEED": 30, "MAX_SPEED": 70,
            "DISPLAY_DEVICE_ID": 48, "TARGET_SPEED": 70, "VSL_ID": 53},
       65.3: {"MIN_VOLUME": 20, "OCCUPANCY_THRESHOLD": 20, "SPEED_THRESHOLD": 60, "MIN_SPEED": 30, "MAX_SPEED": 70,
            "DISPLAY_DEVICE_ID": 50, "TARGET_SPEED": 70, "VSL_ID": 11},
       66.3: {"MIN_VOLUME": 20, "OCCUPANCY_THRESHOLD": 20, "SPEED_THRESHOLD": 60, "MIN_SPEED": 30, "MAX_SPEED": 70,
            "DISPLAY_DEVICE_ID": 52, "TARGET_SPEED": 70, "VSL_ID": 37},
       66.7: {"MIN_VOLUME": 20, "OCCUPANCY_THRESHOLD": 20, "SPEED_THRESHOLD": 60, "MIN_SPEED": 30, "MAX_SPEED": 70,
            "DISPLAY_DEVICE_ID": 54, "TARGET_SPEED": 70, "VSL_ID": 55},
       67.1: {"MIN_VOLUME": 20, "OCCUPANCY_THRESHOLD": 20, "SPEED_THRESHOLD": 60, "MIN_SPEED": 30, "MAX_SPEED": 70,
            "DISPLAY_DEVICE_ID": 55, "TARGET_SPEED": 70, "VSL_ID": 58},
       67.5: {"MIN_VOLUME": 20, "OCCUPANCY_THRESHOLD": 20, "SPEED_THRESHOLD": 60, "MIN_SPEED": 30, "MAX_SPEED": 70,
```

```
            "DISPLAY_DEVICE_ID": 57, "TARGET_SPEED": 70, "VSL_ID": 47},
        68.1: {"MIN_VOLUME": 20, "OCCUPANCY_THRESHOLD": 20, "SPEED_THRESHOLD": 60, "MIN_SPEED": 30, "MAX_SPEED": 70,
            "DISPLAY_DEVICE_ID": 59, "TARGET_SPEED": 70, "VSL_ID": 66},
        68.6: {"MIN_VOLUME": 20, "OCCUPANCY_THRESHOLD": 20, "SPEED_THRESHOLD": 60, "MIN_SPEED": 30, "MAX_SPEED": 70,
            "DISPLAY_DEVICE_ID": 61, "TARGET_SPEED": 70, "VSL_ID": 67},
        69.1: {"MIN_VOLUME": 20, "OCCUPANCY_THRESHOLD": 20, "SPEED_THRESHOLD": 60, "MIN_SPEED": 30, "MAX_SPEED": 70,
            "DISPLAY_DEVICE_ID": 62, "TARGET_SPEED": 70, "VSL_ID": 60},
        69.6: {"MIN_VOLUME": 20, "OCCUPANCY_THRESHOLD": 20, "SPEED_THRESHOLD": 60, "MIN_SPEED": 30, "MAX_SPEED": 70,
            "DISPLAY_DEVICE_ID": 64, "TARGET_SPEED": 70, "VSL_ID": 8}
    }

    # Downstream is the smallest mile marker
    i24w_vsl_segments_info = {
        53.2: {"MIN_VOLUME": 20, "OCCUPANCY_THRESHOLD": 20, "SPEED_THRESHOLD": 60, "MIN_SPEED": 30, "MAX_SPEED": 70,
            "DISPLAY_DEVICE_ID": 1, "TARGET_SPEED": 70, "VSL_ID": 63},
        53.7: {"MIN_VOLUME": 20, "OCCUPANCY_THRESHOLD": 20, "SPEED_THRESHOLD": 60, "MIN_SPEED": 30, "MAX_SPEED": 70,
            "DISPLAY_DEVICE_ID": 3, "TARGET_SPEED": 70, "VSL_ID": 15},
        54.0: {"MIN_VOLUME": 20, "OCCUPANCY_THRESHOLD": 20, "SPEED_THRESHOLD": 60, "MIN_SPEED": 30, "MAX_SPEED": 70,
            "DISPLAY_DEVICE_ID": 5, "TARGET_SPEED": 70, "VSL_ID": 48},
        54.1: {"MIN_VOLUME": 20, "OCCUPANCY_THRESHOLD": 20, "SPEED_THRESHOLD": 60, "MIN_SPEED": 30, "MAX_SPEED": 70,
            "DISPLAY_DEVICE_ID": 6, "TARGET_SPEED": 70, "VSL_ID": 14},
        54.9: {"MIN_VOLUME": 20, "OCCUPANCY_THRESHOLD": 20, "SPEED_THRESHOLD": 60, "MIN_SPEED": 30, "MAX_SPEED": 70,
            "DISPLAY_DEVICE_ID": 8, "TARGET_SPEED": 70, "VSL_ID": 1},
        55.3: {"MIN_VOLUME": 20, "OCCUPANCY_THRESHOLD": 20, "SPEED_THRESHOLD": 60, "MIN_SPEED": 30, "MAX_SPEED": 70,
            "DISPLAY_DEVICE_ID": 10, "TARGET_SPEED": 70, "VSL_ID": 51},
        55.7: {"MIN_VOLUME": 20, "OCCUPANCY_THRESHOLD": 20, "SPEED_THRESHOLD": 60, "MIN_SPEED": 30, "MAX_SPEED": 70,
            "DISPLAY_DEVICE_ID": 12, "TARGET_SPEED": 70, "VSL_ID": 62},
        56.0: {"MIN_VOLUME": 20, "OCCUPANCY_THRESHOLD": 20, "SPEED_THRESHOLD": 60, "MIN_SPEED": 30, "MAX_SPEED": 70,
            "DISPLAY_DEVICE_ID": 13, "TARGET_SPEED": 70, "VSL_ID": 9},
        56.5: {"MIN_VOLUME": 20, "OCCUPANCY_THRESHOLD": 20, "SPEED_THRESHOLD": 60, "MIN_SPEED": 30, "MAX_SPEED": 70,
            "DISPLAY_DEVICE_ID": 15, "TARGET_SPEED": 70, "VSL_ID": 20},
        56.9: {"MIN_VOLUME": 20, "OCCUPANCY_THRESHOLD": 20, "SPEED_THRESHOLD": 60, "MIN_SPEED": 30, "MAX_SPEED": 70,
            "DISPLAY_DEVICE_ID": 17, "TARGET_SPEED": 70, "VSL_ID": 27},
        57.3: {"MIN_VOLUME": 20, "OCCUPANCY_THRESHOLD": 20, "SPEED_THRESHOLD": 60, "MIN_SPEED": 30, "MAX_SPEED": 70,
            "DISPLAY_DEVICE_ID": 19, "TARGET_SPEED": 70, "VSL_ID": 33},
        58.2: {"MIN_VOLUME": 20, "OCCUPANCY_THRESHOLD": 20, "SPEED_THRESHOLD": 60, "MIN_SPEED": 30, "MAX_SPEED": 70,
            "DISPLAY_DEVICE_ID": 21, "TARGET_SPEED": 70, "VSL_ID": 42},
        58.9: {"MIN_VOLUME": 20, "OCCUPANCY_THRESHOLD": 20, "SPEED_THRESHOLD": 60, "MIN_SPEED": 30, "MAX_SPEED": 70,
            "DISPLAY_DEVICE_ID": 24, "TARGET_SPEED": 70, "VSL_ID": 32},
        59.4: {"MIN_VOLUME": 20, "OCCUPANCY_THRESHOLD": 20, "SPEED_THRESHOLD": 60, "MIN_SPEED": 30, "MAX_SPEED": 70,
            "DISPLAY_DEVICE_ID": 26, "TARGET_SPEED": 70, "VSL_ID": 10},
        60.1: {"MIN_VOLUME": 20, "OCCUPANCY_THRESHOLD": 20, "SPEED_THRESHOLD": 60, "MIN_SPEED": 30, "MAX_SPEED": 70,
            "DISPLAY_DEVICE_ID": 28, "TARGET_SPEED": 70, "VSL_ID": 56},
        60.6: {"MIN_VOLUME": 20, "OCCUPANCY_THRESHOLD": 20, "SPEED_THRESHOLD": 60, "MIN_SPEED": 30, "MAX_SPEED": 70,
            "DISPLAY_DEVICE_ID": 30, "TARGET_SPEED": 70, "VSL_ID": 5},
        61.8: {"MIN_VOLUME": 20, "OCCUPANCY_THRESHOLD": 20, "SPEED_THRESHOLD": 60, "MIN_SPEED": 30, "MAX_SPEED": 70,
            "DISPLAY_DEVICE_ID": 35, "TARGET_SPEED": 70, "VSL_ID": 61},
        61.2: {"MIN_VOLUME": 20, "OCCUPANCY_THRESHOLD": 20, "SPEED_THRESHOLD": 60, "MIN_SPEED": 30, "MAX_SPEED": 70,
            "DISPLAY_DEVICE_ID": 33, "TARGET_SPEED": 70, "VSL_ID": 34},
        62.3: {"MIN_VOLUME": 20, "OCCUPANCY_THRESHOLD": 20, "SPEED_THRESHOLD": 60, "MIN_SPEED": 30, "MAX_SPEED": 70,
            "DISPLAY_DEVICE_ID": 37, "TARGET_SPEED": 70, "VSL_ID": 22},
        62.7: {"MIN_VOLUME": 20, "OCCUPANCY_THRESHOLD": 20, "SPEED_THRESHOLD": 60, "MIN_SPEED": 30, "MAX_SPEED": 70,
            "DISPLAY_DEVICE_ID": 39, "TARGET_SPEED": 70, "VSL_ID": 43},
        63.3: {"MIN_VOLUME": 20, "OCCUPANCY_THRESHOLD": 20, "SPEED_THRESHOLD": 60, "MIN_SPEED": 30, "MAX_SPEED": 70,
            "DISPLAY_DEVICE_ID": 41, "TARGET_SPEED": 70, "VSL_ID": 44},
        63.9: {"MIN_VOLUME": 20, "OCCUPANCY_THRESHOLD": 20, "SPEED_THRESHOLD": 60, "MIN_SPEED": 30, "MAX_SPEED": 70,
            "DISPLAY_DEVICE_ID": 43, "TARGET_SPEED": 70, "VSL_ID": 50},
        64.4: {"MIN_VOLUME": 20, "OCCUPANCY_THRESHOLD": 20, "SPEED_THRESHOLD": 60, "MIN_SPEED": 30, "MAX_SPEED": 70,
            "DISPLAY_DEVICE_ID": 45, "TARGET_SPEED": 70, "VSL_ID": 29},
        64.8: {"MIN_VOLUME": 20, "OCCUPANCY_THRESHOLD": 20, "SPEED_THRESHOLD": 60, "MIN_SPEED": 30, "MAX_SPEED": 70,
            "DISPLAY_DEVICE_ID": 47, "TARGET_SPEED": 70, "VSL_ID": 31},
        65.3: {"MIN_VOLUME": 20, "OCCUPANCY_THRESHOLD": 20, "SPEED_THRESHOLD": 60, "MIN_SPEED": 30, "MAX_SPEED": 70,
            "DISPLAY_DEVICE_ID": 49, "TARGET_SPEED": 70, "VSL_ID": 26},
        65.8: {"MIN_VOLUME": 20, "OCCUPANCY_THRESHOLD": 20, "SPEED_THRESHOLD": 60, "MIN_SPEED": 30, "MAX_SPEED": 70,
            "DISPLAY_DEVICE_ID": 51, "TARGET_SPEED": 70, "VSL_ID": 40},
```

```
        66.4: {"MIN_VOLUME": 20, "OCCUPANCY_THRESHOLD": 20, "SPEED_THRESHOLD": 60, "MIN_SPEED": 30, "MAX_SPEED": 70,
              "DISPLAY_DEVICE_ID": 53, "TARGET_SPEED": 70, "VSL_ID": 25},
        67.0: {"MIN_VOLUME": 20, "OCCUPANCY_THRESHOLD": 20, "SPEED_THRESHOLD": 60, "MIN_SPEED": 30, "MAX_SPEED": 70,
              "DISPLAY_DEVICE_ID": 56, "TARGET_SPEED": 70, "VSL_ID": 3},
        67.6: {"MIN_VOLUME": 20, "OCCUPANCY_THRESHOLD": 20, "SPEED_THRESHOLD": 60, "MIN_SPEED": 30, "MAX_SPEED": 70,
              "DISPLAY_DEVICE_ID": 58, "TARGET_SPEED": 70, "VSL_ID": 41},
        68.3: {"MIN_VOLUME": 20, "OCCUPANCY_THRESHOLD": 20, "SPEED_THRESHOLD": 60, "MIN_SPEED": 30, "MAX_SPEED": 70,
              "DISPLAY_DEVICE_ID": 60, "TARGET_SPEED": 70, "VSL_ID": 13},
        68.9: {"MIN_VOLUME": 20, "OCCUPANCY_THRESHOLD": 20, "SPEED_THRESHOLD": 60, "MIN_SPEED": 30, "MAX_SPEED": 70,
              "DISPLAY_DEVICE_ID": 63, "TARGET_SPEED": 70, "VSL_ID": 21},
        69.5: {"MIN_VOLUME": 20, "OCCUPANCY_THRESHOLD": 20, "SPEED_THRESHOLD": 60, "MIN_SPEED": 30, "MAX_SPEED": 70,
              "DISPLAY_DEVICE_ID": 65, "TARGET_SPEED": 70, "VSL_ID": 24},
        69.9: {"MIN_VOLUME": 20, "OCCUPANCY_THRESHOLD": 20, "SPEED_THRESHOLD": 60, "MIN_SPEED": 30, "MAX_SPEED": 70,
              "DISPLAY_DEVICE_ID": 66, "TARGET_SPEED": 70, "VSL_ID": 45},
        70.3: {"MIN_VOLUME": 20, "OCCUPANCY_THRESHOLD": 20, "SPEED_THRESHOLD": 60, "MIN_SPEED": 30, "MAX_SPEED": 70,
              "DISPLAY_DEVICE_ID": 67, "TARGET_SPEED": 70, "VSL_ID": 23}
    }

    # Check which traffic direction
    if direction == 'Westbound':
        message_queue.put(('DEBUG', "EVALUATE: VSL operating on Westbound side."))
        vsl_segments_info = i24w_vsl_segments_info
    else:
        message_queue.put(('DEBUG', "EVALUATE: VSL operating on Eastbound side."))
        vsl_segments_info = i24e_vsl_segments_info
    # Set all vsl segments' TARGET_SPEED to their MAX_SPEED
    reset_vsl_target_speed(vsl_segments_info)

    # Sort i24e vsl segments in an order from downstream to upstream
    if vsl_segments_info == i24e_vsl_segments_info:
        vsl_segments_info = dict(reversed(i24e_vsl_segments_info.items()))
        message_queue.put(('DEBUG', "EVALUATE: VSL items re-sorted moving upstream for Eastbound values."))

    # Iterate over ordered vsl segments starting from the most downstream one
    mile_marker_list = list(vsl_segments_info)
    for index in range(len(mile_marker_list)):
        key = mile_marker_list[index]
        # Real speed for current evaluating vsl segment
        CURRENT_SPEED = get_real_speed(vsl_segments_info, key, vsl_snapshot)
        # Target speed for current evaluating vsl segment
        TARGET_SPEED = vsl_segments_info[key]["TARGET_SPEED"]
        # MIN_SPEED = vsl_segments_info[key]["MIN_SPEED"]
        # MAX_SPEED = vsl_segments_info[key]["MAX_SPEED"]
        # Determine whether activate current vsl or not
        if vsl_activation_condition(vsl_segments_info, key, vsl_snapshot) and CURRENT_SPEED < TARGET_SPEED:
            vsl_segments_info[key]["TARGET_SPEED"] = round_up_five(CURRENT_SPEED)
            # Correct target speed if over min or max
            if vsl_segments_info[key]["TARGET_SPEED"] < vsl_segments_info[key]["MIN_SPEED"]:
                vsl_segments_info[key]["TARGET_SPEED"] = vsl_segments_info[key]["MIN_SPEED"]
                message_queue.put(('DEBUG', "EVALUATE: Target speed was less than min speed at MM {}; target speed "
                                   "re-assigned min speed.".format(key), vsl_segments_info))
            elif vsl_segments_info[key]["TARGET_SPEED"] > vsl_segments_info[key]["MAX_SPEED"]:
                vsl_segments_info[key]["TARGET_SPEED"] = vsl_segments_info[key]["MAX_SPEED"]
                message_queue.put(('DEBUG', "EVALUATE: Target speed was greater than max speed at MM {}; target "
                                   "speed re-assigned max speed.".format(key), vsl_segments_info))

            # Traverse current vsl segment's upstream segments
            for upstream_index in range(index + 1, len(mile_marker_list)):
                # Step down correction
                if vsl_segments_info[mile_marker_list[index]]["TARGET_SPEED"] % 10 == 0:
                    vsl_segments_info[mile_marker_list[upstream_index]]["TARGET_SPEED"] = \
                        vsl_segments_info[mile_marker_list[upstream_index - 1]]["TARGET_SPEED"] + step_down

                else:
                    if upstream_index == index + 1:
```

```
                    vsl_segments_info[mile_marker_list[upstream_index]]["TARGET_SPEED"] = \
                        vsl_segments_info[mile_marker_list[upstream_index - 1]]["TARGET_SPEED"] + int(
                            step_down / 2)
                else:
                    vsl_segments_info[mile_marker_list[upstream_index]]["TARGET_SPEED"] = \
                        vsl_segments_info[mile_marker_list[upstream_index - 1]]["TARGET_SPEED"] + step_down
            # Maximum speed limit constraint
            if vsl_segments_info[mile_marker_list[upstream_index]]["TARGET_SPEED"] >= \
                    vsl_segments_info[mile_marker_list[upstream_index]]["MAX_SPEED"]:
                vsl_segments_info[mile_marker_list[upstream_index]]["TARGET_SPEED"] = \
                    vsl_segments_info[mile_marker_list[upstream_index]]["MAX_SPEED"]

        # Check downstream vsl segments
        if index >= num_bounce_segments:
            if vsl_segments_info[mile_marker_list[index]]["TARGET_SPEED"] < \
                    vsl_segments_info[mile_marker_list[index - 1]]["TARGET_SPEED"]:
                bounce_target_speed = min([vsl_segments_info[mile_marker_list[i]]["TARGET_SPEED"] for i in
                                           range(index - num_bounce_segments, index)])
                for downstream_index in range(index - num_bounce_segments, index):
                    if vsl_segments_info[mile_marker_list[downstream_index]]["TARGET_SPEED"] > \
                            bounce_target_speed:
                        vsl_segments_info[mile_marker_list[downstream_index]]["TARGET_SPEED"] = \
                            bounce_target_speed

    return get_target_speed(vsl_segments_info, 'Westbound')

# Return all our target speeds from our evaluation to the VSL subsystem
westbound_target_speed = swri_vsl_response('Westbound')
eastbound_target_speed = swri_vsl_response('Eastbound')
message_queue.put(('DEBUG', "EVALUATE: Finalized SwRI Eastbound target speeds.", eastbound_target_speed))
message_queue.put(('DEBUG', "EVALUATE: Finalized SwRI Westbound target speeds.", westbound_target_speed))
return {**westbound_target_speed, **eastbound_target_speed}


def compute_lcs_boards(event_id, event_cache, corridor_graph, message_queue):
    """
    Determine LCS config to be sent to SwCS.

    :param event_id: ID of event to be evaluated
    :param event_cache: Shared dictionary of event data; {eventID: (ts, mm, dir, loc, class, descr, status, lanes)}
    :param corridor_graph: I24Graph object instantiated for use by LCS, only; VSL should have its own copy to use
    :param message_queue: Shared message queue that takes messages to log in the messaging subsystem
    :return: List of non-default LCS board suggestions, grouped by gantry; empty list or None if no response desired
    """

    def event_config_extract(lane_blockage_dict):
        """
        Extract the lcs scenario from "lane blockage" feature of the given event data.

        :param lane_blockage_dict: Dictionary of the lane blockage of the event
        :return: Tuple containing a list of the travel lane config and integer of shoulder config
        """
        # Determine travel lane blockage, if any.
        event_travel_lane_config = list(lane_blockage_dict['lane_blockage']['travel'])
        message_queue.put(('DEBUG', "EVALUATE: Determined travel lane blockage to be {}."
                           .format(event_travel_lane_config)))
        # Determine shoulder blockage, if any.
        event_right_shoulder_config = lane_blockage_dict['lane_blockage']['right_shoulder']
        message_queue.put(('DEBUG', "EVALUATE: Determined right shoulder blockage to be {}."
                           .format(event_right_shoulder_config)))
        return event_travel_lane_config, event_right_shoulder_config

    def event_config_to_lcs_signals(travel_lane_config, num_upstream_gantries):
        """
        Return the corresponding LCS signals when given a specific event type
```

```
        LCS signal signs:
        {1: "Closed (red X)",
         2: "Merge (yellow X)",
         3: "Open (green arrow)",
         4: "HOV Only (2+ only)",
         5: "HOV Open (open to all)",
         6: "NextExit",
         7: "Slow Traffic Ahead",
         8: "Reduce Speed Ahead"}
         :param travel_lane_config: configuration of travel lanes in event
         :param num_upstream_gantries: amount of gantries
         :return: List of lists corresponding to lcs signal configs for each gantry
        """

        # Standardized to six lane slots, some of which may not be occupied
        first_upstream_gantry_signals = [0, 0, 0, 0, 0, 0]
        # Create configuration for the first gantry
        for i in range(len(travel_lane_config)):
            lane_status = travel_lane_config[i]
            # Lane blocked
            if lane_status == 0:
                first_upstream_gantry_signals[i] = 'Closed'
            # Lane open
            elif lane_status == 1:
                first_upstream_gantry_signals[i] = 'Open'
            # Lane does not exist
            elif lane_status == -1:
                first_upstream_gantry_signals[i] = -1
        message_queue.put(('DEBUG', "EVALUATE: Signals for first upstream gantry: {}"
                           .format(first_upstream_gantry_signals)))

        # Create configuration for the second gantry if necessary, based on signals of first gantry
        if num_upstream_gantries >= 2:
            second_upstream_gantry_signals = [0, 0, 0, 0, 0, 0]
            for i in range(len(first_upstream_gantry_signals)):
                upstream_signal = first_upstream_gantry_signals[i]
                if upstream_signal == 'Closed':
                    second_upstream_gantry_signals[i] = 'Warning'
                elif upstream_signal == 'Warning':
                    second_upstream_gantry_signals[i] = 'Warning'
                elif upstream_signal == 'Open':
                    second_upstream_gantry_signals[i] = 'Open'
                elif upstream_signal == -1:
                    second_upstream_gantry_signals[i] = -1
            message_queue.put(('DEBUG', "EVALUATE: Signals for second upstream gantry: {}"
                               .format(second_upstream_gantry_signals)))
        else:
            second_upstream_gantry_signals = None
            message_queue.put(('DEBUG', "EVALUATE: No second upstream gantry available."))

        # Create configuration for the third gantry if necessary, based on signals of the second gantry
        if num_upstream_gantries >= 3:
            third_upstream_gantry_signals = [0, 0, 0, 0, 0, 0]
            for i in range(len(second_upstream_gantry_signals)):
                upstream_signal = second_upstream_gantry_signals[i]
                if upstream_signal == 'Closed':
                    third_upstream_gantry_signals[i] = 'Warning'
                elif upstream_signal == 'Warning':
                    third_upstream_gantry_signals[i] = 'Warning'
                elif upstream_signal == 'Open':
                    third_upstream_gantry_signals[i] = 'Open'
                elif upstream_signal == -1:
                    third_upstream_gantry_signals[i] = -1
                message_queue.put(('DEBUG', "EVALUATE: Signals for third upstream gantry: {}"
```

```
                        .format(third_upstream_gantry_signals)))
    else:
        third_upstream_gantry_signals = None
        message_queue.put(('DEBUG', "EVALUATE: No third upstream gantry available."))

    # Create list of the gantry signal configurations
    gantry_signals = [first_upstream_gantry_signals, second_upstream_gantry_signals,
                third_upstream_gantry_signals]

    # If we have more than three gantries to add, copy the third gantry config and extend it
    if num_upstream_gantries > 3:
        extended_num_gantries = num_upstream_gantries - 3
        message_queue.put(('DEBUG', "EVALUATE: Adding {} extra upstream gantries.".format(extended_num_gantries)))
        return gantry_signals + [third_upstream_gantry_signals.copy()] * extended_num_gantries
    else:
        return gantry_signals

# Copy out the current event info, in case it changes in the meantime.
event_info = deepcopy(event_cache[event_id])
event_mm = event_info['mile_marker']
event_direction = event_info['direction']
# Pre-compute travel and right shoulder lane configurations
travel_lane_config, right_shoulder_config = event_config_extract(event_info)

# Storage for individual gantry configuration/message.
lcs_response_plan = {}

# Override number of upstream gantries if flagged.
if config["TOGGLES"]['add_gantry']:
    num_upstream_gantries = 4
else:
    num_upstream_gantries = 3
message_queue.put(('INFO', 'EVALUATE: Number of upstream gantries set to {}.'.format(num_upstream_gantries)))

# Get the exact upstream gantries from the graph.
graph_gantry_list = corridor_graph.find_gantries_upstream(location=event_mm,
                                        roadway_direction=event_direction,
                                        num_gantries=num_upstream_gantries)
message_queue.put(('DEBUG', "EVALUATE: Upstream gantry ids set as {}".format(graph_gantry_list)))

# Get abstract LCS signal patterns, disregarding gantry IDs.
upstream_gantry_signals = event_config_to_lcs_signals(travel_lane_config=travel_lane_config,
                                    num_upstream_gantries=num_upstream_gantries)
message_queue.put(('DEBUG', "EVALUATE: Upstream gantry signals set as {}".format(upstream_gantry_signals)))

for upstream_index, gantry in enumerate(graph_gantry_list):
    gantry_id = gantry['gantry_id']
    lcs_response_plan[gantry_id] = []

    # Get from the graph the board configuration.
    # Graph gantry items always contain 'lane_1'...'lane_6'.
    # `board_config` refers to number of heads on a gantry
    # 1 = board present and linked to lane
    # 0 = lane exists, but no board linked
    # -1 = lane does not exist
    board_config = [gantry['lane_{}'.format(i)] for i in range(1, 7)]
    message_queue.put(('DEBUG', "EVALUATE: Physical existence of lanes determined to be {}".format(board_config)))
    # Signal configuration for this gantry
    general_signal_config = upstream_gantry_signals[upstream_index]
    # Compare the board (graph) at index i to the signal config (plan) at index i.
    for i, board in enumerate(board_config):
        if board == 1:
            if general_signal_config[i] != -1:
                lcs_response_plan[gantry_id].append(general_signal_config[i])
            else:
```

```
                # This case is when there is a lane on this gantry, but not the upstream one. So leave open.
                lcs_response_plan[gantry_id].append('Open')

    # Hard rule for all lanes closed scenario
    if 1 not in travel_lane_config and right_shoulder_config == 1:
        message_queue.put('DEBUG', "EVALUATE: All travel lanes closed, but shoulder open.")
        # All travel lanes are closed, but shoulder is still open.
        for gantry in graph_gantry_list:
            gantry_id = gantry['gantry_id']
            lcs_response_plan[gantry_id].append('Open')
    elif 1 not in travel_lane_config and right_shoulder_config == 0:
        # Everything, shoulder included, is closed.
        message_queue.put(('DEBUG', "EVALUATE: All lanes (including shoulder) closed."))
        for gantry in graph_gantry_list:
            gantry_id = gantry['gantry_id']
            index = graph_gantry_list.index(gantry)
            if index == 0:
                lcs_response_plan[gantry_id].append('Closed')
            elif index == 1:
                lcs_response_plan[gantry_id].append('NextExit')
            else:
                lcs_response_plan[gantry_id][-1] = 'Open'
                lcs_response_plan[gantry_id].append('NextExit')
    message_queue.put(('INFO', "EVALUATE: Final LCS response plan created: {}".format(lcs_response_plan)))
    return lcs_response_plan


if __name__ == '__main__':
    print("NO CODE TO RUN")
```

B.10 graph.py

```
import networkx as nx
import os
import datetime as dt

"""
Contains class I24Graph, which holds a loaded graph object and holds functions for running queries against the graph.
"""


class I24Graph:
    """
    Wrapper class for I-24 infrastructure graph object and associated "blacklist", which contains sensors or other field
    items that are temporarily out of service or misbehaving that should be excluded from graph computations.
    """

    def __init__(self, graph_directory, blacklist_reload_interval=None):
        self.graph_directory = graph_directory
        self.g = self.load_graph()
        self.last_blacklist_reload_time = None
        self.blacklist = None
        # self.blacklist = self.load_blacklist()
        self.blacklist_reload_interval = blacklist_reload_interval
        self.gantry_west_list = list() # always at downstream direction
        self.gantry_east_list = list() # always at downstream direction
        self.rds_west_list = list() # always at downstream direction
        self.rds_east_list = list() # always at downstream direction
        self.set_gantry_list()
        self.set_rds_list()


    def ready_to_reload(self):
```

```
        """Determines if blacklist is ready to reload."""
        if self.last_blacklist_reload_time is not None:
            secs = (dt.datetime.now() - self.last_blacklist_reload_time).total_seconds()
            if secs / 60 > self.blacklist_reload_interval:
                return True
            else:
                return False

    def get_graph(self):
        """Fetches most recent version of the graph."""
        if self.blacklist_reload_interval is not None and self.ready_to_reload():
            self.blacklist = self.load_blacklist()
        return self.g

    def set_gantry_list(self):
        """Sets relative ordering of gantries in gantry_east_list and gantry_west_list."""
        for edge in self.g.edges.data("gantries"): # edge: (node_1, node_2, gantry_info) in a tuple
            if len(edge[2]) != 0: # if an edge has no gantry, the gantry_info term is []
                for gantry_item in edge[2]:
                    assert gantry_item['side'] in {"West", "East"}, f"gantry_item['side'] returns {gantry_item['side']}"
                    if gantry_item['side'] == "West":
                        self.gantry_west_list.append(gantry_item)
                    else:
                        self.gantry_east_list.append(gantry_item)
        self.gantry_west_list.sort(key=lambda d: d['mm'], reverse=True) # in prevention that graph data is messed up
        self.gantry_east_list.sort(key=lambda d: d['mm'], reverse=False)
        pass

    def set_rds_list(self): # rds is recorded in different dict based on the side of the road
        """Sets relative ordering of RDS units in lists rds_east_list and rds_west_list."""
        # west and east are sorted by whether a unit can detect the westbound/eastbound traffic
        for edge in self.g.edges.data("west_rds_units"): # edge: (node_1, node_2, rds_info) in a tuple
            if len(edge[2]) != 0:
                for rds_item in edge[2]:
                    assert rds_item['sides'] in {"West", "Both"}, f"rds_item['sides'] returns {rds_item['sides']}"
                    self.rds_west_list.append(rds_item)

        for edge in self.g.edges.data("east_rds_units"):
            if len(edge[2]) != 0:
                for rds_item in edge[2]:
                    assert rds_item['sides'] in {"Both", "East"}, f"rds_item['sides'] returns {rds_item['sides']}"
                    self.rds_east_list.append(rds_item)

        self.rds_west_list.sort(key=lambda d: d['mm'], reverse=True)
        self.rds_east_list.sort(key=lambda d: d['mm'], reverse=False)
        pass

    def load_graph(self, version=None):
        """
        Loads from file and returns a graph in the designated directory.

        Filenames should be graph_%v.gpickle, where %v is the version of the graph in that file as an integer.
        Default returns a graph with the largest version number, but a specific version may be specified as a parameter.

        :param version: Specific version to search for in the graph storage directory.
        :return: NetworkX graph object (subtype agnostic) loaded from file.
        """
        graph_filenames = []
        for fn in os.listdir(self.graph_directory):
            fnb, fne = os.path.splitext(os.path.basename(fn))
            if fne == '.gpickle' and 'graph' in fnb:
                graph_filenames.append((int(fnb.split('_')[1]), os.path.join(self.graph_directory, fn)))
        if len(graph_filenames) == 0:
            raise FileNotFoundError("No graphs found in graph directory ({}).".format(self.graph_directory))
        if version is not None:
```

```
        version_lookup = dict(graph_filenames)
        if version not in version_lookup.keys():
            raise FileNotFoundError("Specific version () not found in graph directory.".format(version))
        max_graph_fn = version_lookup[version]
    else:
        max_graph_fn = max(graph_filenames)[1]

    return nx.read_gpickle(max_graph_fn)

def load_blacklist(self):
    """Blacklist not in use temporarily - 11/3/2022"""
    pass
    # blacklist_filename = 'blacklist.txt'
    # self.last_blacklist_reload_time = dt.datetime.now()
    # if blacklist_filename not in os.listdir(self.graph_directory):
    # raise FileNotFoundError("No blacklist found in graph directory ({}).".format(self.graph_directory))
    # else:
    # with open(os.path.join(self.graph_directory, blacklist_filename)) as f:
    # return list(f)

def find_rds_downstream(self, location: float, roadway_direction, num_rds: int):
    """Finds num_rds RDS units downstream on side roadway_direction (roadway_direction) from location.

    :param location: float in range [53.0, 81.0] (e.g. 61.8)
    :param roadway_direction: string "Westbound" or "Eastbound"
    :param num_rds: integer

    :return: a list including rds unit dicts
    """

    assert roadway_direction in {"Westbound", "Eastbound"}, \
        "Direction should be either \"Westbound\" or \"Eastbound\""
    rds_buffer = []
    num_collected = 0
    lookup_list = self.rds_west_list if roadway_direction == "Westbound" else self.rds_east_list

    if roadway_direction == "Eastbound":
        for rds_unit in lookup_list: # downstream order
            if rds_unit["mm"] > location:
                rds_buffer.append(rds_unit)
                num_collected += 1
                if num_collected == num_rds:
                    break
    elif roadway_direction == "Westbound":
        for rds_unit in lookup_list:
            if rds_unit["mm"] < location:
                rds_buffer.append(rds_unit)
                num_collected += 1
                if num_collected == num_rds:
                    break
    return rds_buffer


def find_rds_upstream(self, location, roadway_direction, num_rds):
    """Finds num_rds RDS units upstream on side roadway_direction (roadway_direction) from location.

    :param location: float in range [53.0, 81.0] (e.g. 61.8)
    :param roadway_direction: string "Westbound" or "Eastbound"
    :param num_rds: integer

    :return: a list including rds unit dicts
    """
    assert roadway_direction in {"Westbound", "Eastbound"}, \
        "Direction should be either \"Westbound\" or \"Eastbound\""
    rds_buffer = []
```

```
        num_collected = 0
        lookup_list = self.rds_west_list if roadway_direction == "Westbound" else self.rds_east_list

        if roadway_direction == "Eastbound":
            for rds_unit in list(reversed(lookup_list)): # reverse order to upstream
                if rds_unit["mm"] < location:
                    rds_buffer.append(rds_unit)
                    num_collected += 1
                    if num_collected == num_rds:
                        break
        elif roadway_direction == "Westbound":
            for rds_unit in list(reversed(lookup_list)):
                if rds_unit["mm"] > location:
                    rds_buffer.append(rds_unit)
                    num_collected += 1
                    if num_collected == num_rds:
                        break

        return rds_buffer


    def find_gantries_downstream(self, location, roadway_direction, num_gantries):
        """Finds num_gantries gantry units downstream on side roadway_direction (roadway_direction) from location.

        :param location: float in range [53.0, 81.0] (e.g. 61.8)
        :param roadway_direction: string "Westbound" or "Eastbound"
        :param num_gantries: integer

        :return: a list including gantry unit dicts
        """
        assert roadway_direction in {"Westbound", "Eastbound"}, \
            "Direction should be either \"Westbound\" or \"Eastbound\""
        gantry_buffer = []
        num_collected = 0
        lookup_list = self.gantry_west_list if roadway_direction == "Westbound" else self.gantry_east_list

        if roadway_direction == "Eastbound":
            for gantry_unit in lookup_list: # downstream order
                if gantry_unit["mm"] > location:
                    gantry_buffer.append(gantry_unit)
                    num_collected += 1
                    if num_collected == num_gantries:
                        break
        elif roadway_direction == "Westbound":
            for gantry_unit in lookup_list:
                if gantry_unit["mm"] < location:
                    gantry_buffer.append(gantry_unit)
                    num_collected += 1
                    if num_collected == num_gantries:
                        break

        return gantry_buffer


    def find_gantries_upstream(self, location, roadway_direction, num_gantries):
        """Finds num_gantries gantry units upstream on side roadway_direction (roadway_direction) from location.

        :param location: float in range [53.0, 81.0] (e.g. 61.8)
        :param roadway_direction: string "Westbound" or "Eastbound"
        :param num_gantries: integer

        :return: a list including gantry unit dicts
        """
        assert roadway_direction in {"Westbound", "Eastbound"}, \
            "Direction should be either \"Westbound\" or \"Eastbound\""
```

```
        gantry_buffer = []
        num_collected = 0
        lookup_list = self.gantry_west_list if roadway_direction == "Westbound" else self.gantry_east_list

        if roadway_direction == "Eastbound":
            for gantry_unit in list(reversed(lookup_list)): # reverse order to upstream
                if gantry_unit["mm"] < location:
                    gantry_buffer.append(gantry_unit)
                    num_collected += 1
                    if num_collected == num_gantries:
                        break
        elif roadway_direction == "Westbound":
            for gantry_unit in list(reversed(lookup_list)):
                if gantry_unit["mm"] > location:
                    gantry_buffer.append(gantry_unit)
                    num_collected += 1
                    if num_collected == num_gantries:
                        break

        return gantry_buffer

if __name__ == '__main__':
    # For debugging
    graph = I24Graph(os.getcwd())
    graph.find_gantries_upstream(62.86, "Eastbound", 3)
```

B.11 log_writer.py

```
# -----------------------------
"""
Contains the logging backend for the AI-DSS.
"""
__file__ = 'log_writer.py'
# -----------------------------
import logging
import socket
from logging.handlers import SysLogHandler
from logstash_async.handler import AsynchronousLogstashHandler
from logstash_async.formatter import LogstashFormatter
import ecs_logging
import sys
import os
import struct
import datetime as dt
from typing import Union, Mapping

from config.get_config import base_config

# The below line has been commented out because it causes an error in Linux
# Address = tuple[str, int]

levels = {'CRITICAL': logging.CRITICAL, 'ERROR': logging.ERROR, 'WARNING': logging.WARNING,
          'INFO': logging.INFO, 'DEBUG': logging.DEBUG, None: None}
# Conversion from Python logging numbers (equivalent to strings) to SwRI StatusLogger defined levels.
sl_levelno_mapping = {0: 4, 10: 3, 20: 2, 30: 1, 40: 0, 50: 0}


class MaxLevelFilter(object):
    """
    Filter for keeping log records of a given level or LOWER (as opposed to the normal 'or higher' functionality).
    Does not inherit from logging.Filter, since we need our own __init__ to keep track of the max level.
    Inspired from: https://pythonexamples.org/python-logging-info
    """
```

```
36
37      def __init__(self, level):
38          """
39          Establish the filter with maximum log level to keep.
40          :param level: maximum logging level (e.g., logging.INFO) to allow through the filter
41          """
42          self.__max_level = level
43
44      def __call__(self, log_record: logging.LogRecord) -> bool:
45          """
46          Filter function, implemented as the direct call of this object.
47          :param log_record: logging.LogRecord that contains all the relevant fields and functionality.
48          :return:
49          """
50          return log_record.levelno <= self.__max_level
51
52
53  class ExtraLogger(logging.Logger):
54      """
55      Subclass of logging.Logger that adds "extra" log record information passed as a dictionary as 1) unpacked individual
56          LogRecord attributes (default behavior) and 2) as a single attribute that contains the entire dictionary. This
57          feature is needed in order to unify the logging interface between different code modules that will want to
58          include different "extra" fields depending on context.
59      Inspired from: https://devdreamz.com/question/710484-python-logging-logger-overriding-makerecord
60      """
61
62      def makeRecord(self, name: str, level: int, fn: str, lno: int, msg: object, args, exc_info,
63                     func: Union[str, None] = None, extra: Union[Mapping[str, object], None] = None,
64                     sinfo: Union[str, None] = None) -> logging.LogRecord:
65          """
66          Overrides `makeRecord` in logging.Logger in order to add a single feature: add the attribute 'extra' to each
67              LogRecord that is created and set its value as the entire "extra" dictionary that is passed to the log
68              function that initiated the record creation. The "extra" dictionary still gets unpacked and added as
69              individual attributes through the call to super.makeRecord(...).
70          :param name: passed straight to the LogRecord factory, which by default is the LogRecord class
71          :param level: passed straight to the LogRecord factory, which by default is the LogRecord class
72          :param fn: passed straight to the LogRecord factory, which by default is the LogRecord class
73          :param lno: passed straight to the LogRecord factory, which by default is the LogRecord class
74          :param msg: passed straight to the LogRecord factory, which by default is the LogRecord class
75          :param args: passed straight to the LogRecord factory, which by default is the LogRecord class
76          :param exc_info: passed straight to the LogRecord factory, which by default is the LogRecord class
77          :param func: passed straight to the LogRecord factory, which by default is the LogRecord class
78          :param extra: a dictionary of extra log information that is contextual to the code module logging call
79          :param sinfo: passed straight to the LogRecord factory, which by default is the LogRecord class
80          :return: LogRecord with the desired 'extra' attribute and unpacked "extra" values
81          """
82          # Make the call to the normal `makeRecord` function, which will do the default behavior
83          # DEREK: brutish fix, use logging.Logger.makeRecord as a static method
84          rv = logging.Logger.makeRecord(None, name=name, level=level, fn=fn, lno=lno, msg=msg, args=args,
85                                         exc_info=exc_info, func=func, extra=extra, sinfo=sinfo)
86          # Also add the complete "extra" dictionary as an attribute
87          rv.__dict__['extra'] = extra
88          return rv
89
90
91  class StatusLoggerHandler(logging.handlers.SocketHandler):
92      """
93      Send log messages as the defined StatusLogger format, defined below.
94
95      StatusLogger format: [ version (4 bytes UINT) -- OLE timestamp (8 bytes DOUBLE) -- log level (4 bytes UINT) --
96          log classification (4 bytes UINT) -- error code (4 bytes INT) -- app name (0-18 bytes MFC string) --
97          user name (0-6 bytes MFC string) -- event ID (variable bytes MFC string) -- event descr. (var. bytes MFC string)
98          -- message (var. bytes MFC string) ]
99      """
100
```

```
    # def __init__(self, *args, **kwargs):
    # super(StatusLoggerHandler, self).__init__(*args, **kwargs)

    @staticmethod
    def mfc_string(string):
        """
        Format a string in MFC standard byte format.

            For len < 255: single byte length value + string bytes
            For len < 65534: 0xFF + 2-byte length value + string bytes
            Else: 0xFF 0xFF 0xFF + 4-byte length value + string bytes
        :param string: the string to format
        :return: bytes of length indicators and string
        """
        if len(string) < 255:
            return struct.pack('B', len(string)) + bytes(string, 'utf-8')
        elif len(string) < 65534:
            return b'\xff' + struct.pack('H', len(string)) + bytes(string, 'utf-8')
        else:
            return b'\xff\xff\xff' + struct.pack('I', len(string)) + bytes(string, 'utf-8')

    @staticmethod
    def ole_timestamp(timestamp):
        """
        Convert a datetime object into an OLE timestamp.

        OLE timestamp has integer number of days since epoch, plus decimal portion of the seconds elapsed in the day.
        :param timestamp: datetime.datetime object
        :return: float for OLE timestamp
        """
        old_datum = dt.datetime(1899, 12, 30)
        delta = timestamp - old_datum
        return float(delta.days) + (float(delta.seconds) / 86400)

    def makePickle(self, record: logging.LogRecord):
        """
        Make a bytes representation of the message to send to StatusLogger through a custom implementation.

        :param record: a LogRecord object, assumed to be constructed using the ExtraLogger factory implemented here
        :return: buffer of bytes
        """
        ts = self.ole_timestamp(dt.datetime.fromtimestamp(record.created))
        # Start with the version, timestamp as OLE date, log level number, log classification, error code
        buf = struct.pack('<IdIII', 1, ts, sl_levelno_mapping[record.levelno], 0, 0)
        # Add app name, user name, and host name
        buf += self.mfc_string('AI-DSS') + self.mfc_string(os.getlogin()) + self.mfc_string(os.uname().nodename)
        # Add event ID, event description
        buf += self.mfc_string('event ID') + self.mfc_string('event description')
        # buf += self.mfc_string(record.extra.get('eventID', 'None'))
        # buf += self.mfc_string(record.extra.get('eventDesc', 'None'))
        buf += self.mfc_string(record.msg)
        buf = bytearray(buf)
        return buf


class I24Logger:
    """
    This unified interface is used to abstract log setup from other code modules,
        which we want to have consistent behavior.
    """

    # Python 3.8 compatibility mode...add other typehints if minimum version changes.
    def __init__(self, log_name: str = None, processing_environment: str = None,
                 connect_logstash: bool = False, connect_file: bool = False,
                 connect_syslog: bool = False, connect_console: bool = False, connect_sl: bool = False,
```

```
            logstash_address=None, sl_address=None, syslog_location: str = None,
            all_log_level: str = 'WARNING', logstash_log_level=None, file_log_level=None,
            syslog_log_level=None, console_log_level=None, sl_log_level=None):
        """
        Constructor of the persistent logging interface. It establishes a custom multi-destination logger with the
           option to log different levels to different destinations.
        :param log_name:
        :param processing_environment:
        :param connect_logstash: True/False to connect to Logstash via asynchronous handler.
        :param connect_file: True/False to connect a simple log file (non-rotating) to this logger. If multiple loggers
           are instantiated, multiple files will be produced and need to be differentiated by `file_path`.
        :param connect_syslog: True/False to connect to the host computer's syslog via TCP Socket Stream.
        :param connect_console: True/False to connect to the STDOUT and STDERR available via `sys` package.
        :param connect_sl: True/False to connect to SwRI StatusLogger.
        :param logstash_address: (host, port) tuple for Logstash connection.
        :param sl_address: (host, port) tuple for StatusLogger connection.
        :param syslog_location: Path to syslog.
        :param all_log_level: Available to set a global log level across all handlers; overridden by handler-specific.
        :param logstash_log_level: Logstash log level as string; overrides `all_log_level`.
        :param file_log_level: File log level as string; overrides `all_log_level`.
        :param syslog_log_level: Syslog log level as string; overrides `all_log_level`.
        :param console_log_level: Console log level as string; overrides `all_log_level`.
        :param sl_log_level: StatusLogger log level as string (Python convention); overrides `all_log_level`.
        """
        # There are multiple default LogRecord attributes that are populated automatically, so we don't need to
        # duplicate this functionality unless it's not working for us.
        # - LogRecord.process: process ID (if available, acquired from `os.getpid()`)
        # - LogRecord.processName: process name (default='MainProcess', acquired from `mp.current_process().name`)
        # - LogRecord.thread: thread ID (default=None, acquired from `threading.get_ident()`)
        # - LogRecord.threadName: thread name (default=None, acquired from `threading.current_thread().name`)
        # - LogRecord.filename/pathname: file/path of source file (where this comes from is complicated)
        # - LogRecord.module: filename without extension

        # We have to give the logger a name, but the actual process name is populated in LogRecords automatically.
        self._name = log_name

        self._hostname = socket.gethostname()
        self._environment = processing_environment if processing_environment is not None else 'DEF_ENV'

        self._logstash_addr = logstash_address
        self._statuslogger_addr = sl_address
        self._logfile_path = os.path.join(base_config['install_path'], base_config['logs_path_join'], "aidss.log")
        self._syslog_location = syslog_location
        self._temporary_debug_file_handler = None

        # No need to put in owner process name/ID or parent, since this information will be in the logger name or
        # the LogRecord attributes.
        self._default_logger_extra = {'host': self._hostname, 'env': self._environment}

        if not all([ll in levels.keys() for ll in
                    (logstash_log_level, file_log_level, syslog_log_level, console_log_level, sl_log_level)]):
            raise ValueError("Invalid log level specified. Use: 'CRITICAL', 'ERROR', 'WARNING', 'INFO', 'DEBUG', None.")
        self._log_levels = {'logstash': (levels[logstash_log_level] if logstash_log_level is not None
                                         else levels[all_log_level]),
                            'file': (levels[file_log_level] if file_log_level is not None
                                     else levels[all_log_level]),
                            'syslog': (levels[syslog_log_level] if syslog_log_level is not None
                                       else levels[all_log_level]),
                            'console': (levels[console_log_level] if console_log_level is not None
                                        else levels[all_log_level]),
                            'statuslogger': (levels[sl_log_level] if sl_log_level is not None
                                             else levels[all_log_level]),
                            }
        self._connect = {'logstash': connect_logstash, 'file': connect_file, 'syslog': connect_syslog,
                         'console': connect_console, 'statuslogger': connect_sl}
```

```
231
232         if self._connect['logstash'] is True and self._log_levels['logstash'] is None:
233             raise ValueError("Logstash logging activated, but no log level specified during construction.")
234         if self._connect['file'] is True and self._log_levels['file'] is None:
235             raise ValueError("File logging activated, but no log level specified during construction.")
236         if self._connect['syslog'] is True and self._log_levels['syslog'] is None:
237             raise ValueError("Syslog logging activated, but no log level specified during construction.")
238         if self._connect['console'] is True and self._log_levels['console'] is None:
239             raise ValueError("Console logging activated, but no log level specified during construction.")
240         if self._connect['statuslogger'] is True and self._log_levels['statuslogger'] is None:
241             raise ValueError("StatusLogger logging activated, but no log level specified during construction.")
242
243         if self._connect['logstash'] is True and self._logstash_addr is None:
244             raise ValueError("Logstash logging activated, but no connection address given (host, port).")
245         if self._connect['file'] is True and (self._logfile_path is None or self._logfile_path == ''):
246             raise ValueError("File logging activated, but no file path given.")
247         if self._connect['syslog'] is True and self._syslog_location is None:
248             raise ValueError("Syslog logging activated, but no location (path or host/port tuple) given.")
249         if self._connect['statuslogger'] is True and self._statuslogger_addr is None:
250             raise ValueError("StatusLogger logging activated, but no connection address given (host, port).")
251
252         logging.setLoggerClass(ExtraLogger)
253         self._logger = logging.getLogger(self._name)
254
255         self._logger.propagate = False
256         # Set overall logger level at the minimum of the specified levels (no need to set it any lower).
257         self._logger.setLevel(min(self._log_levels.values()))
258
259         if self._connect['logstash'] is True:
260             self._setup_logstash()
261         if self._connect['file'] is True:
262             self._setup_regular_file()
263         if self._connect['syslog'] is True:
264             self._setup_syslog()
265         if self._connect['console'] is True:
266             self._setup_console()
267         if self._connect['statuslogger'] is True:
268             self._setup_statuslogger()
269
270     def connect_logstash(self, logstash_address, logstash_log_level=None):
271         """
272         External-access function for setting up Logstash AFTER construction of I24Logger.
273         :param logstash_address: Since Logstash was not set up at construction, need to pass in (host, port).
274         :param logstash_log_level: Logstash log level as string; overrides any level specified in constructor for LS.
275         :return: None
276         """
277         if self._connect['logstash'] is True:
278             self.warning("Logstash logging is already connected!")
279             return
280         self._connect['logstash'] = True
281         self._logstash_addr = logstash_address
282         if logstash_log_level is not None:
283             self._log_levels['logstash'] = levels[logstash_log_level]
284         self._setup_logstash()
285
286     def connect_syslog(self, syslog_location, syslog_log_level=None):
287         """
288         External-access function for setting up syslog AFTER construction of I24Logger.
289         :param syslog_location: Since syslog was not set up at construction, need to pass in its location.
290         :param syslog_log_level: Syslog log level as string; overrides any level specified in constructor for syslog.
291         :return: None
292         """
293         if self._connect['syslog'] is True:
294             self.warning("Syslog logging is already connected!")
295             return
```

```
        self._connect['syslog'] = True
        self._syslog_location = syslog_location
        if syslog_log_level is not None:
            self._log_levels['syslog'] = levels[syslog_log_level]
        self._setup_syslog()

    def connect_file(self, file_path, file_log_level=None):
        """
        External-access function for setting up Logstash AFTER construction of I24Logger.
        :param file_path: Since file log was not set up at construction, need to pass in a path for it.
        :param file_log_level: File log level as string; overrides any level specified in constructor for file.
        :return: None
        """
        if self._connect['file'] is True:
            self.warning("File logging is already connected!")
            return
        self._connect['file'] = True
        self._logfile_path = file_path
        if file_log_level is not None:
            self._log_levels['file'] = levels[file_log_level]
        self._setup_regular_file()

    def connect_console(self, console_log_level=None):
        """
        External-access function for setting up console AFTER construction of I24Logger.
        :param console_log_level: Console log level as string; overrides any level specified in constructor for console.
        :return: None
        """
        if self._connect['console'] is True:
            self.warning("Console logging is already connected!")
            return
        self._connect['console'] = True
        if console_log_level is not None:
            self._log_levels['console'] = levels[console_log_level]
        self._setup_console()

    def connect_statuslogger(self, sl_address, sl_log_level=None):
        """
        External-access function for setting up StatusLogger AFTER construction of I24Logger.
        :param sl_address: Since StatusLogger was not setup during construction, need (host, port) address.
        :param sl_log_level: StatusLogger log level as string (Python convention); overrides any level specified in
            constructor for StatusLogger.
        :return: None
        """
        if self._connect['statuslogger'] is True:
            self.warning("StatusLogger logging is already connected!")
            return
        self._connect['statuslogger'] = True
        self._statuslogger_addr = sl_address
        if sl_log_level is not None:
            self._log_levels['statuslogger'] = levels[sl_log_level]
        self._setup_statuslogger()

    def _setup_logstash(self):
        """
        Attaches a Logstash asynchronous handler, which executes transactions without blocking primary code. Uses
            connection information given in the I24Logger constructor. Log level is also set in the constructor.
            Formatter is currently the LogstashFormatter with only 'message_type='python-logstash'', which appears
             to be purely cosmetic and not a behavior change.
        :return: None
        """
        # Set database_path to None to use in-memory caching.
        logstash_host, logstash_port = self._logstash_addr
        lsth = AsynchronousLogstashHandler(logstash_host, logstash_port, database_path=None)
        lsth.setLevel(self._log_levels['logstash'])
```

```
        # Not using the "extra" feature of the LogstashFormatter, since we already have the desired merge behavior
        # in our own logger object.
        lstf = LogstashFormatter(message_type='python-logstash', extra_prefix=None)
        lsth.setFormatter(lstf)
        self._logger.addHandler(lsth)

    def _setup_syslog(self, elastic_format: bool = False):
        """
        Attaches a syslog handler for this machine. The path of the syslog is needed in the I24Logger constructor, since
            platforms have different destinations (e.g., Mac appears to be '/var/run/syslog' and Linux is usually
            '/var/log/syslog'). There are two formatting options: ECS, which makes logs easily importable into Elastic,
            and a default time/level/name/message/extra line format.
        :param elastic_format: True/False to use Elastic-compatible formatting.
        :return: None
        """
        sysh = SysLogHandler(address=self._syslog_location, socktype=socket.SOCK_STREAM)
        sysh.setLevel(self._log_levels['syslog'])
        if elastic_format is True:
            ecsfmt = ecs_logging.StdlibFormatter()
            sysh.setFormatter(ecsfmt)
        else:
            # Other fields may include: %(module)s, %(processName)s, %(thread)d, %(threadName)s
            fmtstr = '%(asctime)s | %(levelname)s | %(name)s | %(process)d | %(message)s | %(extra)s'
            exfmt = logging.Formatter(fmtstr)
            sysh.setFormatter(exfmt)
        self._logger.addHandler(sysh)

    def _setup_regular_file(self, elastic_format: bool = False):
        """
        Attaches a timed rotating file handler that rolls over at midnight. The file path is given during I24Logger
            construction and backup files are given the datetime as an extension. Formatting is by default a simple
            line of information that is easily readble, but can also be made compatible with Elastic.
        :param elastic_format: True/False to use Elastic-compatible formatting.
        :return: None
        """
        rflh = logging.handlers.TimedRotatingFileHandler(filename=self._logfile_path, when='midnight', interval=1,
                                            backupCount=31, utc=False, delay=False)
        rflh.setLevel(self._log_levels['file'])
        if elastic_format is True:
            ecsfmt = ecs_logging.StdlibFormatter()
            rflh.setFormatter(ecsfmt)
        else:
            # Other fields may include: %(module)s, %(processName)s, %(thread)d, %(threadName)s
            # Process ID (%(process)d) was not included, since the files are separated already by process.
            fmtstr = '%(asctime)s | %(levelname)s | %(name)s | %(message)s | %(extra)s'
            exfmt = logging.Formatter(fmtstr)
            rflh.setFormatter(exfmt)
        self._logger.addHandler(rflh)

    def _setup_console(self, stdout_max_level=logging.INFO):
        """
        Attaches a STDOUT/STDERR handler. Messages at INFO/DEBUG level are handled through STDOUT and WARNING and higher
            are handled through STDERR in order to take advantage of typically built-in formatting (e.g., red text).
            That filtering is accomplished through the custom MaxLevelFilter, which can be set with `stdout_max_level`.
        :param stdout_max_level: Option to set STDOUT max log level, everything higher goes to STDERR. *Not currently
            configurable/implemented in constructor.*
        :return: None
        """
        if stdout_max_level not in (logging.DEBUG, logging.INFO, logging.WARNING, logging.ERROR, logging.CRITICAL):
            raise ValueError("Must provide valid logging level for maximum log level to STDOUT.")
        fmtstr = '%(levelname)s | %(name)s | %(process)d | %(message)s ' # | %(extra)s'
        csfmt = logging.Formatter(fmtstr)
        if self._log_levels['console'] <= logging.INFO:
            outh = logging.StreamHandler(stream=sys.stdout)
            outh.setLevel(self._log_levels['console'])
```

```
            outh.addFilter(filter=MaxLevelFilter(level=stdout_max_level))
            outh.setFormatter(csfmt)
            self._logger.addHandler(outh)
        errh = logging.StreamHandler(stream=sys.stderr)
        errh.setLevel(max(self._log_levels['console'], logging.WARNING))
        errh.setFormatter(csfmt)
        self._logger.addHandler(errh)

    def _setup_statuslogger(self):
        """
        Create an instance of the custom StatusLoggerHandler class, which is a subclass of SocketHandler. Uses
            connection information given in the I24Logger constructor. Log level is also set in the constructor.
            Formatter is implicit in `makePickle` function, which packs log messages in bytes representation for
            transmittal to StatusLogger.
        :return: None
        """
        sl_host, sl_port = self._statuslogger_addr
        slh = StatusLoggerHandler(host=sl_host, port=sl_port)
        slh.setLevel(self._log_levels['statuslogger'])
        self._logger.addHandler(slh)

    def _setup_debug_file(self, elastic_format: bool = False):
        """
        Attaches a non-rotating file handler. The file path is given during I24Logger construction. Formatting is by
            default a simple line of information that is easily readble, but can also be made compatible with Elastic.
        :param elastic_format: True/False to use Elastic-compatible formatting.
        :return: None
        """
        dflh = logging.FileHandler(
            filename=os.path.join(base_config['install_path'], base_config['logs_path_join'],
                                  'DEBUG_{}.log'.format(dt.datetime.now().strftime('%Y-%m-%d_%H-%M-%S'))))
        dflh.setLevel(logging.DEBUG)
        if elastic_format is True:
            ecsfmt = ecs_logging.StdlibFormatter()
            dflh.setFormatter(ecsfmt)
        else:
            # Other fields may include: %(module)s, %(processName)s, %(thread)d, %(threadName)s
            # Process ID (%(process)d) was not included, since the files are separated already by process.
            fmtstr = '%(asctime)s | %(levelname)s | %(name)s | %(message)s | %(extra)s'
            exfmt = logging.Formatter(fmtstr)
            dflh.setFormatter(exfmt)
        return dflh

    def set_temporary_debug(self):
        if self._temporary_debug_file_handler is not None:
            self.warning("Asked to set temporary debug but the debug file handler class variable is not None.")
            self._temporary_debug_file_handler.close()
            self.warning("Closed debug file handler to make room for a new one.")
        self._temporary_debug_file_handler = self._setup_debug_file()
        self._logger.addHandler(self._temporary_debug_file_handler)
        self.warning("Created temporary single file handler for DEBUG logs at 'DEBUG_%Y-%m-%d_%H-%M-%S.log'")
        for handler in self._logger.handlers:
            if isinstance(handler, AsynchronousLogstashHandler) or isinstance(handler, StatusLoggerHandler):
                handler.setLevel(logging.DEBUG)
                self.warning("Set {} handler to DEBUG level.".format(type(handler)))

    def unset_temporary_debug(self):
        print("{} active handlers".format(len(self._logger.handlers)))
        self._temporary_debug_file_handler.close()
        self._temporary_debug_file_handler = None
        self.warning("Closed temporary DEBUG log file.")
        print("{} active handlers".format(len(self._logger.handlers)))
        for handler in self._logger.handlers:
            if isinstance(handler, AsynchronousLogstashHandler):
                handler.setLevel(self._log_levels['logstash'])
```

```
            self.warning("Set {} handler back to {} level.".format(
                type(handler), logging.getLevelName(self._log_levels['logstash'])))
        if isinstance(handler, StatusLoggerHandler):
            handler.setLevel(self._log_levels['statuslogger'])
            self.warning("Set {} handler back to {} level.".format(
                type(handler), logging.getLevelName(self._log_levels['statuslogger'])))

    def debug(self, message: Union[str, BaseException], extra: Union[dict, None] = None, exc_info: bool = False):
        """
        Logs a message at the DEBUG level, which is the lowest order of precedence.
        Anything given in `extra` is merged with the values in the I24Logger constructor. This is the location in
            which contextual information should be passed. This allows, particularly in LogStash, this information
            to be separated automatically from the log message and to maintain its type. For example, one might include
            information about processing rate (e.g., frames per second, trajectories per minute) or status of monitored
            assets (e.g., cameras).
        In order to log an exception traceback, pass the exception or a message as `message`, and set `exc_info`=True;
            or write a message and pass the exception object as `exc_info`. Support is available for just setting
            `exc_info`=True and letting `logging` automatically gather the traceback, but it is recommended to be
            explicit about including the exception.
        ```
        try:
            raise ValueError("Parameter invalid.")
        except ValueError as e:
            my_logger.warning(e, exc_info=True) # Option 1
            my_logger.warning("Got an exception!", exc_info=e) # Option 2
        ```
        :param message: Either a log message as a string, or an exception.
        :param extra: Dictionary of extra contextual information about the log message.
        :param exc_info: True/False to automatically include exception info, or the exception itself (recommended).
        :return: None
        """
        extra = extra if extra is not None else {}
        self._logger.debug(message, extra={**self._default_logger_extra, **extra}, exc_info=exc_info)

    def info(self, message: Union[str, BaseException], extra: Union[dict, None] = None, exc_info: bool = False):
        """
        Logs a message at the INFO level. See .debug(...) for more information.
        """
        extra = extra if extra is not None else {}
        self._logger.info(message, extra={**self._default_logger_extra, **extra}, exc_info=exc_info)

    def warning(self, message: Union[str, BaseException], extra: Union[dict, None] = None, exc_info: bool = False):
        """
        Logs a message at the WARNING level. See .debug(...) for more information.
        """
        extra = extra if extra is not None else {}
        self._logger.warning(message, extra={**self._default_logger_extra, **extra}, exc_info=exc_info)

    def error(self, message: Union[str, BaseException], extra: Union[dict, None] = None, exc_info: bool = False):
        """
        Logs a message at the ERROR level. See .debug(...) for more information.
        """
        extra = extra if extra is not None else {}
        self._logger.error(message, extra={**self._default_logger_extra, **extra}, exc_info=exc_info)

    def critical(self, message: Union[str, BaseException], extra: Union[dict, None] = None, exc_info: bool = False):
        """
        Logs a message at the CRITICAL level. See .debug(...) for more information.
        """
        extra = extra if extra is not None else {}
        self._logger.critical(message, extra={**self._default_logger_extra, **extra}, exc_info=exc_info)

    def log(self, level: str, message: Union[str, BaseException],
            extra: Union[dict, None] = None, exc_info: bool = False):
        """
```

```
        Logs a message at the level specified in 'level' (as a string). Otherwise, behavior is the same as .debug(...).
        """
        level_upper = level.upper()
        if level_upper == 'DEBUG':
            self.debug(message=message, extra=extra, exc_info=exc_info)
        elif level_upper == 'INFO':
            self.info(message=message, extra=extra, exc_info=exc_info)
        elif level_upper == 'WARNING':
            self.warning(message=message, extra=extra, exc_info=exc_info)
        elif level_upper == 'ERROR':
            self.error(message=message, extra=extra, exc_info=exc_info)
        elif level_upper == 'CRITICAL':
            self.critical(message=message, extra=extra, exc_info=exc_info)

    def set_name(self, name):
        self._logger.name = name

    def __del__(self):
        for h in reversed(self._logger.handlers):
            h.close()
            try:
                logging._removeHandlerRef(h)
            except:
                pass
            del h
        self._logger.handlers.clear()
        self._logger.handlers = []
        del self._logger
```

B.12 subsys_data.py

```
# ------------------------------
"""
Contains the data subsystem that manages data processes for the AI-DSS.

1. Ingests data for each data type from SwCS
2. Parses data for various necessary information
3. Dumps data to database as caches fill
"""
__file__ = 'subsys_data.py'

# ------------------------------

import multiprocessing
import I24customwebsocket as mws
import pandas as pd
import requests
import time
import datetime
from datetime import timedelta
import xml.etree.ElementTree as ET

from config.get_config import config
from utility import error_handler, xml_parse


def detectors_data_receiver(data_queue: multiprocessing.Queue, message_queue: multiprocessing.Queue, request_url: str,
                            websocket_url: str) -> None:
    """
    Receive data from detector requests and websockets to be processed.

    :param data_queue: Data queue from the manage_data function to temporarily store data before storing in data_cache
    :param message_queue: Shared message queue that takes messages to log in the messaging subsystem
```

```
    :param request_url: URL of HTTP request for initial data pull
    :param websocket_url: URL of TCP websocket for continuous data stream
    :return: None
    """

    def on_open(ws):
        """
        Indicate websocket connection for detectors in data subsystem.

        :param ws: Detectors websocket
        :return: None
        """
        ws.send("Client connected")
        message_queue.put(('INFO', "STARTUP: Detectors websocket connected."))

    def parse_detectors(tree):
        """
        Parse detector XML for most recent changes.

        :param tree: An XML containing the features of the detectors
        :return: None
        """
        if xml_parse(tree, './/roadway') == 'Interstate 24':
            id = int(xml_parse(tree, './/id'))
            message_queue.put(('DEBUG', 'UPDATE: Attempting to parse detectors ID {}'.format(id)))
            direction = xml_parse(tree, './/direction')
            message_queue.put(('DEBUG', 'UPDATE: Detectors ID {} assigned direction {}.'
                               .format(id, direction)))
            latitude = float(xml_parse(tree, './/latitude'))
            message_queue.put(('DEBUG', 'UPDATE: Detectors ID {} assigned latitude {}.'
                               .format(id, latitude)))
            longitude = float(xml_parse(tree, './/longitude'))
            message_queue.put(('DEBUG', 'UPDATE: Detectors ID {} assigned longitude {}.'
                               .format(id, longitude)))
            message_queue.put(('DEBUG', "UPDATE: Detector ID {} update processed.".format(id),
                               {'message_type': 'detectors', 'id': id, 'direction': direction, 'latitude': latitude,
                                'longitude': longitude}))
            data_queue.put(('detectors', id, {'direction': direction, 'latitude': latitude, 'longitude': longitude}))
            message_queue.put(('DEBUG', 'UPDATE: Detector ID {} placed in data queue.'.format(id)))

    def on_detectors_message(ws, message):
        """
        Handle every new message across the detectors websocket for parsing or pings.

        :param ws: Detectors websocket
        :param message: An XML string containing detector updates
        :return: None
        """
        if message == "OK":
            reconnect_count = 0
            message_queue.put(('DEBUG', "DETECTORS PING: Got an OK from SwCS.", {'message_type': 'detectors_ping'}))
            return
        root = ET.fromstring(message)
        message_queue.put(('DEBUG', 'UPDATE: Detectors message root found.', {'XML': str(root)}))
        parse_detectors(tree=root)

    # Delay reconnection to the websocket after multiple failed attempts
    reconnect_count = 0
    reconnect_map = {0: 0, 1: 10, 2: 60, 3: 1800}
    while True:
        time.sleep(reconnect_map[reconnect_count])
        if reconnect_count != 3:
            reconnect_count += 1
            message_queue.put(('DEBUG', 'UPDATE: Detectors reconnect count incremented to {}.'.format(reconnect_count)))
        try:
```

```
98          r = requests.get(url=request_url, headers={'Authorization': 'Bearer banana'})
99          tree = ET.fromstring(r.content)
100         message_queue.put(('DEBUG', 'UPDATE: Got Detectors XML for initial request.', {'XML': str(tree)}))
101         for detector in tree.findall('.//data/detector'):
102            parse_detectors(tree=detector)
103         ws = mws.WebSocketApp(url=websocket_url,
104                               header={'Authorization': 'Bearer banana'},
105                               on_message=on_detectors_message,
106                               on_open=on_open)
107         # Run the websocket until error occurs
108         ws.run_forever(ping_interval=10, ping_timeout=5, ping_payload="OK")
109      except Exception as e:
110         message_queue.put(error_handler(e))
111
112
113 def links_data_receiver(data_queue: multiprocessing.Queue, message_queue: multiprocessing.Queue, request_url: str,
114                         websocket_url: str) -> None:
115    """
116    Receive data from links requests and websockets to be processed.
117
118    :param data_queue: Data queue from the manage_data function to temporarily store data before storing in data_cache
119    :param message_queue: Shared message queue that takes messages to log in the messaging subsystem
120    :param request_url: URL of HTTP request for initial data pull
121    :param websocket_url: URL of TCP websocket for continuous data stream
122    :return: None
123    """
124
125    def on_open(ws):
126       """
127       Indicate websocket connection for links in data subsystem.
128
129       :param ws: Links websocket
130       :return: None
131       """
132       message_queue.put(('INFO', "STARTUP: Links websocket connected."))
133
134    def parse_links(tree):
135       """
136       Parse links XML for most recent changes.
137
138       :param tree: An xml containing the features of the links
139       :return: None
140       """
141       name = xml_parse(tree, './/displayName')
142       if "I24" in name:
143          id = int(xml_parse(tree, './/link/id'))
144          message_queue.put(('DEBUG', 'UPDATE: Attempting to parse Link ID {}'.format(id)))
145          mm_idx = name.rfind('-') + 1
146          rest_of_str = name[mm_idx:]
147          end = name.rfind('(')
148          if end != -1:
149             end -= 2
150             rest_of_str = name[mm_idx:end]
151             direction = name[end]
152          else:
153             direction = name[mm_idx - 2]
154          mm = float(rest_of_str)
155          message_queue.put(('DEBUG', 'UPDATE: Assigned mile marker {} to Link ID {}'.format(mm, id)))
156          if 53 <= mm <= 81:
157             timestamp = xml_parse(tree, './/timestamp')
158             message_queue.put(('DEBUG', 'UPDATE: Assigned timestamp {} to Link ID {}'.format(timestamp, id)))
159             try:
160                speed = int(tree.find('.//linkAvgData/rawData/speed').text)
161                message_queue.put(('DEBUG', 'UPDATE: Assigned speed {} to Link ID {}'.format(speed, id)))
162                vol = int(tree.find('.//linkAvgData/rawData/volume').text)
```

```
                message_queue.put(('DEBUG', 'UPDATE: Assigned volume {} to Link ID {}'.format(vol, id)))
                occ = int(tree.find('.//linkAvgData/rawData/occupancy').text)
                message_queue.put(('DEBUG', 'UPDATE: Assigned occupancy {} to Link ID {}'.format(occ, id)))
                data_queue.put(('links', id, {'mile_marker': mm, 'direction': direction, 'timestamp': timestamp,
                                              'speed': speed, 'vol': vol, 'occ': occ}))
                message_queue.put(('DEBUG', 'UPDATE: Link ID {} placed in data queue.'.format(id)))
            except AttributeError:
                speed, vol, occ = 0, 0, 0
                data_queue.put(('links', id, {'mile_marker': mm, 'direction': direction, 'timestamp': timestamp,
                                              'speed': speed, 'vol': vol, 'occ': occ}))
                message_queue.put(('DEBUG', 'UPDATE: Assigned 0 to speed, occupancy, and volume on Link ID {} and dumped to database.'.format(id)
                      ))

    def on_links_message(ws, message):
        """
        Handle every new message across the links websocket for parsing or pings.

        :param ws: Links websocket
        :param message: An XML string containing links updates
        :return: None
        """
        if message == "OK":
            reconnect_count = 0
            message_queue.put(('DEBUG', "LINKS PING: Got an OK from SwCS.", {'message_type': 'links_ping'}))
            return
        root = ET.fromstring(message)
        message_queue.put(('DEBUG', 'UPDATE: Links message root found.', {'XML': str(root)}))
        parse_links(tree=root)

    # Delay reconnection to the websocket after multiple failed attempts
    reconnect_count = 0
    reconnect_map = {0: 0, 1: 10, 2: 60, 3: 1800}
    while True:
        time.sleep(reconnect_map[reconnect_count])
        if reconnect_count != 3:
            reconnect_count += 1
            message_queue.put(('DEBUG', 'UPDATE: Links reconnect count incremented to {}.'.format(reconnect_count)))
        try:
            r = requests.get(url=request_url, headers={'Authorization': 'Bearer banana'})
            tree = ET.fromstring(r.content)
            message_queue.put(('DEBUG', 'UPDATE: Got Links XML for initial request.', {'XML': str(tree)}))
            for link in tree.findall('.//data/linkAndStatus'):
                parse_links(tree=link)
            ws = mws.WebSocketApp(url=websocket_url,
                                  header={'Authorization': 'Bearer banana'},
                                  on_message=on_links_message,
                                  on_open=on_open)
            # Run the websocket until error occurs
            ws.run_forever(ping_interval=10, ping_timeout=5, ping_payload="OK")
        except Exception as e:
            message_queue.put(error_handler(e))


def linkgeometry_data_receiver(data_queue: multiprocessing.Queue, message_queue: multiprocessing.Queue,
                               request_url: str, websocket_url: str) -> None:
    """
    Receive data from linkgeometry requests and websockets to be processed.

    :param data_queue: Data queue from the manage_data function to temporarily store data before storing in data_cache
    :param message_queue: Shared message queue that takes messages to log in the messaging subsystem
    :param request_url: URL of HTTP request for initial data pull
    :param websocket_url: URL of TCP websocket for continuous data stream
    :return: None
    """
```

```
227     def on_open(ws):
228         """
229         Indicate websocket connection for linkgeometry in data subsystem.
230 
231         :param ws: Linkgeometry websocket
232         :return: None
233         """
234         ws.send("Client connected")
235         message_queue.put(('INFO', "STARTUP: Linkgeometry websocket connected."))
236 
237     def parse_linkgeometry(tree):
238         """
239         Parse linkgeometry XML for most recent changes.
240 
241         :param tree: An xml containing the features of the linkgeometry
242         :return: None
243         """
244         name = xml_parse(tree, './/displayName')
245         if "I24" in name:
246             id = int(xml_parse(tree, 'id'))
247             message_queue.put(('DEBUG', 'UPDATE: Attempting to parse Linkgeometry ID {}'.format(id)))
248             mm_idx = name.rfind("-") + 1
249             end = name.rfind('(')
250             rest_of_str = name[mm_idx:end]
251             for char in rest_of_str[::-1]:
252                 if not char.isdigit() and not (char == '.'):
253                     end -= 1
254                 else:
255                     break
256             mm = float(name[mm_idx:end])
257             message_queue.put(('DEBUG', 'UPDATE: Assigned mile marker {} to Linkgeometry ID {}'.format(mm, id)))
258             if 53 <= mm <= 81:
259                 direction = xml_parse(tree, './/direction')
260                 message_queue.put(('DEBUG', 'UPDATE: Assigned direction {} to Linkgeometry ID {}'
261                                    .format(direction, id)))
262                 message_queue.put(('DEBUG', "UPDATE: Linkgeometry ID {} update processed".format(id),
263                                    {'message_type': 'linkgeometry', 'id': id, 'mile_marker': mm,
264                                     'direction': direction}))
265                 data_queue.put(('linkgeometry', id, {'mile_marker': mm, 'direction': direction}))
266                 message_queue.put(('DEBUG', 'UPDATE: Linkgeometry ID {} placed in data queue.'.format(id)))
267 
268     def on_linkgeometry_message(ws, message):
269         """
270         Handle every new message across the linkgeometry websocket for parsing or pings.
271 
272         :param ws: Linkgeometry websocket
273         :param message: An XML string containing linkgeometry updates
274         :return: None
275         """
276         if message == "OK":
277             reconnect_count = 0
278             message_queue.put(
279                 ('DEBUG', "LINKGEOMETRY PING: Got an OK from SwCS.", {'message_type': 'linkgeometry_ping'}))
280             return
281         root = ET.fromstring(message)
282         message_queue.put(('DEBUG', 'UPDATE: Linkgeometry message root found.', {'XML': str(root)}))
283         parse_linkgeometry(root)
284 
285     # Delay reconnection to the websocket after multiple failed attempts
286     reconnect_count = 0
287     reconnect_map = {0: 0, 1: 10, 2: 60, 3: 1800}
288     while True:
289         time.sleep(reconnect_map[reconnect_count])
290         if reconnect_count != 3:
291             reconnect_count += 1
```

```
292             message_queue.put(('DEBUG', 'UPDATE: Linkgeometry reconnect count incremented to {}.'
293                                .format(reconnect_count)))
294         try:
295             r = requests.get(url=request_url, headers={'Authorization': 'Bearer banana'})
296             tree = ET.fromstring(r.content)
297             message_queue.put(('DEBUG', 'UPDATE: Got Linkgeometry XML for initial request.', {'XML': str(tree)}))
298             for link in tree.findall('.//links/linkGeometry'):
299                 parse_linkgeometry(link)
300             ws = mws.WebSocketApp(url=websocket_url,
301                                   header={'Authorization': 'Bearer banana'},
302                                   on_message=on_linkgeometry_message,
303                                   on_open=on_open)
304             # Run the websocket until error occurs
305             ws.run_forever(ping_interval=10, ping_timeout=5, ping_payload="OK")
306         except Exception as e:
307             message_queue.put(error_handler(e))
308
309
310 def dms_data_receiver(data_queue: multiprocessing.Queue, message_queue: multiprocessing.Queue, request_url: str,
311                       websocket_url: str) -> None:
312     """
313     Receive data from DMS requests and websockets to be processed.
314
315     :param data_queue: Data queue from the manage_data function to temporarily store data before storing in data_cache
316     :param message_queue: Shared message queue that takes messages to log in the messaging subsystem
317     :param request_url: URL of HTTP request for initial data pull
318     :param websocket_url: URL of TCP websocket for continuous data stream
319     :return: None
320     """
321
322     def on_open(ws):
323         """
324         Indicate websocket connection for DMS in data subsystem.
325
326         :param ws: DMS websocket
327         :return: None
328         """
329         ws.send("Client connected")
330         message_queue.put(('INFO', "STARTUP: DMS websocket connected."))
331
332     def parse_dms(tree):
333         """
334         Parse DMS XML for most recent changes.
335
336         :param tree: An XML containing the features of the DMS
337         :return: None
338         """
339         # Make sure the DMS is on I-24
340         if xml_parse(tree, './/roadway') == 'Interstate 24':
341             id = int(xml_parse(tree, 'id'))
342             message_queue.put(('DEBUG', 'UPDATE: Attempting to parse DMS ID {}'.format(id)))
343             name = xml_parse(tree, './/displayName')
344             # Make sure DMS name has the mile marker in it
345             if not any(i.isdigit() for i in name):
346                 return
347             mm_idx = name.find("(") - 6
348             mm = float(name[mm_idx:mm_idx + 4])
349             message_queue.put(('DEBUG', 'UPDATE: Assigned mile marker {} to DMS ID {}'.format(mm, id)))
350             # Only include DMS in the I-24 Smart Corridor
351             if 53 <= mm <= 81:
352                 direction = xml_parse(tree, './/direction')
353                 message_queue.put(('DEBUG', 'UPDATE: Assigned direction {} to DMS ID {}'.format(direction, id)))
354                 latitude = float(xml_parse(tree, './/latitude'))
355                 message_queue.put(('DEBUG', 'UPDATE: Assigned latitude {} to DMS ID {}'.format(latitude, id)))
356                 longitude = float(xml_parse(tree, './/longitude'))
```

```
            message_queue.put(('DEBUG', 'UPDATE: Assigned longitude {} to DMS ID {}'.format(longitude, id)))
            dms_msg = xml_parse(tree, './/multiMsg/multiText')
            message_queue.put(('DEBUG', 'UPDATE: Assigned message {} to DMS ID {}'.format(dms_msg, id)))
            message_queue.put(('DEBUG', 'UPDATE: DMS ID {} update processed.'.format(id),
                               {'message_type': 'dms', 'id': id, 'direction': direction, 'latitude': latitude,
                                'longitude': longitude, 'mile_marker': mm, 'dms_msg': dms_msg}))
            data_queue.put(('dms', id, {'direction': direction, 'latitude': latitude, 'longitude': longitude,
                                        'mile_marker': mm, 'dms_msg': dms_msg}))
            message_queue.put(('DEBUG', 'UPDATE: DMS ID {} placed in data queue.'.format(id)))

    def on_dms_message(ws, message):
        """
        Handle every new message across the DMS websocket for parsing or pings.

        :param ws: DMS websocket
        :param message: An XML string containing DMS updates
        :return: None
        """
        if message == "OK":
            reconnect_count = 0
            message_queue.put(('DEBUG', "DMS PING: Got an OK from SwCS.", {'message_type': 'dms_ping'}))
            return
        root = ET.fromstring(message)
        message_queue.put(('DEBUG', 'UPDATE: DMS message root found.', {'XML': str(root)}))
        parse_dms(tree=root)

    # Delay reconnection to the websocket after multiple failed attempts
    reconnect_count = 0
    reconnect_map = {0: 0, 1: 10, 2: 60, 3: 1800}
    while True:
        time.sleep(reconnect_map[reconnect_count])
        if reconnect_count != 3:
            reconnect_count += 1
            message_queue.put(('DEBUG', 'UPDATE: DMS reconnect count incremented to {}.'.format(reconnect_count)))
        try:
            r = requests.get(url=request_url, headers={'Authorization': 'Bearer banana'})
            tree = ET.fromstring(r.content)
            message_queue.put(('DEBUG', 'UPDATE: Got DMS XML for initial request.', {'XML': str(tree)}))
            for dms in tree.findall('.//data/dms'):
                parse_dms(tree=dms)
            ws = mws.WebSocketApp(url=websocket_url,
                                  header={'Authorization': 'Bearer banana'},
                                  on_message=on_dms_message,
                                  on_open=on_open)
            # Run the websocket until error occurs
            ws.run_forever(ping_interval=10, ping_timeout=5, ping_payload="OK")
        except Exception as e:
            message_queue.put(error_handler(e))


def lcs_data_receiver(data_queue: multiprocessing.Queue, message_queue: multiprocessing.Queue, request_url: str,
                      websocket_url: str) -> None:
    """
    Receive data from LCS requests and websockets to be processed.

    :param data_queue: Data queue from the manage_data function to temporarily store data before storing in data_cache
    :param message_queue: Shared message queue that takes messages to log in the messaging subsystem
    :param request_url: URL of HTTP request for initial data pull
    :param websocket_url: URL of TCP websocket for continuous data stream
    :return: None
    """

    def on_open(ws):
        """
        Indicate websocket connection for LCS in data subsystem.
```

```
422
423         :param ws: LCS websocket
424         :return: None
425         """
426         ws.send("Client connected")
427         message_queue.put(('INFO', "STARTUP: LCS websocket connected."))
428
429     def parse_lcs(tree):
430         """
431         Parse LCS XML for most recent changes.
432
433         :param tree: An XML containing the features of the LCS
434         :return: None
435         """
436         id = int(xml_parse(tree, './/id'))
437         message_queue.put(('DEBUG', 'UPDATE: Attempting to parse LCS ID {}'.format(id)))
438         mm = xml_parse(tree, './/mileMarker')
439         message_queue.put(('DEBUG', 'UPDATE: Assigned mile marker {} to LCS ID {}'.format(mm, id)))
440         direction = xml_parse(tree, './/direction')
441         message_queue.put(('DEBUG', 'UPDATE: Assigned direction {} to LCS ID {}'.format(direction, id)))
442         latitude = float(xml_parse(tree, './/latitude'))
443         message_queue.put(('DEBUG', 'UPDATE: Assigned latitude {} to LCS ID {}'.format(latitude, id)))
444         longitude = float(xml_parse(tree, './/longitude'))
445         message_queue.put(('DEBUG', 'UPDATE: Assigned longitude {} to LCS ID {}'.format(longitude, id)))
446         message_queue.put(('DEBUG', "UPDATE: LCS ID {} update processed.".format(id),
447                            {'message_type': 'lcs', 'id': id, 'mile_marker': mm,
448                             'direction': direction, 'latitude': latitude, 'longitude': longitude}))
449         data_queue.put(('lcs', id, {'mile_marker': mm, 'direction': direction, 'latitude': latitude,
450                                     'longitude': longitude}))
451         message_queue.put(('DEBUG', 'UPDATE: LCS ID {} placed in data queue.'.format(id)))
452
453     def on_lcs_message(ws, message):
454         """
455         Handle every new message across the LCS websocket for parsing or pings.
456
457         :param ws: LCS websocket
458         :param message: An XML string containing LCS updates
459         :return: None
460         """
461         if message == "OK":
462             reconnect_count = 0
463             message_queue.put(('DEBUG', "LCS PING: Got an OK from SwCS.", {'message_type': 'lcs_ping'}))
464             return
465         root = ET.fromstring(message)
466         message_queue.put(('DEBUG', 'UPDATE: LCS message root found.', {'XML': str(root)}))
467         parse_lcs(tree=root)
468
469     # Delay reconnection to the websocket after multiple failed attempts
470     reconnect_count = 0
471     reconnect_map = {0: 0, 1: 10, 2: 60, 3: 1800}
472     while True:
473         time.sleep(reconnect_map[reconnect_count])
474         if reconnect_count != 3:
475             reconnect_count += 1
476             message_queue.put(('DEBUG', 'UPDATE: LCS reconnect count incremented to {}.'.format(reconnect_count)))
477         try:
478             r = requests.get(url=request_url, headers={'Authorization': 'Bearer banana'})
479             tree = ET.fromstring(r.content)
480             message_queue.put(('DEBUG', 'UPDATE: Got LCS XML for initial request.', {'XML': str(tree)}))
481             for lcs in tree.findall('.//data/lcs'):
482                 parse_lcs(tree=lcs)
483             ws = mws.WebSocketApp(url=websocket_url,
484                                   header={'Authorization': 'Bearer banana'},
485                                   on_message=on_lcs_message,
486                                   on_open=on_open)
```

```
            # Run the websocket until error occurs
            ws.run_forever(ping_interval=10, ping_timeout=5, ping_payload="OK")
        except Exception as e:
            message_queue.put(error_handler(e))


def vsl_data_receiver(data_queue: multiprocessing.Queue, message_queue: multiprocessing.Queue, request_url: str,
                      websocket_url: str) -> None:
    """
    Receive data from VSL requests and websockets to be processed.

    :param data_queue: Data queue from the manage_data function to temporarily store data before storing in data_cache
    :param message_queue: Shared message queue that takes messages to log in the messaging subsystem
    :param request_url: URL of HTTP request for initial data pull
    :param websocket_url: URL of TCP websocket for continuous data stream
    :return: None
    """

    def on_open(ws):
        """
        Indicate websocket connection for VSL in data subsystem.

        :param ws: VSL websocket
        :return: None
        """
        ws.send("Client connected")
        message_queue.put(('INFO', "STARTUP: VSL websocket connected."))

    def parse_vsl(tree):
        """
        Parse VSL XML for most recent changes.

        :param tree: An XML containing the features of the VSL
        :return: None
        """
        name = xml_parse(tree, './/config/displayName')
        if "I24" in name:
            mm_idx = name.rfind('-') + 1
            rest_of_str = name[mm_idx:]
            end = name.rfind('(')
            if end != -1:
                end -= 2
                rest_of_str = name[mm_idx:end]
                direction = name[end]
            else:
                direction = name[mm_idx - 2]
            id = int(xml_parse(tree, 'id'))
            message_queue.put(('DEBUG', 'Attempting to parse VSL ID {}.'.format(id)))
            mm = float(rest_of_str)
            message_queue.put(('DEBUG', 'UPDATE: Assigned mile marker {} to VSL ID {}'.format(mm, id)))
            message_queue.put(('DEBUG', 'UPDATE: Assigned direction {} to VSL ID {}'.format(direction, id)))
            timestamp = xml_parse(tree, './/status/timestamp')
            message_queue.put(('DEBUG', 'UPDATE: Assigned timestamp {} to VSL ID {}'.format(timestamp, id)))
            state = xml_parse(tree, './/status/state')
            message_queue.put(('DEBUG', 'UPDATE: Assigned state {} to VSL ID {}'.format(state, id)))
            target_speed = int(xml_parse(tree, './/status/targetSpeed'))
            message_queue.put(('DEBUG', 'UPDATE: Assigned target speed {} to VSL ID {}'.format(target_speed, id)))
            link_id = int(xml_parse(tree, './/status/links/link/linkId'))
            message_queue.put(('DEBUG', 'UPDATE: Assigned Link ID {} to VSL ID {}'.format(link_id, id)))
            message_queue.put(('DEBUG', "UPDATE: VSL ID {} update processed.".format(id),
                               {'message_type': 'vsl', 'id': id, 'timestamp': timestamp, 'mile_marker': mm,
                                'direction': direction, 'state': state, 'target_speed': target_speed,
                                'link_id': link_id}))
            data_queue.put(
                ('vsl', id, {'timestamp': timestamp, 'mile_marker': mm, 'direction': direction, 'state': state,
```

```
                    'target_speed': target_speed, 'link_id': link_id}))
        message_queue.put(('DEBUG', 'UPDATE: VSL ID {} placed in data queue.'.format(id)))

    def on_vsl_message(ws, message):
        """
        Handle every new message across the VSL websocket for parsing or pings.

        :param ws: VSL websocket
        :param message: An XML string containing VSL updates
        :return: None
        """
        if message == "OK":
            reconnect_count = 0
            message_queue.put(('DEBUG', "VSL PING: Got an OK from SwCS.", {'message_type': 'vsl_ping'}))
            return
        root = ET.fromstring(message)
        message_queue.put(('DEBUG', 'UPDATE: VSL message root found.', {'XML': str(root)}))
        parse_vsl(tree=root)

    # Delay reconnection to the websocket after multiple failed attempts
    reconnect_count = 0
    reconnect_map = {0: 0, 1: 10, 2: 60, 3: 1800}
    while True:
        time.sleep(reconnect_map[reconnect_count])
        if reconnect_count != 3:
            reconnect_count += 1
            message_queue.put(('DEBUG', 'UPDATE: VSL reconnect count incremented to {}.'.format(reconnect_count)))
        try:
            r = requests.get(url=request_url, headers={'Authorization': 'Bearer banana'})
            tree = ET.fromstring(r.content)
            message_queue.put(('DEBUG', 'UPDATE: Got VSL XML for initial request.', {'XML': str(tree)}))
            for vsl in tree.findall('.//data/vslSegment'):
                parse_vsl(tree=vsl)
            ws = mws.WebSocketApp(url=websocket_url,
                                  header={'Authorization': 'Bearer banana'},
                                  on_message=on_vsl_message,
                                  on_open=on_open)
            # Run websocket until error occurs
            ws.run_forever(ping_interval=10, ping_timeout=5, ping_payload="OK")
        except Exception as e:
            message_queue.put(error_handler(e))


def aggregate_links(links_queue: multiprocessing.Queue, message_queue: multiprocessing.Queue) -> None:
    """
    Manage the aggregation of links data for visualization purposes.

    :param links_queue: Queue specifically used for sensor data aggregation for links
    :param message_queue: Shared message queue that takes messages to log in the messaging subsystem
    :return: None
    """

    # Determine information for time buckets
    current_time = datetime.datetime.now()
    mins = current_time.minute
    secs = current_time.second

    start_bucket = current_time - timedelta(minutes=(mins % 5), seconds=secs)
    log_time = start_bucket + timedelta(minutes=2, seconds=30)
    log_time = log_time.strftime("%a %b %d %H:%M:%S %Y")
    end_time = start_bucket + timedelta(minutes=5)

    message_queue.put(('DEBUG', 'UPDATE: Set aggregate links times as {} to {}.'.format(start_bucket, end_time)))

    links_df = pd.DataFrame.from_records([links_queue.get(block=True, timeout=None)])
```

```
    while True:
        current_time = datetime.datetime.now()

        # Aggregate every five minutes
        if current_time >= end_time:
            # Aggregate the data, first make sure numeric columns are numeric then group by each link
            links_df['avg_speed'] = pd.to_numeric(links_df['avg_speed'])
            links_df['avg_vol'] = pd.to_numeric(links_df['avg_vol'])
            links_df['avg_occ'] = pd.to_numeric(links_df['avg_occ'])
            links_df = links_df.groupby(['mile_marker', 'direction']).mean(numeric_only=True)

            # Iterate through all the links and log the results of aggregation
            for i in range(links_df.shape[0]):
                avg_speed = links_df['avg_speed'].iloc[i]
                avg_vol = links_df['avg_vol'].iloc[i]
                avg_occ = links_df['avg_occ'].iloc[i]
                tmp_dict = {'ts': log_time, 'mile_marker': links_df.index[i][0], 'direction': links_df.index[i][1],
                            'avg_speed': avg_speed, 'avg_vol': avg_vol, 'avg_occ': avg_occ, 'env': 'RDS-speed'}

                message_queue.put(('INFO',
                                   "UPDATE: {} LINK MM {}{} 5 minute update processed with an avg speed of {:.2f}, "
                                   "avg volume of {:.2f}, and avg occupancy of {:.2f}.".format(log_time,
                                                                                              tmp_dict['mile_marker'],
                                                                                              tmp_dict['direction'],
                                                                                              tmp_dict['avg_speed'],
                                                                                              tmp_dict['avg_vol'],
                                                                                              tmp_dict['avg_occ']),
                                   tmp_dict))
            # Reset the dataframe for next aggregation
            links_df = pd.DataFrame(columns=['mile_marker', 'direction', 'timestamp', 'avg_speed', 'avg_vol',
                                             'avg_occ'])

            # Reset the timer
            mins = current_time.minute
            secs = current_time.second
            start_bucket = current_time - timedelta(minutes=(mins % 5), seconds=secs)
            log_time = start_bucket + timedelta(minutes=2, seconds=30)
            log_time = log_time.strftime("%a %b %d %H:%M:%S %Y")
            end_time = start_bucket + timedelta(minutes=5)
        else:
            # Append to the dataframe since it is not time to aggregate yet
            tmp_df = pd.DataFrame.from_records([links_queue.get(block=True, timeout=None)])
            links_df = pd.concat([links_df, tmp_df], ignore_index=True)


def data_cache_manager(data_cache: dict, data_queue: multiprocessing.Queue, links_queue: multiprocessing.Queue,
                       message_queue: multiprocessing.Queue) -> None:
    """
    Manage the transfer of data from the queue to the data cache, links aggregation, and database.

    :param data_cache: Shared data structure created by AI-DSS manager for data storage
    :param data_queue: Data queue from the manage_data function to temporarily store data before storing in data_cache
    :param links_queue: Queue specifically used for sensor data aggregation for links
    :param message_queue: Shared message queue that takes messages to log in the messaging subsystem
    :return: None
    """
    message_queue.put(('INFO', "STARTUP: Data subsystem cache manager function started."))
    while True:
        data_name, data_id, new_data = data_queue.get(block=True, timeout=None)
        # If there is no data for this ID, initialize list
        if data_id not in data_cache[data_name]:
            message_queue.put(('DEBUG', 'UPDATE: Placing new ID {} in data cache.'.format(data_id)))
            # https://stackoverflow.com/questions/35202278/cannot-append-items-to-multiprocessing-shared-list
            # Below initializes an empty list in weird syntax, explained in the above link
```

```
            item = data_cache[data_name][data_id] = list()
            item.append(new_data)
            data_cache[data_name][data_id] = item
        else:
            # Create tmp list to modify current structure
            tmp = list(data_cache[data_name][data_id])
            # If the list is longer than DATA_SIZE entries of data, then roll off the old data
            if len(data_cache[data_name][data_id]) > int(config['DATASIZES']['DATA_SIZE']):
                tmp.pop(0)
                message_queue.put(('DEBUG', 'UPDATE: Rolled off {} data from data cache.'.format(data_name)))
            # Append to the list with new data
            tmp.append(new_data)
            data_cache[data_name][data_id] = tmp
            # Put element on data aggregation queue
        # Put element on links aggregation queue if it is links data
        if data_name == 'links':
            links_info = {'mile_marker': new_data['mile_marker'], 'direction': new_data['direction'],
                          'timestamp': new_data['timestamp'], 'avg_speed': new_data['speed'],
                          'avg_vol': new_data['vol'], 'avg_occ': new_data['occ']}
            links_queue.put(links_info)
            message_queue.put(('DEBUG', 'UPDATE: Placed links data into links queue.', links_info))


def manage_data(data_cache, message_queue: multiprocessing.Queue, pid_tracker) -> None:
    """
    Manage the data subsystem processes by spawning and respawning if they need to be restarted.

    :param data_cache: Shared data structure created by AI-DSS manager for data storage
    :param message_queue: Shared message queue that takes messages to log in the messaging subsystem
    :param pid_tracker: Tracker containing all the process IDs
    :return: None
    """
    # Data queue for writing to data cache and database
    # - - - - - - - - - - - - - - - - - - - - - - - - - -
    # Format for writing to the data queue includes the data name as it appears in the data cache (e.g., 'tss')
    # as the first value in a tuple, with the second value being the data contents.
    data_queue = multiprocessing.Queue(maxsize=config['DATASIZES']['DATA_QUEUE_SIZE'])
    links_queue = multiprocessing.Queue(maxsize=config['DATASIZES']['LINKS_QUEUE_SIZE'])
    message_queue.put(('INFO', "STARTUP: Data subsystem beginning to spawn processes."))
    processes_to_spawn = {'cache_manager': (data_cache_manager, (data_cache, data_queue, links_queue, message_queue)),
                          'links_aggregate': (aggregate_links, (links_queue, message_queue)),
                          'data_detectors': (detectors_data_receiver, (
                              data_queue, message_queue, config['CONNECTIONS']['detectors_request_url'],
                              config['CONNECTIONS']['detectors_websocket_url'])),
                          'data_links': (links_data_receiver, (
                              data_queue, message_queue, config['CONNECTIONS']['links_request_url'],
                              config['CONNECTIONS']['links_websocket_url'])),
                          'data_linkgeometry': (linkgeometry_data_receiver, (
                              data_queue, message_queue, config['CONNECTIONS']['linkgeometry_request_url'],
                              config['CONNECTIONS']['linkgeometry_websocket_url'])),
                          'data_dms': (dms_data_receiver, (
                              data_queue, message_queue, config['CONNECTIONS']['dms_request_url'],
                              config['CONNECTIONS']['dms_websocket_url'])),
                          'data_lcs': (lcs_data_receiver, (
                              data_queue, message_queue, config['CONNECTIONS']['lcs_request_url'],
                              config['CONNECTIONS']['lcs_websocket_url'])),
                          'data_vsl': (vsl_data_receiver, (
                              data_queue, message_queue, config['CONNECTIONS']['vsl_request_url'],
                              config['CONNECTIONS']['vsl_websocket_url']))
                          }

    # Here we set which processes are set to ON or OFF based on values specified in the config
    data_process_control = config['PROCESSES']
    for name in list(processes_to_spawn):
        # If we specify to data process to be false in config, we delete it and do not spawn it
```

```
747         if not data_process_control[name]:
748             del processes_to_spawn[name]
749 
750     # Keep a store of the mp.Process objects for the data receivers and the cache manager.
751     data_process_objects = {}
752 
753     for process_name, (process_function, process_args) in processes_to_spawn.items():
754         message_queue.put(('INFO', "STARTUP: Data subsystem spawning {}.".format(process_name)))
755         # Start up each data subsystem process.
756         data_process = multiprocessing.Process(target=process_function, args=process_args,
757                                                name=process_name, daemon=True)
758         data_process.start()
759         # Put the process object in the dictionary, keyed by the process name.
760         data_process_objects[process_name] = data_process
761         # Each process is responsible for putting its own children's PIDs in the tracker upon creation.
762         pid_tracker[process_name] = data_process.pid
763         message_queue.put(('INFO', "STARTUP: {} process has PID={}.".format(process_name, data_process.pid)))
764 
765     while True:
766         # Every few seconds do a check of the process status.
767         time.sleep(29.5)
768         # For each process that is being managed at this level, check if it's still running
769         running = []
770         dead = []
771         for child_key in data_process_objects.keys():
772             child_process = data_process_objects[child_key]
773             if child_process.is_alive():
774                 # Process is running; do nothing.
775                 message_queue.put(('DEBUG', 'UPDATE: {} process is still running.'.format(child_process.name)))
776                 running.append(True)
777             else:
778                 # Process has died. Let's restart it.
779                 running.append(False)
780                 # Copy its name out of the existing process object for lookup and restart.
781                 process_name = child_process.name
782                 message_queue.put(('ERROR', "Restarting process: {}".format(process_name)))
783                 # Add to list of processes needing restart
784                 dead.append(process_name)
785                 # Get the function handle and function arguments to spawn this process again.
786                 process_function, process_args = processes_to_spawn[process_name]
787                 # Restart the process the same way we did originally.
788                 data_process = multiprocessing.Process(target=process_function, args=process_args,
789                                                        name=process_name, daemon=True)
790                 data_process.start()
791                 # Re-write the process object in the dictionary and update its PID.
792                 data_process_objects[child_key] = data_process
793                 pid_tracker[process_name] = data_process.pid
794         if all(running):
795             message_queue.put(('INFO', "DATA HEARTBEAT: All processes are still running.",
796                                {'message_type': 'data_heartbeat', 'restarted_processes': 'none'}))
797         else:
798             message_queue.put(('ERROR', "Process(es) {} not running and were restarted.".format(dead),
799                                {'message_type': 'data_heartbeat', 'restarted_processes': dead}))
800 
801         cache_sizes = {'detectors': 0, 'links': 0, 'linkgeometry': 0, 'dms': 0, 'lcs': 0, 'vsl': 0}
802         for key in cache_sizes.keys():
803             count = 0
804             for data_id in data_cache[key].keys():
805                 count += len(data_cache[key][data_id])
806             cache_sizes[key] = count
807         message_queue.put(('INFO', """CACHE: Detector data cache size: {}, Links data cache size: {},
808                           Link_geometry data cache size: {}, DMS data cache size: {}, LCS data cache size: {},
809                           VSL data cache size: {}""".format(cache_sizes['detectors'], cache_sizes['links'],
810                                                             cache_sizes['linkgeometry'], cache_sizes['dms'],
811                                                             cache_sizes['lcs'], cache_sizes['vsl']), cache_sizes))
```

```
812
813
814 if __name__ == '__main__':
815     print("NO CODE TO RUN")
```

B.13 subsys_events.py

```
1  # -----------------------------
2  """
3  Contains the event subsystem that manages event processes for the AI-DSS.
4
5  1. Ingests event data from SwCS
6  2. Parses event data for various necessary information
7  3. Dumps event data to database on event closure
8  """
9  __file__ = 'subsys_events.py'
10 # -----------------------------
11
12 import multiprocessing
13 import time
14 import I24customwebsocket as mws
15 import requests
16 import xml.etree.ElementTree as ET
17 import pymongo
18 import xmltodict
19
20 from config.get_config import config
21 from utility import error_handler, xml_parse
22
23
24 def manage_events(event_cache, message_queue: multiprocessing.Queue) -> None:
25     """
26     Manage the event subsystem.
27
28     :param event_cache: Shared data structure containing all the event data
29     :param message_queue: Shared message queue that takes messages to log in the messaging subsystem
30     :return: None
31     """
32
33     def lane_blockage(event_info):
34         """
35         Process and parse the lane blockage information for a given event into a usable format
36
37         :param event_info: An XML tree of a given event
38         :return: Dictionary of the lane blockage configuration
39         """
40         try:
41             # 0 indicates Blocked and 1 indicates Clear
42             lane_blockage_dict = {}
43             travel_dict = {}
44             travel_list = []
45             blockage_list = event_info.find(".//laneList")
46             message_queue.put(('DEBUG', 'UPDATE: Event lane blockage config found.', {'XML': str(blockage_list)}))
47             for lane in blockage_list.findall('./lane'):
48                 if lane.find('./laneIndex').text == '1' and lane.find('./laneType/classification').text == 'shoulder':
49                     message_queue.put(('DEBUG', 'UPDATE: Event left shoulder found. Checking blockage.'))
50                     lane_blockage_dict['left_shoulder'] = 1 if lane.find('./laneBlockageCode').text == 'clear' else 0
51                     message_queue.put(('DEBUG', 'UPDATE: Event left shoulder assigned new value {}.'
52                                        .format(lane_blockage_dict['left_shoulder'])))
53                 elif lane.find('./laneType/classification').text == 'shoulder':
54                     message_queue.put(('DEBUG', 'UPDATE: Event right shoulder found. Checking blockage.'))
55                     lane_blockage_dict['right_shoulder'] = 1 if lane.find('./laneBlockageCode').text == 'clear' else 0
56                     message_queue.put(('DEBUG', 'UPDATE: Event right shoulder assigned new value {}.'
```

```
                        .format(lane_blockage_dict['right_shoulder'])))
            elif lane.find('./laneType/classification').text == 'travel':
                cur_index = int(lane.find("./laneIndex").text)
                message_queue.put(('DEBUG', 'UPDATE: Event travel lane found. Checking blockage.'))
                val_to_add = 1 if lane.find('./laneBlockageCode').text == 'clear' else 0
                message_queue.put(('DEBUG', 'UPDATE: Event travel lane assigned new value {}.'
                        .format(val_to_add)))
                travel_dict[cur_index] = val_to_add
            elif lane.find('./laneType/classification').text == 'exit':
                message_queue.put(('DEBUG', 'UPDATE: Event exit lane found. Checking blockage.'))
                lane_blockage_dict['exit'] = 1 if lane.find('./laneBlockageCode').text == 'clear' else 0
                message_queue.put(('DEBUG', 'UPDATE: Event exit lane assigned new value {}.'
                        .format(lane_blockage_dict['exit'])))
            elif lane.find('./laneType/classification').text == 'entry':
                message_queue.put(('DEBUG', 'UPDATE: Event entry lane found. Checking blockage.'))
                lane_blockage_dict['entry'] = 1 if lane.find('./laneBlockageCode').text == 'clear' else 0
                message_queue.put(('DEBUG', 'UPDATE: Event entry lane assigned new value {}.'
                        .format(lane_blockage_dict['entry'])))
        for lane_index in sorted(travel_dict.keys()):
            travel_list.append(travel_dict[lane_index])
        num_lanes = len(travel_list)
        for i in range(6 - num_lanes):
            message_queue.put(('DEBUG', 'UPDATE: Event lane {} assigned value -1.'.format(num_lanes + 1 + i)))
            travel_list.append(-1)

        # Puts travel lanes in configuration consistent with HOV as lane 1
        travel_tup = tuple(travel_list)
        lane_blockage_dict['travel'] = travel_tup
        message_queue.put(('DEBUG', 'UPDATE: Event travel lanes assigned configuration {}.'.format(travel_tup)))
        return lane_blockage_dict
    except AttributeError as ae:
        message_queue.put(error_handler(ae))
        return 'Unknown Lane Configuration'

def parse_event(event_info):
    """
    Parse detector XML for most recent changes.

    :param event_info: An XML tree of a given event
    :return: ID of the event and dictionary of parsed data from the event
    """
    # takes event id, time and return it to the other function currently before data needs to be parsed
    event_id = [i.text for i in event_info.findall('.//id')][0]
    message_queue.put(('DEBUG', "UPDATE: Attempting to parse event ID {}.".format(event_id)))
    event_time = xml_parse(event_info, './/eventTimestamps/statusDateTime')
    message_queue.put(('DEBUG', 'UPDATE: Event ID {} assigned timestamp {}.'.format(event_id, event_time)))
    mm = float(xml_parse(event_info, './/referencePoint/sortOrder'))
    mm = mm / 1000
    message_queue.put(('DEBUG', 'UPDATE: Event ID {} assigned mile marker {}.'.format(event_id, mm)))
    direction = xml_parse(event_info, './/roadwayDirection')
    message_queue.put(('DEBUG', 'UPDATE: Event ID {} assigned direction {}.'.format(event_id, direction)))
    latitude = float(xml_parse(event_info, './/point/latitude')) / 1000000
    message_queue.put(('DEBUG', 'UPDATE: Event ID {} assigned latitude {}.'.format(event_id, latitude)))
    longitude = float(xml_parse(event_info, './/point/longitude')) / 1000000
    message_queue.put(('DEBUG', 'UPDATE: Event ID {} assigned longitude {}.'.format(event_id, longitude)))
    classification = xml_parse(event_info, './/eventType/classification')
    message_queue.put(('DEBUG', 'UPDATE: Event ID {} assigned classification {}.'
                .format(event_id, classification)))
    description = xml_parse(event_info, './/SAEMessageText')
    message_queue.put(('DEBUG', 'UPDATE: Event ID {} assigned description {}.'
                .format(event_id, description)))
    status = xml_parse(event_info, './/eventStatus/shortName')
    message_queue.put(('DEBUG', 'UPDATE: Event ID {} assigned status {}.'.format(event_id, status)))
    blockage = lane_blockage(event_info)
    event_data = {'env': 'event', 'message_type': 'event', 'id': event_id, 'timestamp': event_time,
```

```
                'mile_marker': mm,
                'direction': direction, 'lat': latitude, 'lon': longitude, 'classification': classification,
                'description': description, 'status': status, 'lane_blockage': blockage}
    return event_id, event_data

def add_event_to_database(id, data):
    """
    Save events to the database.

    :param id: ID of the event to be placed in the database
    :param data: A string representation of the XML event data unless it is a stranded event, which is a dict
    :return: None
    """
    client = pymongo.MongoClient(host=config['DATABASE']['host'], port=config['DATABASE']['port'],
                         username=config['DATABASE']['username'], password=config['DATABASE']['password'],
                         connect=True, connectTimeoutMS=5000)
    message_queue.put(('DEBUG', "UPDATE: Adding Event id {} to database.".format(id)))
    db = client['atcmtd']['events']
    # Data is a dictionary if the event is stranded, lost some information due to websocket disconnect
    if type(data) == dict:
        db.insert_one(data)
    # Data is an XML string if we close it on websocket update, convert to dict and dump to database
    else:
        db.insert_one(xmltodict.parse(data))

def on_open(ws):
    """
    Indicate websocket connection for event subsystem.

    :param ws: Event websocket
    :return: None
    """
    ws.send("Client connected")
    message_queue.put(('INFO', "STARTUP: Events websocket connected."))

def on_event_message(ws, message):
    """
    Handle every new message across the event websocket for parsing or pings.

    :param ws: Event websocket
    :param message: An XML string containing event updates
    :return: None
    """
    if message == "OK":
        reconnect_count = 0
        message_queue.put(('DEBUG', "EVENTS PING: Got an OK from SwCS.", {'message_type': 'events_ping'}))
        return
    root = ET.fromstring(message)
    message_queue.put(('DEBUG', 'UPDATE: Event message root found.', {'XML': str(root)}))
    if xml_parse(root, './/roadway/longName') == 'Interstate 24':
        event_id, event_data = parse_event(root)
        message_queue.put(
            ('DEBUG', "UPDATE: Successfully parsed and updated event cache for event ID {}."
             .format(event_id), event_data))
        event_cache[event_id] = event_data
        # If event is resolved, dump to database and delete from cache
        if event_data['status'] == 'Closed':
            if config["DATABASE"]["activate"]:
                add_event_to_database(event_id, message)
                message_queue.put(('DEBUG', "UPDATE: Dumped to database for event ID {}."
                                   .format(event_id), event_data))
            event_cache.pop(event_id)
            message_queue.put(('DEBUG', "UPDATE: Deleted event ID {} from the event cache."
                               .format(event_id), event_data))
```

```
    # Delay reconnection to the websocket after multiple failed attempts
    reconnect_count = 0
    reconnect_map = {0: 0, 1: 10, 2: 60, 3: 1800}
    while True:
        time.sleep(reconnect_map[reconnect_count])
        if reconnect_count != 3:
            reconnect_count += 1
            message_queue.put(('DEBUG', 'UPDATE: Events reconnect count incremented to {}.'.format(reconnect_count)))
        try:
            message_queue.put(('DEBUG', "UPDATE: Making initial event request to SwCS."))
            r = requests.get(url=config['CONNECTIONS']['event_request_url'], headers={'Authorization': 'Bearer banana'},
                             stream=True)
            tree = ET.fromstring(r.content)
            message_queue.put(('DEBUG', 'UPDATE: Got Event XML for initial request.', tree))
            event_id_list = []
            for event in tree.findall('.//data/event'):
                if xml_parse(event, './/roadway/longName') == 'Interstate 24':
                    event_id, event_data = parse_event(event)
                    event_id_list.append(event_id)
                    event_cache[event_id] = event_data
                    message_queue.put(('DEBUG', "UPDATE: Received event ID {} on initial request."
                                       .format(event_id), event_data))
            # This will delete stranded events we may have missed if we disconnected from the websocket for any reason
            for key in event_cache.keys():
                if key not in event_id_list:
                    if config["DATABASE"]["activate"]:
                        add_event_to_database(key, event_cache[key])
                    del event_cache[key]
            message_queue.put(('DEBUG', "STARTUP: Got {} I-24 events on initial request."
                               .format(len(event_cache))))
            ws = mws.WebSocketApp(url=config['CONNECTIONS']['event_websocket_url'],
                                  header={'Authorization': 'Bearer banana'},
                                  on_message=on_event_message,
                                  on_open=on_open)
            # Run the websocket until error occurs
            ws.run_forever(ping_interval=10, ping_timeout=5, ping_payload="OK")
        except Exception as e:
            message_queue.put(error_handler(e))


if __name__ == '__main__':
    print("NO CODE TO RUN")
```

B.14 subsys_messaging.py

```
# ------------------------------
"""
Contains the messaging subsystem that takes messages from other subsystems ; handles their logging / distribution.
"""
__file__ = 'subsys_messaging.py'
# ------------------------------

import multiprocessing

from log_writer import I24Logger
from config.get_config import config


def configure_logger():
    """
    Configure the logger with the necessary parameters.

    :return: None
```

```
    """
    logger_params = ['log_name', 'processing_environment', 'connect_logstash', 'connect_file', 'connect_syslog',
                'connect_console', 'connect_sl', 'logstash_address', 'sl_address', 'file_path', 'syslog_location',
                'all_log_level', 'logstash_log_level', 'file_log_level', 'syslog_log_level', 'console_log_level',
                'sl_log_level']

    logger_input = {}
    # For all items in our logging configuration - populate a dictionary
    for key, value in config['LOGGING'].items():
        if key in logger_params:
            logger_input[key] = value

    return logger_input


def message_handler(message_queue: multiprocessing.Queue) -> None:
    """
    Handle the logging of various messages occurring throughout the system processes.

    :param message_queue: Shared message queue that takes messages to log in the messaging subsystem
    :return: None
    """

    logger_input = configure_logger()
    logger = I24Logger(**logger_input)

    while True:
        new_message = message_queue.get(block=True, timeout=None)
        if isinstance(new_message, BaseException):
            exception_info = True

        else:
            exception_info = False
            if isinstance(new_message, tuple):
                if len(new_message) == 2:
                    logger.log(level=new_message[0], message=new_message[1], exc_info=exception_info)
                elif len(new_message) == 3:
                    logger.log(level=new_message[0], message=new_message[1], extra=new_message[2], exc_info=exception_info)
                else:
                    logger.warning("Invalid tuple length (={}) for logging!".format(len(new_message)))
            elif isinstance(new_message, str):
                if new_message == 'SIGUSR1':
                    logger.warning("Moving to set temporary debug level on logger.")
                    logger.set_temporary_debug()
                elif new_message == 'SIGUSR2':
                    logger.warning("Moving to unset debug level on logger.")
                    logger.unset_temporary_debug()
                else:
                    logger.warning("Received string on message_queue, but it's not one of the signals.")


if __name__ == '__main__':
    print("NO CODE TO RUN")
```

B.15 subsys_recommendations.py

```
# ----------------------------
"""
Contains the recommendation subsystem that manages response plan processes for the AI-DSS.

1. Ingest response plan request from SwCS
2. Provide event ID to evaluator for calculation of a response plan
3. Compare the SwCS response plan and AI-DSS response plan to get necessary items to override
```

```
4. Send our response plan as an XML document to SwCS
"""
__file__ = 'subsys_events.py'
# ------------------------------

import multiprocessing
import xml.etree.ElementTree as ET
from fastavro import writer, parse_schema
from datetime import date, datetime
import os
import time
import evaluator
import I24customwebsocket as mws
from config.get_config import config, base_config
from graph import I24Graph
from utility import error_handler, xml_parse

# Instance of the graph owned by recommendation subsystem and evaluated for LCS gantry association.
response_plan_graph = I24Graph(graph_directory=os.path.join(base_config['install_path'], base_config['repo_path_join'], config['GRAPH']['
      directory']),
                       blacklist_reload_interval=config['GRAPH']['blacklist_reload_interval'])


def build_lcs_xml(tree, lcs_dict):
   """
   Build an LCS response item XML for a specified response plan.

   :param tree: XML tree for the response plan
   :param lcs_dict: Dictionary of the LCS response item
   :return: None
   """
   item = ET.SubElement(tree, 'item')
   ET.SubElement(item, 'itemId').text = 'LCSPLANITEM_0000{}'.format(int(time.time() * 10000))
   lcsPlan = ET.SubElement(item, 'lcsPlanItemData')
   lcsId = ET.SubElement(lcsPlan, 'lcsId')
   ET.SubElement(lcsId, 'id', providerName="lcs", resourceType="lcs", centerId="Region 3").text = str(
      lcs_dict['gantry_id'])
   lcsMsg = ET.SubElement(lcsPlan, 'lcsMsg')
   heads = ET.SubElement(lcsMsg, 'heads')
   for element in lcs_dict['board_config']:
      head = ET.SubElement(heads, 'head')
      ET.SubElement(head, 'category').text = element
      if element == 'Speed':
         ET.SubElement(head, 'speedStyle').text = 'WhiteOnBlack'
      elif element == 'Hov':
         ET.SubElement(head, 'hovStyle').text = 'OpenToAll'
   ET.SubElement(lcsMsg, 'owner').text = 'em'
   ET.SubElement(lcsMsg, 'duration').text = '-1'
   ET.SubElement(lcsMsg, 'priority').text = '1'
   if lcs_dict['modified']:
      conflict = ET.SubElement(item, 'conflictInfo')
      original = ET.SubElement(conflict, 'originalSuggestion')
      heads = ET.SubElement(original, 'heads')
      for element in lcs_dict['swcs_config']:
         head = ET.SubElement(heads, 'head')
         ET.SubElement(head, 'category').text = element
         if element == 'Speed':
            ET.SubElement(head, 'speedStyle').text = 'WhiteOnBlack'
         elif element == 'Hov':
            ET.SubElement(head, 'hovStyle').text = 'OpenToAll'
      ET.SubElement(original, 'owner').text = 'em'
      ET.SubElement(original, 'duration').text = '-1'
      ET.SubElement(original, 'priority').text = lcs_dict['priority']


```

```
def create_response_plan_xml(ref_id, event_id, compared_response_plan_list: list):
    """
    Take both response plans and compare them in order to be transformed into an XML document.

    :param ref_id: ID of the XML reference from SwCS
    :param event_id: ID of event to be evaluated
    :param compared_response_plan_list: List containing information for each gantry
    :return: XML of the finalized response plan
    """
    root = ET.Element("modifyResponsePlanSuggestionResp", providerName="DecisionSupportApi",
                providerType="DecisionSupportApi")
    ET.SubElement(root, "refId").text = str(ref_id)
    data = ET.SubElement(root, 'data')
    ET.SubElement(data, 'eventId', providerName="em", resourceType="event", centerId="Region 3").text = str(event_id)
    # Specify our add, modify, delete items for LCS - delete is currently unused
    add = ET.SubElement(data, 'addedItems')
    modify = ET.SubElement(data, 'modifiedItems')
    delete = ET.SubElement(data, 'deletedItems')
    for item in compared_response_plan_list:
        if item['modified']:
            build_lcs_xml(modify, item)
        else:
            break
    # If we wish to add a gantry to RP, set it in config
    if config["TOGGLES"]["add_gantry"]:
        build_lcs_xml(add, compared_response_plan_list[3])
    return ET.tostring(root, encoding='unicode', method='xml')


def compare_response_plans(swcs_response_plan, lcs_boards_dict):
    """
    Takes both response plans and compares them in order to be transformed into an XML document.

    :param swcs_response_plan: Response plan from SwCS
    :param lcs_boards_dict: Response plan from evaluator
    :return: Response plan containing compared metrics of both plans
    """
    compared_response_plans_list = []
    # For all LCS items that SwCS passes to us
    for i, item in enumerate(swcs_response_plan):
        # The fourth gantry given by SwCS is a downstream gantry we will not deal with (always green arrows), skip
        if i != 3:
            # Get their gantry ID and priority value
            gantry_id = int(xml_parse(item, './/lcsId/id'))
            priority = xml_parse(item, './/deviceMsg/priority')
            lane_list = []
            # Parameter to know whether our created plan is different
            modified = False
            swcs_lane_list = []
            # For each head on each gantry
            for j, head in enumerate(item.findall('.//lcsMsg/heads/head')):
                swcs_category = xml_parse(head, './/category')
                if swcs_category == 'Speed':
                    lane_list.append(swcs_category)
                    swcs_lane_list.append(swcs_category)
                elif swcs_category == "Hov":
                    lane_list.append(swcs_category)
                    swcs_lane_list.append(swcs_category)
                else:
                    # If our configurations for LCS are different
                    if swcs_category != lcs_boards_dict[gantry_id][j]:
                        modified = True
                    # We append to our lane list (we will always use) and swcs lane list (to see difference)
                    lane_list.append(lcs_boards_dict[gantry_id][j])
                    swcs_lane_list.append(swcs_category)
```

```
            # Append to list to be passed for creation of the XML diff
            compared_response_plans_list.append({'modified': modified, 'gantry_id': gantry_id, 'priority': priority,
                                    'board_config': lane_list, 'swcs_config': swcs_lane_list})
    # If we are extending upstream gantries by one gantry, copy the last gantry and add it
    if config["TOGGLES"]["add_gantry"]:
        add_id = list(lcs_boards_dict.items())[-1][0]
        add_config = compared_response_plans_list[2]['board_config']
        compared_response_plans_list.append({'modified': False, 'gantry_id': add_id, 'board_config': add_config})
    return compared_response_plans_list


def record_lcs(input_rp, input_time, output_rp, output_time):
    """
    Record RP from SwCS and RP from our evaluator in Apache Avro for later verification.

    Reminder: use ast.literal_eval (type:str) to parse string
    :param input_rp: XML of the SwCS response plan
    :param input_time: Timestamp before evaluating response plan
    :param output_rp: XML of the evaluator response plan
    :param output_time: Timestamp after evaluation has completed
    :return: None
    """
    schema = {
        'doc': 'LCS Response',
        'name': 'LCS Response',
        'type': 'record',
        'fields': [
            {'name': 'input_rp', 'type': 'string'},
            {'name': 'input_time', 'type': 'string'},
            {'name': 'output_rp', 'type': 'string'},
            {'name': 'output_time', 'type': 'string'},
        ],
    }
    parsed_schema = parse_schema(schema)
    input_rp = [{'input_rp': f"{input_rp}", 'input_time': f"{input_time}",
              'output_rp': f"{output_rp}", 'output_time': f"{output_time}"
              }]
    with open(os.path.join(base_config['install_path'], base_config['data_path_join'],
                      f"lcs_{date.today()}.avro"), 'a+b') as f:
        writer(f, parsed_schema, input_rp)


def parse_response_plan(rp_info, message_queue=None):
    """
    Takes an XML of a requested response plan and parses the refID, eventID, and responsePlanItemList.

    :param rp_info: XML of requested response plan, as a string straight off the WebSocket.
    :param message_queue: Shared message queue that takes messages to log in the messaging subsystem
    :return: Tuple of parsed values (ref_id, event_id, response_plan_items)
    """
    root = ET.fromstring(rp_info)
    if message_queue is not None:
        message_queue.put(('DEBUG', 'RESPONSE: Response plan request message root found.', {'XML': rp_info}))
    ref_id = xml_parse(root, 'refId', required=True)
    if message_queue is not None:
        message_queue.put(('DEBUG', 'RESPONSE: Parsed refID {}.'.format(ref_id)))
    event_id = xml_parse(root, 'eventId', required=True)
    if message_queue is not None:
        message_queue.put(('DEBUG', 'RESPONSE: Parsed eventID {}.'.format(event_id)))
    # Get list of LCS response plan items active and return it for inspection
    swcs_response_plan = []
    for item in root.findall('.//responsePlanItemList/responsePlanItem'):
        if "LCS" in xml_parse(item, 'itemId'):
            swcs_response_plan.append(item)
    if message_queue is not None:
```

```
        message_queue.put(('DEBUG', 'RESPONSE: Parsed responsePlanItemList {}.'.format(swcs_response_plan)))
    return ref_id, event_id, swcs_response_plan


def get_response_plan(event_id, event_cache, message_queue, result_queue):
    """
    Handle the operation of passing necessary information and response plans to be evaluated to the evaluator.

    :param event_id: ID of event to be evaluated
    :param event_cache: Shared data structure containing all the event data
    :param message_queue: Shared message queue that takes messages to log in the messaging subsystem
    :param result_queue: Queue of evaluated response plans
    :return: None
    """

    message_queue.put(('INFO', "RESPONSE: Requesting RP from evaluator; event_ID {}.".format(event_id)))
    # lcs_boards_dict has structure organized by gantry ID with corresponding list of config ex. {32: [1, 3, 3, 3], ...}
    lcs_boards_dict = evaluator.compute_lcs_boards(event_id=event_id, event_cache=event_cache,
                                       corridor_graph=response_plan_graph, message_queue=message_queue)
    # Put the result of the evaluator function on the queue for the function caller to read off.
    result_queue.put(lcs_boards_dict)
    message_queue.put(('DEBUG', "Placed LCS (event_ID {}) evaluation result on queue. Exiting process.".format(
        event_id)))


def manage_recommendations(event_cache: dict, data_cache: dict, message_queue: multiprocessing.Queue,
                           pid_tracker: dict) -> None:
    """
    Manage the processes of the recommendation subsystem.

    :param event_cache: Shared data structure containing all the event data
    :param data_cache: Shared data structure created by AI-DSS manager for data storage
    :param message_queue: Shared message queue that takes messages to log in the messaging subsystem
    :param pid_tracker: Tracker containing all the process IDs
    :return: None
    """

    response_eval_process_name = 'response_eval'

    def on_recommendation_ws_open(ws):
        """
        Indicate websocket connection for recommendations subsystem.

        :param ws: Recommendations websocket
        :return: None
        """
        ws.send("AI-DSS client connected")
        message_queue.put(('INFO', "STARTUP: Recommendations websocket connected."))

    def on_rp_request_message(ws, rp_info):
        """
        Function called when we get message over recommendations WebSocket.

        1. If message is 'OK', this was a ping response.
        2. Otherwise, initiate response plan evaluation and communication back to SwCS.
            2a. Parse response plan according to the schema (using `parse_response_plan()` function).
            2b. Create a process for the evaluation result, compare it to SwCS plan, and send necessary changes as XML
        :param ws: Recommendations websocket
        :param rp_info: Response plan request message. Should contain 'OK' (ping-related) or XML following RP schema.
        :return: None
        """
        # Initiate evaluation using a separate process, so that we can get the timeout on our side
        if rp_info == "OK":
            message_queue.put(('DEBUG', "RP PING: Got an OK from SwCS."))
            return
```

```
        message_queue.put(('INFO', "RESPONSE: Received request for response plan."))
        # Send XML for parsing of relevant information.
        # Current parse fields are response plan ID, event referenced ID, and SwCS response plan items.

        ref_id, event_id, swcs_response_plan = parse_response_plan(rp_info, message_queue=message_queue)
        message_queue.put(('INFO', "Response plan parsed. RP_ID {}, Event_ID {}".format(ref_id, event_id)))
        eval_result = multiprocessing.Queue()

        # Evaluation of the response plan
        # -------------------------------
        message_queue.put(('INFO', "RESPONSE: Starting response plan evaluation."))

        try:
            time1 = datetime.now()
            eval_process = multiprocessing.Process(target=get_response_plan,
                                        args=(event_id, event_cache, message_queue, eval_result),
                                        name=response_eval_process_name, daemon=True)
            eval_process.start()
            # Add the evaluator PID to the tracker while it's running.
            pid_tracker[response_eval_process_name] = eval_process.pid
            # Wait for the evaluator process to finish, with timeout.
            eval_process.join(timeout=config['TIMEOUTS']['RESPONSE_EVAL_TIMEOUT'])
            lcs_boards_dict = eval_result.get(block=True, timeout=2)
            # If we got past get_nowait then the queue was not empty
            message_queue.put(('INFO', "RESPONSE: Finished evaluating response plan.",
                               {'response_plan': lcs_boards_dict}))
            compared_response_plan_dict = compare_response_plans(swcs_response_plan, lcs_boards_dict)
            message_queue.put(('INFO', "RESPONSE: Response plan comparison complete.",
                               {'compared_plan': compared_response_plan_dict}))
            response_plan = create_response_plan_xml(ref_id, event_id, compared_response_plan_dict)
            if config["STORAGE"]["activate"]:
                record_lcs(rp_info, time1, response_plan, datetime.now())
            message_queue.put(('INFO', "RESPONSE: Converted response plan comparison to XML.",
                               {'response_plan_xml': response_plan}))
            # Remove the evaluator PID from the tracker, since it is finished.
            if response_eval_process_name in pid_tracker:
                del pid_tracker[response_eval_process_name]
            # Probably overkill, but go ahead and explicitly delete these multiprocessing objects.
            del eval_result
            del eval_process

        except Exception as e:
            message_queue.put(error_handler(e))
            message_queue.put(('ERROR', "RESPONSE: Response plan evaluation failed."))
            return

        # Send response plan to SwCS
        ws.send(response_plan)
        message_queue.put(('INFO', "RESPONSE: Sent updated response plan to SwCS."))

    # Run this loop continuously to get past any errors. WebSocketApp itself should run forever.
    while True:
        try:
            ws = mws.WebSocketApp(url=config['CONNECTIONS']['rp_websocket_url'],
                                  on_message=on_rp_request_message,
                                  on_open=on_recommendation_ws_open)
            ws.run_forever(ping_interval=10, ping_timeout=5, ping_payload="OK")
        except mws.WebSocketTimeoutException:
            message_queue.put(('WARNING', "RESPONSE: Timeout exception on WebSocket. Reconnecting..."))
        except Exception as e:
            message_queue.put(error_handler(e))


if __name__ == '__main__':
    print("NO CODE TO RUN")
```

B.16 subsys_vsl.py

```
# ------------------------------
"""
Contains VSL evaluation subsystem that watches for active events and computes VSL evaluation on each event until
the event closes and congestion surrounding the event dissipates.
"""
__file__ = 'subsys_vsl.py'

# ------------------------------

import math
import multiprocessing
import time
import xml.etree.ElementTree as ET
from fastavro import writer, parse_schema
from datetime import datetime, date
import os

import I24customwebsocket as mws
import evaluator
from graph import I24Graph
from utility import error_handler
from config.get_config import config, base_config


def create_override_xml(vsl_override: dict, cancel=False):
    """
    Build an XML of the necessary gantries to be overriden or canceled.

    :param vsl_override: Dictionary of gantry IDs and target speeds to be overriden
    :param cancel: Boolean value that is set to True if we need to send a cancel override
    :return: None
    """
    # Create ref ID to send to SwCS
    ref_id = math.trunc(time.time() * 1000)
    # If this is not a cancel VSL override request
    if not cancel:
        root = ET.Element("setSystemOverrideReq")
        ET.SubElement(root, "refId").text = str(ref_id)
        overrides = ET.SubElement(root, "overrides")
        for key in vsl_override.keys():
            override = ET.SubElement(overrides, 'override')
            ET.SubElement(override, "segmentId", providerName="vsl", resourceType="vslSegment",
                      centerId="Region 3").text = str(key)
            ET.SubElement(override, "speed").text = str(vsl_override[key])
        return ET.tostring(root, encoding='unicode', method='xml')
    # This is a cancel request
    else:
        root = ET.Element("cancelSystemOverrideReq")
        ET.SubElement(root, "refId").text = str(ref_id)
        for key in vsl_override.keys():
            segmentIds = ET.SubElement(root, 'segmentIds')
            ET.SubElement(segmentIds, "segmentId", providerName="vsl", resourceType="vslSegment",
                      centerId="Region 3").text = str(key)
        return ET.tostring(root, encoding='unicode', method='xml')


def compare_vsl_status(vsl_snapshot, computed_override):
    """
    Compare the status of the AI-DSS override and the configuration in SwCS.
```

```
60
61      :param vsl_snapshot: Dictionary containing the current configuration of VSL gantries in SwCS
62      :param computed_override: Dictionary containing the configuration calculated in the evaluator
63      :return: Dictionary containing VSL gantry IDs and their respective override speeds
64      """
65      compared_dict = {}
66      # Iterate through all gantry IDs
67      for id in vsl_snapshot.keys():
68          # If the configurations do not match, then add the AI-DSS configuration to the override
69          if vsl_snapshot[id][-1]['target_speed'] != computed_override[id]:
70              compared_dict[id] = computed_override[id]
71      return compared_dict
72
73
74  def send_override(override: str, message_queue: multiprocessing.Queue):
75      """
76      Connects to the VSL override websocket, sends the desired override, and does not close until confirmation/timeout.
77
78      :param override: String of contents of VSL override message to send over websocket
79      :param message_queue: Shared message queue that takes messages to log in the messaging subsystem
80      :return: Websocket response or False if timeout reached (logging handled by function)
81      """
82      try:
83          ws = mws.WebSocket()
84          ws.connect(url=config["CONNECTIONS"]["vsl_override_websocket_url"],
85                  timeout=config["TIMEOUTS"]["VSL_OVERRIDE_TIMEOUT"],
86                  header={'Authorization': 'Bearer banana'})
87          message_queue.put(('DEBUG', "Connected to VSL override websocket successfully. Preparing to send VSL override."))
88          ws.send(override)
89          message_queue.put(('DEBUG', "Sent VSL override to SwCS."))
90          ws.recv()
91          message_queue.put(('INFO', "VSL override successfully sent and received by SwCS."))
92          ws.close()
93      except Exception as e:
94          message_queue.put(error_handler(e))
95
96
97  def record_vsl(input_rds, input_vsl, input_time, output_vsl, send_xml, output_time):
98      """
99      Record RDS data from SwCS and VSL override from our evaluator in Apache Avro for later verification.
100
101     Reminder: use ast.literal_eval (type:str) to parse string
102     :param input_data: Snapshot of the RDS data for the relevant time period
103     :param input_time: Timestamp before VSL override evaluation
104     :param output_vsl: XML of the VSL override
105     :param output_time: Timestamp after VSL override evaluation has completed
106     :return: None
107     """
108     schema = {
109         'doc': 'VSL Response',
110         'name': 'VSL Response',
111         'type': 'record',
112         'fields': [
113             {'name': 'input_rds', 'type': 'string'},
114             {'name': 'input_vsl', 'type': 'string'},
115             {'name': 'input_time', 'type': 'string'},
116             {'name': 'output_vsl', 'type': 'string'},
117             {'name': 'send_xml', 'type': 'string'},
118             {'name': 'output_time', 'type': 'string'},
119         ],
120     }
121     parsed_schema = parse_schema(schema)
122     record = [{'input_rds': f"{input_rds}", 'input_vsl': f"{input_vsl}", 'input_time': f"{input_time}",
123             'output_vsl': f"{output_vsl}", 'send_xml': f"{send_xml}", 'output_time': f"{output_time}"
124             }]
```

```
    with open(os.path.join(base_config['install_path'], base_config['data_path_join'],
                           f"vsl_{date.today()}.avro"), 'a+b') as f:
        writer(f, parsed_schema, record)


def manage_vsl_eval(event_cache, data_cache, message_queue: multiprocessing.Queue) -> None:
    """
    Manage the VSL subsystem and VSL eval cache by running a 1-minute clock cycle.

    1. Check event cache to determine if any events are present which are not reflected in the VSL eval cache
    2. For each active VSL eval event, call the VSL evaluation function
    3. Determine if closed and non-congestion events should be removed from the VSL eval cache
    4. Communicate VSL overrides over websocket to SmartwayCS
    :param event_cache: Shared data structure containing all the event data
    :param data_cache: Shared data structure containing all other data streams
    :param message_queue: Shared message queue that takes messages to log in the messaging subsystem
    :return: None
    """
    # VSL evaluation cache
    # ----------------------
    # -- holds events that are currently being evaluated for VSL override (can be open or closed events)
    # -- NOTE: this doesn't need to be a multiprocessing/shared data structure since it's only used in one process
    vsl_eval_cache = {
        'events': {} # structure should be {eventID: (status, timestamp, milemarker, direction)}
    }

    # Load a stand-alone instance of the I24Graph object for use in VSL/RDS computation.
    vsl_graph = I24Graph(graph_directory=os.path.join(base_config['install_path'], base_config['repo_path_join'], config['GRAPH']['directory']),
                    blacklist_reload_interval=config['GRAPH']['blacklist_reload_interval'])

    # Infinite loop across the vsl_eval_cache to keep it updated and run evaluations.
    while True:
        time.sleep(60)
        # Populate vsl_eval_cache with all active events and their current data.
        for eid, edata in event_cache.items():
            vsl_eval_cache['events'][eid] = ('open', edata['timestamp'], edata['mile_marker'], edata['direction'])
        message_queue.put(('DEBUG', "Updated VSL evaluation cache with {} active events.".format(len(event_cache))))
        # Make sure every event in vsl_eval_cache that isn't in the event_cache is marked 'closed'.
        active_event_ids = list(event_cache.keys())

        for eid in vsl_eval_cache['events'].keys():
            if eid not in active_event_ids:
                # For now, we are just deleting events that are closed, even if congestion still occurring
                del vsl_eval_cache['events'][eid]
                # Need to copy out of eval_cache so that we can change one item, also it's in a shared data structure
                # closed_event_tuple = list(vsl_eval_cache['eval_events'][eid])
                # closed_event_tuple[0] = 'closed'
                # vsl_eval_cache['eval_events'][eid] = tuple(closed_event_tuple)
                # message_queue.put(('DEBUG', "Still evaluating closed event (ID={}) for VSL.".format(eid)))
        # Snapshot of RDS data created here to be sent for VSL override
        rds_snapshot = dict(data_cache['links'])
        vsl_snapshot = dict(data_cache['vsl'])
        # Evaluator is responsible for taking closed events out of the vsl_eval_cache when they're no longer relevant
        time1 = datetime.now()
        try:
            computed_override = evaluator.compute_vsl_override(vsl_eval_cache=vsl_eval_cache, rds_cache=rds_snapshot,
                                                vsl_snapshot=vsl_snapshot, message_queue=message_queue)
        except Exception as e:
            message_queue.put(error_handler(e))
            continue

        # Computed override only sends if it didn't error out.
        message_queue.put(('DEBUG', "Got VSL override (len={}) from evaluator.".format(len(computed_override))))

        try:
```

```
        # Compared override is a dictionary of differences between vsl_snapshot and computed_override; may be empty.
        compared_override = compare_vsl_status(vsl_snapshot, computed_override)
        message_queue.put(('DEBUG', "Override for VSL created: {}".format(compared_override)))
        # Override gets put into XML string. If empty, XML contains empty override field.
        formatted_override = create_override_xml(compared_override)
        # Save VSL data if activated in config.
        if config["STORAGE"]["activate"]:
            rds_tail = {id: val[-1] for id, val in rds_snapshot.items()}
            vsl_tail = {id: val[-1]['target_speed'] for id, val in vsl_snapshot.items()}
            record_vsl(rds_tail, vsl_tail, time1, computed_override, formatted_override, datetime.now())
        send_override(override=formatted_override, message_queue=message_queue)
    except Exception as e:
        message_queue.put(error_handler(e))


if __name__ == '__main__':
    print("NO CODE TO RUN")
```

B.17 utility.py

```
# ------------------------------
"""
Contains utilities to be used throughout the AI-DSS repository.
"""
__file__ = 'utility.py'

# ------------------------------

import traceback


def error_handler(e: BaseException):
    """
    Format a caught exception for handling and logging.
    Usage: message_queue.put(utility.error_handler(e))

    :param e: The exception that was caught
    :returns: Tuple of exception headline, 'ERROR' log level indicator, and extra dictionary containing more info
    """
    stacktrace = traceback.format_exc()
    msg = str(type(e))
    beg = msg.index("'")
    end = msg.index("'", beg + 1)
    error_type = msg[beg + 1:end]
    return (('ERROR', '{} has occurred. Stacktrace: {}'.format(error_type, stacktrace),
           {'message_type': 'error', 'error_type': error_type,
            'stacktrace': stacktrace}))


def xml_parse(tree, target, required=False):
    """
    Parse an XML tree for a specified target.

    :param tree: An XML tree
    :param target: The attribute of the tree we are looking for
    :param required: Boolean value set to True if the target is required
    :returns: String of the target, or None if not required
    """
    try:
        result = tree.find(target).text
        return result
    except AttributeError as e:
        # If the target is not required, don't raise an error - return None
```

```
        if not required:
            return None
        else:
            raise e
```

B.18 base_config.toml

```
install_path = '/AI-DSS/'
repo_path_join = 'I24-AI-DSS/'
logs_path_join = 'outputs/logs/'
data_path_join = 'outputs/responses/'
```

B.19 get_config.py

```
# ------------------------------
"""Converts appropriate .toml config file into python dictionary."""
# ------------------------------

import toml
import os


def convert_address(address):
    """
    Take address string from vandy_dev_config.toml and format it to a tuple recognized by log_writer.

    Example: sl_address = "atcmtd-scs.isis.vanderbilt.edu, 8000" converts to ("atcmtd-scs.isis.vanderbilt.edu", 8000).
    :param address: string of server address/IP and port
    :return: tuple format recognized by the logger
    """
    elements = address.split(",")
    return tuple((elements[0], int(elements[1])))


# Base config contains basic system values that are needed immediately without going into user-facing configs.
base_config_path = os.path.join(os.path.dirname(os.path.abspath(__file__)), 'base_config.toml')
base_config = toml.load(base_config_path)

# Config selector is located in user-facing configs.
config_selector_path = os.path.join(base_config['install_path'], base_config['repo_path_join'], 'config/choose_config.toml')
config_selector = toml.load(config_selector_path)

# verify that only one config is selected
got_true = False
filename = None

for key, boolean in config_selector.items():
    if boolean and got_true:
        raise Exception("Multiple config files selected.")
    if boolean:
        got_true = True
        filename = str(key)


# Get the absolute path to the X_config.toml file
if filename is not None:
    toml_path = os.path.join(base_config['install_path'], base_config['repo_path_join'],
                             'config/{}_config.toml'.format(filename))
else:
    raise Exception("Could not find config file.")
# Load the x_config.toml as a dictionary to be used by files in the AI-DSS
```

```
config = toml.load(toml_path)
# Convert addresses from the vandy_dev_config.toml for the log_writer.py file to interpret
if config['LOGGING']['connect_sl']:
    config['LOGGING']['sl_address'] = convert_address(config['LOGGING']['sl_address'])
if config['LOGGING']['connect_logstash']:
    config['LOGGING']['logstash_address'] = convert_address(config['LOGGING']['logstash_address'])
```

Appendix C

UAT document

Shown on the following pages is the UAT document from 11/10/2022 testing with TDOT.

1 AI-DSS Tests

Test Start Date/ Time	

Subsystems Required

- ☐ LCS, VSL, SAA, MAS

1.1 AI-DSS Startup

Step	Instructions	Expected Result	Pass/Fail	Notes
1	Start SwCS on the VM as well as the LCS and TSS simulators. Once those are booted up, start the AI-DSS system service.	Logs begin to send to Status Logger.		
2	Within StatusLogger, with log records filtered to the AI-DSS, check for recent messages filtered to "INFO".	There should be a recent log message showing "STARTUP: AI-DSS manager starting up." with several messages below indicating other processes starting up labeled STARTUP.	☐ Pass ☐ Fail	

1.2 AI-DSS Manager Heartbeat

Step	Instructions	Expected Result	Pass/Fail	Notes
1	Within StatusLogger, with log records filtered to the AI-DSS, check for recent messages after at least 30 seconds of startup.	There should be a recent log message showing "MANAGER HEARTBEAT: All processes are still running. Total system CPU percent usage is x%".	☐ Pass ☐ Fail	

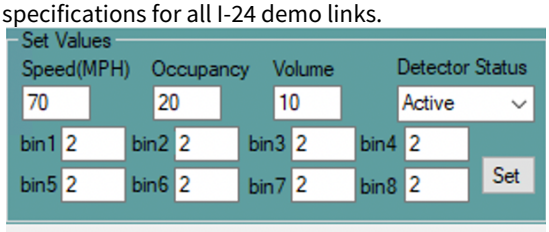
1.3 AI-DSS Data Cache Update

Step	Instructions	Expected Result	Pass/Fail	Notes
1	Within StatusLogger, with log records filtered to the AI-DSS, check for recent messages filtered to "INFO".	There should be a recent log message showing "CACHE: Detector data cache size: , Links data cache size: x, Link_geometry data cache size: x, DMS data cache size: x, LCS data cache size: x, VSL data cache size: x" which will gradually increase throughout testing.	☐ Pass ☐ Fail	

1.4 AI-DSS Event Heartbeat

Step	Instructions	Expected Result	Pass/Fail	Notes
1	Within StatusLogger, with log records filtered to the AI-DSS, check for recent messages filtered to "INFO".	There should be a recent log message showing "EVENTS HEARTBEAT: 0 Active Events." which will change later in testing once an event has been created.	☐ Pass ☐ Fail	

1.5 VSL Free Flow

Step	Instructions	Expected Result	Pass/Fail	Notes
1	Configure the TSS simulator to these below specifications for all I-24 demo links. 	Links begin to display values reflective of the set values.		
2	Let the simulator run for 2 minutes and settle at the values set. Go to the VSL status screen.	There should be no overrides sent to the VSL status screen from the AI-DSS since the system is in free flow.	☐ Pass ☐ Fail	

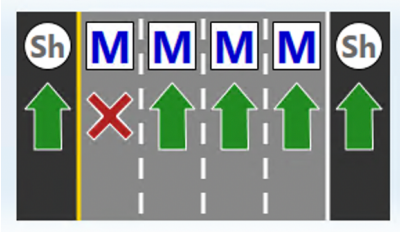
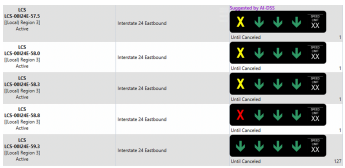
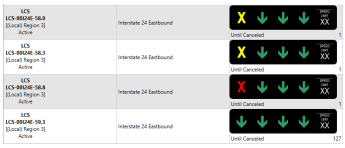
1.6 VSL Override for link speed reduction

Step	Instructions	Expected Result	Pass/Fail	Notes
1	Enter the TSS simulator and set the link segment speed for **demo-link-00I24W-58.2** to **35** mph, occupancy to **30**, and volume to **30**.	The "Current Values" (the upper blue section) of **demo-link-00I24W-58.2** should show a speed of **35,** occupancy of **30**, and volume of **30**.		

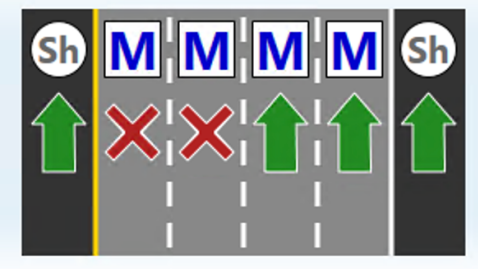
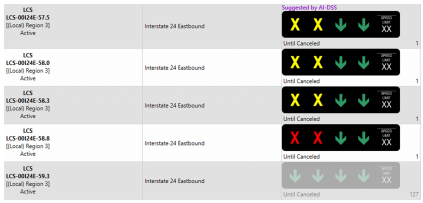
Step	Instructions	Expected Result	Pass/Fail	Notes
2	Let the simulator run for 2 minutes and settle at the values set. Go to the VSL status screen.	The following override values are set by the AI DSS: • VSL-00I24W-58.2 - **35** • VSL-00I24W-58.9 - **40** • VSL-00I24W-59.4 - **50** • VSL-00I24W-60.1 - **60** Each of the above segments should have an Override value. All VSL segments should display a speed of **70** not set by the AI-DSS.	☐ Pass ☐ Fail	
3	Review the log file that is configured at **systemOverridesLogPath**	There is a corresponding entry for the above system overrides.	☐ Pass ☐ Fail	
4	In the VSL status dialog, click the "Disable External Overrides" button. Take note of the previously overridden segments' Override column.	Each segment has an Override of **None**	☐ Pass ☐ Fail	

1.7 LCS - Single Left Lane Closure

Step	Instructions	Expected Result	Pass/Fail	Notes
1	Create an event, any type, any notifying contact, any status at: • County: DAVIDSON • Roadway: Interstate 24 • Direction: Eastbound • Reference Point: 59 MILE MARKER • Offset: Any	The event is created and the event creation window pops up.		

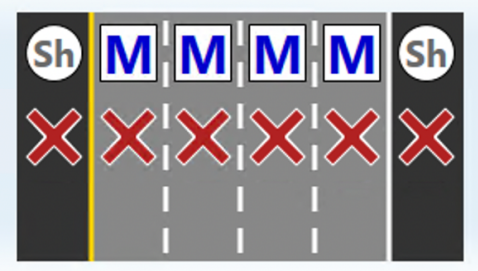
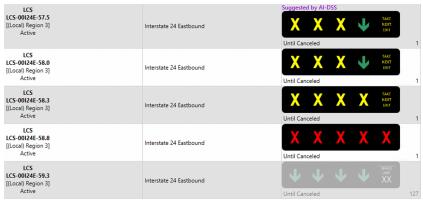
Step	Instructions	Expected Result	Pass /Fail	Notes
2	Using that window, block the leftmost lane only as shown below. Click save and suggest response plan. 	The following configuration should pop up indicating that it was suggested by the AI-DSS. 		
3	Accept and activate the response plan.	The response plan is activated on the corresponding devices.	☐ Pass ☐ Fail	
4	Review the log file that is configured at **suggestionsLogPath**	There is a new entry that corresponds to the response plan item marked as "Suggested by AI-DSS" (see image in the **Expected Results** of **Step 2**)	☐ Pass ☐ Fail	
5	On the System Settings dialog, check the "**Disable DSS Suggestions**" option and click the Save button. On the Event Details dialog, click Save and Suggest Response Plan.	The following response plan suggestion should appear (without a suggestion from the AI-DSS): 	☐ Pass ☐ Fail	
6	On the System Settings dialog, uncheck the "**Disable DSS Suggestions**" option and click the Save button.			

1.8 LCS - Multi Left Lane Closure

Step	Instructions	Expected Result	Pass/ Fail	Notes
1	Select the event used in the previous test.	The event window pops up.		
2	Using that window, block the two leftmost lanes only as shown below. Click save and suggest response plan. 	The following configuration should pop up indicating that it was suggested by the AI-DSS. 		
3	Accept and activate the response plan.	The response plan is activated on the corresponding devices.	☐ Pass ☐ Fail	

1.9 LCS - Full Lane Closure

Step	Instructions	Expected Result	Pass/ Fail	Notes
1	Select the event used in the previous test.	The event creation window pops up.		

Step	Instructions	Expected Result	Pass/ Fail	Notes
2	Using that window, block all lanes as shown below. Click save and suggest response plan. 	The following configuration should pop up indicating that it was suggested by the AI-DSS. 		
3	Accept and activate the response plan.	The response plan is activated on the corresponding devices.	☐ Pass ☐ Fail	

Test Result	
TDOT Witness	
SwRI Witness	
Test End Date/Time	

(end AI-DSS Tests)

References

(2013). Optimal variable speed limit control for real-time freeway congestions. *Procedia - Social and Behavioral Sciences*, 96:2362–2372. Intelligent and Integrated Sustainable Multimodal Transportation Systems Proceedings from the 13th COTA International Conference of Transportation Professionals (CICTP2013).

Akhtar Ali Shah, S., Kim, H., Baek, S., Chang, H., and Ahn, B. H. (2008). System architecture of a decision support system for freeway incident management in republic of korea. *Transportation Research Part A: Policy and Practice*, 42(5):799–810.

Carlson, R. C., Papamichail, I., and Papageorgiou, M. (2011). Local feedback-based mainstream traffic flow control on motorways using variable speed limits. *IEEE Transactions on intelligent transportation systems*, 12(4):1261–1276.

Chang, G.-L., Park, S. Y., and Paracha, J. (1999). Evaluation of freeway lane control for incident management. *Journal of Transportation Engineering*, 125.

Chang, G.-L., Park, S. Y., and Paracha, J. (2011). Intelligent transportation system field demonstration: Integration of variable speed limit control and travel time estimation for a recurrently congested highway. *Transportation Research Record*, 2243(1):55–66.

Cheng, Q. and Zheng, C. (2016). Research on the speed limit for highway traffic safety. *Material Science and Environmental Engineering*.

de Souza, A. M. and Villas, L. A. (2016). A fully-distributed traffic management system to improve the overall traffic efficiency. In *Proceedings of the 19th ACM International Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems*, page 19–26. Association for Computing Machinery.

Elasticsearch (2022). Elastic docs.

FHWA (2020). Integrated corridor management, transit, and mobility on demand.

Gregurić, M., Kušić, K., and Ivanjko, E. (2022). Impact of deep reinforcement learning on variable speed limit strategies in connected vehicles environments. *Engineering Applications of Artificial Intelligence*, 112.

Grumert, E. F., Tapani, A., and Ma, X. (2018). Characteristics of variable speed limit systems. *European Transport Research Review*, 10(21).

Guo, Y., Xu, H., Zhang, Y., and Yao, D. (2020). Integrated variable speed limits and lane-changing control for freeway lane-drop bottlenecks. *IEEE Access*, 8:54710–54721.

Hawas, Y. E. (2002). Calibrating simulation models for advanced traveler information systems/advanced traffic management systems applications. *Journal of transportation engineering*, 128(1):80–88.

Hegyi, A., Hoogendoorn, S. P., Schreuder, M., Stoelhorst, H., and Viti, F. (2008). Specialist: A dynamic speed limit control algorithm based on shock wave theory. In *2008 11th international ieee conference on intelligent transportation systems*, pages 827–832. IEEE.

INRIX (2022). NHTSA early estimates show overall increase in roadway deaths in first half of 2022, second quarter 2022 projects first decline since 2020.

Jayakrishnan, R., Oh, J.-S., and Sahraoui, A.-E.-K. (2001). Calibration and path dynamics issues in microscopic simulation for advanced traffic management and information systems. *Transportation Research Record*, 1771(1):9–17.

Kuhn, B., Balke, K., Brydia, R., Theiss, L., Tsapakis, I., Ruback, L., and Le, M. (2015). Evaluation of txdot variable speed limit pilot projects.

Lee, C., Hellinga, B., and Saccomanno, F. (2006). Evaluation of variable speed limits to improve traffic safety. *Transportation research part C: emerging technologies*, 14(3):213–228.

Meier, R., Harrington, A., and Cahill, V. (2005). A framework for integrating existing and novel intelligent transportation systems. In *Proceedings. 2005 IEEE Intelligent Transportation Systems, 2005.*, pages 154–159.

Müller, E. R., Carlson, R. C., Kraus, W., and Papageorgiou, M. (2015). Microsimulation analysis of practical aspects of traffic control with variable speed limits. *IEEE Transactions on Intelligent Transportation Systems*, 16(1):512–523.

Murley, P., Ma, Z., Mason, J., Bailey, M., and Kharraz, A. (2021). Websocket adoption and the landscape of the real-time web. In *Proceedings of the Web Conference 2021*, WWW '21, page 1192–1203, New York, NY, USA. Association for Computing Machinery.

NHTSA (2022). Nhtsa early estimates show overall increase in roadway deaths in first half of 2022, second quarter 2022 projects first decline since 2020.

Osaba, E., Onieva, E., Moreno, A., Lopez-Garcia, P., Perallos, A., and Bringas, P. G. (2016). Decentralised intelligent transport system with distributed intelligence based on classification techniques. *IET Intelligent Transport Systems*, 10(10):674–682.

Stantec (2021). High level design for TDOT ICM DSS.

Van Toorenburg, J. and De Kok, M. (1999). Automatic incident detection in the motorway control system mtm. *Bureau Transpute, Gouda*.

Vrbanic, F., Ivanjko, E., Kušić, K., and Cakija, D. (2021). Variable speed limit and ramp metering for mixed traffic flows: A review and open questions. *Applied Sciences*, 11.

Yang, Q., Koutsopoulos, H. N., and Ben-Akiva, M. E. (2000). Simulation laboratory for evaluating dynamic traffic management systems. *Transportation Research Record*, 1710(1):122–130.

Zhang, N., Wang, F.-Y., Zhu, F., Zhao, D., and Tang, S. (2008). Dynacas: Computational experiments and decision support for its. *IEEE Intelligent Systems*, 23(6):19–23.

Zhang, Y. and Ioannou, P. (2016). Combined variable speed limit and lane change control for highway traffic. *IEEE Transactions on Intelligent Transportation Systems*, PP:1–12.

Zhang, Y., Quinones-Grueiro, M., Barbour, W., Weston, C., Biswas, G., and Work, D. (2022). Quantifying the impact of driver compliance on the effectiveness of variable speed limits and lane control systems. In *2022 IEEE 25th International Conference on Intelligent Transportation Systems (ITSC)*, pages 3638–3644.