



# Decision provenance in a real-time system with microservice architecture

Master's thesis in Computer science and engineering

Daniel Willim
Erik Ljungdahl

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2022

Master's thesis 2022

# Decision provenance in a real-time system with microservice architecture

Daniel Willim
Erik Ljungdahl

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2022

Decision provenance in a real-time system with microservice architecture
Daniel Willim
Erik Ljungdahl



Supervisor: Alejandro Russo, Department of Computer Science and Engineering
Advisor: Philip Andreasson, Carmenta Automotive
Examiner: Andreas Abel, Department of Computer Science and Engineering

Master's Thesis 2022
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2022

# Abstract

This thesis investigates the feasibility of decision provenance, by implementing it in Carmenta TrafficWatch, a real-time system with a microservice architecture. We add provenance by creating a separate service that listens to the message-bus and stores all events and potential changes to those events. This provenance information can later be queried by other services. To have provenance of how events interact with each other, we also implement a system for event aggregation. Our findings indicate that adding provenance to real-time systems with microservice architecture will not impact the overall performance of the system significantly, and is a viable solution to add accountability and increase the understanding of complex systems.



Keywords: decision provenance, microservice, provenance, Computer science, thesis, real-time.

# Contents

# List of Figures

# List of Tables

# 1 Introduction

A recent paper by Singh *et al.* [1] argue that provenance can be used as a way to track decisions in complex systems and increase accountability in these systems, they call this *decision provenance*. Their paper motivates the use of decision provenance and points out that it needs to be tested in real systems to see if the idea is viable in reality. This is what we aim to investigate within the field of connected and autonomous vehicles (CAVs).

In recent years there has been a lot of research in the field of CAVs with many successful examples [2]. Most of these vehicles only rely on on-board sensors to see what is in the immediate surroundings of the car. However, there are lots of other off-board information that could be useful, for example, an emergency vehicle is approaching or other vehicles have experienced the curve ahead to be slippery.

To make this off-board information accessible Carmenta has developed Carmenta TrafficWatch [3], a cloud platform for CAVs. It collects data from different sources, such as road condition, weather and current road work, in order to send recommendations to CAVs based on the data. For example, there might be data indicating that a road segment is slippery. For CAVs soon approaching that road, a warning can be received recommending to lower the speed.

Another application of Carmenta TrafficWatch is as a control tower, where a fleet manager can manage several CAVs and have an overview of the status of all monitored roads. In this case, when it received data that a road segment was slippery, this would immediately be visualized on their map, which they can use to re-route their fleet, take manual control of the CAV or lower speeds as necessary.

**Real time requirements** Traffic situations might change in the blink of an eye, which makes Carmenta TrafficWatch a real-time system. A disabled vehicle may have started working again or been towed away, road works might be extended, etc. Since the most important thing is current traffic situations, the old data is often thrown away and there is not much historic data stored. Data that could be used for analysis in order to get a better understanding of the system and to increase accountability. This data together with some metadata is provenance data. But there are several challenges with storing provenance over a longer period of time. The system still needs to work in real-time to react instantly to incoming data, and hence any solution would need to not slow down the overall performance.

**Incident aggregation** Another interesting aspect of provenance in Carmenta TrafficWatch is how incidents on the same road interact with each other. First, there can be incidents that enhance each other and make the traffic situation more dangerous. An example is combining fog and road work since drivers might not see road signs or people on the road in time. Secondly, if three data sources

all report that there is an accident on a specific road, does that mean there are three accidents or are they the same accident? Probably the latter, but it could also be that there are three different vehicles in the same accident, and they have all sent a message to the cloud that they have been in an accident. Finally, different data sources may report different types of incidents on a road that do not enhance each other, but are still related. For example, a weather service reporting sub-zero temperatures, sensors on vehicles noticing low friction, and another weather service reporting risk of precipitation. They may all indicate that there can be a slippery road due to ice, but the combination of all three incident makes the situation more certain.

When there are several incidents on the same road segment, it poses the question on how you inform the user of these. Do you send them out one by one? However, as discussed above, they may all point to the same accident, and then if the user were to get notified about each and every one separately, it provides duplicate information. As an operator, too much information requires more analyzing on how these incidents interact with each other, figuring out if they are the same thing or multiple things that enhance each others severity. This takes time which can distract from things that may be more severe and need urgent attention. As a driver one only wants to see the relevant information and not be overwhelmed with unnecessary information that can take away focus from the road. For these reasons we want incident aggregation, which also offers a good playing ground for provenance.

**Solution** Our thesis tackles these problems by collaborating with Carmenta [4]. We will use their product Carmenta TrafficWatch as a test bed to investigate if decision provenance is feasible. Through the reminder of this report we will refer to this test bed as TrafficWatch but our implementation and results are generalizable to any system with a micro-service architecture that communicate over a message bus.

**Thesis overview** The remainder of the thesis is organized as follows, section 2 *TrafficWatch* and 3 *Provenance* gives background knowledge about TrafficWatch and related research topics that the reader need to understand the thesis. section 4 *Aggregated incidents* and 5 *Provenance in TrafficWatch* describe how we solved the problems outlined in the introduction and what decisions we made along the way. Finally section 6 *Performance tests* and 7 *Discussion* describe and discuss our findings.

## 2 TrafficWatch

TrafficWatch is built upon a microservice architecture, that is, the system is built from small, loosely-coupled services, each with one clear purpose. They communicate with each other over a message bus. With this architecture, TrafficWatch is modular and easy to adapt to the current application by, for example, adding new data sources or different services.

**Table 1: Properties of Situations and Incidents in TrafficWatch.** A ✓indicate that the message contains that property.

| Property | Purpose | Situations | Incidents |
|---|---|---|---|
| Id | Unique identifier | ✓ | ✓ |
| Metadata | Related information about the situation, stored in a JSON string e.g. name of related road | ✓ | ✓ |
| Severity | 5 step scale from *None* to *Highest* of how severe the event is. | ✓ | ✓ |
| RoadIds | Ids of related roads to the event | ✓ | ✓ |
| Time | Start and endtime of the event. | ✓ | ✓ |
| Probablity | 3 step scale of how probable the event is, *RiskOf*, *Probable* or *Certain* | ✓ | ✓ |
| Type | Type of event, e.g. a road work, accident or a slippery road | ✓ | ✓ |
| Source | Data source that sent the event | ✓ | ✓ |
| Caused by | Information about what situation caused the incident | | ✓ |
| Status | Indicates if the event is *open*, *handled*, *closed* or *archived* | | ✓ |
| Creation time | Time that the event was created | | ✓ |

**Message bus** There are two ways to communicate via the message bus. A direct service-to-service message, where the sender specifies which service should receive the message. The other option is a message broadcast, where the receiver chooses what message types to listen for. When the sender sends such a message it is called that they publish the message. In TrafficWatch, mostly message broadcasts are used, the only time when direct service-to-service messages are used is when a service sends a command to another service.

**Supervised roads** One can dynamically specify which roads are of interest and should be monitored, these are called supervised roads. This can be useful in different ways, one might only be interested in roads of a certain size, or for a control tower, only the roads that their fleet is expected to use is of interest.

**Event types** In TrafficWatch there are two types of events, *situations* and *incidents*, they are structured according to Table 1. Situations indicate that something have happened at a location. This information is supplied by one of the connected data sources. For example, a reading of how slippery a road is would create a situation of type *Slippery Road* with a severity according to the friction of the road. Incidents on the other hand are caused by situations that occur on a supervised road and can for example be used to notify a CAV of dangers ahead or the operator of a control-tower.

## 2.1 A common use-case

Because of the microservice architecture, TrafficWatch is very modular and can have different functionality according to what services are running. One common configuration is illustrated in Figure 1, where TrafficWatch functions as a control tower. In this configuration there are two running data services, connected to the external APIs. Data service 1 publishes situations with traffic information, for example current or planned roadworks and data service 2 publishes situations with information about how slippery roads are.

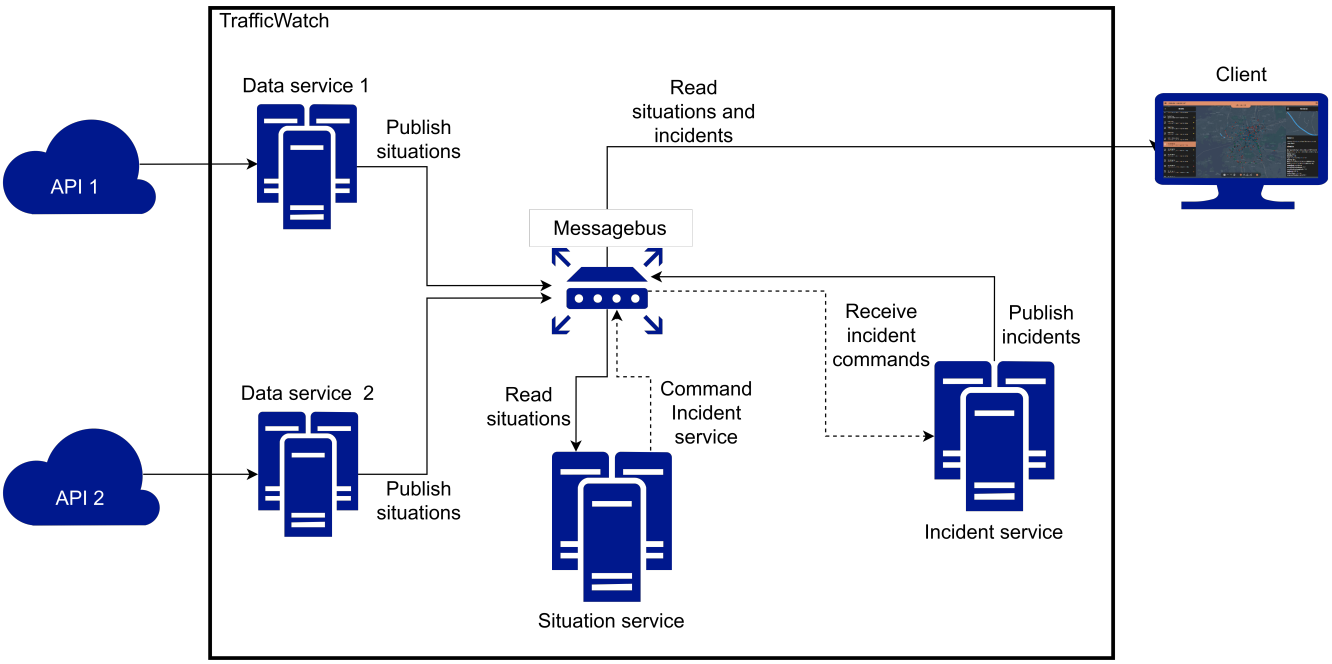


**Figure 1: An overview of a common use-case of TrafficWatch.** Here it is connected to two external data sources and a control tower client. Arrows indicate that a service publishes or reads something to or from the message-bus. Dashed arrows indicate that the service sends a direct message to another service.

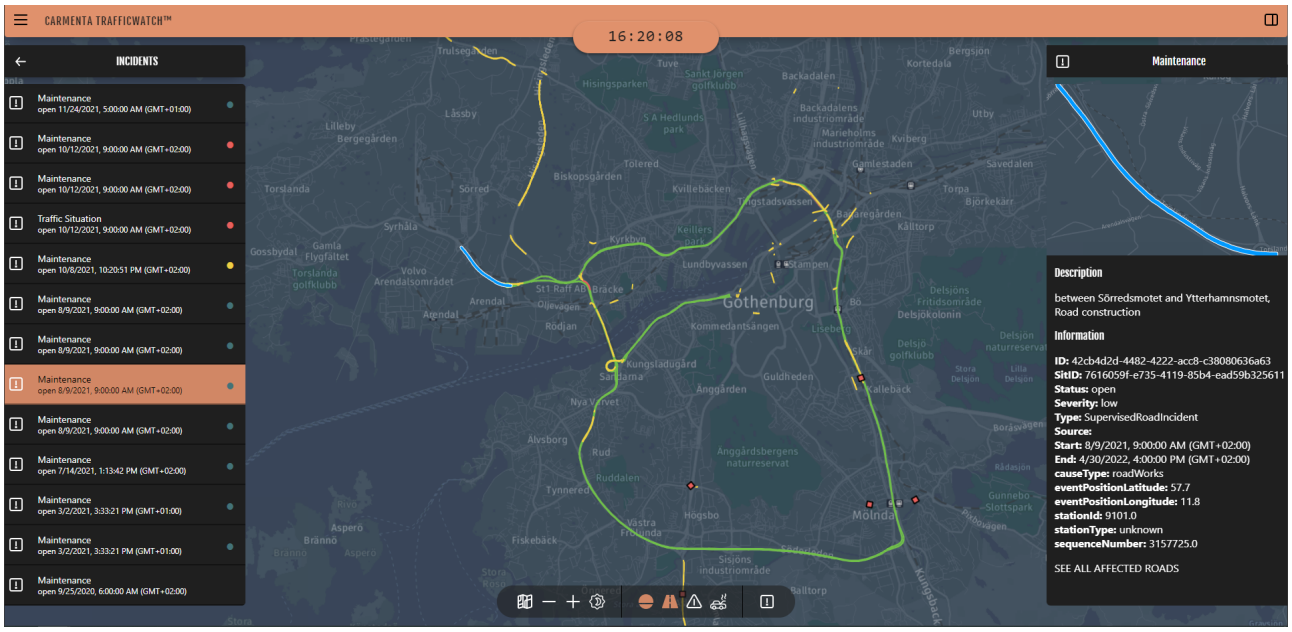


**Figure 2: A screenshot of TrafficWatch functioning as a control tower over Gothenburg.** In the middle of the figure is a map of Gothenburg, where the green roads are supervised. In the left panel is a list of all currently active incidents and in the right panel is information about the currently selected incident. The red squares are connected vehicles updating the system with their current position and status.

These situations are then consumed by the situation service which checks if they are severe enough to create an incident from the situation, this can for example be that the situation happened on a supervised road. If this is the case the situation service sends a command to the incident service to create an incident from the situation. This incident is then published on the bus by the incident service.

Both situations and incidents will be consumed by the control tower client to give the end user an overview of the current situation and be able to inspect both event types. A screenshot of the client can be seen in Figure 2, where an incident is selected. The incidents are most important since they happen on supervised roads and will likely affect the fleet. One might think that situations are not interesting at all, however this information can be used when a vehicle inevitably needs to be rerouted. Or there might be an incident warning about traffic congestion, but to know why, one has to look at an unsupervised road nearby, where there might have occurred an accident.

When something related to an event changes in a data source, for example that the end time of a roadworks is extended, that data service will publish a

new situation with the same id but with updated parameters, this will then be received by the situation service who commands the incident service to create a new incident with the same id but updated parameters.

# 3 Provenance

As described by Herschel *et al.* [5], "Provenance refers to any information describing the production process of an end product, which can be anything from a piece of digital data to a physical object" and have been used in different applications such as food supply chains and data handling in scientific experiments at CERN.

Provenance can further be broken into sub-levels; provenance meta-data, information system provenance, workflow provenance, and data provenance. There are no clear boundaries between the levels but the higher you go there are more data tracked about each item, but requires that you have better control and knowledge of the system [5], [6].

Provenance can use different capturing methods, disclosed and observed [1], [7]. Disclosed provenance is implemented into the code and can thus have full control over what information to capture. Observed provenance only sees things as an outsider, it can only capture information based on the input and output of an application. A practical solution might use a mix of both methods to achieve the best result.

## 3.1 Workflow provenance

In workflow provenance we track information about the flow of data though the system, how the data changes over time and system parameters. Workflow provenance can take different forms, depending on what the goal is, but is often one or a combination of the following; *prospective provenance*, captures static content about the workflow, often to get an abstract view of the workflow. *Retrospective provenance* captures information about execution, for example accessed or generated resources, often in the form of an execution trace. *Evolution provenance* captures information about the difference between different runs, that is changes in for example input or system configuration [8].

## 3.2 Decision provenance

As previously mentioned, Singh *et al.* [1] introduces the concept of decision provenance, a subcategory of workflow provenance where the focus is on what data caused a certain decision.

They further argue that as we increasingly depend on technology in our everyday lives these systems grow more complex and the root of an action is often hard or impossible to know. But at the same time these systems control important parts of our lives and there is a need to know what caused an action. This is where they propose to use decision provenance to increase the system transparency and accountability. However, more research is needed to show that this is a feasible idea and how this could be implemented in real system [1].

The paper by Singh *et al.* [1] is the main cornerstone that we build our thesis on,

since it is a new paper that clearly argues that their theory of decision provenance needs to be tested in reality, which is our goal. As decision provenance is the main focus of this thesis, we will sometimes refer to *decision provenance* as just *provenance* to improve readability.

# 4 Aggregated incidents

In order to show how incidents interact with each other and the provenance of that, we have implemented aggregation of incidents in TrafficWatch. When receiving incidents which overlap in time and are on the same road segment, we combine it to one bigger aggregated incident with several sub-incidents. It sends this aggregated incident to the client instead of the client receiving several incidents with no clear information about how they interact.

An aggregated incident show what weights the different sub-incidents have on the combined incident. This is based on their severity and probability, first introduced in section 2 *Event types*. On top of that it takes a configurable policy specification into account, specifying which incidents enhance each other. In Figure 3 you can see a simple visualization of an aggregated incident in a client as a proof of concept.

How an aggregated incidents is visualized is important for an operator to easily get a hold of the situation. In our simple visualization the sub-incidents are ordered according to weight in order to show the most important thing first. For each sub-incident we decided to show its weight, incident type, severity, and probability. This is also the main information used for aggregating incidents in the back-end. Supplying the same information to the operator gives a transparent system, where it is possible to understand how the system combined the incidents. However, in a real product one might want the option to expand a sub-incident to get a more detailed view, while showing a simplified view with the most important information by default.

In Table 2 we have listed what data the whole aggregated incident consists of. We have structured an aggregated incident so that it has the joint information of the sub-incidents at the top level, so that the sub-incidents can omit those parts. This results in less duplicate information and more space efficiency.

## 4.1 Policy specification

The incident aggregation follows a policy specification to allow for smarter aggregation. The focus has mainly been different incident types that can enhance each other. Some examples of incident combinations that might enhance each other, and the reason why, are:

1. Accident and slippery road - braking and swerving becomes more difficult, which can result in another accident.
2. Road work and fog - might not see road signs or people on the road in time.
3. Raining and sub-zero temperatures - risk for black ice.
4. Accident and road work - lots of people on the road.


**Figure 3: Aggregated Incident in a UI.** Here you can see that an aggregated incident consists of several sub-incidents and what information we display for each one.

**Table 2:** Properties of an aggregated incident.

| Property | Purpose |
|---|---|
| Id | Unique identifier |
| Status | Indicates if the incident is *open*, *handled*, *closed* or *archived* |
| Severity | 5 step scale from *None* to *Highest* of how severe the aggregated incident is |
| RoadId | Id of the road related to the incidents |
| Geometry | The geometry of the road |
| Source | A comma-separated set of sources |
| Time | Overall Start and end time of the incidents |
| Sub-incidents | Each one contains mostly the same thing as Incidents in Table 1, except things that are the same for the whole aggregated incident, such as RoadID, Geometry, and Status. |
| Weights | Each sub-incident has a weight indicating how much it contributes to the aggregated incident. |
| Triggered Rules | A list of what rules in the policy specification have triggered for this aggregated incident. |

The policy specification has been modelled so that it has a list of rules that can trigger. Each rule consists of a list of conditions that all need to be met, and a multiplier of how much the triggered rule should increase the severities of an aggregated incident. Multiple rules can trigger and they are then all multiplied with each other. As multiplication is associative we get something that is independent of order and thus easy to customize.

Each condition always has an incident type and that is also the simplest case. There are three possible additions to the condition. The first is a requirement of a minimum severity of the incident. Another is that it can require that there are several incidents of the same type, such as requiring three slippery road incidents. The final one is that it can require those incidents to be from different sources.

The policy specification is supplied as a JSON file. Below we have an example of such a policy specification with three rules: the example incident combinations 1 and 5, as well as requiring three slippery roads from different sources.

```
[
    {
        "name": "Road Work Accident",
        "sevInc": 1.6,
        "conds": [
            {
                "type": "Accident",
                "minSev": "Low",
```

```
                "reps": 1,
                "unique": false
            },
            {
                "type": "CONSTRUCTION",
                "minSev": "Medium",
                "reps": 1,
                "unique": false
            }
        ]
    },
    {
        "name": "Slippery Accident",
        "sevInc": 2,
        "conds": [
            {
                "type": "Accident",
                "minSev": "Medium",
                "reps": 1,
                "unique": false
            },
            {
                "type": "Slippery Road",
                "minSev": "Low",
                "reps": 1,
                "unique": false
            }
        ]
    },
    {
        "name": "Certain Slippery Road",
        "sevInc": 1.5,
        "conds": [
            {
                "type": "Slippery Road",
                "minSev": "Low",
                "reps": 3,
                "unique": true
            }
        ]
    }
]
```

As you can see, each rule also has a name. This is used as an identifier for keeping track of what rules have triggered. This is stored together with the aggregated incident in the provenance service that we will talk more about in

subsection 5.3 *Provenance Service*. With this information we can know what changes an aggregated incident has gone through.

As we assume that the name of a rule is unique in order to be used as an identifier, if one changes a rule without changing its name, one loses the exact meaning of a rule. Especially since the provenance service has persistent storage and can store things from a long time ago. This could be mitigated by either adding provenance for the policy specification, or setting recommendations for the operator to name the rules carefully. Either give them meaningful names, such as "3 slippery roads", or make a separate specification document with unique keys, where one needs to look it up, which would allow less meaningful names, such as "Rule 6".

We made the policy specification configurable due to several reasons. The first is that the domains that TrafficWatch can be used for are several. A car, truck, and a boat might not all be affected in the same way. But they are all vehicles that could use this application. This means that the incident types might differ, and what combinations are bad might change depending on the domain. The second reason is that we are not experts in this field and could only speculate on what incident combinations are bad, so we want to leave that to the more knowledgeable in the field. These are some of the reasons why we wanted a configurable policy specification that the operator can set and easily tune.

## 4.2 Weight calculation

For each sub-incident in an aggregated incident we calculate a weight based on their probability, severity, and the policy specification. This weight later transforms into a percentage of how much an sub-incident contributes to the aggregated incident. This percentage is useful to give the user a clear overview about what the most important sub-incidents are, so they do not focus on sub-incidents that are minor in comparison. In order to calculate weights with the probabilities and severities, we had to transform them into numbers. Since TrafficWatch has a wide range of use cases that all have different needs, the specific numbers of what the severities and probabilities correspond to are supplied via a custom JSON file and are thus configurable by the system's end-user. An example of how these numbers could be set up, which will be used in coming examples, is as follows:

```
{
  "Severity": {
    "None": 0,
    "Low": 1,
    "Medium": 5,
    "High": 25,
    "Highest": 125
  },
  "Probability": {
```

```
    "RiskOf": 1,
    "Probable": 1.1,
    "Certain": 1.2
  }
}
```

These provided weights along with the potentially fulfilled rules of the policy specification are used to calculate the weights for each sub-incident, with the formula

$$w_{si} = Severity_{si} \cdot Probability_{si} \cdot \prod FulfilledRules$$

As an example: If we have an *Accident* with *High* severity and *Probable* probability, while the first two rules of the example policy specification triggered, the calculation would be:

$$w_{Accident} = 25 \cdot 1.1 \cdot (2 \cdot 1.6) = 88$$

The sum of all sub-incident's weights are used to determine the aggregated severity by checking which threshold it passes. Example: If the sum is 27.3, the severity becomes *High*, or 5 becomes *Medium*. The formula for the sum is

$$sum = \sum_{si=1}^{n} w_{si}$$

This sum is also used to normalize the weights so that the sub-incidents' weights becomes an percentage of how much an sub-incident contributes to the aggregated incident. The normalized weight is calculated as

$$\hat{w}_{si} = \frac{w_{si}}{sum}$$

In the example JSON configuration we have modelled the weights exponentially, since we thought that was fitting when the total weight is a sum of products. However, this might not be the right semantic meaning of the set definitions of the severities and probabilities. A linear or nonlinear model might be more correct, or the base of the exponential might need to be changed. That is why we decided to have the weights customizable so that the operator or an expert can tune the systems to one's needs. With an exponential model, a base of 5 means that with 5 *Low* severity sub-incidents, the combined severity becomes *Medium*, and with 25 *Low* severity sub-incidents it becomes *High*, disregarding other factors.

If one of the probabilities' corresponding weights are below 1, one might actually get a severity that is less than the maximum severity of an sub-incident. This is

probably not wanted unless one wants different semantic meanings of severities depending on if it is a single incident or an aggregated incident. An example of when such a situation could occur is with the probability weights

```
{
 "RiskOf":   0.9 ,
 "Probable": 1.0 ,
 "Certain":  1.1
}
```

together wih one sub-incident with *Low* severity and *Certain* probability, and one sub-incident with *High* severity and *RiskOf* probability:

$$w_1 = 1 \cdot 1.1 = 1.1$$

$$w_2 = 25 \cdot 0.9 = 22.5$$

$$sum = 1.1 + 22.5 = 23.6$$

The sum would not quite meet the threshold for High severity, 25, and would thus result in *Medium* severity, which is lower than if we only had the second sub-incident which had *High* severity. Having a probability weight below 1 is therefore not supported.

## 4.3 DSL for policy specification

We had ideas of creating a domain specific language (DSL) for more easily expressing these policies. The DSL could generate the JSON needed for instantiating the rules. That could help the operator to write correct specifications, since a JSON file can not check for spelling errors in incident types and severities, something that a DSL could.

We started to look at how one would like to express oneself in a DSL by sketching on a labelled BNF grammer. BNF is a common notation for syntax of languages [9], [10]. We propose the following BNF grammer for these policies:

```
Policy. Policy ::= [Rule] ;

Rule.   Rule ::= String Double ":" [Condition] ;
separator Rule "" ;

Inc.    Condition1 ::= Incident ;
MinSev. Condition1 ::= Severity Incident ;
_.      Condition  ::= Condition1 ;
Times.  Condition  ::= Integer "*" Condition1 ;
UTimes. Condition  ::= Integer "x" Condition1 ;

separator Condition "&&" ;
```

```
rules Incident ::= "Slippery Road" | "Accident" | "Wind" ;
rules Severity ::= "None" | "Low" | "Medium" | "High"
                         | "Highest" | "Unknown" ;

comment "--" ;
comment "{-" "-}" ;
```

Where "Times" is for indicating how many times a incident type has to occur for the rule to trigger. And "UTimes" refers to the same thing but with unique times instead, so the sources has to be different from each other. To express several conditions that need to be met for a rule, we combine them with the logical and-operator `&&`. To get a feel of how this actually would look like, here is an example policy:

```
Certain Slippery Road 1.3 : 3 * Slippery Road
Very Slippery Road    1.9 : 2 x High Slippery Road
Slippery Accident     2.3 : Medium Slippery Road && Accident
```

### 4.3.1 Digression - Potential syntactic sugar

We had the idea that one might want to use the logical or-operator `||` as syntactic sugar. This would enable having one line for everything interacting with 1 type of incident, which would make it organized when there are things that combinate badly with lots of things. One such example is accidents, whose rule could then be expressed as:

```
Accident comb 1.5  : Accident &&
                     ( Slippery Road || Fog || Precipitation )
```

By using the distributive property of the logical operators it would be equivalent to:

```
Accident Slippery Road 1.5  : Accident && Slippery Road
Accident Fog           1.5  : Accident && Fog
Accident Precipitation 1.5  : Accident && Precipitation
```

It looks quite clean with the syntactic sugar, but we lose the ability to customize the severity multiple per incident type. That could maybe be fixed if we were to incorporate the severity multiplier within the parentheses. However, that would start looking quite messy. Moreover, when adding more rules we realize that it doesn't achieve the goal of having one line for everything interacting with 1 type of incident. As a Slippery Road might have other bad combinations:

```
Slippery Road comb 1.6  : Slippery Road && ( Fog || Road Work )
```

Now we have two rows where Slippery road is mentioned. We also get a dilemma, do we add "Accident" within the parentheses here as well, but then we have duplicate information that "Accident" and "Slippery Road" combinate badly which is not wanted. So in this case we would give the user an not obvious

choice on where to put the rule, and if there are multiple people involved, which there often are in software projects, they might choose differently and it becomes harder to read. This is not ideal for a DSL that aims to make it easier for the user, and that is why we didn't include the "||" operator in the grammar.

# 5 Provenance in TrafficWatch

When planning how to implement a system for decision provenance in TrafficWatch, three different ideas were considered; adding provenance to the metadata of each message described in subsection 5.1 *Message metadata*, altering the message broker to store provenance for all messages described in subsection 5.2 *Message broker* or creating a separate service that handles provenance described in subsection 5.3 *Provenance Service*. As discussed below each idea have pros and cons and is suited for different use-cases. For TrafficWatch we found that the third idea, the provenance service, is the most suitable solution.

## 5.1 Message metadata

Our first idea to achieve decision provenance was to add provenance information to the metadata of each message. Here each service that consumes or creates messages would add information to the metadata about the origin of the message and what changed in that service. This information could then be displayed in the client by reading the provenance from the metadata of received situations and incidents.

This implementation would fall in the disclosed category according to the categorisation of Singh *et al.* [1] and since the provenance is created inside the services, a highly detailed provenance could be achieved. But there are also several problems with this implementation for TrafficWatch. First, there is not much data that is interesting to add to a single message, the situations and incidents already contain the interesting parts and it is a quite linear system and not many advanced decisions are being made. Other interesting data could first be achieved when a situation is changed. However, to know what has changed one would need the original message, but that has already been sent away and is not stored within each service. Second, since the provenance is only stored in each message there is no inherent way to get a history of events and their provenance. This solution would also not be that flexible, for each old or new service extra code would need to be written to have working provenance for that service. Another problem with this solution is that there would be no guarantees that all running services have implemented provenance.

Because of the problems with this implementation mentioned above, we did not explore this solution further. However, in other situations this approach might be viable, and we will discuss when what is in subsection 7.2 *Use cases*.

## 5.2 Message broker

Another solution that was discussed, would be to implement some kind of provenance directly in the message broker, for example by storing both sender and receiver of each message as well as the contents of the message. This would fall much closer to the observed provenance category [1], seeing how and what services communicate with each-other but not exactly what happens in each

service. But as we are still quite near the service level, and not observing from the operating system level, we still fall somewhat into the disclosed category. Thus we are somewhere in between disclosed and observed provenance

As described in section 2 *TrafficWatch* there are two ways to send a message via the message broker, either sending a direct message or publishing a message. Thus, a message broker with provenance could listen to and store information about all messages sent between services. The design could otherwise be similar to how the provenance service function, which will be described in subsection 5.3 *Provenance Service*. But with the advantage that it could listen to all messages and not only published messages. Another advantage with this solution is that the provenance would always be collected since the message broker need to be running in order for the system to work.

But to implement this solution one would need to modify or implement a new message broker, something that is both time consuming and complex. Since all relevant messages are published in TrafficWatch the same result could be accomplished by simply having another service listening to all published messages. For these reasons we did not choose this solution but in other domains where there are more direct messages sent between services this solution might be relevant, as we will discuss in subsection 7.2 *Use cases*.

## 5.3 Provenance Service

Our final idea and the one that we implemented was to create a separate service that listens to messages published to the bus and stores them in a database, but as discussed above it cannot listen to direct messages between services. This is currently not a limitation since all TrafficWatch messages that are of interest is already published messages instead of direct messages. And with control over the code one could choose to publish messages instead, when needed. The service also detects and stores changes when events are updated, this information can then be requested from the provenance service, for example to be displayed in the client. An example of how the provenance service would fit into the TrafficWatch setup, described in subsection 2.1 *A common use-case*, is shown in Figure 4.

As with the message broker, this implementation falls somewhere in the middle of disclosed and observed provenance, since we still only listen to the input and output of a service. If we wanted full disclosed provenance, it could have been implemented by adding provenance in each service, as discussed in subsection 5.1 *Message metadata*. If we in some case want more disclosed provenance, we can still achieve that by changing the service to send a message with the information we are interested in to the provenance service.

By having provenance in a separate service we also get modularity, provenance can easily be turned off when wanted. A drawback of this solution is that we cannot as easy guarantee that the provenance service is running, so we might lose out on provenance data.

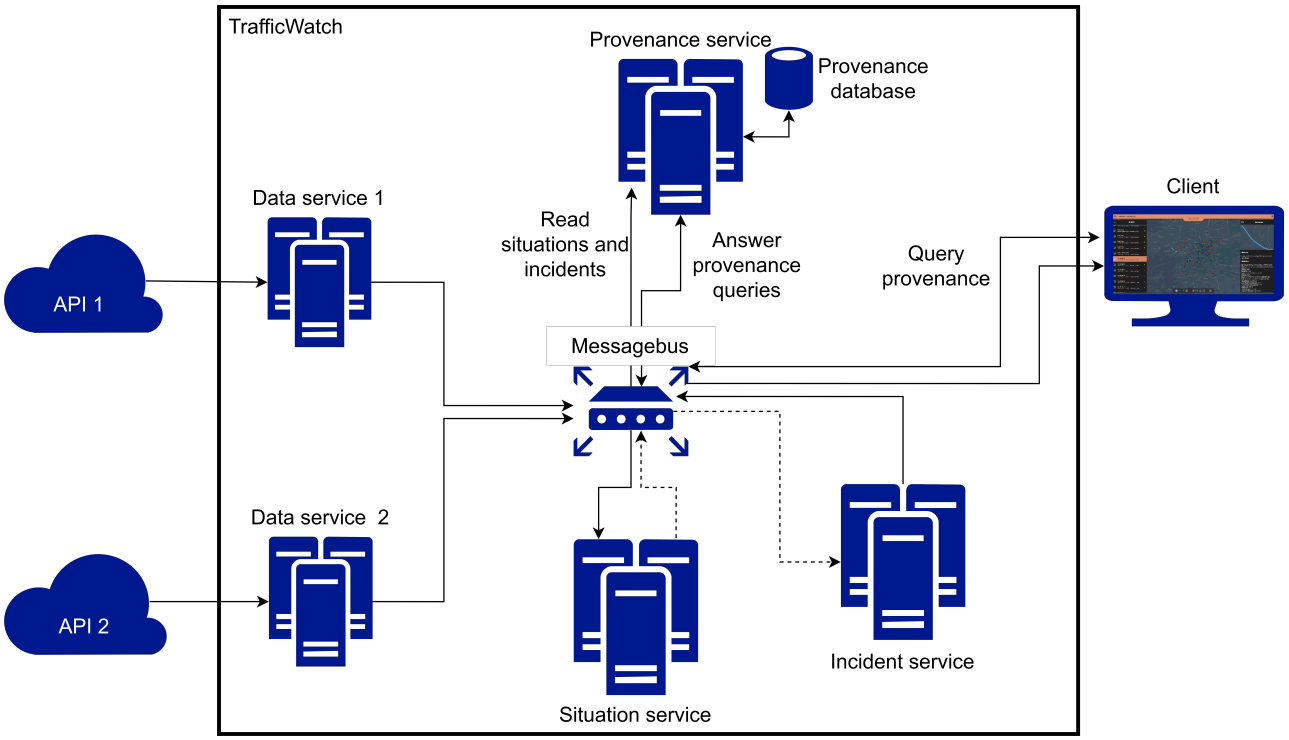


**Figure 4: An overview of one common use-case of TrafficWatch with the provenance service connected.** The provenance service connects to the message bus listening to all published messages, these are then stored in a database together with records of any changes in the events. This data can then be requested from the client. Other than that all services still perform the same actions as described in Figure 1.

### 5.3.1 Provenance Database

As mentioned, the service listens to all situations and incidents published to the message bus. When a new event is received the service first checks if the event have been received before, by comparing IDs. If this is the case, we check what has been updated in the event and store the changes. Otherwise the new event is simply stored in the database. This will be described in more detail in subsubsection 5.3.2 *Detecting and storing updated events*.

There are several advantages to persistent storage of provenance, as this will allow the service to detect changes in updated events even after the service has restarted. Also, as a relational database is designed to efficiently be queried on different parameters, provenance data is easily retrieved on different conditions.

The database tables are designed to mimic the fields of the stored events, in order to recreate events when retrieved and to be able to query on relevant

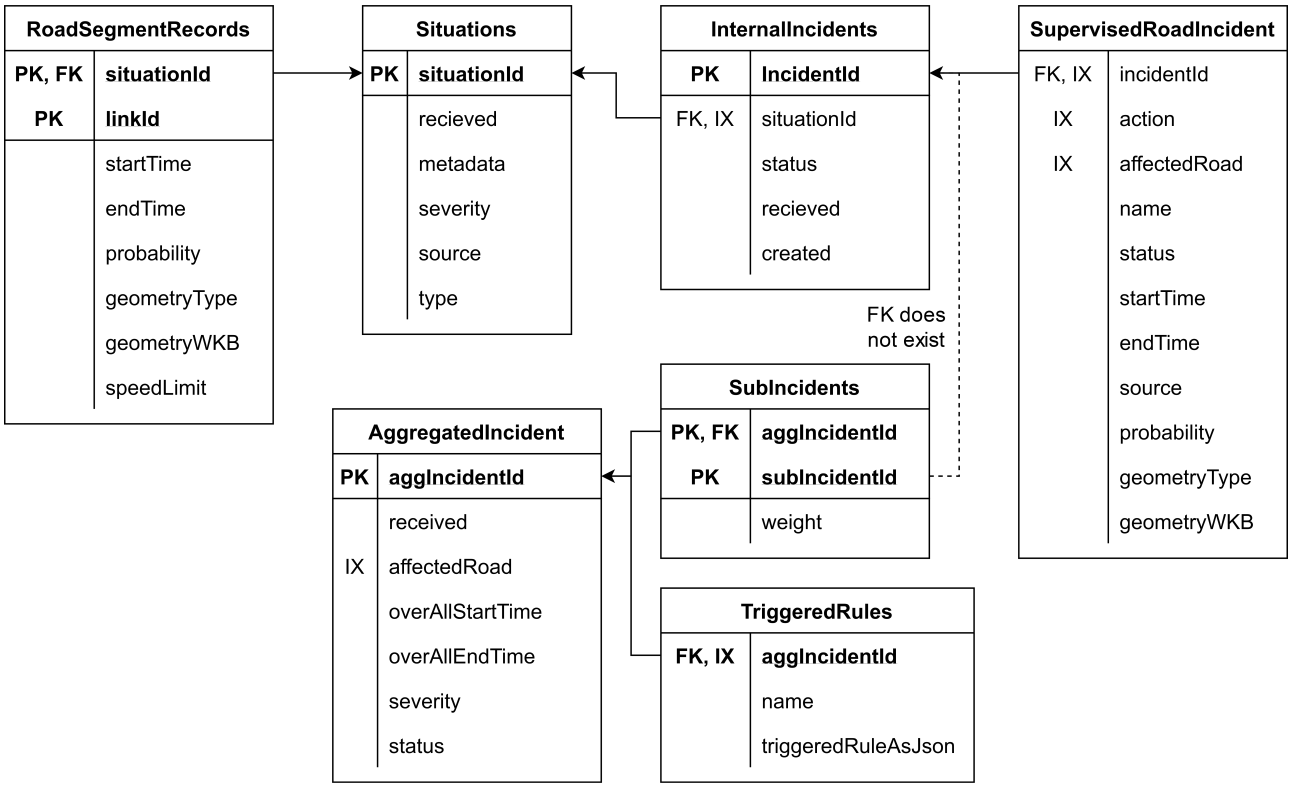


**Figure 5: Database schema of provenance database.** Each box indicate one database table and each row one column in that table. Rows annotated with PK are part of the primary key. Rows annotated with IX are part of the index and rows annotated with FK have a foreign key to another table (following the arrow).

parameters. The database schema is described in Figure 5. Following database design principals [11] no information is stored multiple times, instead relations between tables are implemented. The observant reader might see that there is a missing relation, from Subincidents Id to Incident Id. This is intentional as when an aggregated incident is created both the subincident that triggered the aggregation and the aggregated incident are published on the bus at the same time and as TrafficWatch is a distributed system there is no guarantees of the order that the provenance service will retrieve the events in. Hence if the aggregated incident is received before the subincident there would be a database error as the aggregated incident would reference a incident Id that does not exist. This could be solved by storing the aggregated incidents in memory until all connected subincidents have been retrieved and then store it in the database. But this would complicate the code a lot, especially when retrieving from the database, for this reason we have not implemented this feature.

**Table 3:** Properties of change-entries for different event types.

| Property | Purpose | Situations | Incidents | Aggregated Incidents |
|---|---|---|---|---|
| Time | Timestamp when change was detected. | ✓ | ✓ | ✓ |
| SituationId | Id of changed situation. | ✓ | ✓ | ✓ |
| Property | Name of changed property, for example Severity or RoadLink.StartTime. | ✓ | ✓ | ✓ |
| NewValue | New value of property, null if there is no new value. | ✓ | ✓ | ✓ |
| OldValue | Old value of property, null if there is no old value | ✓ | ✓ | ✓ |
| RoadId | Id of road if changed property is connected to a specific road. Null values indicate there is no related roadId. | ✓ | ✓ | |
| TriggeredRule | Name of triggered rule if change is related to rules, otherwise null. | | | ✓ |
| SubIncidentId | Id of subincident if change is related to subincidents, otherwise an empty guid. | | | ✓ |

Since we store information about past events, the database is expected to be large and hence indexes have been created on all columns that occur in queries, mostly event- and road ids, see Figure 5 for exact details. This will slow down insertion some, since the index also need to be updated, but it will significantly speed up retrieval of the data [11], [12]. Fast retrieval of data is paramount as this information will be used to make time critical decisions.

### 5.3.2 Detecting and storing updated events

As described in section 2 *TrafficWatch*, when an event in a data source is updated, TrafficWatch will publish an updated situation. If that situation triggered an incident, a new updated incident is also published. The result of this is that when a new event is received, the provenance service will need to know if that event has been received before or not so that it can check the event for updated parameters or store it in the database.

The simplest solution to this problem would be to, each time a new event is received, query the database to check if the event exists in the corresponding database. If it exists, compare the two events too each other, otherwise store it. The problem with this solution is that each query to the database takes time, as this is an external connection that need to be established. With this solution

the database would need to be queried at least two times, even more to recreate lists such as sub-incidents in aggregated incidents.

To minimize the amount of database queries and therefore speed up handling of a received event, a cache of recently received and accessed events is used. This allows to keep likely candidates for changed events in the provenance service memory where a lookup is almost instant. To not overflow the memory of the provenance service, the cache is limited to a maximum amount of events. This limit is set when configuring the system, by default it is 1000 events per category. For simplicity, the cache replacement algorithm used is: *first in, first out* (FIFO). That is when the cache is full, the oldest event is removed to make space for the new event. This is probably not the best cache replacement algorithm but to know what to change it to, would need further analysis of event update patterns, something that depends on the application of TrafficWatch. Hence FIFO replacement was chosen as it is simple and quick to implement.

With the help of the cache there is now an efficient way to detect if an event is new or updated. Given that an event is updated the next step is to detect what has changed. This is done by comparing the properties to each other and for each difference a change-entry is created. For each event type there is a different change entry with some type specific properties, which are described in Table 3. The type specific properties are needed to be able to identify what property have changed, for example an incident can have several related roads with different start- and endtimes for those roads hence if one of those times change we need to be able to identify at what road the time changed. Once all changed are identified, the generated change-entries are stored in a database with the same columns as properties of that change-entry.

### 5.3.3 Analysis of provenance

As the provenance service store information about all events in TrafficWatch and how they change overtime, it might be interesting to perform analysis on this data. An example would be how the road friction have changed overtime or how often a certain parameter is updated. This would give a developer or operator of TrafficWatch better insights in the system.

For this reason we implemented a proof of concept of how these analysis might work. In our implementation the user can request the friction history of a selected road. This request is then consumed by the Provenance Service who queries the database for all relevant situations on that road. From these situations the friction values over time are processed and then sent back to the client to be displayed. In Figure 6 we have an example of how this data could be visualized very simply with just text, almost like a simple table. A table is not such a good visualization though, a better solution would be to have some sort of graph, plotting the friction values over time.

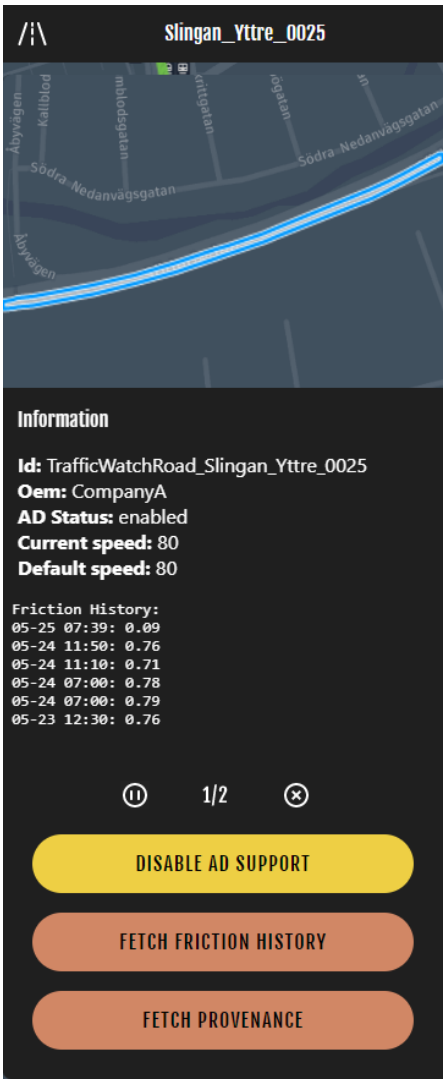


**Figure 6: Example visualization of friction history.** Under the "Friction History" header, each update of the friction is displayed on its own row. A row consists of a short version of the date and time, paired with the friction value.

# 6 Performance tests

To test how our solution impacts the system performance we replaced the two data sources from Figure 1 and 4 with another data service designed to test the performance of the system. This service generates a burst of situations on supervised roads and then measures both the time until all related incidents have been received and until the provenance service have stored all situations, incidents and aggregated incidents.

As we have implemented two separate features, incident aggregation and the provenance service, we will run the tests on four different configurations of TrafficWatch. As a baseline we will test the system without any of our modifications, then with each feature separately and lastly with both features running at the same time. As a normal use-case of TrafficWatch expects loads of at most a few situations/incidents per seconds with occasional bursts of a few hundred situations. We will therefore test each setup by sending a burst of 10, 100, 500, 1000 or 10000 situations at the start of the test to investigate how TrafficWatch handles normal and higher than expected loads.

The results from the tests are presented in Figure 7, there are several interesting observations to make in the figure. In Figure 7 a) we see that while adding the provenance service to TrafficWatch will increase the time to handle one situation by some, the delay is constant and not growing with the amount of situations handled. Hence adding the provenance service to TrafficWatch will not slow down the overall system response times. From Figure 7 b) we see that the provenance service is slower to handle situations than TrafficWatch in general which can handle around 100 situations per second. This slowness is due to the need to communicate with an external database, something that is comparatively slow. At current speeds the provenance service can handle around 25 situations per second, much higher then the expected loads of a few situations per second. But if this becomes a problem there are several optimizations that could be done in the provenance service, for example by using bulk insertion techniques [13].

From the figure we also see that the provenance service slows down significantly when handling 10000 situations. This is probably due to the fact that in the TrafficWatch setup there the performance tests were run there were around 200 supervised roads, hence this created abnormally large aggregated incidents, which take longer time to handle since all sub-incidents need to be updated each time, resulting in many reads and writes to the database. In a normal use-case it would not be the expected to have that large aggregated incidents.

Another rather unexpected result is that when incident aggregations is activated the time to handle one situation is decreased, this is probably due to the changes in the general logic of the incident service implemented to support aggregation of incidents. These change could probably also be applied to the case without aggregation but more work is needed to identify what changes actually resulted in the speedup.

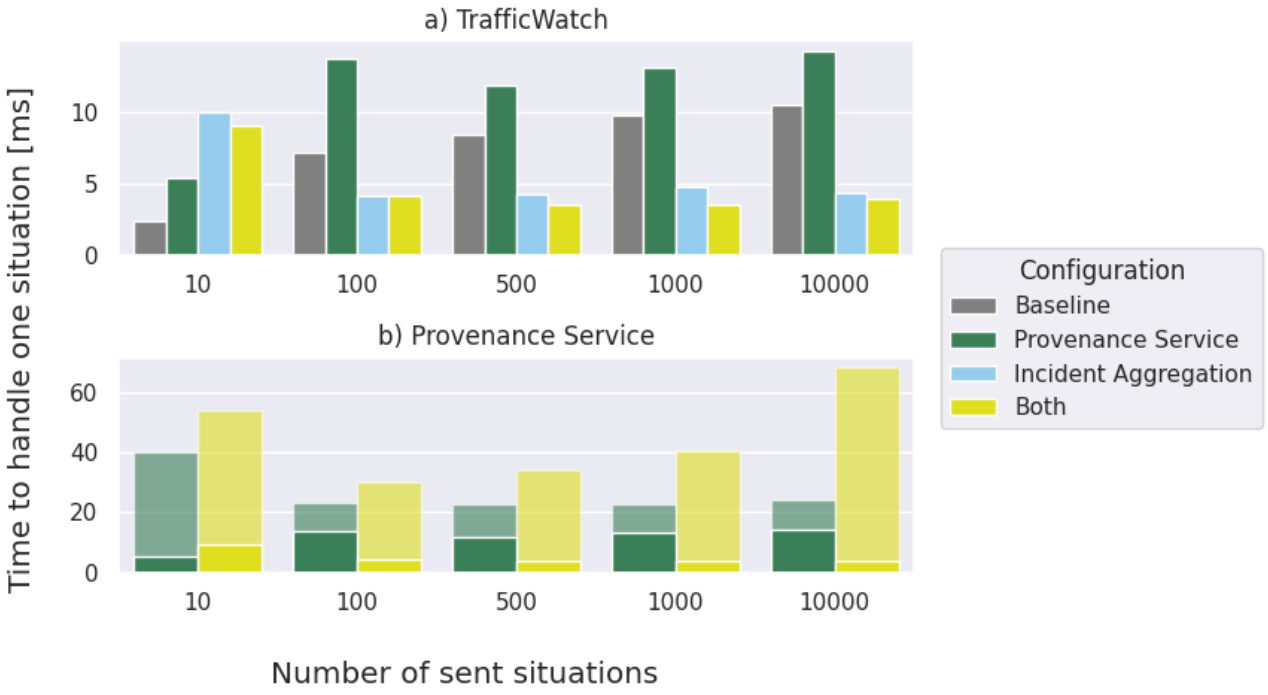


**Figure 7: Results from running performance tests on TrafficWatch with different configurations.** Both plots show the time in milliseconds for the system to process one situation at different loads. a) shows the time for the test-client to receive incidents for all created situation divided by the number of situations sent. b) shows the time for the provenance service to process all created situations, incidents and aggregated incidents divided by the number of situations sent, the darker part of the bar indicate where the last incident was received. See section 6 *Performance tests* for a description of test setups.

# 7 Discussion

We have until now talked about how we implemented decision provenance and aggregated incidents in TrafficWatch, as well as shown how this affects the performance. What this can be used for more generally, and shortcomings of the solution is discussed here.

## 7.1 Shortcomings and reasoning

As mentioned in subsection 4.2, an aggregated incident can get a higher severity than any of its sub-incidents without any rules in the policy specification triggering, if there are enough incidents. However, this might not always be desired, it might be the case that there are lots of data sources warning about the same thing. Does this justify the severity being raised? Just because something is over-reported does not mean that it is more severe, we can only be more certain about it. Thus it might be fitting to also have a probability scale for an aggregated incident that can change depending on its sub-incidents and what rules trigger. For now we have to stick to changing the severity, and to mitigate the original problem, one could hot patch it by supplying a rule in the policy specification that decreases the severity when there are incidents of the same type.

We made the decision to aggregate all incidents that happen on the same road and overlap in time. This was partly due to simplicity but also that most incidents can interact with each other in some way. Although, if the warnings are about vastly different things, they might not affect each other in any way. If one incident says that there is a 3 month-long road work, and we get another incident saying that there is a low risk of snow for 40 minutes, there may be no point in combining these into an aggregated incident. It might be clearer to have them as two separate things because they differ significantly in time spans and impact.

Designing a weight calculation for traffic incidents is tricky. There is no clear way to normalize based on the number of incidents, as this can result in a lower total severity. An example is that with normalization, adding a *Low* severity incident to a *high* severity incident would result in a *Medium* severity. Adding more incidents should only be able to raise or keep the same severity, otherwise we lose the meaning of the severities.

## 7.2 Use cases

Aggregating incidents have uses for both autonomous vehicles and drivers of connected vehicles, as well as for control tower operators. Since a driver does not have much time to read through warnings as they need to focus on the road, they get more relevant information from an aggregated incident. Aggregating incidents lowers the amount of separate warnings they get which reduces the chances of overwhelming them with too many notifications. When it comes to

autonomous vehicles, having aggregated incidents let the cloud computers do the heavy-lifting of calculating how the incidents interact with each other, and supplying the result to lots of vehicles. It also centralizes the decision-making so that different vehicle manufacturers do not have to implement their own decision-making algorithms. Doing one calculation in the cloud instead of each vehicle doing their own calculation reduces the overall amount of computing power used. For an operator of a control tower, aggregated incidents give a clearer view of the current traffic situation and let them take decisions quicker. Aggregating incidents will also allow for the possibility to connect more data sources with less regard to overwhelming an operator or CAV driver.

The provenance data can be used for both developers and operators to get a better insight into the system. It can be used to analyze the data quality of data sources and frequency of data. As knowing how often something occurs, can give an indication of how important it is to implement a nice user experience for such an occurrence. Something that rarely happens is not something worth laying a lot of time on, while something that is frequent is more important to implement nicely.

With lots of provenance data it can be interesting to do different analysis on it. Some interesting analytics one could gather and show is:

- For multiple connecting road segments, if there are events that are expected to overlap but do not.
- How many incidents are reported for each category?
- Does any data sources over-report events, and need to be tuned down?
- What is the probability that an incident will occur on a specific road segment tomorrow?
- What is the update frequency for different data sources, and what do they update?
- How many changes have an event been through?
- What correlation does slippery roads have with accidents? And other correlations between different incident types.

Decision provenance can be used for accountability as Singh *et al.* [1] proposed. When computer systems get more and more complex, we humans need tools to be able to have a better understanding of what the system does. One field that could especially make use of decision provenance is machine learning and AI, since those systems can become very complex and their decision-making is not directly designed by humans. It is easier in man-made code, because then we can reason about what the code does and there is hopefully some documentation.

In subsection 5.1 *Message metadata* we reasoned why adding provenance metadata to each message is not that interesting in TrafficWatch. However, we see

potential for this approach in other complex systems where there are more decisions and changes being made. If provenance metadata simply becomes static data, it is not that interesting, and the same thing could be achieved with simpler and more fitting methods, such as static analysis.

If more messages were sent directly between services and not published to the bus, implementing some kind of provenance in the Message broker could be a more fitting solution, as first discussed in subsection 5.2 *Message broker*. Implementing it in the message broker has the advantage that all services gets affected. Hopefully, without needing to change how the message broker library is used. Moreover, having to implement and maintain provenance in several services is tedious and not so modular.

## 7.3 Future Work

This thesis has only implemented decision provenance within a single system, TrafficWatch. However, it is actually a part of a system-of-systems, as the data sources have their own systems and TrafficWatch sends decisions to other systems who act on it. System-of-systems are a common occurrence and implementing decision provenance to a whole system-of-systems could be of interest as it would lead to better tracking of the whole data flow and increase accountability. Since there are usually different organisations responsible for the different systems, it might be good to have a common format for decision provenance, as Singh *et al.* [1] suggested. We did not use such a a standard since this is a proof of concept and contained within a single system, thus being confined to a standard would be a hindrance.

Apart from things already discussed like the potential need to speed up the database connection. There are several areas where the provenance could be extended or improved. One such area is provenance of running services, since there is no guarantees that all parts of the system is currently online and this could explain unexpected behavior. Another similar improvement is to add provenance of data that almost triggered a situations, as if something is close to a threshold it might still be of interest. Both of these proposed improvements are things that would contribute to explain the stored provenance, for example why a situation did or did not trigger an incident.

Singh *et al.* [1] suggested the future work: "Another consideration are the record-keeping requirements: for how long should provenance data be kept? Data retention considerations are particularly important if serving as evidence supporting investigations or other accountability processes. There is scope for methods that aggregate, summarise and compress such information to assist storage/management overheads." This is still relevant and is especially interesting for real-time systems where data gets less relevant over time.

## 7.4 Conclusion

This report has shown that decision provenance is feasible for systems like TrafficWatch, real-time systems with microservice architecture communicating via a message bus. Decision provenance gives developers and operators better insight into how the system works and increases accountability. We believe it should also be feasible in other complex systems, but the approach for implementing it might differ. We have argued that all three approaches that we have considered could have its uses, not just in every system. However, to prove the feasibility in other setups requires further research.

# References


[1] J. Singh, J. Cobbe, and C. Norval, "Decision provenance: Harnessing data flow for accountable systems," *IEEE Access*, vol. 7, pp. 6562–6574, 2019. DOI: 10.1109/ACCESS.2018.2887201.

[2] X. Zhao, R. Darwish, and A. Pernestål, "Automated vehicle traffic control tower: A solution to support the next level automation," *International Journal of Transport and Vehicle Engineering*, vol. 14, no. 7, pp. 283–293, 2020.

[3] Carmenta. "Carmenta trafficwatch." (2021), [Online]. Available: https://carmenta.com/en/automotive/carmenta-trafficwatch/ (visited on 12/10/2021).

[4] ——, "Carmenta automotive." (2021), [Online]. Available: https://carmenta.com/en/automotive/ (visited on 01/24/2022).

[5] M. Herschel, R. Diestelkämper, and H. B. Lahmar, "A survey on provenance: What for? what form? what from?" *The VLDB Journal*, vol. 26, no. 6, pp. 881–906, 2017.

[6] M. Herschel and M. Hlawatsch, "Provenance: On and behind the screens," in *Proceedings of the 2016 International Conference on Management of Data*, 2016, pp. 2213–2217.

[7] L. Carata, S. Akoush, N. Balakrishnan, *et al.*, "A primer on provenance," *Communications of the ACM*, vol. 57, no. 5, pp. 52–60, 2014.

[8] J. Freire, D. Koop, E. Santos, and C. T. Silva, "Provenance for computational tasks: A survey," *Computing in Science & Engineering*, vol. 10, no. 3, pp. 11–21, 2008.

[9] C. for Language Technology. "Bnfc webpage." (2022), [Online]. Available: https://bnfc.digitalgrammars.com/ (visited on 05/24/2022).

[10] ——, "Bnfc documentation." (2022), [Online]. Available: https://bnfc.readthedocs.io/en/latest/lbnf.html (visited on 05/24/2022).

[11] J. D. U. Hector Garcia-Molina and J. Widom, *Database Systems: Pearson New International Edition: The Complete Book, 2nd Edition*. London: Pearson, 2009.

[12] Microsoft. "Sql server and azure sql index architecture and design guide." (2022), [Online]. Available: https://docs.microsoft.com/en-us/sql/relational-databases/sql-server-index-design-guide?view=sql-server-ver16 (visited on 05/24/2022).

[13] T. Deschryver. "Faster sql bulk inserts with c#." (2022), [Online]. Available: https://timdeschryver.dev/blog/faster-sql-bulk-inserts-with-csharp (visited on 05/24/2022).