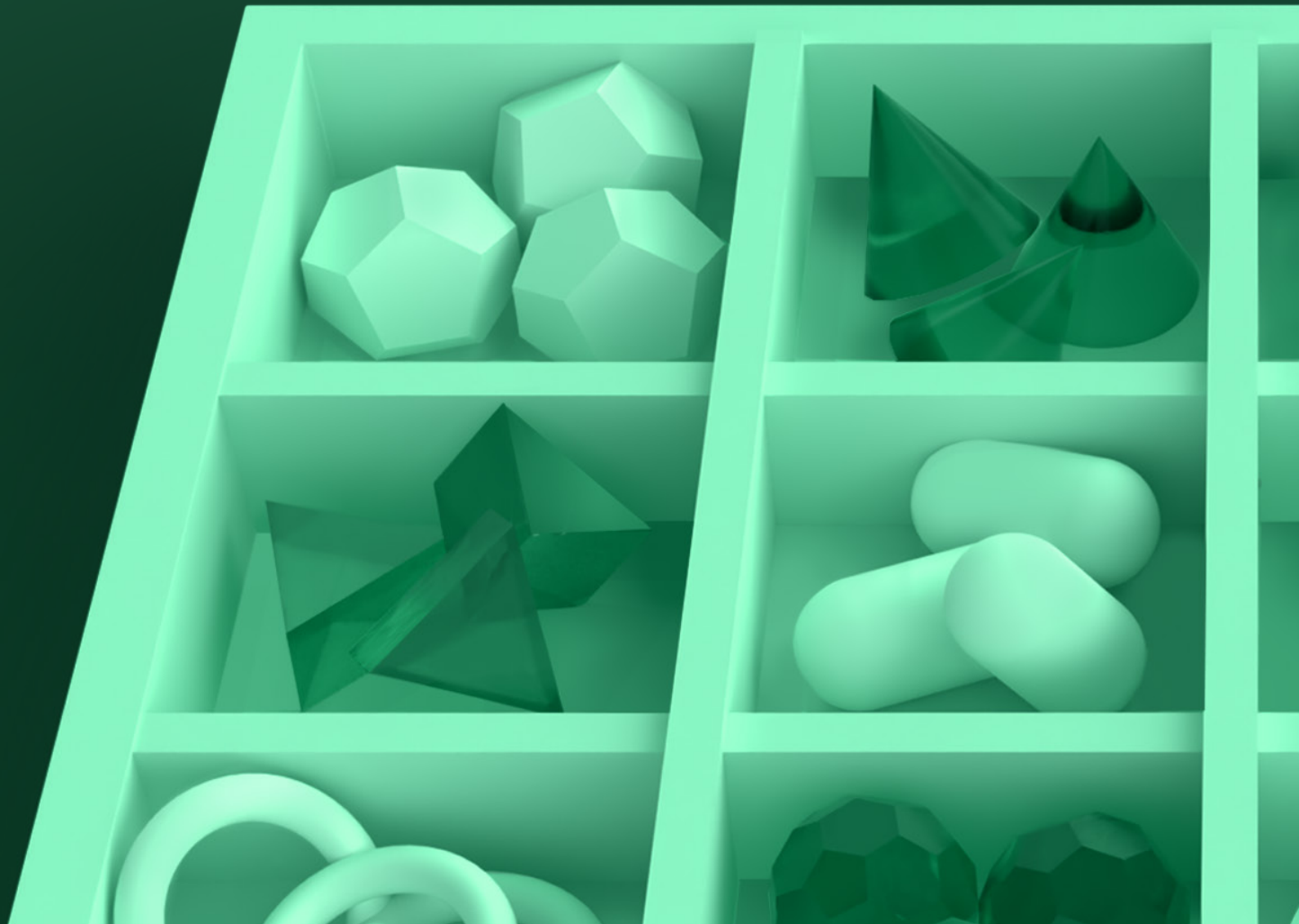


Powering Autonomy With Log Data

Handbook for log data management in
autonomous systems development

1ST EDITION • OCTOBER 2022



Contents

- Executive Summary 3**
- Introduction 5**
 - A. What is log data? 5
 - B. What does effective log data management look like? 5
 - C. Who should read this handbook? 5
 - D. What are the different workflows powered by log data? 5
- I. Log Collection 7**
 - A. Choosing the right log collection methods 7
 - B. Challenges with recording log data 8
 - C. Collecting logs from a test fleet 10
- II. Log Exploration 13**
 - A. Goal: Extracting value from logs efficiently 13
 - B. Processing the firehose of incoming data 13
 - C. Building indices for fast data retrieval 14
 - D. Enriching log data with offline algorithms 15
 - E. Surfacing interesting events for additional review 16
 - F. Enabling easy visualization of all data 18
 - G. Understanding current performance and coverage 19
- III. Log-Based Workflows 21**
 - A. Triaging issues from the field 21
 - B. Curating datasets to train ML models 22
 - C. Using ground-truth labels to analyze perception performance 23
 - D. Improving prediction performance 24
 - E. Validating a supplier module 25
 - F. Improving motion planning performance 26
- IV. Creating Test Cases From a Log 28**
 - A. Reproducing an issue: Test case creation 28
 - B. Resolving the issue 31
 - C. Ensuring the issue does not reappear: Regression and progression tests 31
 - D. Advanced re-simulation: Fuzzing 32
- V. Log Storage and Archival 34**
 - A. Storage types and container formats 34
 - B. Data retention and archival 35
- Conclusion 38**
- Glossary 39**

Executive Summary

A. Challenge

The collection and management of log data is one of the most important tasks that every autonomy program needs to master. A test fleet collects on average four terabytes (TB) of log data per vehicle every day. A production fleet (i.e., vehicles purchased by individual consumers) can generate millions of events per day. This firehose of data from test and production fleets has enormous potential to drive development velocity across an entire organization.

Due to the costs and risks involved in real-world testing, it is crucial that autonomy programs collect and manage their log data effectively. For example, the vehicles themselves, their sensors, and the team of safety operators all incur significant costs to operate and maintain a test fleet. Additionally, the safety implications of real-world testing are very high: One safety-critical mistake can put human lives at risk. Autonomy programs should thus implement practices to scale their data collection effectively, build efficient log data management pipelines, and create scalable workflows to ensure they use all of the collected data to its full potential.

B. Goals

This handbook outlines the benefits of an expansive log data management process, defines the components involved, and provides tactical steps to reduce the cost of implementing and executing such a process. It does not prescribe a one-size-fits-all solution but rather explores common industry practices for autonomy programs to pick and choose from. As every autonomy program has different needs and challenges, no log management process looks the same.

The structure of this handbook follows the journey of a log file from inception to long-term storage (Figure 1). First, an autonomous system collects the log file ([Log Collection](#)). Next, the log file is distributed through data processing pipelines and data exploration frameworks ([Log Exploration](#), [Log-Based Workflows](#), and [Creating Test](#)

[Cases From a Log](#)). Finally, it lands in long-term storage ([Log Storage and Archival](#)).

Applied Intuition has partnered with leading organizations to deploy automated [ingestion](#), [triage](#), and [re-simulation](#) pipelines for their autonomous systems development. Through our extensive industry experience, our team has helped shape log management practices at some of the world's top autonomy programs. We look forward to collaborating on solutions to the topics laid out in this handbook and welcome your feedback for future iterations of it.



Figure 1: Components of a log data workflow.

Introduction

A. What is log data?

In autonomous systems development, log data is any real-world data collected on the system corresponding to the autonomous task at hand. For autonomous vehicles, log data is collected during a drive and ranges from raw sensor inputs to pedal or wheel actuation commands.

B. What does effective log data management look like?

Successful autonomy programs typically strive to make log data accessible to their engineering team as quickly as possible. They are also able to maintain efficient data collection while rapidly scaling up real-world testing.

By contrast, teams who under-invest in their data infrastructure might experience the following pain points:

- Development teams do not regularly use real-world log data, leading to producing modules or algorithms that underperform in real-world conditions.
- When using log data, teams might spend 30 minutes or more to find and visualize a single incident. This is especially problematic as less than 10% of collected log data is useful for development.
- Finding events from previous log data may be difficult and result in the expensive collection of new data to test new versions of the software.
- Known issues remain unresolved and regularly occur during testing.
- It is difficult or impossible to answer key questions on the autonomous system's progress (e.g., which root cause was the largest source of issues in the most recent software release).
- With increased real-world log collection, an autonomy program's data storage costs increase every single day. Teams might find it difficult to decide which log files to delete and which ones to store in the long term.

C. Who should read this handbook?

This handbook puts particular emphasis on supporting the end users of log data (e.g., algorithm developers, systems

engineers, machine learning (ML) engineers, and triage teams).

The concepts, principles, and approaches laid out in this handbook apply to autonomy programs of all sizes and across industries. Most of the metrics and examples used to illustrate different topics in this handbook relate to automotive [SAE Level 2](#) (L2), L3, and L4 systems, but the content of this handbook is equally relevant to autonomous trucking, construction, mining, and agriculture as well as warehouse robots, unmanned aerial systems, and other types of autonomous systems.

D. What are the different workflows powered by log data?

This handbook will cover the following log-based workflows in autonomous systems development:

- **Data science:** Building a platform to mine data, run analytics, and extract metrics from fleet data.
- **Diagnosing issues:** Triageing issues from real-world testing effectively and assigning problems for development teams to solve.
- **Module development:** Scaling the development of perception, prediction, motion planning, and localization modules in a cost-effective manner.
- **Curating labeled datasets:** Detecting events of interest and creating labels for ML training.
- **Simulation:** Creating simulations from log data to power development and triage.
- **Acceptance testing:** Verifying if the release of a new software version is ready for further use.
- **Validating supplier solutions:** Determining if a system provided by a Tier 1 or Tier 2 supplier performs to the required specifications.
- **Regression testing:** Ensuring that previously solved issues do not reappear.

Implementing these workflows requires the careful collection, processing, and storage of log data. This

handbook explores these challenges and recommends practices to implement each workflow effectively.

Log data can also be leveraged for the following use cases, which will not be covered in this handbook:

- **Mapping:** Updating a map based on changes in the real world.
- **Cybersecurity:** Detecting threats and measuring the effectiveness of mitigations.
- **Regulatory compliance:** Reaching and maintaining compliance with privacy, safety, and other regulations, and providing a mechanism to report incidents to regulatory bodies.
- **Insurance compliance:** Providing scheduled reports about operational safety to insurance companies.

I. Log Collection

The scale of log data that autonomy programs collect grows rapidly with the number of vehicles in their fleet, the number of sensors in their sensor suite, and the types of events they aim to track. The infrastructure for collecting, uploading, and processing log data needs to enable autonomy programs to parallelize and scale all of these processes.

Logging and recording autonomous system data presents a unique systems engineering challenge, which requires solutions different from those applied in other domains of software engineering. For example, the best practices for logging distributed systems and web applications assume a persistent network connection, a small volume of log data, and consistent formats. Logging autonomous system data implies a variable network connection, a huge volume of data, and extremely varied formats being recorded.

This section of the handbook discusses how to choose the correct testing methodology, challenges in recording logs, and how to scale log collection operationally.

A. Choosing the right log collection methods

As part of their log collection efforts, autonomy programs need to first choose the collection method (or combination of methods) that best accomplishes their goals. The following tables lay out different log collection methods for test fleets (Figure 2) and production fleets (Figure 3) along with their benefits and challenges. For test fleets, the log recording method should stay the same no matter which log collection methods a team chooses. For production fleets, the log recording method is vehicle and program specific.

Log collection method	Description	Benefits	Challenges
Structured testing	<ul style="list-style-type: none">A test run on a test track (i.e., a private testing ground) associated with a specific requirement	<ul style="list-style-type: none">ControlledRepeatableSafe	<ul style="list-style-type: none">Limited variabilityNot fully representative of the real world
Human driving	<ul style="list-style-type: none">Human driving in a vehicle with the full sensor suite in an unstructured environment (i.e., on a public road) that represents the operational design domain (ODD)Analysis of the collected data takes place either offline or in real time with a “shadow-mode” system deployed on the vehicle	<ul style="list-style-type: none">Safe: A human driver is in control of the vehicle the entire timeAbility to target specific types of drive data (e.g., left turns, red flashing traffic lights, high-speed highway merges)	<ul style="list-style-type: none">New insights are limited to discovering edge casesDoes not test the vehicle’s controls module
Unstructured testing	<ul style="list-style-type: none">A test run on public roads with external traffic in the true ODDInvestigation of operator disengagements and system escalations	<ul style="list-style-type: none">Ability to uncover new unknowns and errorsRepresentative of the real world	<ul style="list-style-type: none">Dangerous if the system has not yet been validated in simulation and structured testingComprehensive training required for safety operatorsLegal approval required

Figure 2: Different log collection methods for a test fleet along with their benefits and challenges.

Log collection method	Description	Benefits	Challenges
Static triggering (e.g., event data recorder (EDR) logs)	<ul style="list-style-type: none"> A vehicle owner drives their vehicle A predefined set of rules governs when to save a log file Rules do not undergo regular updates 	<ul style="list-style-type: none"> Relatively cost-effective to build Predictable volume of data 	<ul style="list-style-type: none"> New insights are limited by vehicle owner behavior Updating rules is cumbersome and may require software changes at a dealership
Dynamic data collection	<ul style="list-style-type: none"> A vehicle owner drives their vehicle A rules-based system discovers interesting events automatically and saves log files to later upload them to a collection server Developers can update rules over the air on a regular basis 	<ul style="list-style-type: none"> Scalable Events are sent as soon as possible (based on network availability) 	<ul style="list-style-type: none"> Determining when an event occurred presents a tough engineering challenge Creating and saving segments of a log requires additional computing power Unpredictable volume of events, especially when adding new rules Large network bandwidth required to transmit recorded data (via Wi-Fi or cellular network)

Figure 3: Different log collection methods for a production fleet along with their benefits and challenges.

As seen in Figure 2, structured testing helps programs test specific requirements of a test fleet in a private, safe environment. Teams can also leverage human driving to collect and test large amounts of sensor data. Unstructured testing provides the largest amount of insights, but it also bears the highest cost due to safety, training, and legal requirements. Autonomy programs should thus leverage unstructured testing only after they have validated a release candidate in simulation environments and with a large number of structured tests. As an autonomy program matures, unstructured testing will make up a growing percentage of its testing efforts.

As seen in Figure 3, static triggering is relatively cost-effective with a predictable volume of data, but new insights are limited and rules are difficult to update. Dynamic data collection is scalable and fast, but its challenges include clock management, computing power, data volume, and network bandwidth.

Implementing programmable log collection from a production fleet requires a number of different technical considerations regarding the hardware and software

deployed on the vehicle as well as data formats, security, transmission methods, and more. A detailed discussion of these topics is outside the scope of this handbook.

B. Challenges with recording log data

As autonomous system data is inherently multi-dimensional and complex, log collection can pose various challenges to autonomy programs. Examples of these challenges include:

- **Recording fidelity:** During the collection phase, it is unclear how the team will use the collected data. This uncertainty makes it difficult to decide at which fidelity logs should be recorded.
- **Clock synchronization:** Teams might log each sensor with its own clock timestamp, which may disagree with other clocks in the system.
- **Container format:** Choosing a correct format to use in real-time logging requires teams to balance tradeoffs between flexibility, robustness, and compatibility with supplier software.

Recording fidelity

Autonomy programs should record logs losslessly

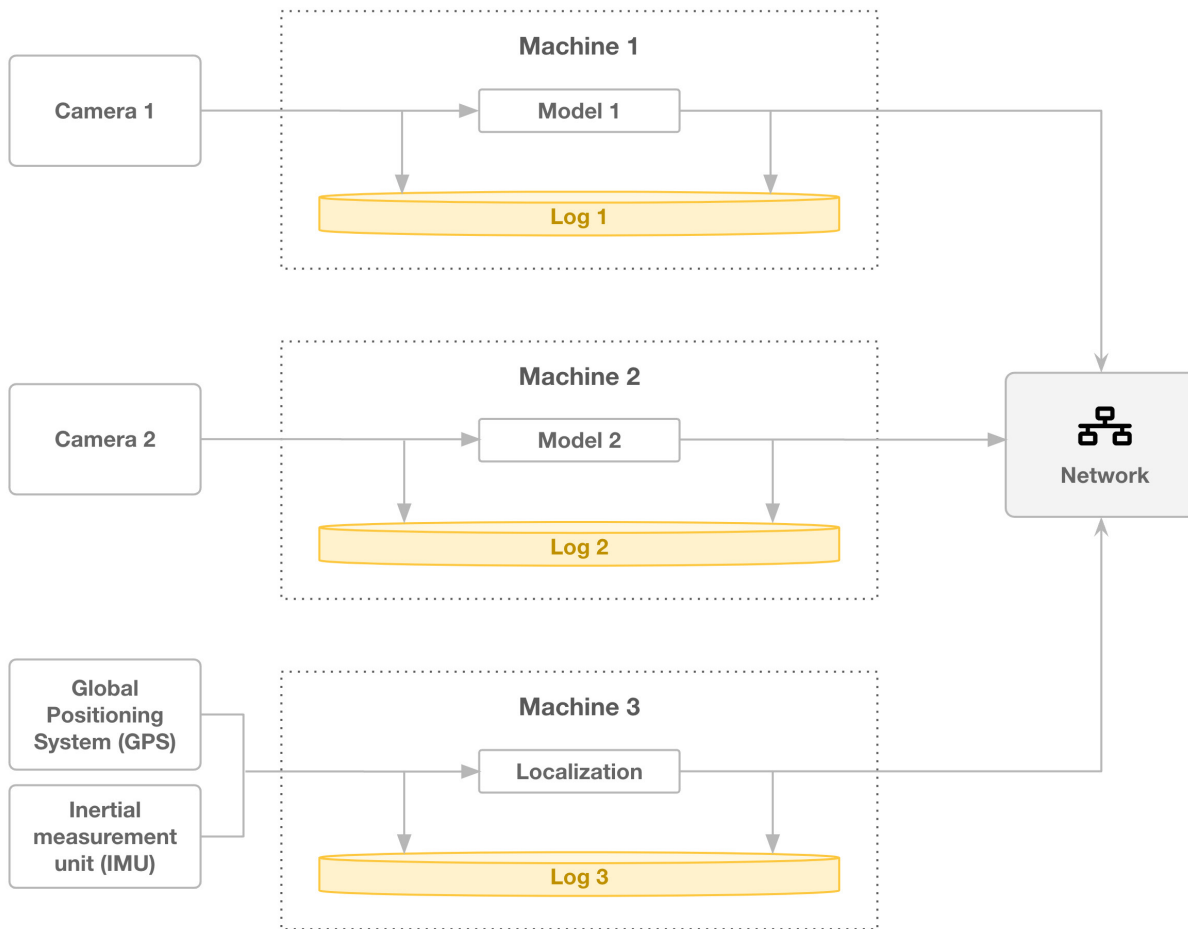


Figure 4: Autonomy programs should record log data into a log file located on the machine that is processing the data.

without any downsampling (e.g., compressing images). This recommendation stems from downstream log data requirements that call for lossless fidelity. For systems that are constrained by disk space, such as many deployed L2 fleets, downsampling is required. However, the downsampling should be limited to sensor data, and the remaining signals should be kept as high-fidelity as possible.

The two types of log data with distinct usage patterns are sensor data and structured data. Sensor data, especially camera images, should be labeled and provided to ML models for training at the full resolution. Teams should execute model training at the full resolution. Otherwise, their model will either be too slow in production usage or fail to perform its tasks. Structured data, which is made up of numerical and string components, represents the state of various modules. Teams should store structured data without downsampling, as missing individual messages

from a signal would make it impossible to accurately reconstruct an issue and reproduce it for testing. Structured data is also very cost-effective to store, especially in a wire format optimized for sending data over a network, such as Protocol Buffers (Protobuf) or MessagePack.

In addition to the above recommendations, downsampling data also incurs central processing unit (CPU) costs, thus reducing the CPU budget for core autonomy tasks. When writing at full fidelity instead, teams can increase their disk utilization. Disk I/O (i.e., the measure of how long read and write operations require on a hard disk) is generally uncontested, as core robotic tasks are handled in memory to reduce latency.

Besides avoiding downsampling, autonomy programs should record log data into a log file located on the machine or electronic control unit (ECU) that is processing the data (Figure 4). For example, the computer that is reading from

a camera and processing an image should be in charge of recording that image along with the outputs generated during processing. This solves two issues: First, it avoids saturating the network with traffic only required for logging. Second, it ensures that logging is more accurate, as data gets recorded into a log file more quickly than if it first needed to traverse across a network.

Clock synchronization

When separating log files across machines or ECUs as described in the previous section of this handbook, it becomes especially complex to reconstruct an accurate view of the order of recorded data. To determine this order, autonomy programs need to reliably determine the first exact timing of each message in each log. Clock synchronization helps solve this challenge.

Clock synchronization is a well-studied topic in software and hardware engineering. Clock misalignment occurs either due to incorrect syncing at the start of a process or due to clock drift from small errors in clock hardware that accumulate over time.

The correct clock synchronization solution differs based on the hardware and software that an autonomy program uses for its vehicle architecture. At a minimum, programs should use a time protocol such as Network Time Protocol (NTP) or, if available, Precision Time Protocol (PTP) at system startup time. This way, each machine starts up, receives a synchronization signal, and then uses that time as the clock signal when recording data.

Choosing the correct logging format

Log data has many downstream users and is recorded in a constrained environment. Autonomy programs thus have many requirements and tradeoffs to consider when choosing the correct logging format.

In order of priority, the following requirements are critical when choosing a logging format:

1. Ability to handle the disk I/O throughput of the sensor data and module communication without falling behind
2. Low CPU footprint

3. Robustness to corrupted inputs and interrupted writes

Generally, autonomy programs should collect logs in a format that is optimized for real-time logging. This format should be append-only, quick to write, and able to store many different data types flexibly. The canonical example of an ideal recording format is rosbag. Many autonomy programs record logs in this format even if they do not use Robot Operating System (ROS) as their middleware. Rosbag is not optimized for compatibility with other toolchains such as query engines or web frameworks. If an autonomy program chooses to record to rosbag, the first step in their data processing pipeline is usually to convert files from rosbag to a different container format.

The following features help ensure that downstream systems are able to effectively use the logged data. These features are recommended but not required:

1. Inclusion of the message schema
2. Support for high frequency with small amounts of data as well as low frequency with larger amounts of data
3. Readability by standard software

C. Collecting logs from a test fleet

The size of a test fleet is a major factor that influences an autonomy program's log collection process. The optimal size of a test fleet usually depends on several different factors: The needs of the engineering organization, the scope of the required validation work, and the algorithm engineering team's organic demand for hours of testing. This section lays out different log collection practices that autonomy programs typically adopt depending on the size of their test fleet.

Test fleet of 1-10 vehicles: Empowering safety operators

In the early development stages, an autonomy program might have a fleet size of 1-10 vehicles. At this stage, vehicles usually return to a garage daily, where the team uploads each vehicle's entire logs to a blob store such as Amazon S3 or Azure blob storage, or to a local machine.

To use all vehicles in their fleet effectively, autonomy

programs should form a team that manages the fleet's daily operations. Beyond regular daily operations, this team also prioritizes, schedules, and executes requests for specific testing or data collection campaigns. Before submitting requests for such campaigns, engineering teams should first use available offline data and prove that existing data is insufficient for their use case.

During testing, two safety operators are typically present in the vehicle. One operator is responsible for ensuring that the vehicle operates safely. In order to assess safety risks during the test drive, they need to understand the software changes that the team is testing. The other operator triggers data collection and notes down additional comments and insights regarding issues that the vehicle is encountering during testing.

To support the work of the safety operators, each vehicle usually features a rudimentary user interface (UI) that shows the vehicle's status. This UI should also monitor the health of the logging system. This helps prevent data loss and logging errors, thus saving operators valuable time that they might otherwise spend collecting more log data or debugging issues. For example, statistics on machine usage such as the CPU, random-access memory (RAM), and disk space as well as the current processing bandwidth can indicate whether the logging system is in a non-functional state. In addition to monitoring the logging system during testing, operators should always check the logging system's health before each trip as part of their pre-test checklist.

The logging system itself should make it easy for safety operators to add comments to an event immediately. Operators should also be able to edit their actions in case they press a button by accident or trigger an event they did not mean to trigger.

As they expand their testing and log collection efforts, autonomy programs should increase the amount of data types available for safety operators to collect and analyze. For example, a basic visualization of some sensors and their health allows operators to manually bookmark events

when issues occur (e.g., if the perception stack does not correctly detect a pedestrian on the road). Additional options for useful visualizations include fused perception outputs, map data, and vehicle controller area network (CAN) signals. Ideally, operators can also hide or customize each sensor visualization based on their test requirements and preference.

Finally, autonomy programs should keep the hardware resources for their logging system, visualization functionality, and stack separate in order to prevent issues in one system from blocking or delaying other systems.

Test fleet of 10-100 vehicles: Automating event detection

As their fleet reaches more than 10 vehicles, autonomy programs may hit scaling limits in processing through all logs and finding relevant events. Programs train their safety operators to manually trigger data collection events when issues occur and archive all other logs that do not contain interesting events. However, operator feedback tends to vary in quality. It can also be error-prone, especially if multiple issues occur at the same time. Additionally, programs may spread testing over a wider geographic area without a persistent high-bandwidth network connection. To solve these issues, programs should develop a method of determining when an interesting event has occurred and automatically uploading partial log artifacts.

Autonomy programs typically implement automatic rules that determine when an interesting event has occurred. For example, programs can implement diagnostic checks on signals which automatically trigger data collection events as soon as the vehicle fails to perform within certain thresholds (e.g., if a camera fails to capture a frame for more than 100 milliseconds or a planner outputs an empty plan). For L4 autonomous systems, operator disengagements (i.e., situations where the safety operator intervenes to disengage autonomous mode) should automatically trigger data collection events. Throughout development, teams should be able to create new automatic rules for new types of events and implement them across vehicles on their fleet.

Test fleet of 100+ vehicles: Scaling to production size

When their fleet reaches over 100 vehicles, autonomy programs should focus on minimizing issue resolution times and scaling their fleet.

The primary goal of an autonomy program's fleet of this size is to identify and resolve issues and minimize the amount of time that passes between issue discovery and resolution. To minimize issue resolution time, programs need to identify issues automatically, automate issue reproduction, and assign bugs accurately to the correct engineers.

Another goal at this stage is to efficiently grow the fleet. Autonomy programs need to conduct careful supply chain management to source vehicles while anticipating long lead times and frequent interruptions. The engineering team should focus on building the systems required to eventually scale the fleet to production size.

A note on log collection from a production fleet

While L2-3 autonomy programs might already deploy their software on production vehicles and record certain types of log data for further research and development, L4 autonomy programs are still preparing to build and deploy production fleets with logging systems operating in production environments. As vehicles have become increasingly network-dependent, the production log collection system needs to be robust to network failures. Additionally, logging systems must be accessible to the engineering team, and protected by a firewall to avoid interference with the rest of the vehicle's system.

The intricacies of production log collection and management are deeply tied to the unique characteristics of each autonomy program and its vehicle architecture. The program's organizational structure, the architecture of its vehicles, and the ODD's regulatory requirements all shape what the program's ideal back-office systems look like. Due to the deeply specific nature of these systems, a description of specific requirements for production fleet log collection is outside the scope of this handbook.

II. Log Exploration

Once an autonomy program has collected log data, it needs to enable different teams to explore and extract value from the recorded log data. In order to achieve this, programs need to process the large amounts of collected data and build indices to enable faster data retrieval. They can then enrich log data with offline algorithms to better evaluate the performance of their ML models, surface interesting events for additional review, visualize data to understand it more easily, and draw conclusions about system performance and ODD coverage.

This section of the handbook discusses how autonomy programs can build a processing pipeline, add indices to enable efficient queries, enrich log data with additional metadata, and create a data platform for downstream workflows.

A. Goal: Extracting value from logs efficiently

The goal of log exploration is to extract high-level insights about the autonomous system's current performance

and coverage across its ODD. Log exploration also allows engineers to quickly find and investigate specific events.

As autonomy programs scale their log collection efforts, they also need to efficiently extract value from more and more collected logs. If they lack the infrastructure to scale their log exploration processes effectively, the time and cost programs spend exploring and understanding logs will grow linearly relative to the amount of collected data.

B. Processing the firehose of incoming data

Data processing

To make log data usable for downstream teams, autonomy programs must process the data effectively. Examples of different data processing jobs include:

- **Transforming** data into a different format to make it more easily accessible and understandable.
- **Encoding** sensor and media data to optimize it for

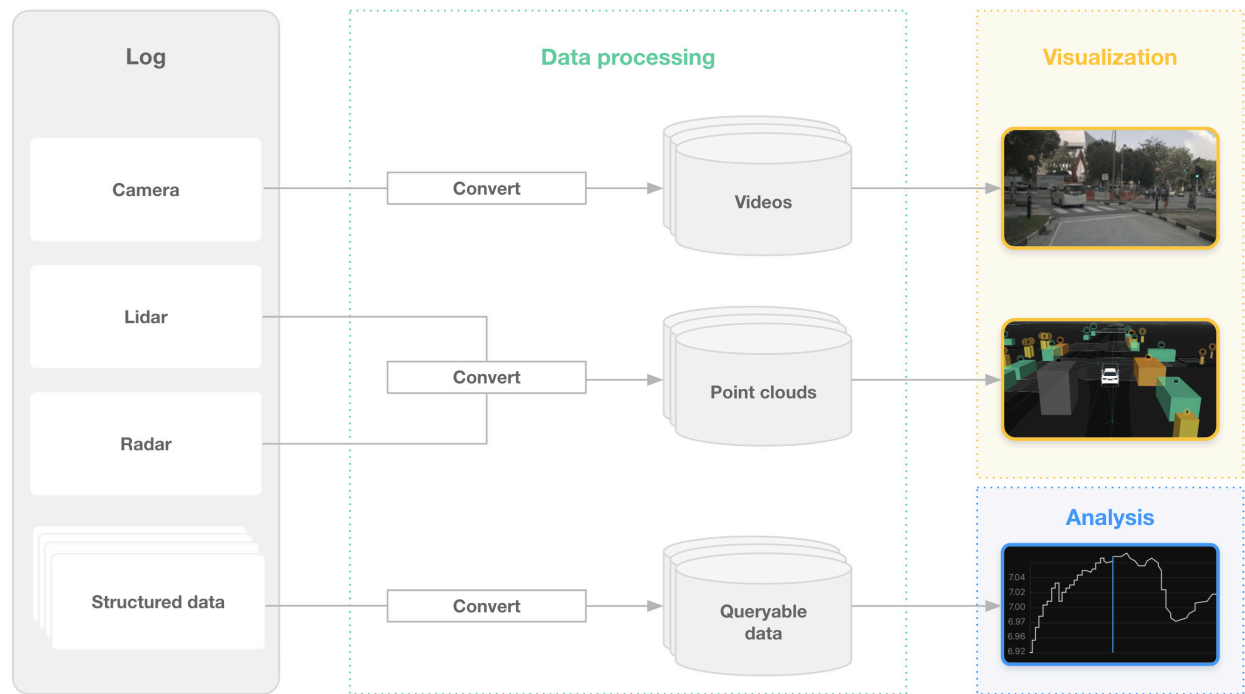


Figure 5: Data processing pipelines operate on log data to convert it into formats usable for analysis or exploration.

playback (Figure 5). Sensor data artifacts include video, compressed or downsampled lidar, and radar. The most important sensor data artifact is usually video, as triage, operations, and engineering teams rely on video streams to quickly understand the situation they are investigating.

- **Cropping** the output files to ensure that each file has a manageable size or to extract certain events of interest.
- **Optimizing** time series metrics for querying.
- **Removing** personal information in compliance with regulations such as the General Data Protection Regulation (GDPR).
- **Indexing** the data to improve query performance and enable filtering for interesting subsets of data (see [Building indices for fast data retrieval](#)).

Beyond executing individual data processing jobs, autonomy programs should implement an entire data processing pipeline to inform time-sensitive decision making such as the evaluation of safety-critical issues. As the data processing pipeline can make or break a program’s overall data availability, it needs to efficiently scale alongside an increasing amount of data collection.

A data processing pipeline’s effectiveness can be quantified with the help of specific target metrics. The following table provides examples of such target metrics (Figure 6), but programs should define additional metrics based on their current priorities.

As seen in Figure 6, the recommended log ingestion success rate is 99.9%. In other words, if 10 vehicles each collect two logs per day, less than 10 logs should fail to

ingest over the course of one year. Failures might occur due to erroneous data formats or infrastructure issues.

In general, the data processing pipeline’s target metrics should incentivize autonomy programs to shorten their iteration loop (i.e., the time that passes between the introduction of a new change and the availability of new logs that the team can access and query). This ensures that autonomy programs make decisions based on the most recent data possible instead of waiting days or weeks before making an informed decision.

C. Building indices for fast data retrieval

During data processing, autonomy programs should index their ingested log data so that team members can quickly find the subsets of logs relevant to them. Programs can use several different indices concurrently. The following four indices usually provide the most value:

Event index

The outcome of event indexing is a list of events that match certain predefined criteria. An event index should include the log of the event, the reason why this event is of interest, specific sensors related to the event (if any), and the time window during which the event occurs in its source log. Additionally, autonomy programs can attach associated taxonomy information to the event. The [Log-Based Workflows](#) section discusses the types of events that teams can index and extract. These types are not limited to safety-critical events only but instead range from information gathering for training to edge case reporting and diagnostics.

Metric	Recommended target
Time to ingestion in proportion to the original log duration	0.5-1x the original log duration
Time to query availability (i.e., how much time passes between recording a log and being able to query it)	24 hours from the time of recording
Log ingestion success rate	99.9%

Figure 6: Metrics to measure a data processing pipeline’s effectiveness.

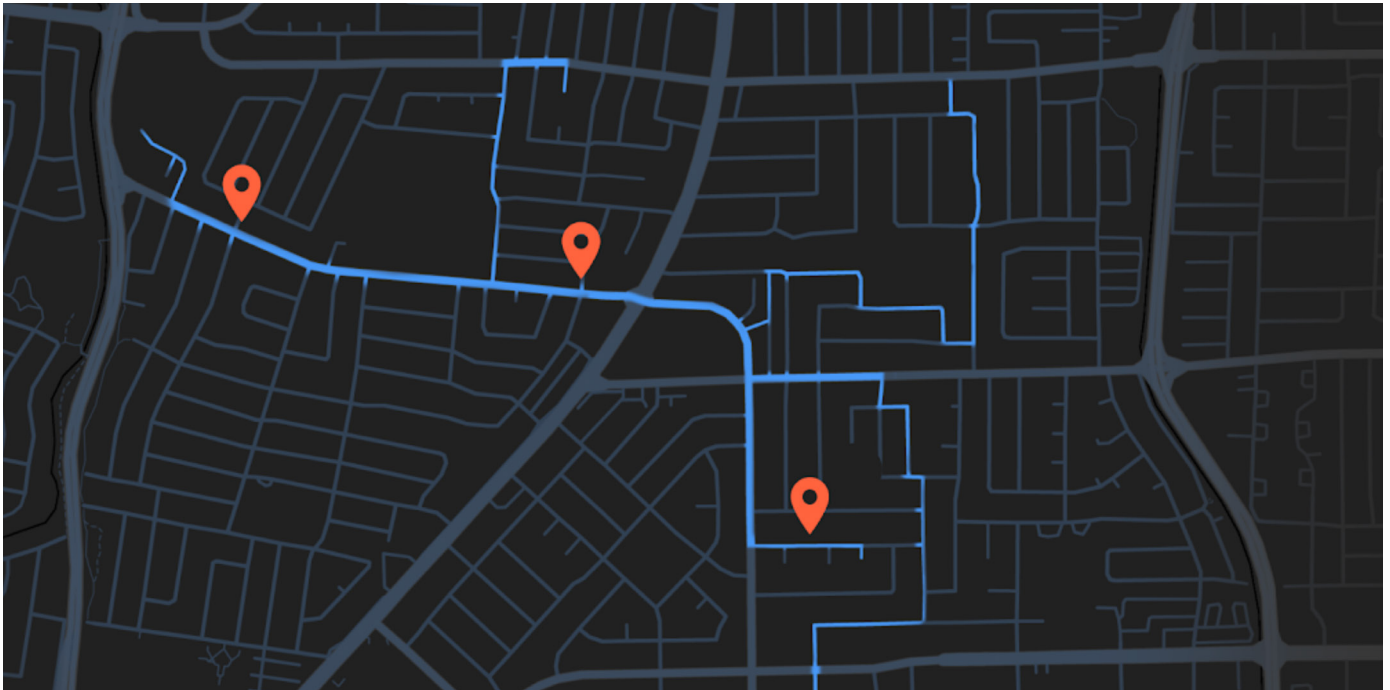


Figure 7: A geospatial index visualized as a map of a driven route, with events highlighted.

Geospatial index

Geospatial indexing (Figure 7) allows autonomy programs to query logs based on geographic filters. More specifically, geospatial indexing makes it easier for teams to build complex location-based queries. For example, a team might want to evaluate an autonomous system's performance on a certain stretch of road or terrain or, in a more complex query, find all left turns that occur on a steep grade. In both cases, programs can speed up the query process by joining on geospatial indices.

Metrics index

Metrics indexing uses a columnar data store to enable performant log queries based on the value of any time series metric. This allows autonomy programs to quickly find log snippets of when the autonomous system was in a specific state. These metrics can be highly specific to each autonomy program's domain, but they should generally be useful to specific teams and easy to compute. Examples of metrics include:

1. Engagement status
2. System's velocity
3. Number of detected pedestrians

4. System's chosen behavior
5. CPU usage

Log index

Log indexing enables autonomy programs to query their collected log data based on metadata associated with each log. This metadata includes the specific test vehicle, the safety operator, the route, the date of recording, and other mission-level parameters. Programs often use log indexing to hone in on an event of interest on a specific drive or test. For example, log indexing enables a query to find all routes driven during a one-month period on a certain vehicle.

D. Enriching log data with offline algorithms

Autonomy programs should enrich their ingested log data by running additional algorithms offline. This might seem like a complex and daunting task, but it is typically worth the effort. Offline algorithms can add metrics from external sources or run ML models on the recorded log data instead. This way, data enrichment allows programs to augment logs with data that would be impossible to collect in real

time. For example, heavy ML models that cannot run on an automotive ECU can be run offline and inserted into the log. Common data enrichment methods include map-based annotations, running computer vision algorithms, and annotating the behavior of perceived actors.

Map data

Autonomy programs can enrich the autonomous system's location in the log data with data from high-definition (HD) maps or open-source satellite data. For each frame, teams should enrich map data with:

- Semantic information regarding the entity on which the autonomous system is currently located (e.g., the type of road, such as a highway, entry ramp, tunnel, or surface street)
- Metrics associated with the autonomous system's position on the current map element (e.g., the lane curvature or the distance from the lane center)

Computer vision models

In addition to ML models that can run online directly on the test vehicle, autonomy programs can execute more powerful computer vision models offline on the collected sensor data. Offline models are less constrained by hardware and timing requirements compared to online models. The output of an offline computer vision model might include:

- The raw detection positions in the sensor data (e.g., 2D and 3D bounding boxes, lane line geometry)
- Metrics associated with object detection (e.g., obstacle class, count per obstacle)
- Discrepancy metrics between offline and online perception systems (e.g., intersection over union—i.e., the measure of how much the predicted boundary overlaps with the ground truth—per frame and per object class) (see [Automatic error detection](#) for more detail)

Using an offline model also allows programs to detect object classes that the on-vehicle model has failed to detect. For example, an offline model can distinguish between “bicyclist walking their bicycle” versus “pedestrian” or “bicyclist.”

Behavior annotation

Behavior annotation involves tagging the autonomous system's behavior and the behavior of other actors in the scene. To annotate behavior accurately, autonomy programs should use a combination of end-to-end learned models alongside heuristics based on map information and localization.

Behavior annotation is particularly important for motion planning and controls teams, who spend much of their time tuning behavior around cut-in maneuvers from other vehicles. L2 through L4 systems all struggle to be robust to cut-in and merging behaviors. The ability to detect cut-ins and benchmark new algorithms against historic scenarios helps accelerate motion planning and controls development.

E. Surfacing interesting events for additional review

Triage operations teams, algorithm engineers, and other teams that use an autonomy program's log data can utilize both the original and the enriched data as inputs to surface relevant events.

Operator disengagements and system escalations

All interventions that occur while the autonomous system is engaged should be surfaced for review. Interventions include operator disengagements (i.e., situations where the safety operator intervenes to disengage autonomous mode in L4 autonomous systems) and system escalations (i.e., situations where the safety operator or the vehicle owner intervenes to take control of an L2 or L3 autonomous system). If available, the surfaced events should include relevant safety operator comments. Safety operator comments can help teams quickly review and root cause events that may have been caused by a system failure. Surfaced events might also contain the autonomous system's intended maneuver, the behavior of nearby actors, and taxonomy information such as map information. Taxonomy information helps teams classify an event as occurring inside or outside of the autonomous system's ODD.

Scenario tags

Scenario tags enable validation teams to find scenarios matching a specific situation. Scenario tags often combine behavior annotation, map or geographic information, and ODD data. For example, teams should surface and tag scenarios such as “lane changes on highway” and “actor cut-ins during rain.” They can then perform aggregate analysis on scenarios with the same tags to understand how the autonomous system tends to perform in specific situations.

Autonomy programs can combine tags to create human descriptions of especially difficult situations. For example, filtering by “vehicle pulling out of parking spot” and “bicyclist adjacent” will surface edge cases that are relevant to every L4 planning team.

Automatic error detection

Autonomous systems development involves an abundance of unlabeled raw data. Unfortunately, high-quality labels are often expensive to obtain, and labeling common scenarios such as standard highway driving has diminishing returns on perception model performance. Perception models should be trained on data that previous versions classified

incorrectly (Figure 8). Autonomy programs should have a system that identifies these cases in logged data automatically.

To surface potential errors automatically, teams should compare the output of the on-vehicle model to the output of their more powerful offline computer vision model and find cases where the two models disagree. These discrepancies likely indicate errors in the on-vehicle perception stack and should be sent to a team member for manual review and labeling.

Anomaly detection

Beyond finding specific predefined events, autonomy programs typically find it useful to surface unpredicted anomalies in their log data for manual review. Anomalies can include known scenarios with unexpected metrics for pre-chosen dimensions (e.g., the most aggressive cut-ins) as well as unseen edge cases surfaced through ML. A common ML-based anomaly detection technique leverages unsupervised learning to detect unexpected data. Teams train an ML model on a subset of the most relevant channels in their log data to predict the next timestamp based on a window of historical data. This model then runs on a log

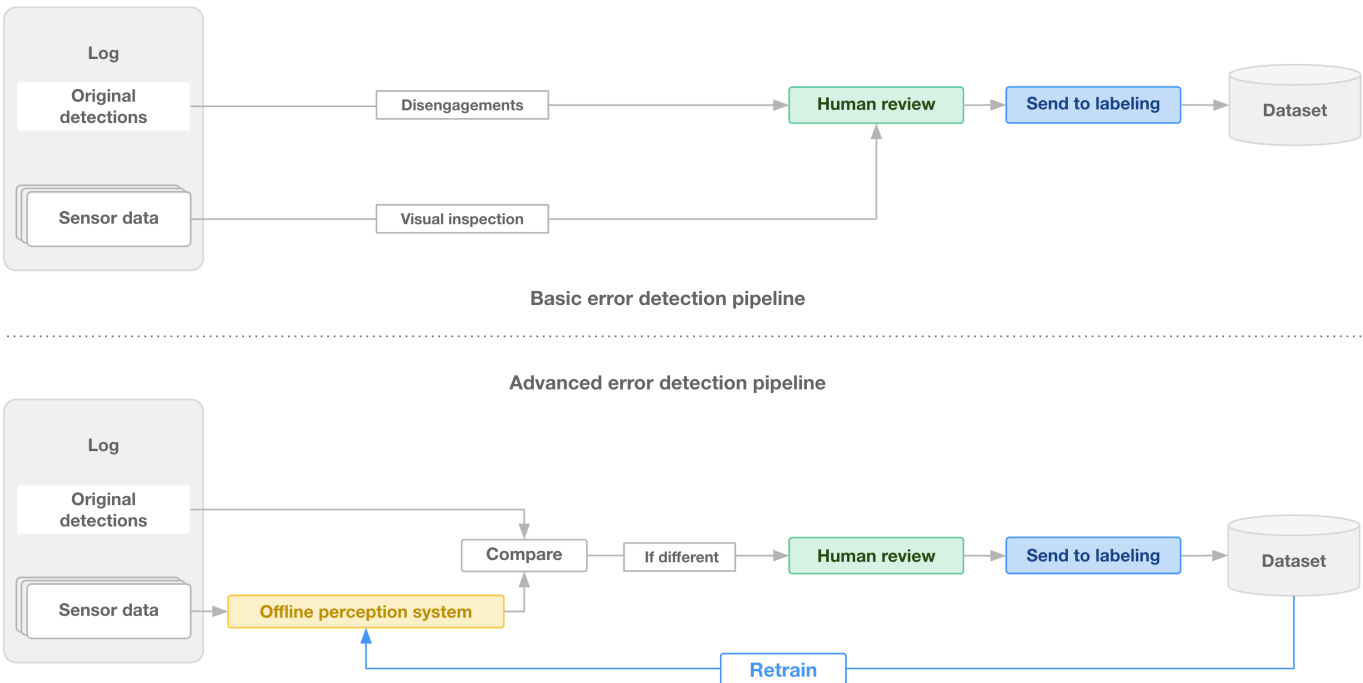


Figure 8: An offline perception system can highlight failures in the on-vehicle perception system. A human review of the disagreements provides final arbitration over which perception system is correct. Advanced autonomy programs retrain and rerun the offline model on the same data.

and surfaces events for which it has the largest prediction gap. These are the cases where the autonomous system behaved in the most unexpected way according to existing data, or a sub-system behaved unpredictably.

Automated anomaly detection is an ongoing area of research. However, it is increasingly popular among autonomy programs that face the limits of automation when combing through petabytes of log data.

F. Enabling easy visualization of all data

To make manual data reviews more efficient, autonomy programs should leverage a web-based visualizer that can play back the autonomous system’s captured state alongside its sensor data (Figure 9). The visualizer should transform and encode sensor data to achieve performant playback capabilities.

The visualizer should be web-based, easy to use, and allow users to share specific log snippets via links. A simple and convenient user experience allows teams to review on-road events more quickly and facilitate discussions about

on-road performance across their entire organization. If a visualizer is difficult to use or only usable by engineering teams, this can severely reduce the value of a program’s collected log data.

For example, viewing videos and visualizations for a specific timestamp should be possible on a low-powered laptop and take 10 seconds or less. This way, all team members across an organization can use and discuss autonomy data. Senior managers and executives might not have access to powerful development desktop machines in their day-to-day work, but supporting their ability to view collected log data is especially important.

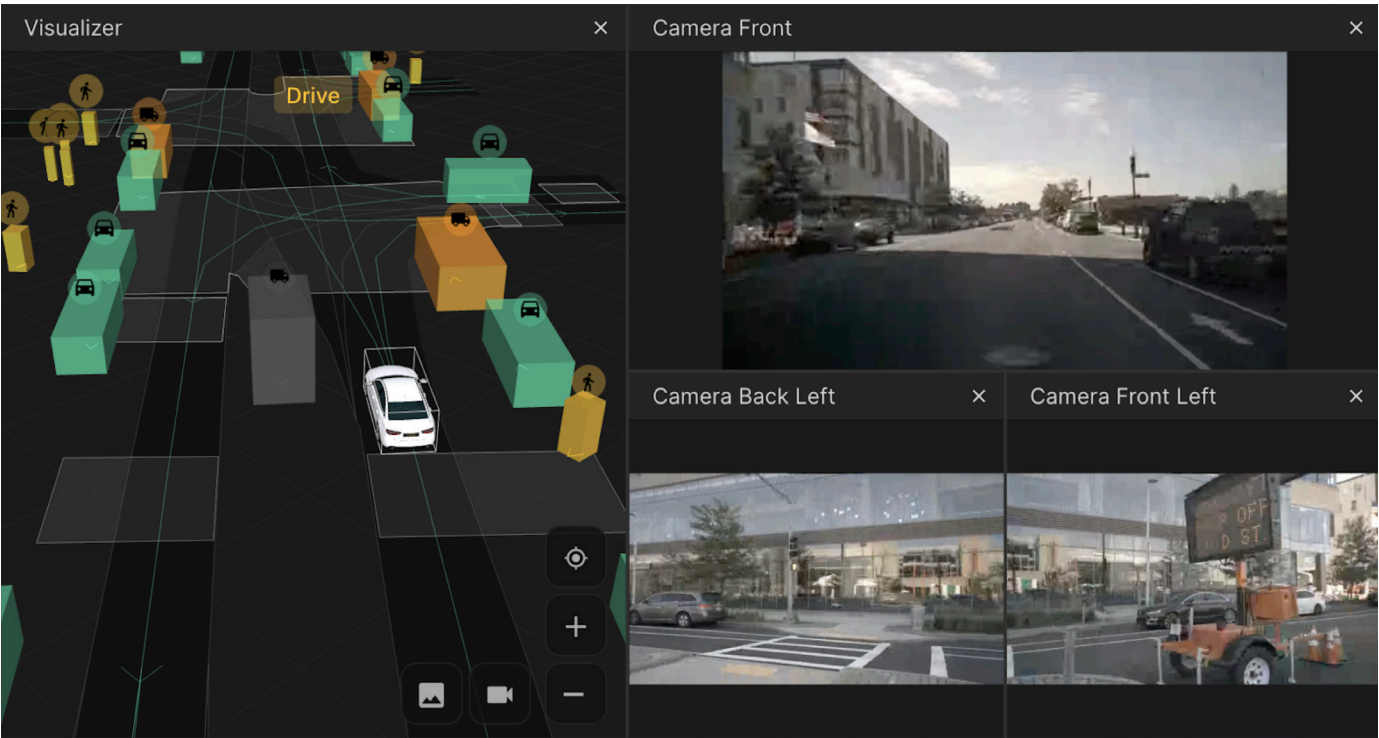


Figure 9: An urban scene with dense traffic, visualized in a web-based tool with easy access to any point of the log.

G. Understanding current performance and coverage

Log data provides autonomy programs with an understanding of their autonomous system's performance and coverage across the target ODD. Coverage is the ratio between what is known and tested versus the total space of possible situations that the system may encounter. Initially, early-stage autonomy programs focus on building broad coverage across known requirements, limited geographies, and common test categories. As the autonomous system matures, this focus shifts toward uncovering edge cases, finding and filling coverage gaps, and refining the ODD definition.

Visualizations of coverage should allow team members to answer the following questions:

- How does stack performance vary across different dimensions of the ODD?
- Which areas of the ODD are underrepresented in the collected data?
- Do newly uncovered issues stem from regressions or new edge cases?

Our [verification and validation \(V&V\) handbook](#) contains further details regarding defining and measuring coverage.

Performance and coverage can both serve as inputs for systems engineers to refine the ODD definition and work toward the autonomous system's deployment. They also permit teams to find specific areas in the ODD where the system performs poorly or where there is a lack of data. Programs can address these issues by using synthetic simulations to fill in coverage gaps and by increasing on-road data collection for relevant sections of the ODD if possible.

Keeping existing log data usable

Whenever autonomy programs make changes to their autonomous system, they need to quickly validate these changes. If indexed correctly, recorded logs serve as a powerful resource to quickly validate stack changes, thus sparing teams the need to conduct slower and more expensive real-world tests. Teams can leverage log re-

simulation to test their new stack version on millions of existing logs. Log re-simulation is a deterministic reproduction of how the autonomous system would have behaved in a specific situation. In contrast to regular log replay, log re-simulation achieves determinism by adding a simulator into the loop (see [Log re-simulation](#) for more detail).

Autonomy programs should thus keep some of their existing log data “fresh” so that it is usable in re-simulation. Other than re-simulation, there are other use cases, such as data visualization and reporting to regulatory bodies, which require log data to stay fresh. Log data usually belongs to one of the following three categories:

- **Unimportant data:** This includes recordings of the inside of an autonomy program's garage, many uninteresting miles of unstructured testing, and false interventions where the safety operator needed a break.
- **Data that is important in the short term:** This includes non-critical interventions and interventions based on discomfort. Once these events go through a triage process and are assigned, fixed, and deployed, they are not necessarily needed for re-simulation in the future.
- **Data that is important in the long term:** This includes all critical interventions such as collision-prevention interventions. This also includes a few non-critical intervention events to track metrics such as comfort and efficiency, and a number of events to track ODD coverage. Autonomy programs should keep log data in this category fresh to leverage it for re-simulation when needed. This data should be kept for regulatory compliance as well.

There are several strategies to keep log data fresh. Some examples include:

- Maintaining a careful versioning system for log data. For any log, it should be obvious which log schema version and stack version were used to record it.
- Writing an associated data migration for every change

to the log schema. This data migration should be able to update existing data to the latest version. Teams should consider including a test in their continuous integration (CI) system that prevents merging schema changes unless they have a migration associated with it. For ROS-based stacks, rosbag migrations are a useful framework.

- Ensuring that every data change is backward-compatible.
- Replaying sensor data through newer vehicle code to generate the most up-to-date outputs.

This section of the handbook laid out the systems required to explore log data, enrich it, and manage the complexity of processing tasks. The next section discusses core workflows involving logs that power an autonomy program's entire development life cycle.

III. Log-Based Workflows

Throughout an autonomy program, a variety of different teams leverage log data for their work. Each team has its unique workflow with specific tooling requirements. Triage teams need to investigate each field issue quickly as it occurs. Perception engineers need to prepare data for ML model development. Motion planning teams need to create test cases based on logged data. On the surface, all of these workflows vary so profoundly that they seem to require their own, separate application built on custom infrastructure. Efficient data platforms offer log-based workflows to solve this challenge.

On the data layer of log-based workflows, “events” are the flexible building blocks that support all of the required workflows. Events are easy to represent—all that is needed is a log identifier, a start timestamp, an end timestamp, and any needed metadata. A “log-based” data platform uses a relatively small common set of processing primitives to construct all of the custom workflows that different teams require. This section of the handbook discusses how triage, perception, prediction, and motion planning teams can leverage log-based workflows to make the most of their collected log data.

A. Triage issues from the field

One of the most important log-based workflows is triage (i.e., the investigation of safety or operational issues from testing). In this workflow, a team triages incoming issues by assigning them to responsible teams or engineers.

Mature autonomy programs should monitor how fast they can go from issue identification to issue resolution (i.e., issue resolution time). To support a large production fleet, issue reporting from the vehicle should be a seamless process—from triggering an event all the way to resolving the issue. It should be quick and easy for safety operators to log comments or tag events during their drives. It should also be easy for software modules to escalate events for investigation at a later point (e.g., a planning module flagging that it is in an unsolvable state).

To support triage teams, autonomy programs should develop an automated triage pipeline that processes an entire log, determines events of interest, and then takes automated action based on the event type (Figure 10).

Once the triage team has identified a problematic event, another set of rules should determine which automated actions will occur. These automated actions augment the event data and inform the triage team’s decision regarding the manual steps they should carry out next. For example, an operator disengagement that matches certain criteria can automatically be scheduled for re-simulation. The triage team can then look at the event and the re-simulation results together to determine the severity of the event.

The table below shows a set of rules that help triage teams filter and populate a queue of events to process (Figure 11). Triage teams should have enough information so that they can inspect and accurately confirm the severity of



Figure 10: A triage pipeline has four key steps: Event classification, triage, issue assignment, and regression testing.

Event	Query	Action	Manual review
Operator disengagement	<ul style="list-style-type: none"> State of the stack changes from “engaged” to “disengaged” 	<ul style="list-style-type: none"> Re-simulate motion planning and controls modules 	<ul style="list-style-type: none"> Inspect failures in re-simulation: Collisions, ride-safety violations (i.e., metrics governing the subjective safety experience, including hard braking, tailgating, and coming too close to a pedestrian), software faults
Unprotected left turn	<ul style="list-style-type: none"> Autonomous system is in intersection Intersection has green light Intersection has no green arrow Velocity > 0 m/s Vehicle leaves intersection 	<ul style="list-style-type: none"> Send to dataset Re-simulate motion planning and controls modules with fuzzed actor positions 	<ul style="list-style-type: none"> Inspect failures in re-simulation: Collisions, ride-safety violations, software faults
Cyclist cuts into the autonomous system’s lane	<ul style="list-style-type: none"> Cyclist is detected Cyclist is in bike lane Cyclist’s path moves into autonomous system’s path 	<ul style="list-style-type: none"> Send to dataset for cyclist detection Send to dataset for motion planning around cyclists 	<ul style="list-style-type: none"> No manual review needed

Figure 11: A set of rules that help triage teams filter and populate a queue of events to process.

the issue. They can then assign the issue to the correct engineering team to root cause and fix.

Throughout their work, successful triage teams are able to use their engineering time effectively. Due to the nature of real-world testing, the vast majority of collected log data is uneventful, and triage teams must filter out these uneventful portions. With a log-based workflow, triage teams can click directly into a problematic event, visualize the relevant signals, and scrub around timestamps in the event.

Efficient triage teams are able to triage an entire day’s worth of testing within a few hours. This speed allows the team to focus its next day of testing on certain parts of the ODD, or provide emergency guidance to safety operators to avoid certain situations. For example, a triage team may uncover a severe fault in driving near bicyclists. The triage team can provide quick guidance to halt the fleet or avoid testing when cyclists are nearby.

B. Curating datasets to train ML models

Perception teams require access to large sets of labeled data to train their ML algorithms, raw sensor data to develop their modules, and additional edge cases or examples of their algorithms misbehaving to continually improve these algorithms. This section discusses how autonomy programs can curate datasets to use for ML model training, with particular focus on the perception use case.

Labeling all logs is prohibitively expensive. To control costs, autonomy programs should develop a workflow that:

1. Finds logs with a specific characteristic.
2. Crops those logs to only the relevant portions.
3. Creates ground-truth labels via an in-house or external labeling team.

Labeling workflows should support a variety of labeling providers and transform incoming labels into a consistent format that the entire organization can use. This flexibility

avoids supplier lock-in and provides a layer of protection against sudden changes in labeling quality from one supplier. Ideally, labeling is targeted toward cases where the perception system misbehaves.

As ground-truth data is required to evaluate perception outputs from a test drive, it can seem challenging to identify perception issues automatically. Offline computer vision models can solve this challenge. As discussed in the [automatic error detection](#) section, a more accurate offline computer vision model can provide pseudo ground truth to compare against the output of the online computer vision model. This is possible as the compute available in a cloud or data center is much larger and more modern than the compute available on a vehicle. For later-stage autonomy programs, this automated method is the most cost-effective in scaling perception improvements. For early-stage autonomy programs, it may be sufficient for humans to provide ground-truth data by viewing videos and other sensor data and then determining if a perception system has failed.

Once perception teams have identified a failure, they should create ground-truth labels on raw sensor data and build a set of tests containing these ground-truth labels. Teams can then run these tests to evaluate their perception stack's overall performance compared to the ground-truth labels. Ground-truth labels then feed into the improved algorithm and enable regression tests.

C. Using ground-truth labels to analyze perception performance

Ground-truth labels are required to train ML models, but they are also important in grading the performance of a perception task. To grade the performance of a perception output, each individual detection is associated with the closest ground-truth label. From these associations, autonomy programs can compute metrics about the accuracy, precision, and aggregate tracking performance. These methods apply to both perception systems developed in-house and third-party perception or computer vision systems.

Throughout testing, teams should choose specific metrics to evaluate the performance of their perception stack in a quantifiable way over time. These metrics usually depend on the specifics of the autonomy program and its goals. The table below contains some examples of such metrics (Figure 12). The metrics “true positive” (TP), “false negative” (FN), and “false positive” (FP) refer to the result of comparing an associated detection with a ground-truth detection.

Perception teams should be able to filter these metrics by zones of interest relative to the autonomous system. Region-based metrics give a sense of how perception performance changes from the left to the front of the vehicle, or from a close range to a medium range in front of the vehicle. This enables perception teams to understand underperforming areas that need the most improvement.

Metric name	Metric meaning
Ground truth (GT)	Total count of ground-truth labels available
ID switches (IS)	Number of times a tracked object switches assigned IDs during a time frame. This captures the failure to recall and stably track an object over a period of time.
Multiple object tracking accuracy (MOTA)	<p>A widely used metric to evaluate a tracker's performance (see this paper for more detail)</p> $\text{MOTA} = 1 - \frac{\text{sum of FP, FN, IS over all frames}}{\text{sum of ground truth track count in a frame, over all frames}}$
Multiple object tracking precision (MOTP)	<p>The average dissimilarity between all true positives and their corresponding ground-truth targets (see this paper for more detail)</p> $\text{MOTP} = \frac{\text{sum, over all tracks in a frame \& over all frames, of the dissimilarity b/t each estimated track and its GT track}}{\text{sum of ground truth track count in a frame, over all frames}}$
Precision	<p>How likely it is that any particular ground-truth object is detected correctly. High precision means that the perception stack is correctly detecting the objects.</p> $\text{Precision} = \frac{TP}{TP + FP}$
Recall	<p>How likely it is that any particular ground-truth object is perceived. High recall means that the perception stack is detecting most of the existing ground-truth objects.</p> $\text{Recall} = \frac{TP}{TP + FN}$
Mean average precision (mAP)	An aggregate metric measuring the overall performance of the detections. Combines precision and recall together across all classes to create a single numerical grade of the performance of the tracking task.

Figure 12: Examples of metrics that help evaluate a perception stack's performance.

D. Improving prediction performance

The prediction of vehicle, bicyclist, or pedestrian motions is a crucial capability required for safe driving. Prediction modules output an estimated trajectory of an actor, typically powered by an ML model. The development of prediction systems requires different processes and infrastructure compared to the development of perception systems.

Prediction teams require large amounts of labeled trajectory data for a variety of actors to train prediction models effectively. Contrary to the perception workflows

described above, it is possible to auto-label prediction data without human involvement. The auto-labeling process uses a robust perception system to observe future positions of an actor and then assign a specific label to the actor's present position.

A log-based workflow can facilitate the identification of relevant training data and the development of prediction systems as a whole. The building blocks of this workflow are similar to the triage and perception workflows: A system

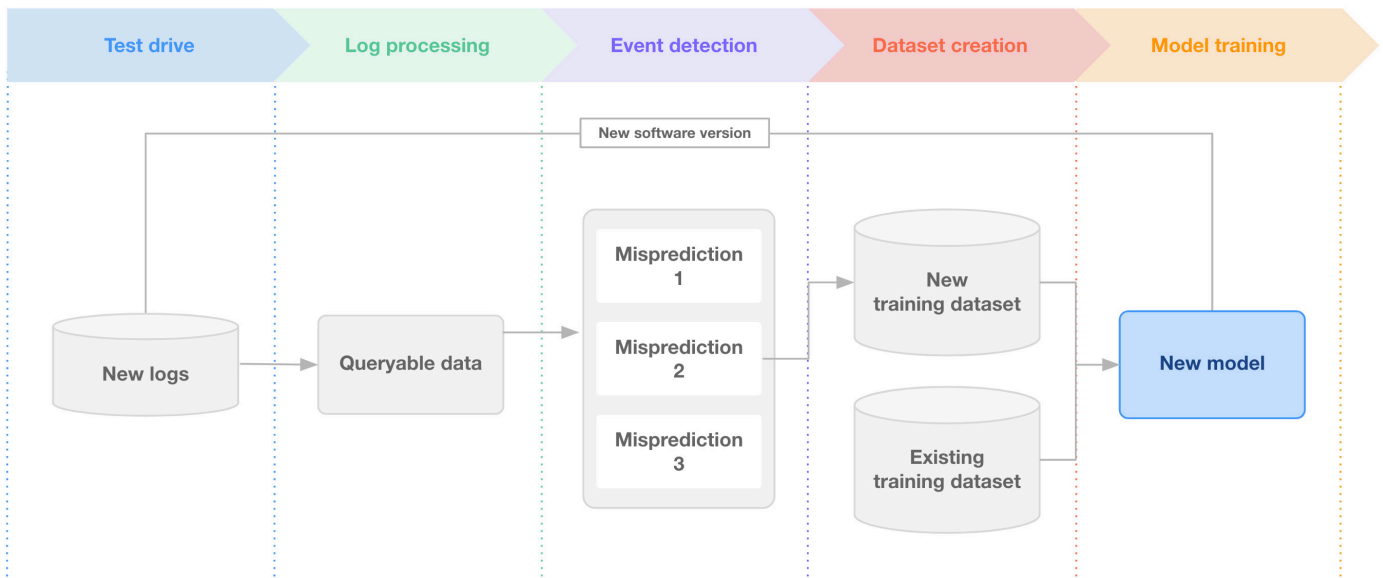


Figure 13: The iterative workflow for prediction development helps detect failed predictions, create a new dataset, and then train a new prediction model to create a positive feedback loop.

identifies and collects events and then takes automated actions on them.

The workflow for prediction development follows six steps that should be repeated to create a positive feedback loop (Figure 13). Each iteration helps the prediction module improve and learn from new situations.

1. **Test drive:** Drive a test vehicle with a prediction system running. Record a log.
2. **Log processing:** Process the log to be queryable by downstream systems.
3. **Event detection:** Search the log to find events where an actor's predicted trajectory does not match its actual trajectory.
4. **Dataset creation:** For each event, package the actor state with the future trajectory to create a label.
5. **Model training:** Retrain the prediction model with the new dataset. Test the model for regressions. If the model passes regression testing, deploy it to a vehicle as part of a new software version.
6. **Repeat:** Collect new logs and start the process again.

A helpful example of prediction systems development is provided in the article [Cruise's Continuous Learning Machine Predicts the Unpredictable on San Francisco Roads](#).

E. Validating a supplier module

Purchasing an autonomy module from a supplier can help autonomy programs reduce risk, speed up development, and keep costs predictable. However, programs might find it challenging to validate a supplier module due to a lack of control over the module, a lack of visibility into module internals, and long iteration cycles. This section describes a log-based workflow that helps autonomy programs detect errors in a supplier module, convert them into regression tests, and curate datasets of failures to send back to their supplier. To illustrate this workflow, this section explores the example of validating an object detection system.

To validate a supplier module, autonomy programs need to grade the module's output by comparing it against ground-truth data. Programs can obtain ground-truth data either from a "twin" module developed in-house, from an additional sensor, or through human labeling. Teams should choose a different method depending on the type of module they seek to validate.

For example, to validate a software-only module, autonomy programs should compare the module to a twin module with a similar output. Developing this twin module is a much lower expense than developing the entire supplier solution, as the in-house module has much less strict requirements.

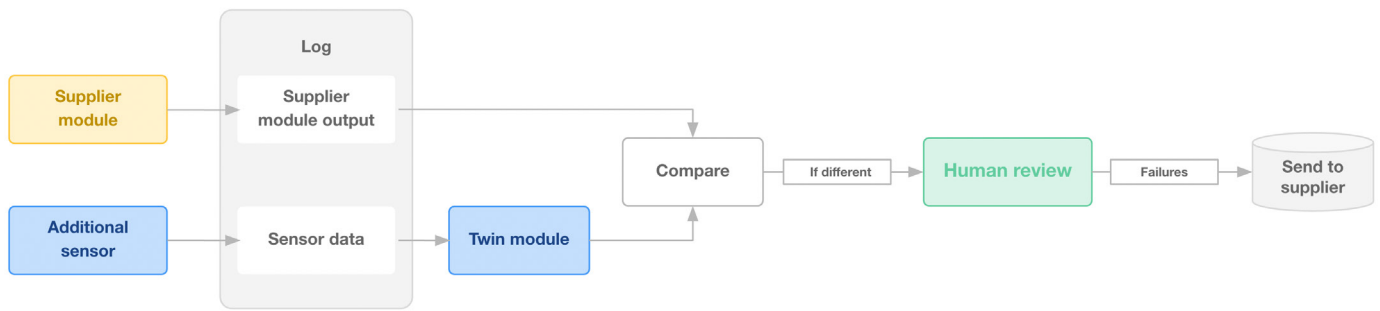


Figure 14: Finding failures in a supplier module using an additional sensor and a twin module.

The in-house module must have a reasonably correct output, but it does not need to obtain safety certifications or run on an embedded system. Teams can run the module offline only. A small team of algorithm engineers can typically build the in-house module using open-source solutions.

Validating a supplier solution that combines hardware and software typically requires a twin module and an additional sensor for validation. It is often impossible to access the input sensor data used in the module. This is very common for a combination of camera and object detection software. During testing, programs should record the additional sensor data alongside the supplier module's output. For example, a radar-based detection module from a supplier can be graded against an additional lidar sensor. The lidar sensor can provide significantly higher-quality detections than a radar sensor and helps highlight areas where the supplier solution fails.

After implementing a twin module (and optionally adding an additional sensor), the next step is to collect, analyze, and process log data. The twin module runs offline and creates an output similar to the supplier module's output. Then, teams compare the outputs of the supplier module and the offline module and flag disagreements for manual review.

Figure 14 shows how to implement this workflow for object detection systems. In this example, the supplier solution provides a camera and software, but accessing the input images is impossible. To validate this supplier module, teams should add an extra camera in the same

relative position as in the supplier solution. They should then compare the output of the supplier's module to the output of the internally-built object detection module. The internally-built module should run offline. It does not need to achieve real-time performance and can be compute-intensive.

To compare the two detections, autonomy programs should use an approach similar to [Using ground-truth labels to analyze perception performance](#). They should associate each detection with the most similar detection in the other output and compare detected attributes or classes.

This workflow for validating a camera and object detection solution is conceptually similar to the [Enriching log data with offline algorithms](#) section. Autonomy programs should weigh the cost of retraining the offline algorithm to improve its performance against the cost of human labeling.

Finally, teams collect the supplier module's failures and send a dataset to the supplier for review. They should also create test cases from the detected failures and add them to a testing suite to ensure that the supplier's next software version resolves each failure. The [Creating Test Cases From a Log](#) section of this handbook discusses how autonomy programs can create test cases from logs.

F. Improving motion planning performance

Autonomy programs in the early development stages typically use synthetic simulation and real-world testing exclusively to make their autonomous system operational

in ideal conditions. Programs in later development stages, however, spend the majority of their effort solving issues in the long tail of possible events. A long-tail issue is a combination of many different variables coming together to create adverse conditions for an autonomous system. Safety-critical issues and stack failures are two common examples of long-tail issues.

As long-tail issues are circumstantial and difficult to anticipate, synthetic simulation and real-world testing alone do not suffice to efficiently solve those issues. Instead, motion planning teams should use a log-based workflow to create test cases from logs, solve long-tail issues, and ensure that those issues do not resurface. The log-based test case creation workflow contains the following steps (Figure 15):

- 1. Create a test case to reproduce the issue.
- 2. Resolve the issue.
- 3. Add the test case to a regression suite to ensure the issue does not reappear.

The log-based test case creation workflow is not only beneficial for motion planning teams. It also allows autonomy programs to improve perception and localization modules. The following section lays out the steps required to create test cases from logs and the benefits that these test cases provide to autonomous system module development.

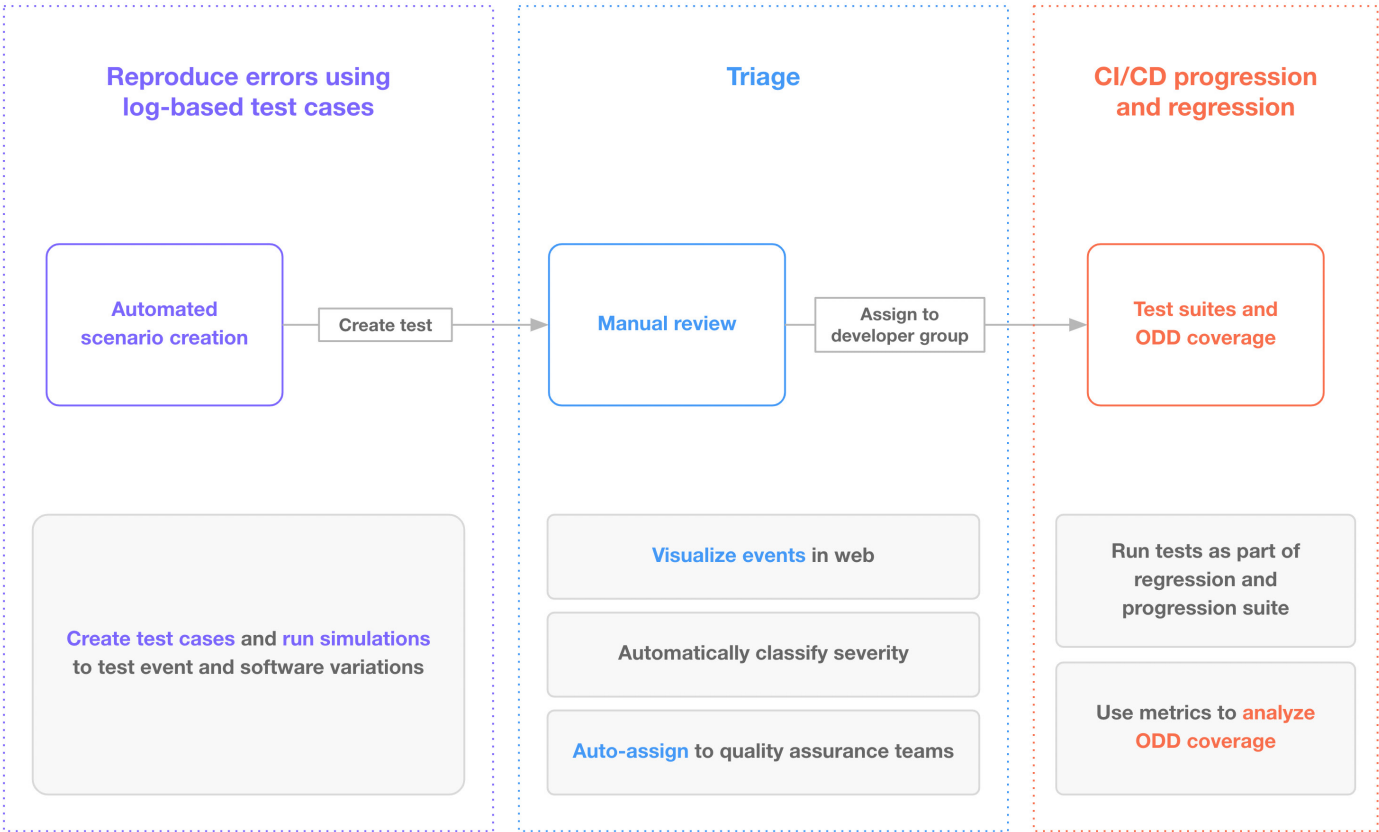


Figure 15: A log-based workflow enables motion planning teams to create test cases from logs, fix long-tail issues, and ensure these issues do not reappear.

IV. Creating Test Cases From a Log

The previous section of this handbook discussed a number of workflows involving logs, including diagnosing issues from testing, curating datasets to train ML models, improving perception and prediction performance, and validating a supplier module. This section dives deeper into the log-based test case creation workflow, which helps autonomy programs improve perception, localization, and motion planning modules.

The log-based test case creation workflow helps solve long-tail issues found during real-world testing. This workflow involves the following steps:

1. Create a test case to reproduce the issue.
2. Resolve the issue.
3. Add the test case to a regression suite to ensure the issue does not reappear.

Optionally, autonomy programs can [fuzz](#) the created test case to stress test their autonomy software and find adjacent scenarios that may cause failures.

A. Reproducing an issue: Test case creation

Creating a test case from a log should be quick—ideally one or two clicks. Alongside the test case, teams should also generate pass/fail rules that determine the test's outcome. Once they have created a test case, teams should run it on their current autonomy stack to reproduce the long-tail issue.

There are two ways to create test cases from a log: Scenario extraction and log re-simulation. Scenario extraction creates a synthetic test with actor behaviors sampled from the perception outputs in the log. Log re-simulation replays the original logged data to the autonomy stack without any synthetic signals. Both these types of test case creation have strengths and weaknesses. Behavior extraction is typically portable between vehicle programs (e.g., an L2 and an L4 autonomy program within

the same organization) and is robust to stack changes. Log re-simulation has higher fidelity and is able to losslessly recreate the exact timing and content of signals sent to the autonomy stack. The following sections describe scenario extraction and log re-simulation in more detail.

Scenario extraction

When extracting a scenario from a log, autonomy programs create a synthetic test case with actor behaviors sampled from the existing log's perception outputs. To generate the actor behavior, teams sample actor detections from the perception system and apply a realistic behavior to achieve the desired motion. They then run the resulting synthetic test case in a target simulator to reproduce the long-tail issue. Scenario extraction usually takes place in the form of a script or tool that selects a portion of the log for extraction and then outputs a file that describes the test case.

Test cases that are created using scenario extraction can contain anything from actor poses and behaviors to traffic control device states and even the entire synthetic environment including the base map and buildings. Autonomy programs usually leverage scenario extraction at the object level: They extract actor poses, actor behavior, and traffic control devices from the log but generate the underlying map and environment separately.

The strengths of scenario creation include portability and robustness to stack changes. The created test cases can also be stack-agnostic. Thanks to these strengths, autonomy programs can create test cases from logs that are many years old and run them on their current autonomy stack. Scenario extraction also makes it easy to share interesting situations among different vehicle programs.

The weaknesses of scenario extraction include fidelity and perception dependence. When extracting a synthetic scenario from a real-world log, the log's message

content, timing, and latency information are lost. Many safety-critical issues arise from complex, time-sensitive interactions between different modules in the autonomy stack. These issues are often impossible to reproduce using scenario extraction. Scenario extraction is unable to preserve noise in logged signals. For example, if a camera sensor and an object detection module provide data on a delay, the downstream fusion system may be late to detect an obstacle. This exact timing would be lost in scenario extraction but reproduced correctly in log re-simulation.

To successfully extract synthetic actors or object behaviors, the perception stack needs to output high-quality detections. If the logged perception stack fails to populate object classifications (e.g., pedestrian, car, truck, bicyclist) or the shape of the detections is incorrect, then the created test case will have all of these deficiencies as well.

Log re-simulation

The conceptually simplest way to reproduce an issue from a log is to run the autonomy stack against the previously recorded log. This method is called log replay. Log replay and log re-simulation both run the original messages or sensor data from the log against the current autonomy stack. However, log re-simulation adds a simulator into the loop (Figure 16). This simulator controls timing and dynamics.

The addition of the simulator thus achieves lossless fidelity and determinism (i.e., the guarantee that, given the same inputs, a simulation will always produce the same result, no matter how often teams run it). This provides autonomy programs a greater chance to successfully resolve the long-tail issue. Because log re-simulation provides a higher degree of determinism than log replay, it is well suited for test case creation.

Autonomy programs can execute log re-simulation in two specific modes: Closed-loop re-simulation and open-loop re-simulation. Closed-loop re-simulation is most useful for disengagement analysis and motion planning development (Figure 17). It helps answer the question “What would have happened if the system continued without intervention”? Closed loop re-simulation involves a vehicle dynamics model interacting with the controller, and it can become inaccurate if the behavior of the re-simulated autonomous system diverges too much from the behavior in the original log. Teams should be careful to keep a re-simulation test accurate when the loop is closed.

Open-loop re-simulation does not involve a vehicle dynamics model, so the position of the autonomous system in the re-simulation is identical to the position in the original log. Instead of testing the motion planning module, open-loop re-simulation helps test perception and localization

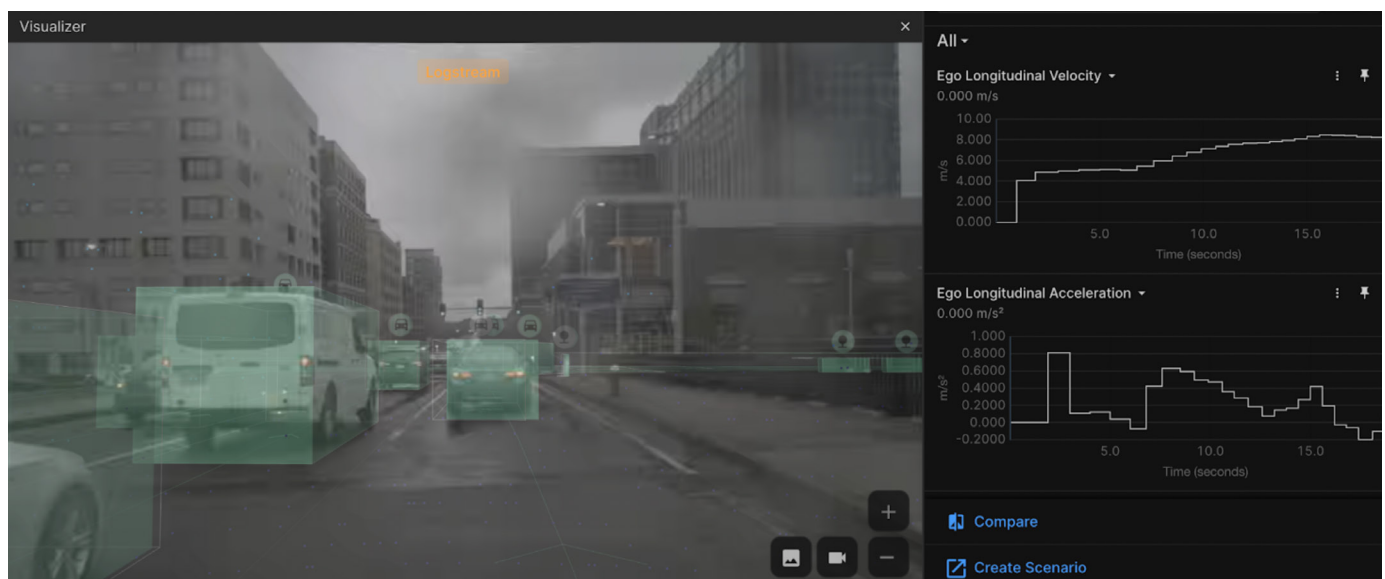


Figure 16: Perception stack detections with pinned front camera detections.

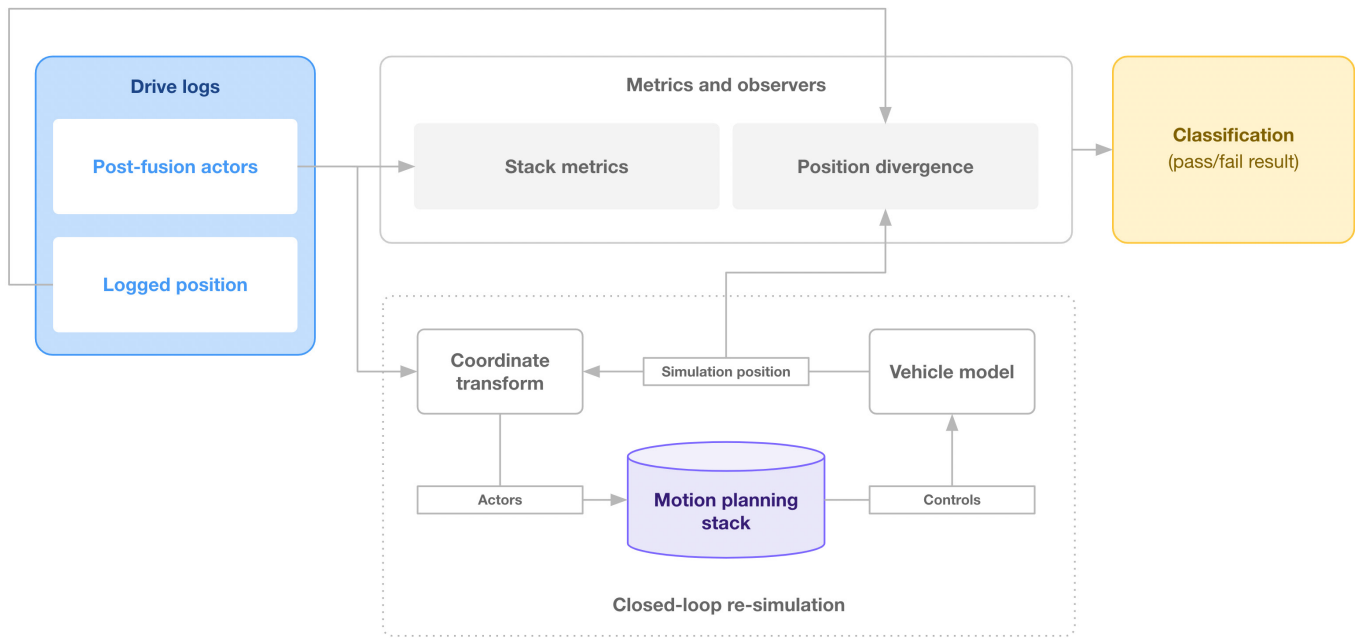


Figure 17: Closed-loop re-simulation helps test the motion planning stack. A coordinate transformation ensures the motion planning stack receives actor positions that accurately relate to the autonomous system's position in the closed loop re-simulation.

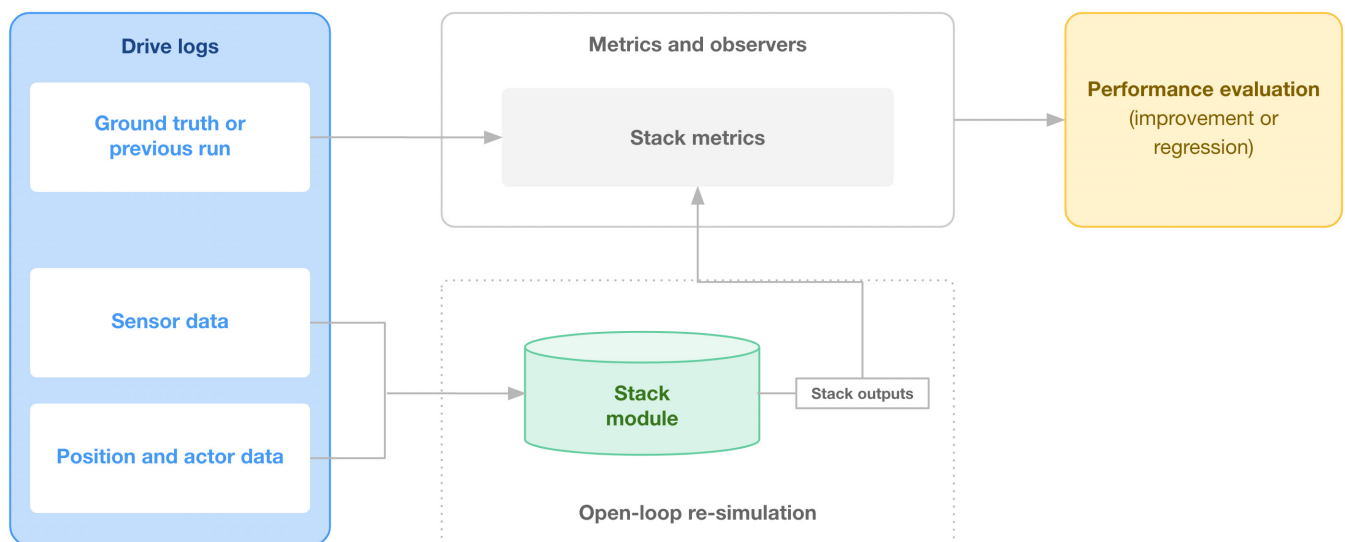


Figure 18: Open-loop re-simulation helps test perception and localization modules.

modules (Figure 18). For example, testing the perception system in open-loop re-simulation involves playing sensor data from a log into the perception software, and then grading the output via a scoring mechanism.

Systems engineers, triage teams, and motion planning engineers typically rely on closed-loop re-simulations,

as they must assess the safety of the vehicle's motion. Perception, prediction, and localization teams typically rely on open-loop re-simulation, as those module outputs can be graded without observing a change in vehicle position. For example, a localization system can be graded based on observing the outputs of the localization module, without involving a downstream system such as the planning

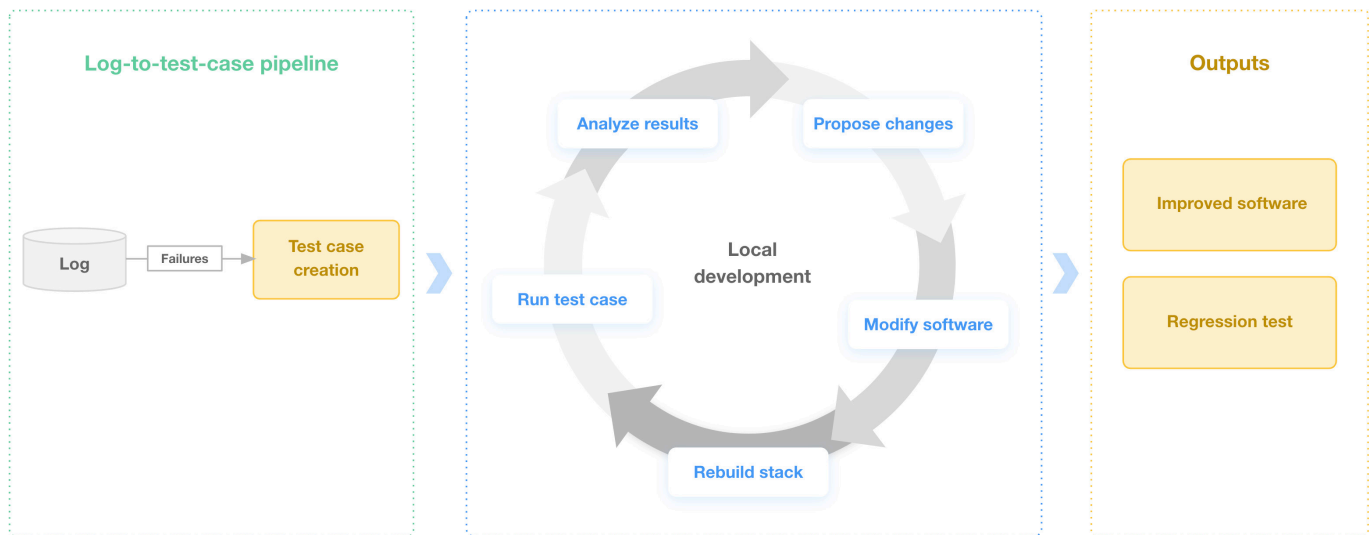


Figure 19: Log-based test case creation lets teams use real data to improve their autonomy stack.

module. Teams should choose the type of re-simulation that is right for them depending on the type of test case they want to create from a log.

The strengths of log re-simulation lie in its determinism and fidelity. However, it is less flexible than scenario extraction. Log re-simulation is inherently tied to the original log. If a new version of the autonomy program's software renders data in the log invalid, then the re-simulation can become invalid as well. There are a few techniques to keep older log data relevant. The [Keeping existing log data usable](#) section discusses these techniques in more detail. However, every log eventually becomes too old to be useful in re-simulation. Once this happens, autonomy programs should either use scenario extraction or deprecate the test case entirely.

Choosing between scenario extraction and log re-simulation

Log re-simulation is an extremely powerful asset to the autonomy programs that decide to invest in this method. Later-stage autonomy programs typically create the majority of their simulated miles from logs using re-simulation, thanks to the advantages that log re-simulation brings compared to scenario extraction.

That being said, the correct tool to create test cases

from scenarios is whatever helps an autonomy program reproduce a long-tail issue fastest. Scenario extraction usually suffices for simple test cases. Log re-simulation may be more affordable than scenario extraction for perception or object detection issues. When reproducing long-tail issues caused by noise, latency, or hardware, re-simulation is the only available choice.

B. Resolving the issue

Once autonomy programs have reproduced a long-tail issue by creating a test case and receiving a failing result, they can now resolve the issue locally by using real data from the log to improve their autonomy stack. The log-based test case creation workflow makes this possible (Figure 19). The team should iteratively modify the autonomy stack and re-run the created test case. This iteration loop should be as quick as possible. Once the test passes, the stack changes are considered a true fix. It is important to note that a passing result for the created test case does not reduce the need for other software testing such as integration and unit tests.

C. Ensuring the issue does not reappear: Regression and progression tests

Successful autonomy programs ensure that their autonomous system does not fail twice in the same way.

After reproducing and resolving a long-tail issue locally, teams should add the created test case to a regression test suite that regularly executes comprehensive tests in CI. If a long-tail issue appears unsolvable due to sensor or compute deficiencies, programs can add it to a progression test suite that evaluates the stack's progress toward aspirational goals. Programs should also create dashboards that monitor the overall health of their autonomy stack and give all team members a high-level overview of stack performance over time.

D. Advanced re-simulation: Fuzzing

In addition to effectively reproducing issues encountered during tests, re-simulation enables teams to create variations of the same test case to stress-test the autonomous system in simulation. Autonomy programs can use a technique called fuzzing to achieve this. For example, fuzzing allows teams to make a cut-in scenario more aggressive by changing the distance between the autonomous system and the vehicle that is cutting in front of it. It also allows programs to make an unprotected left-turn scenario more difficult by adding more traffic to it. Autonomy programs in the later stages of development typically set up rules to automatically fuzz re-simulation test cases with the goal of exposing situations that cause the stack to fail.

Resolving and uncovering issues

Re-simulation fuzzing solves two problems. First, when resolving a long-tail issue, it gives individual engineers

the confidence that they have not accidentally overfit a stack change to an individual test case. Second, it helps validation teams expose unknown issues that the team could otherwise only catch with more unstructured testing.

With the help of fuzzing, every mile of real-world testing thus turns into tens or hundreds of miles of valuable re-simulation tests. Slight changes in multiple variables such as actor behavior, traffic signal timing, or object detection accuracy can create large variations in a scenario (Figure 20), which may cause failures that are important for validation teams to learn from.

In a 2017 article entitled [How simulation turns one flashing yellow light into thousands of hours of experience](#), Waymo explains how it fuzzed a re-simulation to teach its vehicles how to behave with respect to flashing yellow traffic signals.

Object-level and sensor-level fuzzing

Autonomy programs that do not yet use fuzzing on their existing re-simulation tests should build object-level fuzzing capabilities to implement fuzzing efficiently. Object-level fuzzing capabilities allow teams to manipulate traffic signals, actor classifications, actor behavior, and road geometry. These capabilities typically provide the most immediate value to autonomy programs that are just starting to use this technique.

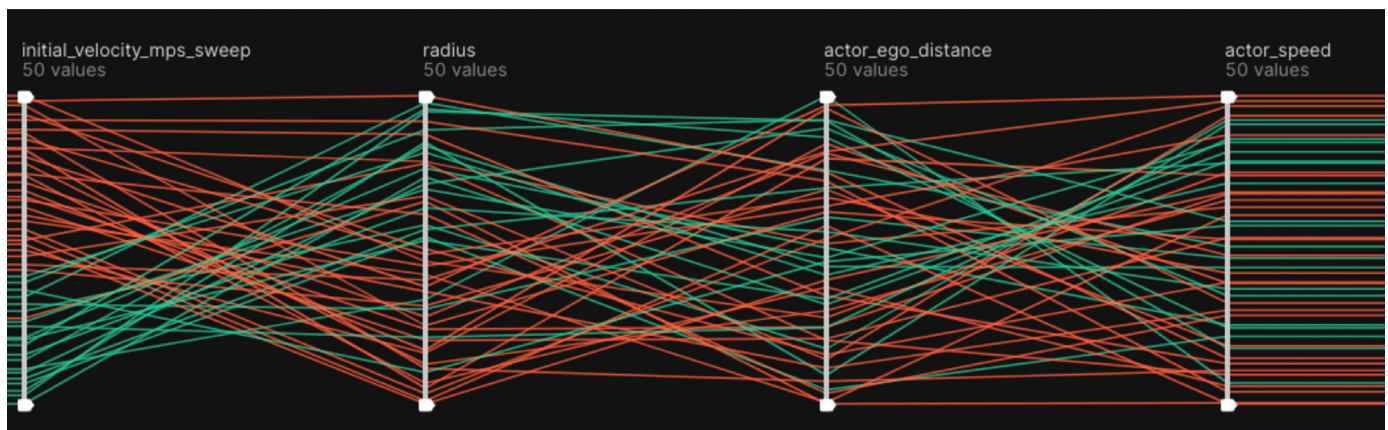


Figure 20: Slight changes in multiple variables can create large variations in a scenario.

In contrast to object-level fuzzing, some autonomy programs prefer using sensor-level fuzzing. Sensor-level fuzzing includes repainting camera images, injecting new object detection or removing original detections in point clouds, or otherwise synthetically modifying sensor data. This method is significantly more difficult to carry out compared to object-level fuzzing. It also widens the simulation-to-real gap (i.e., a degradation in object detection performance due to a difference between synthetic data and the target domain in the real world) more dramatically than object-level fuzzing does. For these two reasons, autonomy programs should usually opt for object-level fuzzing.

This section discussed a workflow for creating test cases from logs, choosing the right type of test case, picking the correct re-simulation setup, and fuzzing test cases to find new failure modes. The following section discusses how to store log data, generate artifacts, and control costs for an autonomy program's log storage life cycle.

V. Log Storage and Archival

Once autonomy programs have collected, processed, and explored a log file, they should store it in a data center long-term. As their data pipeline processes logs, teams end up duplicating the data and transforming log files into different formats. Autonomy programs should generally save all of these formats for as long as possible. However, as their fleet grows in size, the amount of data that teams collect increases, and long-term log storage costs tend to rise as a result.

Different teams consume log data in different ways. While one team may require raw time-series data, another one may watch video recordings. A third team may need access to raw sensor data such as camera images or lidar point clouds. In order to efficiently support a diverse set of use cases, an autonomy program should store its log data in specialized formats.

To reduce costs, autonomy programs should carefully choose the architecture of their storage system. They should apply different storage policies to different classes of log data and measure log storage costs directly against the cost of collecting new data.

The following sections discuss common storage formats and cost-effective storage architectures in more detail.

A. Storage types and container formats

During real-world testing, vehicles collect two high-level classes of data: Raw sensor data and derived structured data. Autonomy programs should separate these two types of data to make processing, storage, and querying more cost-effective. Raw sensor data should be stored in visualization formats that browsers or visualization tools can read. Structured data should be stored in formats that are queryable by large query engines.

Visualization formats

Engineering and operations teams who investigate issues from real-world testing and explore events of interest need to visualize raw sensor data. For example, the most common visualization is a view of the autonomous system's camera data alongside its position and state. Autonomy programs should transform raw sensor data into a visualization format for storage because it provides the following benefits:

- Reduces storage costs
- Lowers latency to visualize data
- Increases playback performance

Examples of common transformations include:

- Transforming raw camera data from images into a video format; this can reduce size on disk by 90%
- Transforming raw lidar data from a byte-packed form into a general lidar point cloud format
- Transforming raw radar data into a visualization message such as `radar_msgs`

To decide on the correct visualization format, autonomy programs should primarily consider their disk size and the readability by other tools in their stack.

Video conversion

The conversion of raw camera frames into a video format is one of the most common transformations in a data pipeline. However, it can introduce timing artifacts, which, in turn, can cause data integrity problems. Real cameras occasionally drop or fail to record frames. When converting raw camera frames into a video format, teams should carefully synchronize the individual frames with the remainder of the messages in their system. Otherwise, missed frames will cause the video to be displayed alongside incorrectly synchronized graphs or time series data.

Query formats

Structured data originates from the communication between an autonomous system’s individual software modules. Structured data usually consists of groups of primitive types that are sent across a robotics middleware or CAN system. These primitive types include booleans, integers, floats, enums, and strings, grouped into lists or sub-structures.

Structured data is essential to scenario search, V&V, measurement of key performance indicators (KPIs), and algorithm development. Teams often need to query structured data (Figure 21) across long periods of time (e.g., weeks, months, or years) and across entire fleets of vehicles. They also need to be able to access specific subsets of structured data. For example, a query might state: “Find all instances where the autonomous system was at a red light, turning left, and had zero velocity for over one minute.” To service this query, an autonomy program needs to have readily available structured data about the system’s velocity, the state of traffic control devices, and the system’s intention to turn left. For efficient query performance on petabytes of log data, the query engine should scan only the data required to filter.

Ideally, a structured data query format should support a flexible schema, accept nested data, be optimized for reading, and support popular query engines and file formats such as [Parquet](#), [ORC](#), or [Avro](#). File formats like these allow teams to read only the data that is relevant to the query at hand.

Autonomy programs should transform a log file into a query data format as soon as the original log arrives in a production log management system. After the log is transformed, teams can schedule various downstream tasks based on it. For example, teams might want to schedule a task to create snippets of original log files that include sensor data around certain events of interest.

B. Data retention and archival

Length of storage

Autonomy programs should retain original log files for many years. This is a high-cost requirement, but teams can offset these costs by leveraging strategies to reduce data storage costs.

The general advantage of keeping original log files for as long as possible is the positive effect this has on the

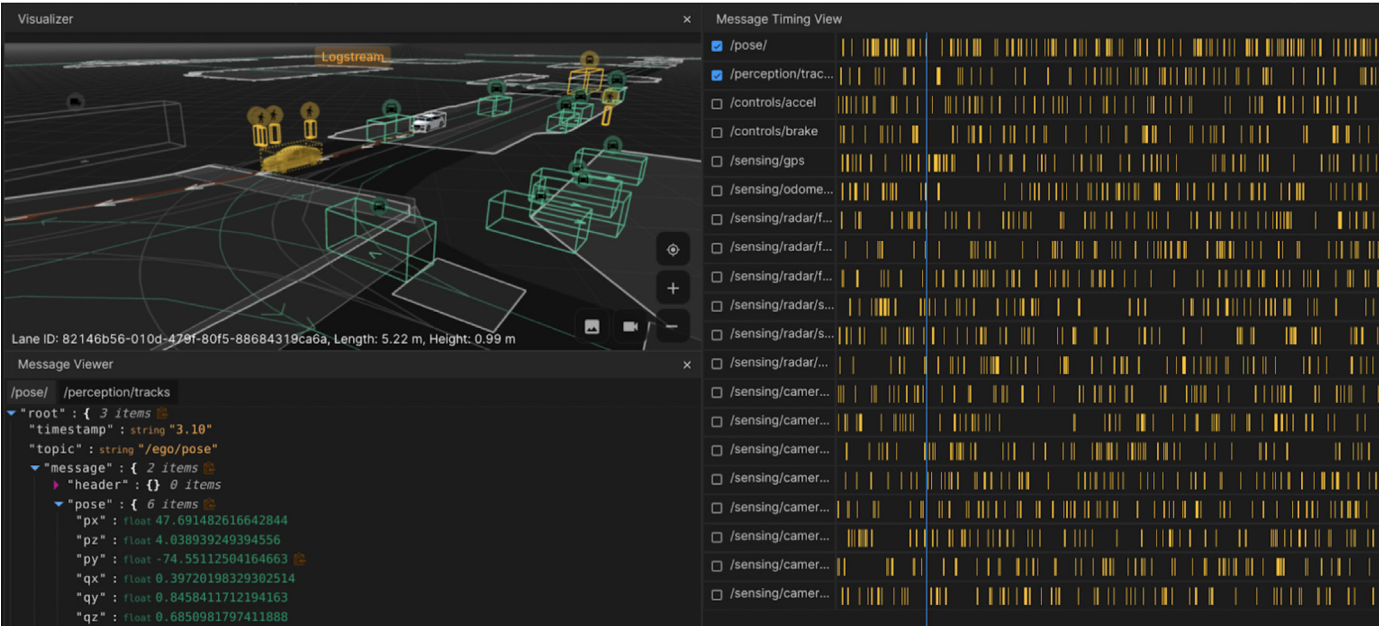


Figure 21: A web-based frontend showing structured log data.

velocity of engineering and operations teams. Those teams benefit greatly if they are able to access the largest possible unique set of events their fleet has experienced so far.

Of course, there is a financial tradeoff between keeping original log files stored for longer periods of time and collecting new logs instead. To decide when to archive a log file, autonomy programs should thus consider the cost of storage, how often they tend to re-access the log, and how much it would cost to collect a new log instead. The cost of storing a 1-TB log for one year is around \$50 at the time of writing. Reproducing the exact scenario of interest in real-world testing, however, is much more expensive in terms of labor cost and operational overhead. It is also often technically infeasible in urban environments, where the general public interacts with the autonomous system. Additionally, teams might find it difficult to know a priori whether they will need a specific segment of a log file in the future.

Given the difficulty of reproduction, the low cost of storage relative to the cost of collecting new data, and the uncertainty of needing to access the file again, long-term retention is the correct choice for most autonomy programs. Luckily, storage costs have been decreasing continuously over time as hard drive storage has become cheaper and more powerful. Additionally, autonomy programs with [fleets of 10-100 test vehicles](#) typically start automating and minimizing their data collection efforts to only collect relevant logs. This practice further increases the chances that long-term storage is the right approach for every collected log.

Data retention policies

Once autonomy programs have decided for how long they wish to keep each log, they should leverage data retention policies to manage storage costs. By aggressively moving data into cheaper storage tiers, teams can reduce storage costs by about 40% compared to an unconfigured public cloud storage solution (Figure 22).

When storing data in a public cloud, autonomy programs should store original log files in a “cold” storage tier. Cold storage tiers provide a cheaper per-month cost, but teams pay an additional fee if they wish to access the data. Alternatively to an additional fee, data access might be slower (up to 12 hours, depending on the type of storage medium). In a cold storage tier, one TB of data costs between \$50-\$100 per month at the time of writing. Teams should place original log files into this tier as soon as the files have finished processing.

Data that the team queries or visualizes should be kept in “hot” storage tiers. Hot storage tiers provide cheap and fast access to the stored data at the cost of higher per-month storage bills. For example, teams often retrieve and view GPS data, perception outputs, and converted video files live on a visualizer. Since a human makes the request to visualize this data, waiting multiple minutes for the data to be returned would be untenable. A hot storage tier satisfies these requirements.

The table below contains a sample data retention policy with costs that are typical for public cloud storage at the time of writing in the United States¹ (Figure 22). The outlined pricing assumes 500 TB of original logs with 25% camera images, 25% lidar points, and 50% structured data. The “Monthly tiered cost” column shows potential prices from different storage types or “tiers.”

¹Based on pricing from [AWS](#) and [Azure](#) as of October 2022





Data format	Size (in TB)	Monthly unconfigured cost		Monthly tiered cost	
					
Original logs	500	\$10,500	\$9,000	\$2,000 \$0.004 per gigabyte (GB) via Glacier tier	\$495 \$0.00099 per GB via Archive tier
Queryable format	250	\$5,250	\$4,500	\$3,125 \$0.0125 per GB via Infrequent Access (IA) tier	\$4,500 \$0.018 per GB via Hot tier
Visualization formats	100	\$2,100	\$1,800	\$1,250 \$0.0125 per GB via IA tier	\$1,800 \$0.018 per GB via Hot tier
Total	850	\$17,850	\$15,300	\$6,375	\$6,795

Figure 22: Typical costs of cloud storage in the United States.

Conclusion

We hope that this handbook provides autonomy programs with useful concepts, benefits, and industry practices of an expansive log data management process. From log collection and exploration to triage and improvements to various modules in an autonomy stack, log data is an essential building block for successful autonomy development. This handbook also aims to provide guidance on how autonomy programs can execute log data management effectively, utilize their resources efficiently, and reduce costs.

While this handbook lays out many important concepts and tactical steps, it may not address all of our readers' questions. The Applied team is happy to discuss these questions and support autonomy programs of all sizes and industries with their growing data collection and management needs. Learn more on [our website](#) and [contact us](#) to speak with our team.

Glossary

CAN: Controller area network
CD: Continuous deployment
CI: Continuous integration
CPU: Central processing unit
ECU: Electronic control unit
EDR: Event data recorder
ETL: Extract, transform, load
FN: False negative
FP: False positive
GB: Gigabyte
GDPR: General Data Protection Regulation
GPS: Global Positioning System
GT: Ground truth
HD: High-definition
IA: Infrequent Access
IMU: Inertial measurement unit
IS: ID switches
KPI: Key performance indicator
L2: SAE Level 2
L3: SAE Level 3
L4: SAE Level 4
mAP: Mean average precision
ML: Machine learning
MOTA: Multiple object tracking accuracy
MOTP: Multiple object tracking precision
NTP: Network Time Protocol
ODD: Operational design domain
Protobuf: Protocol Buffers
PTP: Precision Time Protocol
RAM: Random-access memory
ROS: Robot Operating System
SAE: Society of Automotive Engineers
TB: Terabyte
TP: True positive
UI: User interface
V&V: Verification and validation



Applied Intuition

applied.co/contact