D3: A Dynamic Deadline-Driven Approach for Building Autonomous Vehicles

Ionel Gog
UC Berkeley

Sukrit Kalra
UC Berkeley

Peter Schafhalter
UC Berkeley

Joseph E. Gonzalez
UC Berkeley

Ion Stoica
UC Berkeley

Abstract

Autonomous vehicles (AVs) must drive across a variety of challenging environments that impose continuously-varying deadlines and runtime-accuracy tradeoffs on their software pipelines. A deadline-driven execution of such AV pipelines requires a new class of systems that enable the computation to maximize accuracy under dynamically-varying deadlines. Designing these systems presents interesting challenges that arise from combining ease-of-development of AV pipelines with deadline specification and enforcement mechanisms.

Our work addresses these challenges through D3 (**D**ynamic **D**eadline-**D**riven), a novel execution model that centralizes the deadline management, and allows applications to adjust their computation by modeling missed deadlines as *exceptions*. Further, we design and implement ERDOS, an open-source realization of D3 for AV pipelines that exposes fine-grained execution events to applications, and provides mechanisms to speculatively execute computation and enforce deadlines between an arbitrary set of events. Finally, we address the crucial lack of AV benchmarks through our state-of-the-art open-source AV pipeline, Pylot, that works seamlessly across simulators and real AVs. We evaluate the efficacy of D3 and ERDOS by driving Pylot across challenging driving scenarios spanning 50km, and observe a 68% reduction in collisions as compared to prior execution models.

CCS Concepts: • **Computer systems organization** → **Heterogeneous (hybrid) systems**; **Special purpose systems**.

ACM Reference Format:
Ionel Gog, Sukrit Kalra, Peter Schafhalter, Joseph E. Gonzalez, and Ion Stoica. 2022. D3: A Dynamic Deadline-Driven Approach for Building Autonomous Vehicles. In *Seventeenth European Conference on Computer Systems (EuroSys '22), April 5–8, 2022, Rennes, France.* ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/3492321.3519576

EuroSys '22, April 5–8, 2022, Rennes, France
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9162-7/22/04.
https://doi.org/10.1145/3492321.3519576

1 Introduction

The National Highway Traffic Safety Administration [4], expects advances in autonomous vehicles (AVs) to: (*i*) reduce human error from traffic accidents, which made up for 94% of the 37,133 vehicle related deaths in the U.S. in 2017 [75], (*ii*) increase traffic flow, which could free up as much as 50 minutes per person per day [73], and (*iii*) provide new employment opportunities to around 2 million people with disabilities [46]. Despite the potential benefits and investment [5], AV systems research is still in its infancy [12, 14–16, 27, 61].

While most AV research has focused on the models and algorithms that underpin the perception, planning and control decisions, there has been little work on the software systems that support their execution. To safely drive in complex environments, AVs must ensure highly-accurate results by executing complex pipelines with hundreds of computationally-intensive algorithms and neural networks [95] using multiple parallel processors and hardware accelerators [66]. As a result, the software systems for AVs must support a *deadline-driven* execution of such pipelines that allows them to maximize their accuracy under a given deadline, which is complicated by their two unique characteristics (discussed further in §2):

C1: Environment-dependent deadlines. AVs need to complete their computation at varying timescales to safely drive across the wide array of scenarios in the real-world. For example, navigating a crowded urban street requires different algorithms and can tolerate longer computation times than swerving in response to an obstacle on the freeway [25, 45].

C2: Environment-dependent runtimes. The runtime of various stages of an AV pipeline like pedestrian tracking vary with the input (e.g., the number of pedestrians). As a result, these stages exhibit *runtime-accuracy* tradeoffs that must be addressed *dynamically* according to the environment [44, 97].

The current state-of-the-art systems for autonomous driving (e.g., Autoware [28], Cruise [95], BMW [20] etc. [53, 94]) are built atop the Robot Operating System (ROS) [77]. ROS was designed as an execution platform for enabling robotics research, and achieves its key goal of supporting the construction of complex pipelines through its modular design [19] and best-effort execution of the stages. However, these systems lack mechanisms to specify and enforce deadlines on the computation thus precluding a deadline-driven execution of an AV pipeline (**C1**), which is critical for vehicle safety.

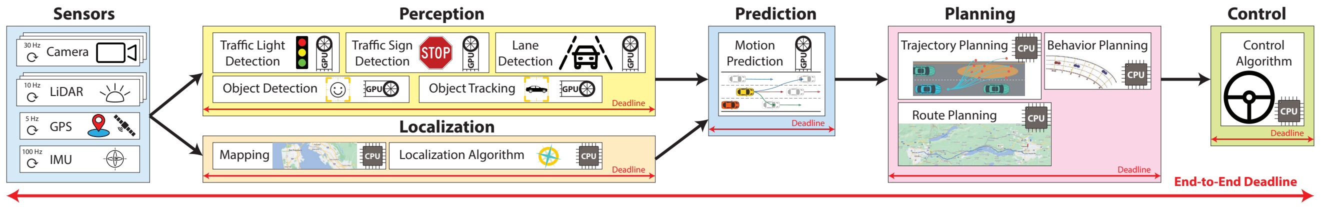


Figure 1. The architecture of a state-of-the-art AV pipeline. A modern AV uses multiple sensors to perceive the environment around it. These sensor readings are used by the *perception* module to detect other agents, and by the *localization* module to compute the location of the AV itself. The *prediction* module uses their output to predict the future trajectories of other agents, and the *planning* module computes a safe and feasible trajectory for the AV using these predictions. Finally, the *control* module produces steering and acceleration commands.

Conversely, decades of work in cyber-physical systems has produced sophisticated techniques for safety-critical applications that ensure the fulfillment of strict deadlines [32, 33, 36, 49, 56, 69]. However, these techniques require a comprehensive, time-consuming analysis of the schedulability of the stages driven by estimates of their worst-case runtimes. Since various stages of the pipeline exhibit environment-dependent runtimes (**C2**), there exists a wide variance between their average and worst-case runtimes. Thus, any schedulability analysis driven by the latter is overly-conservative and leads to an under-utilization of the compute resources, which could be used to execute higher-accuracy algorithms and optimize the runtime-accuracy tradeoff [44, 97] (elaborated in §3.1).

We posit that a new class of systems is required to enable a deadline-driven execution of applications that must interact with a continuously-evolving environment (e.g., robotics, AVs), and exhibit **C1-C2**. Such systems must combine the ease-of-development of state-of-the-art robotics platforms with the deadline specification and enforcement mechanisms of cyber-physical systems. Specifically, such systems must enable applications to specify deadlines that evolve with the environment (**C1**), and adapt their computation to such deadlines to maximize the runtime-accuracy tradeoff (**C2**).

Our work seeks to provide a general execution model for such applications and propose the design of a proof-of-concept system that realizes this model. To achieve this goal, this paper makes the following two key contributions:

(**A**) We propose D3 (**D**ynamic **D**eadline-**D**riven), an execution model for applications that interact with a continuously-evolving environment, and exhibit **C1-C2**. D3 decomposes the application as a graph of computation along with a *deadline policy*, which determines the deadline according to the environment (**C1**). While applications proactively try to meet deadlines, D3 models missed deadlines due to **C2** as *exceptions* and allows the execution of *reactive measures*. Further, D3 notifies downstream computation about missed deadlines, allowing it to *eagerly execute* on incomplete input or adjust to fit in the reduced time upon arrival of the input (see §5.3).

(**B**) We design and implement ERDOS, a proof-of-concept realization of D3 built specifically for AV pipelines. ERDOS exposes fine-grained execution events to the application and provides abstractions for the specification of dynamically-varying deadlines that restrict the wall-clock time elapsed between such events. ERDOS' speculative execution mechanism then aims to fulfill a deadline by executing the appropriate implementation automatically (see §5.3). However, if deadlines are missed due to **C2**, ERDOS executes *exception handlers* that allow computation to convey intermediate results to enable the execution of downstream computation.

The remainder of the paper elaborates on the contributions that enable D3 and ERDOS, and is organized as follows:

(**1**) We introduce and underscore the importance of the two unique characteristics of applications that interact with a continuously-evolving environments (**C1-C2**) by analyzing data collected from both our own real AV and the sensor data released by multiple state-of-the-art AV vendors (§2).

(**2**) We elaborate on D^3, a novel execution model that enables such applications to maximize their accuracy (§3).

(**3**) We present the techniques that enable ERDOS to support D3 (§4-§5) and provide the first open-source implementation of a deadline-driven system built for AVs (§6).

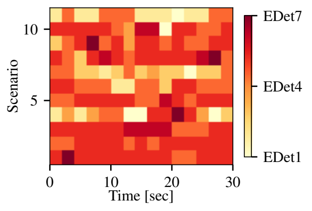
(**4**) We address the crucial lack of AV benchmarks by providing the first open-source state-of-the-art AV pipeline, Pylot (§7.1). Pylot works across simulators and real-vehicles, and achieves the top score in a simulated AV challenge.

(**5**) We evaluate the efficacy of the dynamic deadline-driven execution enabled by D3 and ERDOS by driving Pylot across 50 km of challenging driving scenarios in simulation (§7), and observe a 68% reduction in collisions as compared to the execution model of state-of-the-art robotics platforms.

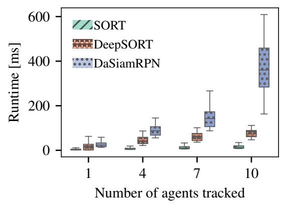
2 Background and Motivation

An AV is equipped with multiple instances of sensors such as cameras, LiDARs, radars etc., that complement each other and enable the AV to build a representation of the environment [7–9]. These sensors, operating at different frequencies, collectively generate $\sim$ 1 GB/s of data, which is processed by a computational pipeline consisting of five modules: perception, localization, prediction, planning, and control (Fig. 1). These modules are implemented by hundreds of components [95] that execute atop several machines and accelerators [66, 72].

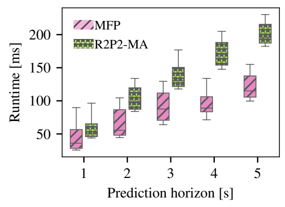
The *perception* module synchronizes the camera and LiDAR data streams and uses machine learning (ML) models to detect pedestrians, vehicles, traffic signals, and lanes. These detected objects along with the AV's location (computed by the *localization* module) are used by the *prediction* module



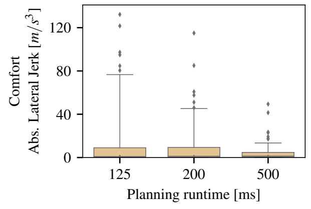
(a) Object detection.



(b) Object tracking.



(c) Trajectory prediction.



(d) Trajectory planning.

Figure 2. No silver bullet. We underscore the need for dynamically-varying deadlines by showing that: the choice of the optimum object detector varies widely both within and across driving scenarios (**a**), the runtimes of various components increase with an increased complexity of the environment (**b**, **c**), and components benefit from an increased allocation of time, which leads to more comfortable rides (**d**).

to predict the trajectories of other vehicles and pedestrians, and create a representation of the environment around the AV. This representation is used by the *planning* module, which first computes coarse-grained waypoints from the AV's location to the destination (using a route planner), and then refines these waypoints to ensure a comfortable ride (e.g. by minimizing jerk) while avoiding collisions (using a trajectory planner). Finally, the *control* module converts these fine-grained waypoints to steering and acceleration commands.

It is imperative that while the pipeline produces accurate results, it also computes them within a specific environment-dependent deadline (**C1**) in order to prevent collisions or unnecessary emergency maneuvers that affect the comfort of the passengers [66]. However, these two requirements are often at odds since higher-accuracy components typically incur an increased response time, and the optimization of this runtime-accuracy tradeoff is further complicated by **C1-C2**. In the remainder of the section, we analyze these two unique characteristics using data collected from both our own real AV and the sensor data released by state-of-the art vendors.

2.1 C1: Environment-dependent deadlines

Ensuring safety across the wide-range of complex scenarios encountered in general driving requires an AV to dynamically change its response time to meet the varying deadlines demanded by the environment. To demonstrate this, we divide 12 driving scenarios from a real-world dataset [3] into 2 second intervals, and plot the object detection model with the highest accuracy (adjusted by its runtime [65]) from the EfficientDet family [88], which provide multiple points in the runtime-accuracy tradeoff curve. Fig. 2a shows that models with differing runtimes and accuracies perform better at different times, which renders the selection of a static point on the tradeoff curve during development inadequate.

To further support this, we develop a scenario using our real vehicle where a replica of a pedestrian walks out in front of the AV, and requires the AV to brake upon its detection[1]. In order to check if the AV can safely stop in time, we measure the *stopping sight distance* [11], which is the sum of the distance traveled by the AV during the detector's response time and the distance required to come to a halt (i.e., braking distance). To

[1]A simulation of this scenario can be found at https://tinyurl.com/j4mhezze

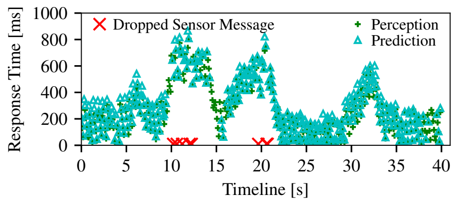


Figure 3. Response time variability. Baidu's Apollo production-grade perception and prediction suffer from response time variability.

explore the tradeoff, we choose detectors EDet6 and EDet2 from the EfficientDet family where EDet6 is accurate at the expense of a higher response time, and EDet2 is faster but less accurate. Hence, while EDet6 can detect the pedestrian 72m away, EDet2 can only do so at a distance of 40m.

As a result, the AV must ensure safety by dynamically choosing between the two detectors based on its speed and the distance to the pedestrian. Specifically, an AV driving at 7m/s requires 7.66m to stop with EDet2 and 11.14m with EDet6, and hence must use EDet2 if the pedestrian walks out 12m away from the AV to be able to stop in time. On the other hand, an AV driving at 17m/s requires 43.43m to stop with EDet2, while it can only detect the pedestrian at a distance of 40m, which requires the AV to use EDet6 to stop safely.

2.2 C2: Environment-dependent runtime

Meeting constantly-evolving deadlines imposed by **C1** is complicated by the impact of the environment on the runtimes of AV components. For example, the number of agents (i.e. vehicles or pedestrians) in the scene affects the runtime of the perception module. Quantifying this impact, Fig. 2b plots how increasing the number of agents changes the runtimes of several object trackers, which are critical components of the perception module that track the trajectories of detected objects. To obtain these results, we drive an AV in the CARLA simulator [52] while increasing the number of agents, and observe an increase in the median runtime for all object trackers. Note that while SORT [34] provides a lower runtime, both DeepSORT [99] and DaSiamRPN [102] offer high accuracy.

In addition, the runtime of the prediction module depends on the velocity of the AV itself. An AV driving at a greater speed requires a higher prediction horizon i.e. it must be able to forecast the trajectories of other agents for longer

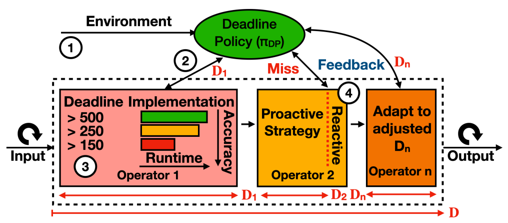


Figure 4. D3 Model structures an application as an operator graph with a policy π_{DP} that decides the deadline $\mathcal{D}$ as per the environment (1), and assigns a $\mathcal{D}_i$ to each operator (2). The operators *proactively* try to meet $\mathcal{D}_i$ (3). However, if $\mathcal{D}_i$ is missed, D3 executes reactive measures (4), and adjusts downstream $\mathcal{D}_i$s using a *feedback* loop.

into the future in order to ensure safety of the vehicle. Many prediction approaches (e.g., MFP [89] and R2P2-MA [79]) use recurrent neural networks, which have a linear runtime dependence on the prediction horizon as shown in Fig. 2c.

The compounding of the runtime variability of individual components leads to a large skew between the mean and the maximum response time of the AV pipeline, which renders worst-case execution time analysis inefficient [25, 87]. To demonstrate this, Fig. 3 analyzes sensor data from Baidu's Apollo AV [31] that drove over 108,000 miles [84, 96]. Specifically, we focus on the traffic light detector [2], a key part of the perception module, that relies on the map and the vehicle's location to choose between multiple cameras in order to obtain bounding box proposals, which are individually refined and classified by multiple neural networks. We find that the response time of the traffic light detector depends on both the choice of the camera and the number of lights in the environment. As a result, the p99 response time latency of perception is 3.3× higher than the mean, which further increases the response time of the downstream prediction component. Moreover, an increase in the response time keeps resources busy, thus forcing the pipeline to drop sensor messages.

3 D3: *D*ynamic *D*eadline-*D*riven Execution

The execution of applications that interact with a continuously-evolving environment (e.g. robots, AVs) requires a careful orchestration of their components. To enable such applications to optimize their accuracy under dynamically-varying deadlines (**C1**), we propose D3, an execution model that centralizes the management of deadlines. D3 models deadlines missed due to runtime variability (**C2**) as exceptions, and enables components to reactively adjust their computation.

D3 structures its application as a directed operator graph along with a *deadline policy* π_{DP} (see Fig. 4). π_{DP} receives the environment's state (e.g., distance to obstacles) and computes an end-to-end deadline $\mathcal{D}$ that ensures safety and prevents unnecessary emergency maneuvers i.e., $\mathcal{D}$ bounds the wall-clock time that can elapse between an input to the graph and its corresponding output (Step 1). Further, π_{DP} splits $\mathcal{D}$ across operators and assigns a per-operator deadline $\mathcal{D}_i$ which aims to maximize the runtime-accuracy tradeoff based on the accuracy and pre-computed runtime profiles (Step 2).

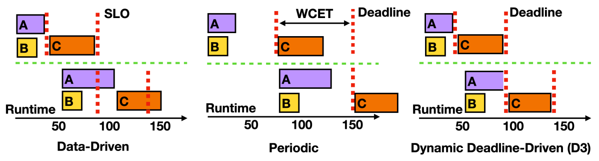


Figure 5. Timeline of execution models when C executes upon receipt of input from A and B. Data-driven models do not enforce deadlines and delay C's execution until both inputs are available. Periodic models use WCET to execute components at a fixed interval that is unable to adjust to slacks or delays, and fails to maximize the runtime-accuracy tradeoff. D3 achieves this by enabling components to either adjust to a constrained deadline or wait for delayed inputs.

D3 expects operators to meet their allocated deadline $\mathcal{D}_i$ by using *proactive strategies* (e.g., running faster models under reduced deadline allocations; Step 3 of Operator 1 in Fig. 4). D3 models $\mathcal{D}_i$s missed due to runtime variability (**C2**) as exceptions and notifies operators to undertake reactive measures to quickly release output (e.g., quickly amending and releasing previous results; Step 4). D3 also notifies the downstream operators of the missed upstream deadline, allowing them to either: (*i*) eagerly execute with incomplete inputs due to a lack of output from the upstream operator that missed its $\mathcal{D}_i$, or (*ii*) reason about the reduction in their available time once the upstream operator's reactive measures release output and modify their computation accordingly.

D3 conveys the occurrence of the missed deadline events to π_{DP} using a *feedback loop*. Upon notification, π_{DP} may adjust the deadline for both downstream operators and future executions of the application. In extreme cases where the application is unable to perform its intended function due to multiple missed deadlines, π_{DP} can choose to execute a safety backup mode that performs simple maneuvers (e.g., braking or pulling over) to ensure a minimal risk condition [82].

3.1 Related Execution Models

We now highlight D3's ability to maximize accuracy under dynamic deadlines by comparing it to two key bodies of work: data-driven execution models and periodic execution models.

Data-driven execution models employed by Service Level Objective-based (SLO) robotics platforms (e.g., ROS [41, 77], Cyber RT [30]) trigger computation on the arrival of input data and hence preclude the initiation of downstream computation in the absence of inputs due to a missed upstream deadline (**C1**). Moreover, such platforms execute computation on a best-effort basis and lack mechanisms to reason about changes in deadlines or variability in runtime [95]. As a result, components are unable to adjust their execution to varying deadlines leading them to miss their $\mathcal{D}_i$ in the presence of runtime variability (**C2**). For example, the lack of mechanisms to reason about a changed deadline coupled with the runtime variability in the second execution of A in Fig. 5 leads to a missed deadline under a data-driven execution model. Further, since downstream components can only trigger computation upon receiving of the input from upstream components, the effects cascade and the second execution of C misses its deadline. Conversely, D3's π_{DP} notifies A of the change in deadline allowing it to proactively modify its computation to meet the new deadline, or reactively release output quickly in case it is missed. D3 also enables C to execute its computation without the input from A, or wait until the input is available and modify its computation to fit within the reduced time.

Periodic execution models, which underpin hard real-time systems [56, 68], use conservative worst-case execution time (WCET) estimates to execute computation at a fixed, periodic interval. While this approach precludes a lack of input from upstream components or a reduction in computation time due to **C2**, it fails to maximize the runtime-accuracy trade-off due to the large skew between the mean and maximum runtime of computation in an AV pipeline (see §2.2). For example, the WCET-driven periodic execution of C in Fig. 5 leaves plenty of slack in the average case, which could be used to execute a higher-accuracy computation. Further, the inflexibility of these models has led to applications adjusting their computation to meet environment-dependent deadlines (**C1**) by defining a fixed set of *mode changes*. However, executing the various modes requires the components to either transition to SLO-based execution [35, 38, 63], or undergo a time-consuming schedulability analysis for each possible deadline and mode transition [32, 33, 36, 78]. By contrast, D3's proactive strategies and reactive measures enable the computation to forego this expensive analysis, and still adjust itself to meet dynamic deadlines (**C1**). We emphasize that D3's execution model subsumes such coarse-grained mode changes by allowing the deadline policy π_{DP} to perform mode changes on either deadline misses or specific environment conditions (e.g., change in the vehicle speed) (see §5.2).

4 Introduction to ERDOS

This section provides an overview of ERDOS, a proof-of-concept instantiation of D3, that builds atop prior work in streaming systems [6, 39, 74, 101]. We first discuss the relevant concepts from streaming systems (§4.1), followed by the structure of an ERDOS application (§4.2). §4.3 exemplifies these concepts through the implementation of a `Planner`.

4.1 Primer on Streaming Systems

Streaming systems (e.g. Cloud Dataflow [6], Flink [39]) structure applications as a directed graph of computational operators, which contains a set of source operators that generate the input and a set of sink operators that consume the output. The sources annotate the inputs with a timestamp derived from an ordered time domain, and notify their downstream operators when they have finished sending all the input for a given timestamp. These messages and notifications cascade along the directed edges of the graph, with each operator potentially transforming its input messages M_t timestamped with t and received along an edge e before sending them along e'.

Further, the operators are notified of the receipt of all messages with time $t' \leq t$ using a watermark message W_t. A watermark [23, 83, 93] informs the operators of the availability of all the inputs required for a computation across all its edges, and thus ensures accurate computation upon synchronized data [22]. While the computation registered with a watermark notification is executed sequentially according to the the timestamp order, the computation that acts on messages is allowed to execute out-of-order, which allows the operators to prevent stragglers while ensuring correctness [22].

4.2 Computation Structure of an ERDOS Application

ERDOS instantiates D3 by modeling the application as a directed graph composed of multiple subgraphs representing the modules (e.g., perception), with each module containing operators representing the components (e.g., lane detection) connected by typed streams. A source of the graph reads data from a sensor and uses an output `WriteStream` to inject it into the graph, while a sink extracts data from the graph using an input `ReadStream`, and sends commands to the vehicle.

Each operator must implement an interface that specifies both the number and types of its input and output streams. This static registration of the input and output allows the system to ensure that the computation graph is well-formed at compile-time, and reduces the runtime errors. Moreover, the static registration enables the system to optimize the allocation of operators to hardware (e.g., colocate operators).

The typed streams allow communication through timestamped messages i.e. a stream s of type $\mathcal{T}$ can carry: (*i*) a `DataMessage` (M_t), with a payload of type $\mathcal{T}$ and a timestamp t, and (*ii*) a `WatermarkMessage` (W_t), with a timestamp t that represents the completion of all incoming messages for $t' \leq t$. Corresponding to the type of message and the input stream, the interface implemented by each operator defines the callbacks that are invoked by ERDOS (see §4.3).

The timestamp t generated by the source operators consists of $t = \left(l \in \mathbb{N}, \hat{c} : \langle c_1, \ldots, c_k \rangle \in \mathbb{N}^k\right)$, where l represents a logical time (see §5.1), and $\hat{c}$ conveys application-specific information (elaborated in §5.3). This abstraction enables applications to seamlessly work across both real-world and simulation, by using l to represent the wall-clock time in real AVs, and simulation time when using a simulator, the latter of which may advance at a different rate than real-time. While a simulator provides a consistent notion of time, ERDOS exploits the presence of a local high-speed network in real AVs to precisely synchronize clocks in order to correctly reason about the wall-clock time across multiple machines [54, 57].

4.3 ERDOS' API

We now provide an overview of the API with the help of a simplified `Planner` (see Lst. 1) that receives the `Obstacles` and `TrafficLights` from perception through `DataMessages`. It then computes a motion plan and returns a set of `Waypoints` i.e., fine-grained points on the road that

```
 1 impl TwoInOneOut<Obstacles, TrafficLights, State<PlanningState>, Waypoints>:
 2   fn setup(objects: ReadStream<Obstacles>, lights: ReadStream<TrafficLights>,
 3            plan: WriteStream<Waypoints>, deadlines: ReadStream<Deadline>) {
 4     // Call `on_watermark` even in the absence of traffic lights.
 5     FrequencyDeadline::new(PlanningOp::on_watermark)
 6         .with_static_deadline(30).on_stream(lights);
 7     // Constrains the completion of local computation.
 8     TimestampDeadline::new(PlanningOp::on_deadline)
 9         .with_end_condition( // and a default start condition
10           |sent_msg_cnt: usize, watermark_status: bool| sent_msg_cnt > 0)
11         .with_dynamic_deadline(deadlines).on_stream(plan);
12   }
13   fn on_left_msg(ctx: Context, objects: Message<Obstacles>,
14         state: State<PlanningState>) {
15     // Change coordinate system of objects and add to state.
16   }
17   fn on_right_msg(..., lights: Message<TrafficLights>, ...) {...}
18   fn on_watermark(ctx: Context, state: State<PlanningState>,
19         plan: WriteStream<Waypoints>) {
20     // Computes a plan upon receiving obstacles and traffic lights.
21   }
22   fn on_deadline(ctx: Context, state: State<PlanningState>,
23         plan: WriteStream<Waypoints) {
24     // Invoked when a deadline is missed.
25   }
```

Listing 1. **`Planner`** computes a trajectory using the `Obstacles` and `TrafficLights`, and specifies deadlines on its response time.

characterize the trajectory of the AV. To register its input and output, the `Planner` implements the `TwoInOneOut` interface where the `ReadStream`s are typed by `Obstacles` and `TrafficLights`, and the `WriteStream` by `Waypoints`.

Further, to invoke the computation, the `Planner` implements the `on_msg` callbacks (lines 13-17) that convert the coordinate system of each `Obstacle` and `TrafficLight`, a task that can be executed out-of-order for each timestamp. However, in order to compute a safe plan, the `Planner` requires a synchronized and complete set of all the obstacles and traffic lights from perception, and hence, waits for a `WatermarkMessage` from both the upstream operators signifying the receipt of all incoming messages for each timestamp. It then uses the converted obstacles and lights to compute the `Waypoints` for the AV in `on_watermark` (line 18).

Moreover, the `Planner` must complete its computation within a deadline, and thus restricts its runtime from the time of the receipt of the input to a dynamically-varying deadline retrieved from the `deadline_stream` provided by the deadline policy π_{DP} (line 11). §5.1 and §5.2 further elaborate on the specification and dynamic-variation of deadlines. ERDOS automatically exposes the deadline for the timestamp computed by π_{DP} to each of the callbacks via the `Context`, allowing the operators to employ proactive strategies to meet deadlines and vary their computation accordingly (see §5.3).

However, to meet its deadline in the presence of a delay of more than 30ms in the receipt of the `TrafficLights`, the `Planner` chooses to eagerly initiate the computation with partial input, and computes the plan using just the obstacles (line 6). Finally, the deadline specification also requires an exception handler that invokes reactive measures to quickly releases output upon a missed deadline (`on_deadline`). The handler receives a `Context` containing information useful to mitigate the deadline miss (e.g., timestamp, deadline) along with the state of the operator, and can be used to output the previous computed plan offset from the AV's current location.

5 Achieving Dynamic End-to-End Deadlines

We now discuss ERDOS' core contributions that enable it to address the following challenges posed by D3's realization:

- ERDOS must initiate computation upon the availability of all required input, and allow components to bound their response time from that event. In addition, components must be allowed to initiate computation in the presence of partial input if upstream components miss their deadline $\mathcal{D}_i$ (§5.1).
- ERDOS must allow the deadline policy π_{DP} to meet strict safety, adaptivity and modularity constraints, owing to its critical effects on the latency-sensitive computation (§5.2).
- ERDOS must provide efficient mechanisms to enable components to utilize different strategies to *proactively* output the highest-accuracy results possible within $\mathcal{D}_i$ (§5.3).
- In case of a missed $\mathcal{D}_i$, ERDOS must enable execution of *reactive measures* that quickly release output, and allow downstream computation to begin. (§5.4).

5.1 Deadline Specification

In an effort to meet both its individual deadline $\mathcal{D}_i$, and the end-to-end deadline $\mathcal{D}$, each component of a D3 application must be able to: (*i*) bound the execution time from the receipt of the inputs and the generation of the corresponding output (**C1**), and (*ii*) bound the time between the invocation of the computation on inputs of successive timestamps in the presence of runtime variability in upstream components (**C2**). While (*i*) ensures that a component adheres to its allocated deadline $\mathcal{D}_i$ for time t, (*ii*) allows components to eagerly initiate their computation on incomplete input for time $t' > t$ to ensure that the end-to-end deadline $\mathcal{D}$ is met for time t' in case the upstream components miss their $\mathcal{D}_i$ for t'.

To achieve these goals, ERDOS must track the initiation and completion of computation for every time t. ERDOS accomplishes this by automatically capturing fine-grained execution events from the components at the granularity of the logical time l. Specifically, for each logical time l of the timestamp t, ERDOS maintains counters of the number of incoming and outgoing messages annotated with t (M_t), and boolean variables indicating the receipt and generation of the watermark for t (W_t) across all the input streams for each component. This allows ERDOS to automatically initiate computation once all required inputs are available (denoted by the receipt of a W_t across all the input streams) (*i*), and register the completion of the computation for the logical time l once the watermark W_t for t is sent on the output streams.

In order to enable flexibility in the events that are constrained by a deadline, ERDOS exposes these events to the components, and allows them to specify relative deadlines, which limit the amount of wall-clock time that can elapse between any two events. Specifically, components can register two boolean functions: *deadline start condition* (D_{SC}) and *deadline end condition* (D_{EC}), which return `True` to signify the initiation and completion of the computation for a

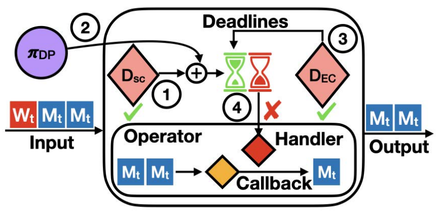


Figure 6. Environment-dependent deadlines. ERDOS evaluates D_{SC} for every message (1). If satisfied, it initiates an absolute deadline according to π_{DP} (2). Similarly, ERDOS evaluates D_{EC} upon generation of messages, and removes any satisfied deadlines (3). If a deadline is missed, ERDOS invokes an exception handler (4).

logical time l respectively. D_{SC} and D_{EC} are evaluated at the receipt and generation of every message, and receive a tuple $\big(n \in \mathbb{N}, w \in \{\texttt{True}, \texttt{False}\}\big)$, where n denotes the number of messages received or sent for that logical time, and w depicts the receipt or generation of the watermark. ERDOS maps the relative deadline $\mathcal{D}_i$ to an absolute deadline by automatically capturing the wall-clock time at which D_{SC} is satisfied (Step 1 in Fig. 6), and offsetting it by $\mathcal{D}_i$ (Step 2 in Fig. 6). ERDOS then tracks the passage of wall-clock time and ensures that D_{EC} is satisfied before the absolute deadline (Step 3 in Fig. 6).

Further, to simplify the specification of the relative deadline $\mathcal{D}_i$ for the enforcement of the response time deadlines (*i*) and (*ii*), ERDOS provides the following two general deadline abstractions that constrain a default set of events:

Timestamp deadlines (lines 7-11 in Lst. 1) bound the execution time. Components define a relative deadline $\mathcal{D}_i$ that constrains the wall-clock time elapsed between a default D_{SC} that specifies the receipt of the first message timestamped with t (M_t), and a default D_{EC} that specifies the output of the first watermark timestamped with $t' \geq t$ ($W_{t'}$). If D_{EC} is not satisfied before $\mathcal{D}_i$ expires, ERDOS invokes the exception handler (`on_deadline` on line 22 in Lst. 1), which quickly releases output to initiate downstream computation (see §5.4).

Frequency deadlines (lines 4-6 in Lst. 1) allow a precise invocation of the computation in the presence of runtime variability. To achieve this, components define a relative deadline $\mathcal{D}_i$ that constrains the maximum wall-clock time that may elapse between a default D_{SC} that specifies the receipt of the watermark timestamped with t (W_t), and a default D_{EC} that specifies the receipt of the first watermark for $t' > t$ ($W_{t'}$). If D_{EC} is not satisfied for t' before $\mathcal{D}_i$ expires, ERDOS automatically inserts $W_{t'}$ on the given input stream to simulate the arrival of all incoming data for t', and invokes the computation with the partial input (see §5.3). For example, if $W_{t'}$ does not arrive on the `lights` stream within 30ms of the receipt of W_t (as specified on line 6 in Lst. 1), ERDOS automatically inserts $W_{t'}$ and invokes `on_watermark` with partial input.

We emphasize that the ability to tailor the above general abstractions using the fine-grained execution events exposed by ERDOS, enables components to specify the full spectrum of deadline constraints discussed in prior work [47]. To exemplify this ability, the `Planner` in Lst. 1 uses this control to tailor the `TimestampDeadline` constraint with a custom D_{EC} (lines 9-10 in Lst. 1) that is satisfied as soon as the first message for a timestamp t is output. Coupled with the default D_{SC}, this constraint allows the `Planner` to bound the time between the receipt and generation of the first message timestamped with t (M_t). This deadline can be used by the `Planner` to quickly release a coarse-grained plan before refining it, thus enabling downstream computation to begin.

5.2 Environment-Dependent Deadlines

Components may use the abstractions discussed in §5.1 to specify static deadlines that do not evolve over time by using static values for $\mathcal{D}_i$ (e.g., 30ms for the `FrequencyDeadline` on line 6 in Lst. 1). However, D3 requires that these abstractions support dynamic deadlines determined by a deadline policy π_{DP}, which evolve according to the environment (**C1**).

We emphasize that the centralization of π_{DP}, which dynamically determines the end-to-end deadline $\mathcal{D}$ and the individual deadline $\mathcal{D}_i$ for each component, is a novel contribution of the D3 execution model. While the development of such a policy raises interesting research challenges orthogonal to this work, this section concerns itself with the following key systems challenges that its placement on the critical path of the computation presents to the design of ERDOS:

Safety. The presence of π_{DP} on the critical path of affecting what computation runs in each component requires it to meet strict deadline constraints. In addition to being able to initiate a safety backup mode that performs simple maneuvers if multiple component deadlines are missed (see §3), π_{DP} must also ensure safety by executing the backup mode if it misses its own deadline (due to delayed inputs or runtime variability).

Adaptivity. In order to reduce π_{DP}'s effect on the latency of the critical path, ERDOS must allow applications to adapt the frequency at which the deadline allocations are recomputed according to the *dynamicity* of the environment. For example, a π_{DP} may change the allocations less frequently on highways than in cities, owing to the infrequent change in environment.

Modularity. Individual modules (e.g., perception) may exploit expert knowledge to specify policies that split deadlines across their components (e.g., detection, tracking) more efficiently than a centralized policy. Thus, ERDOS must enable the decomposition of monolithic policies such that high-level policies provide coarser-grained deadlines to module-specific policies, which further split them across their components.

To achieve these goals, ERDOS executes π_{DP} as a subgraph of operators which receive information about the environment from components on its input streams. π_{DP} processes this information to compute an end-to-end deadline $\mathcal{D}$ and decomposes into individual deadlines $\mathcal{D}_i$, which are sent to components via its output streams. Specifically, lines 8-11 in Lst. 1 show how an operator can adjust its `TimestampDeadline` according to π_{DP} by registering on the `deadlines` stream provided by ERDOS. π_{DP} utilizes the state of the environment to dynamically compute the relative deadline $\mathcal{D}_i$ for each logical time l, and communicates it to the operator using the `deadlines` stream. ERDOS automatically synchronizes the computation for l with the corresponding $\mathcal{D}_i$ provided by π_{DP}, and utilizes it to compute the absolute deadlines for the computation (see Fig. 6). To enable components to adjust their computation to meet the changing deadlines (§5.3), ERDOS exposes the absolute deadlines via the `Context` (§4.3).

Executing π_{DP} as a subgraph enables the policy to exploit ERDOS' graph abstraction to achieve *modularity* by splitting itself across operators, and benefit from co-location with components that share the state of the environment with them (see §5.4). Moreover, π_{DP} can use ERDOS' timestamping mechanism to achieve *adaptivity*. Specifically, a π_{DP} can send a $\mathcal{D}_i$ in a message M_t followed by a watermark $W_{t'}$, where $t' \geq t$, and adaptively evolve the delta between t' and t according to the environment. $W_{t'}$ signifies the completion of all outputs from π_{DP} until timestamp t', and specifies the relative deadline $\mathcal{D}_i$ for the next timestamps from t to t'. Further, π_{DP} can ensure *safety* by using ERDOS' static deadlines (§5.1) to enforce strict constraints on its execution, and invoke the safety backup mode in case it misses a deadline (see §5.4).

5.3 Meeting Deadlines

ERDOS exposes the deadline $\mathcal{D}_i$ (allocated by π_{DP}) to the operators via the `Context` (lines 13, 18 in Lst. 1). We now discuss some general proactive strategies that operators may use to meet $\mathcal{D}_i$ (by satisfying D_{EC} before it expires) below:
Executing anytime algorithms [60, 98, 100] that maximize the accuracy for a given $\mathcal{D}_i$ through iterative refinement [103], and provide a continuous runtime-accuracy tradeoff curve by monotonically increasing accuracy with increasing deadlines. Such algorithms can be interrupted when $\mathcal{D}_i$ expires and ensure the highest-accuracy results possible within the time. Moreover, components can choose to release lower-accuracy results (before $\mathcal{D}_i$ expires) to downstream operators, allowing them to begin computation early and iteratively refine their results. For example, the `Planner` in Lst. 1 could execute an anytime planning algorithm [60, 98, 100] in its `on_watermark` method. The algorithm would release coarse-grained waypoints and iteratively refine them, to allow the downstream control operator to begin generating commands.
Changing the implementation based on the most accurate algorithm that typically completes within $\mathcal{D}_i$ (e.g., mean or p99 runtime is less than $\mathcal{D}_i$). This is facilitated by the existence of multiple algorithms for the components, that enable a tradeoff between runtime and accuracy [59, 88] (see §2).
Executing multiple versions of components to ensure that at least one completes before $\mathcal{D}_i$ expires (similar to [86]). In addition to choosing the highest-accuracy algorithm that fits within $\mathcal{D}_i$, components can execute faster algorithms that are guaranteed to finish execution before $\mathcal{D}_i$ expires, thus maximizing the runtime-accuracy tradeoff, while still meeting deadlines in the presence of runtime variability (**C1**). For example, a detector can run two callbacks in parallel: (*i*) the most-accurate model that *typically* completes within $\mathcal{D}_i$, and (*ii*) a fast, low-accuracy model, and return results from (*ii*) if (*i*) does not meet $\mathcal{D}_i$. Similar to anytime algorithms, components must be allowed to release the lower-accuracy results to unblock downstream operators, or wait until $\mathcal{D}_i$ expires, and return the highest accuracy results available.
Skipping the execution of an algorithm in case of small $\mathcal{D}_i$. Unlike load shedding [40, 90, 91] that does not generate results, AV components can quickly release reduced-accuracy results to unblock downstream computation by amending prior results. For example, the `Planner` in Lst. 1 can release its last computed plan offset to the current location of the AV.
Eagerly executing with partial input if upstream operators cannot meet their $\mathcal{D}_i$ due to runtime variability (**C2**). While previous strategies require the input to be available and must adjust the computation to a reduced deadline in case of upstream runtime variability (**C2**), this strategy allows components to eschew input from certain upstream components in order to maintain its initially allocated $\mathcal{D}_i$. For example, the `Planner` in Lst. 1 eagerly executes without `TrafficLights`, and plans a trajectory using `Obstacles` if the upstream component experiences a runtime variability of more than 30ms.

ERDOS achieves an effortless and efficient realization of these strategies using two novel mechanisms detailed below:
Intermediate Results. ERDOS' extension of timestamps provides first-class support for anytime algorithms, speculative execution of multiple versions, and enables eager execution with partial input. Specifically, anytime algorithms and different versions can annotate outputs with $t = \left(l, \hat{c}\right)$, and increase the value of $\hat{c}$ to notify downstream computation of the accuracy of the results as they become available (with an increased value of $\hat{c}$ signifying increased accuracy of the results). ERDOS orders the execution of computation using $\hat{c}$ and automatically prioritizes computation on higher-accuracy inputs, thus maximizing the accuracy of results. Similarly, upon expiration of a `FrequencyDeadline`, ERDOS automatically inserts a W_t (with a low value of $\hat{c}$) on the stream that failed to generate the required input within the deadline. The computation then conveys the accuracy of these results to its downstream components, and refines its results when missing inputs from upstream components become available.

For example, in the absence of `TrafficLights` or when using anytime algorithms, the `Planner` can output a coarse-grained plan and annotate its accuracy using $t_1 = \left(l, \hat{c}_1\right)$. This allows the downstream control operator to generate commands using the coarse-grained plan, and refine them after a fine-grained plan is available. If multiple plans are available, ERDOS automatically eschews the control operator's execution with a coarse-grain plan tagged with t_1 in favor of a fine-grained plan tagged with $t_2 = \left(l, \hat{c}_2\right)$, where $\hat{c}_2 > \hat{c}_1$.
Speculative Execution. ERDOS automatically chooses to change the implementation, execute multiple versions or skip the execution based on $\mathcal{D}_i$. To achieve this, it requires components to decouple their state from the implementation of

the computation, and specify multiple implementations along with their runtime profiles. Specifically, a component must register its state (e.g., `PlanningState` on line 1 in Lst. 1) with ERDOS. By assuming control of the state, ERDOS' speculative execution mechanism achieves an efficient execution of different implementations for successive timestamps by automatically providing access to the state to different callbacks (lines 14, 18, 22 in Lst. 1). Further, the mechanism enables the parallel execution of multiple versions by providing each implementation with a view of the state without requiring operators to synchronize updates to the state (see §5.4).

5.4 Handling Deadline Misses

If the strategies discussed in §5.3 fail to meet the allocated deadline $\mathcal{D}_i$, D3 requires components to undertake reactive measures, whose execution presents the following challenges:
Fast Invocation of the measures upon expiration of $\mathcal{D}_i$ so as to quickly unblock downstream computation and minimize the reduction of available time for downstream operators.
Access to the state of the partially-executed proactive strategies to enable the measures to quickly release results, and ensure its correctness for executions of future timestamps.
Parallel execution of the measures and the proactive strategies to enable components to quickly unblock downstream computation with lower-accuracy results while using higher-accuracy computation for state updates. For example, if the `Planner` misses $\mathcal{D}_i$ while running a higher-accuracy algorithm, it can release the last computed plan offset to the current location of the AV as its reactive measure, while updating its state with the higher-accuracy plan for future executions.

To address these challenges, ERDOS enables components to accompany the specification of timestamp deadlines (from §5.1) with *deadline exception handlers* (D_{EH}) (lines 22-25), which execute the reactive measures if $\mathcal{D}_i$ is not met. ERDOS orchestrates the execution of the specified D_{EH} alongside the proactive strategies through the following execution policies:
Abort. Terminates the execution of the proactive strategy for time t, and requires D_{EH} to notify the computation's completion by sending a watermark W_t and ensure the correctness of state for t. To achieve this, D_{EH} receives a view of the state for $t' < t$ along with the *dirty state* for t (i.e., mutations made to the state by the partially-executed proactive strategy for t). D_{EH} uses these views to quickly release output and amend the dirty state to ensure its correctness. For example, a D_{EH} in a `Planner` could either use the dirty state at t to output and save the best plan found by the deadline if the operator is anytime, or amend the plan computed for t' otherwise.
Continue. Executes the proactive strategy for t in parallel with the D_{EH}. The latter unblocks downstream computation by releasing output for t (M_t), while the former notifies the computation's completion (W_t) and ensures state consistency. To achieve this, D_{EH} receives a view of the state for $t' < t$, and executes a fast algorithm to quickly release output. In parallel, the proactive strategy continues releasing output

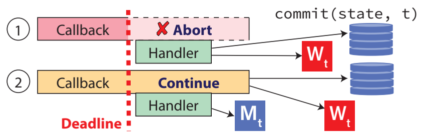


Figure 7. Handling missed deadlines. When a deadline is missed, handlers are invoked to mitigate the consequences. Callbacks which miss their deadline may **Abort** to let the handler rapidly update operator state, or **Continue** to ensure more accurate state updates.

for t, thus providing the downstream computation a choice of more accurate results. Moreover, allowing the proactive strategy to save the state for t enables the computation for $t'' > t$ to use the high-accuracy results, and prevents a cascade of low-accuracy results across time. For example, a D_{EH} in a `Planner` can amend the plan computed for t', while the proactive strategy releases and saves a more-accurate plan.

A seamless execution of D_{EH} under the *Abort* and *Continue* policies requires a careful management of the component's state in order to ensure its consistency. To aid the components in this endeavor, ERDOS provides **system-managed state.** Specifically, ERDOS assumes control over the state of the components decoupled from their implementation (see *Speculative Execution* in §5.3), and enables the state to meet the challenges of executing D_{EH} discussed earlier by ensuring the following two key properties over it:
Transactional Semantics. In order to ensure a fast invocation and parallel execution of π_{DP}, D_{EH} and multiple versions of the computation, ERDOS must provide them with a view of the state and ensure that it is saved from either the D_{EH} or the proactive strategy according to the execution policy. ERDOS achieves this by enforcing transactional semantics on the state at the granularity of a timestamp, and provides the proactive strategy with a view of the last committed state, and automatically commits any mutations made by them upon the successful release of the watermark for the currently executing timestamp. In case of a missed $\mathcal{D}_i$, ERDOS invokes D_{EH} and shares the dirty state along with a view of the last committed state, and automatically commits the changes made to the dirty state by D_{EH} upon its completion.
Time-Versioning. To further ensure the execution of π_{DP}, D_{EH} and proactive strategies across multiple timestamps, ERDOS maintains a version of the state for each timestamp t. For example, multiple executions of the `Planner` for different timestamps (each corresponding to a different set of `Obstacles` and `TrafficLights`) can be executed in parallel with their computed plans being saved in different versions. In case a deadline is missed for t, the D_{EH} gets access to the *committed* state for all timestamps $t' < t$ and can send M_t to unblock downstream computation. Meanwhile, the proactive strategies can continue in parallel for timestamps $t'' \geq t$ and commit state mutations by releasing $W_{t''}$.

Moreover, while ERDOS provides a default `State` implementation with the properties discussed above, it allows components to provide their own states. These states implement an interface that customizes both transactional semantics (through `commit`) and time-versioning (through `get_committed` to retrieve a view of the state at t), and may use techniques such as CRDTs [81]. For example, the `Planner` could implement the interface for `PlanningState`, instead of using the `State` provided by ERDOS (as shown in line 1 of Lst. 1). In such a case, the `PlanningState` could maintain a vector of waypoints for timestamp $t = 0$, and log additions of future waypoints in `commit`, instead of saving the entire set of waypoints for each timestamp $t' > t$.

6 ERDOS' Implementation

ERDOS is an open-source distributed system implemented in $\sim$ 13k lines of Rust, whose type safety and memory semantics are essential for safety-critical applications. Further, to interact with ML frameworks [18] and enable prototyping with simulators [52], ERDOS provides a Python interface.

ERDOS' distributed nature is enabled by a leader-worker architecture where the leader manages a set of worker processes running across several machines. The leader partitions the operator graph and schedules operators to workers, which are responsible for exchanging data along streams (§6.1), executing callbacks (§6.2), managing deadlines and executing their exception handlers (§6.3). We choose the leader-worker architecture due to its implementation simplicity, and ensure its scalability by keeping the leader off the critical path.

6.1 Communication

ERDOS initializes itself by constructing a control plane between the leader and the workers, which is used by the leader to schedule operators to workers and synchronize their initialization, thus ensuring that all operators are ready to execute before transmitting any messages. The workers construct a data plane amongst themselves atop TCP sessions, which is used to communicate the messages sent between the operators. This allows ERDOS to keep the leader off the critical path, while still enabling centralized scheduling decisions.

ERDOS provides a rapid communication of messages by choosing the underlying communication channel based on whether it connects operators: (*i*) on the same worker, or (*ii*) on different workers. While the communication for (*ii*) is multiplexed atop the data plane among the workers, operators on the same worker store data on the heap and communicate a reference to it over Rust's inter-thread channels, enabling rapid delivery of large messages and safe zero-copy communication using Rust's compile-time mutability checks.

6.2 Operator Execution

Workers execute computation by maintaining an execution lattice, a dependency graph of callbacks which guarantees the processing of message and watermark callbacks in timestamp order, thus providing lock-free access to state. Upon receiving a message, a worker retrieves a view of the state using `get_committed` (§5.4), and inserts into the lattice a *bound callback*, consisting of the state, the `Context`, the callback, and the received message. Similarly, upon the receipt of a watermark, the worker verifies if it acts as a low watermark across the operator's input streams, and inserts a callback that commits the state upon completion by invoking `commit`.

This execution lattice serves as a run queue for a worker's multi-threaded runtime. A set of threads retrieve and execute the callbacks, and notify the lattice upon their completion to unlock further dependencies (e.g., callbacks with higher timestamps). ERDOS allows operators to override the ordering semantics of the lattice to fine-tune the parallelism and state-management. For example, an operator may manually synchronize updates to its state, and ask ERDOS to execute all its callbacks in parallel by specifying that all timestamps are equivalent, and thus ready to execute concurrently.

6.3 Deadline Management

The worker also ensures that the deadlines are initialized, and execute the exception handlers in case they are missed. To initialize a deadline, workers maintain per-stream statistics on the receipt of messages and the watermark status for each time t. Upon receipt of a message (M_t) or watermark (W_t), the worker updates the statistics, and invokes D_{SC}. If satisfied, the worker synchronizes the relative deadline $\mathcal{D}_i$ for t sent by the `deadline_stream` (line 3 in Lst. 1), and computes the absolute wall-clock time at which it expires. The deadlines, along with their handlers (D_{EH}), are maintained by the worker in a priority queue ordered by the absolute deadline.

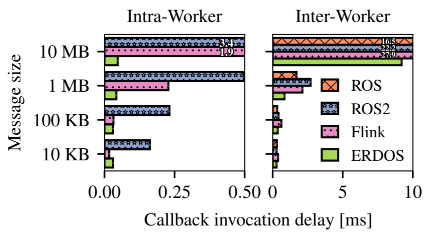
A deadline is removed from the queue when the operator satisfies D_{EC} or misses the deadline. Workers maintain per-stream statistics of the transmission of messages and watermarks, and remove the deadline and the D_{EH} from the queue upon satisfaction of D_{EC}. Further, workers poll the queue, and invoke the D_{EH} according to either the *abort* [37] or *continue* policy upon the expiration of a deadline.

7 Evaluation

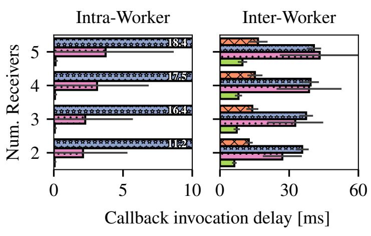
Open-source AV pipelines (e.g., Autoware [28], Apollo [29]) do not include models and lack feature-complete integration with realistic open-source simulators, which are required to measure the efficacy of D3. Thus, we developed Pylot, a state-of-the-art AV that achieves a competitive score on the map track of a simulated AV challenge and drives real AVs. We use Pylot to evaluate D3 and ERDOS, and seek to answer:

1. How does ERDOS compare with other systems? (§7.2)
2. Does ERDOS enable the fulfillment of deadlines? (§7.3)
3. Do D3's dynamic deadlines improve safety? (§7.4)

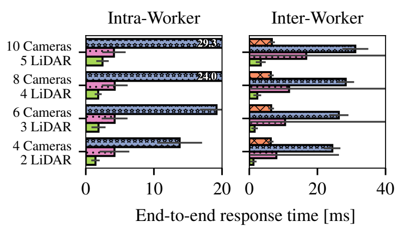
Experimental Setup. We perform all our experiments on a machine having 2× Xeon Gold 6226 CPUs, 128GB of RAM, and 2× Titan-RTX GPUs, running Linux Kernel 5.3.0. This configuration closely reflects the hardware used in our AVs.



(a) Message Delay



(b) Operator Fanout Delay



(c) Sensor Scaling

Figure 8. Messaging Performance. We evaluate the response time for **(a)** varying message sizes, **(b)** operator fanout, and **(c)** pipeline sizes for intra-worker and inter-worker communication. In all cases, we find that ERDOS' optimized implementation results in better performance.

7.1 Pylot: an AV Development Platform

The construction of Pylot was a multi-year effort leading to approximately 28k lines of code, with an additional 434 lines required to port it to a real AV[2]. Pylot contains dozens of components and is, to the best of our knowledge, the most comprehensive open-source AV pipeline with trained models. We now briefly describe a few components relevant to our evaluation (see [55] for an extended discussion).

Pylot's perception module comprises of components that perceive objects, lanes, and traffic lights using multiple cameras. While Pylot provides several implementations for each component (suited for different driving environments), our experiments use EDet2 to EDet6 from the EfficientDet family [88] in the order of increasing accuracy and runtime. This enables us to experimentally evaluate the runtime-accuracy tradeoff as accuracy varies from 39.6 mAP (EDet2) to 51.7 mAP (EDet6), and the runtime varies from 20ms to 262ms.

The Pylot planning component contains implementations of Hybrid A* [51], RRT* [60] and FOT [98, 100] that perform best under different driving scenarios [62, 64, 76]. Since we execute our experiments in an urban environment, we utilize the FOT planner that discretizes the configuration space, and is fast if coarse discretizations are chosen, with poor discretizations producing infeasible plans. We create configurations of the planner by varying the space discretization from 0.3m to 0.7m, and evaluate them in Fig. 2d, which plots the lateral jerk while performing a swerving maneuver. We observe that configurations with longer deadlines, and lower space and time discretization provide increased comfort.

Methodology. The tight coupling between the existing open-source AV pipelines and their underlying systems (e.g., Autoware and ROS [77], Apollo and CyberRT [30]) makes porting these pipelines to ERDOS a time-consuming engineering effort. Similarly, migrating Pylot to the underlying systems used by these pipelines is a challenging undertaking. As a result, our evaluation follows a two-pronged approach. First, we measure low-level system metrics (e.g. callback invocation delay) to show a lack of regression in ERDOS' realization of D3 as compared to the other systems (§7.2). Second, we extend the CARLA challenge [13] to construct a challenging benchmark for AV systems spanning 50km of simulated driving. We port the execution models used by the underlying systems to ERDOS and use the benchmark to highlight the efficacy of D3 when compared to these models (apart from any engineering benefits that come from ERDOS) (§7.3, §7.4).

7.2 ERDOS' Performance vs. Other Systems

We evaluate the latency of ERDOS with respect to message size, operator fanout, and pipeline complexity. We compare against (*i*) ROS, a widely used platform for AVs [20, 28, 53, 95], (*ii*) ROS2, which provides more real-time guarantees [50, 71], and (*iii*) Flink [39] a data-driven streaming system that is closest to ERDOS due to its operator-centric programming model and usage of watermarks for unlocking computation.

Microbenchmarks. We measure the delay incurred by sending messages of increasing sizes between two operators and invoking a callback upon receipt of the message. By measuring the callback invocation delay, we compare how different systems contribute to AV pipeline's response time via the implementation of the communication stack and the scheduling of callbacks. We send messages at 30Hz, the frequency at which AVs process data [1]. Fig. 8a shows the results across both intra-worker and inter-worker placements of the operators. ERDOS' intra-worker callback invocation delay remains constant across message sizes due to its zero-copy communication. Further, ERDOS' inter-worker implementation performs 2.0× better than ROS, and 3.2× better than ROS2, and 2.5× better than Flink when sending 1MB messages. We analyze the systems to attribute the overhead, and find that Flink and ROS have additional data copies and a more inefficient networking path accounting for 80% of the overhead, and slower serialization/deserialization responsible for 20% of the overhead. Moreover, we attribute ROS2's overhead to its use of the Data Distribution Service, which incurs additional costs for data conversion [71].

Next, we compare ERDOS' callback invocation delay to other systems' when broadcasting the output of an operator (e.g., camera image) to multiple operators (e.g., perception components), which is a common pattern in AVs. Fig. 8b shows that ERDOS sends a typical camera image message of 6MB to 5 operators in the same worker at a median latency of 0.12ms, 150× faster than ROS2 and 30× faster than Flink. When communicating across workers, ERDOS broadcasts

[2]A demo of one of our test drives: https://tinyurl.com/yaumb4sn

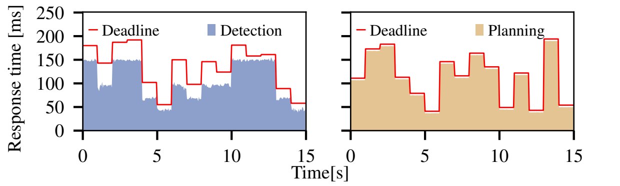


Figure 9. Meeting Deadlines. We vary the deadline every second and show how the modules respond to the new deadlines. Both the detection and planning modules adapt to meet the deadline and the more adaptive planning module is better at using its time allotment.

to 5 operators at a median delay of 9.76ms, which is 1.7×, 4.2×, and 4.4× faster than ROS, ROS2, and Flink.

Response Time Benchmarks on Synthetic Pipelines. In order to measure the scalability to complex AV pipelines [7, 9, 95], we emulate Pylot with an increasing number of sensors sending data at 30Hz. We first instrument Pylot, and retrieve the mean size of each message type. Based on these measurements, we emulate a pipeline with an increasing number of sensors and operators, which sends messages totalling 925MB/s when processing 10 cameras and 5 LiDARs across 75 operators. Moreover, for a worst-case estimate of system overheads, we assume each operator has a 0ms runtime.

Fig. 8c compares the end-to-end response time of the pipeline when executed within a worker and across workers. We find that for 10 cameras and 5 LiDARs, ERDOS' intra-worker implementation exhibits a median response time of 2.5ms, which is 12× and 1.7× better than ROS2 and Flink. When placing each operator in its own worker, ERDOS exhibits a median response time of 3.4ms, which is 2.0×, 9.3×, and 5.0× faster than ROS, ROS2, and Flink. Note that a realistic deployment of Pylot would colocate operators in workers, and thus the worst-case latency would be similar to that observed in the intra-worker graph.

Takeaway: *ERDOS' efficient implementation scales to large pipelines and enables AVs to meet more deadlines by minimizing the amount of time lost to system overheads when invoking computation due to message arrivals.*

7.3 Efficacy of ERDOS' Deadline Mechanisms

We evaluate the latency overhead introduced by the mechanism for implementing π_{DP} policies, the ability of components to proactively meet dynamic deadlines, and the effect of reactive measures in meeting end-to-end deadlines.

Latency Added by the Policy Mechanism. To achieve dynamic deadlines, applications define π_{DP}, which computes deadlines using pipeline data and sends deadlines to components (§5.2). In order to isolate the latency of the mechanism from the latency of the π_{DP} logic, we use a *no-operation* π_{DP} that receives data from Pylot's components and sends static deadlines to the components. We measure Pylot's response time without and with the no-operation π_{DP} during a 35km drive, and we find that the policy mechanism increases the response time by less than 1%. The median and 90th percentile response times increase by 0.9ms and 2.3ms respectively.

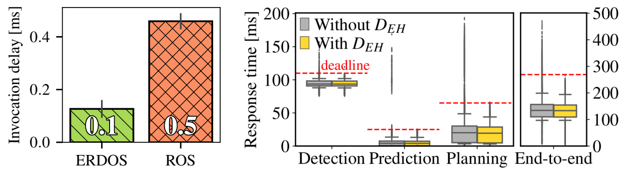


Figure 10. Impact of Deadline Exception Handlers. ERDOS supports fast invocation of handlers (left), and enables Pylot to quickly react to deadline misses (right), ensuring timely responses.

Meeting Deadlines. We evaluate ERDOS' support for fine-grained changes in deadline allocations (§5.3) and if Pylot's components adapt to meet these changes. In the experiment, we use a policy that randomly changes deadline allocations every second. Fig. 9 shows the response time of Pylot's detection and planning during a short drive. We observe that while detection meets its deadline, it fails to utilize its entire time allocation. This is because the EfficientDet [88] family provides 8 models with different runtimes, and ERDOS chooses the model with the highest runtime that fits within the allocated deadline, which may be significantly higher. By contrast, the planning component fully utilizes its time allocation because it executes an anytime algorithm [98, 100].

Handling Deadline Misses. Deadline exception handlers (D_{EH}) ensure that a missed deadline does not delay downstream components (§5.4). In this experiment, we compare against a D_{EH} implementation based on ROS' actionlib, a preemptible task library. Fig. 10 (left) shows that ERDOS invokes D_{EH} 0.1ms after a deadline is missed, and it is 5× faster than ROS. This delay is acceptable for Pylot, as Fig. 10 (right) shows the per-component and end-to-end response time without D_{EH} (i.e., the data-driven execution model described in §3.1) and with D_{EH} during a 50km drive in simulation. Pylot without D_{EH} has a 0.6% end-to-end deadline miss ratio, whereas with D_{EH} it always meets the end-to-end deadline.

Takeaway: *ERDOS implements D3 by swiftly executing π_{DP}, enabling proactive strategies to meet deadlines, and rapidly taking reactive measures when deadlines are missed.*

7.4 Efficacy of the D3 Execution Model

We evaluate the efficacy of D3 by exploring a deadline allocation policy that adjusts the end-to-end deadline to avoid collisions in challenging scenarios. The focus of our work is not the design of policies, but to provide the mechanisms to implement such policies. Therefore, we present a baseline policy that adapts deadlines as a function of the AV speed and the trajectories of other agents. Our policy computes *reaction time*, defined as the sum of time to receive 8 sensor readings, which are sufficient to build a trajectory prediction for the agents, and the end-to-end runtime of the current configuration. The policy uses the reaction time and the AV's

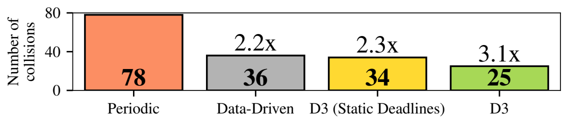


Figure 11. D3 Reduces Collisions. In a challenging 50km drive, ERDOS' realization of D3's dynamic deadlines reduces collisions by 68% over solutions using the periodic execution model.

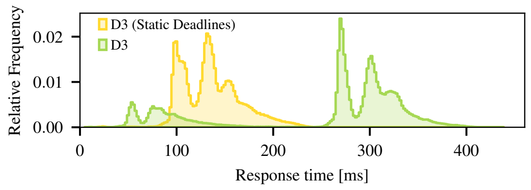


Figure 12. Response Time Histogram. D3 (Static Deadlines) enforces the static deadlines that perform the best during the drive and the variability is due to **C2**. By contrast, the D3 with dynamic deadlines offers faster responses when needed, and executes more accurate computation otherwise.

driving speed to estimate the AV's *stopping distance*. It then adjusts the end-to-end deadline depending on how close to other agents the AV will be at the end of its stopping distance.

We compare the performance of Pylot under the dynamic deadlines computed by this deadline allocation policy to five static deadlines ranging from 125ms to 500ms.

7.4.1 Aggregate Study. We explore if our policy adjusts the deadlines to avoid collisions during a challenging 50km CARLA Challenge drive [13]. In this experiment, we adapt the detector in response to shorter deadlines, but keep all the other components fixed in order to limit the experiment duration (exploring all tradeoffs required 100 days of simulation).

Fig. 11 highlights the efficacy of D3 apart from the engineering of ERDOS by running Pylot atop ERDOS using four execution models: (*i*) a periodic execution derived from WCET estimates (similar to Apollo [1] and Autoware [28], which execute most components periodically), (*ii*) the best data-driven configuration that executes each component upon receipt of all input data (similar to some ROS deployments, see §3.1), (*iii*) the best configuration with static deadlines enforced by D3's D_{EH}, and (*iv*) a D3 execution with dynamic deadlines enabled by our deadline allocation policy and ERDOS. The execution with our policy (D3) reduces collisions by 68% over a periodic execution, and by 26% over the best configuration with static deadlines because the policy reduces the deadlines in challenging scenarios. Finally, we compare the end-to-end response times of Pylot's D3 execution with dynamic deadlines with Pylot's best configuration with static deadlines. Fig. 12 shows that in most situations D3's Pylot execution runs a slow, high-accuracy configuration, but adapts to fast configurations when the environment demands it.

Takeaway: *ERDOS' dynamic deadlines result in significantly fewer collisions compared to the periodic execution and static deadlines used in state-of-the-art platforms.*

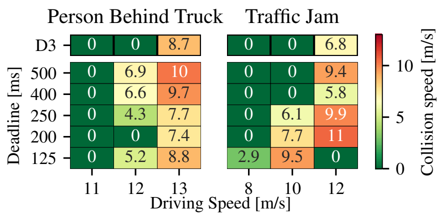


Figure 13. Versatility of D3's Dynamic Deadlines. Configurations with short deadlines reduce collision speed in the *person behind truck* scenario (left), but increase it in the *traffic jam* scenario (right). By contrast, D3 adapts Pylot's deadlines depending on the driving speed and scenario complexity resulting in fewer collisions.

7.4.2 In-depth Study of Scenarios. We study the benefits of ERDOS' realization of D3 using two challenging scenarios that require the AV to adapt in order to avoid collisions.

Person Behind Truck. This scenario simulates a person illegally entering the AV's lane (see video[3]). The scenario is complicated by a truck that occludes the person until they enter the AV's lane. Thus, the AV cannot stop in time and must perform an emergency swerving maneuver. Since this maneuver requires a knee-jerk reaction, we expect the configurations that minimize the response time to perform better.

Traffic Jam. This scenario simulates merging into a traffic jam. The AV is required to come to a halt behind a vehicle and a motorcycle, while the other lane is lined up with vehicles. The motorcycle complicates this scenario as it requires the AV to perceive the object from afar in order to prevent a collision. Moreover, the vehicles on the other lane prevent the AV from performing an emergency swerve. While the previous scenario requires a fast response, this scenario needs consistent high-quality responses from the AV in order to prevent an otherwise-safe scenario from turning into an emergency.

In the experiment, we drive the AV at a fixed speed using a fixed set of hardware resources (see §7). We execute Pylot's five configurations with static deadlines (§7.4) and Pylot's D3 execution with dynamic deadlines computed by our policy. We use the driving speed of the AV at the time of the collision (i.e., *collision speed*) as a proxy for the impact of the collision, where a speed of 0m/s shows that Pylot avoided a collision.

In Fig. 13 we plot the collision speed across varying speeds. As expected, at a speed of 12m/s, the probability of successfully handling the *person behind truck* scenario increases with a decrease in response time. In this scenario all but the fastest configuration detect the person, which is visible 20m away. Thus, the configuration with the lowest response time (200ms) that detects the person 20m away prevents a collision, while configurations with higher response times collide with the person at collision speeds that increase with the response time. We note that the configuration with the lowest response time (125ms) collides as it detects the person too late (12m away) due to its low perception accuracy. On the contrary, in the *traffic jam* scenario, the slower, more accurate configurations allow the AV to reliably stop at 10m/s. This is because the

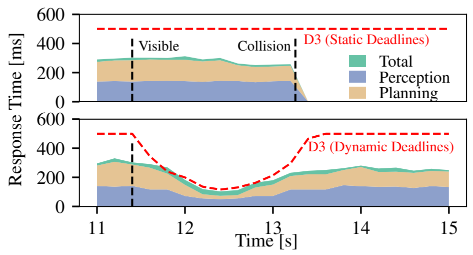


Figure 14. Adapting to Deadlines. D3 enables Pylot's components to meet dynamic deadlines and avoid a collision.

motorcycle is partially occluded, and thus faster, less-accurate models perform poorly. Fig. 14 shows how Pylot adapts as our policy reduces the end-to-end deadline once the person is visible in the *person behind truck* scenario.

Takeaway: *ERDOS' deadlines adapt in both scenarios, and avoid more collisions than any static configuration*[3].

8 Related Work

Data-Driven Execution Model. Vendors [20, 28, 53, 94, 95] are developing AV pipelines atop robotics platforms that provide a modular design and best-effort execution of the components (e.g., ROS [77], ROS2 [58], CyberRT [30]). As a result, vendors execute these pipelines as SLO-based best-effort applications that attempt to meet an environment-agnostic end-to-end deadline [28, 29, 95]. The AV pipelines are deployed as ROS/CyberRT processes, that either use the data-driven execution model to run each component to completion upon receiving all input (§3.1), which may delay downstream components due to runtime variability, or run components periodically [1, 10], which preclude adaptations to meet dynamic deadlines. Moreover, these platforms do not offer a system-managed consistent view of time, and thus lack mechanisms to specify and enforce deadlines, or reason about the available execution time [95]. By contrast, ERDOS enables the development of D3 applications by offering a consistent view of time via logical times, an automatic mapping of logical time to wall-clock time, which components can use to reason about deadlines and available execution time, and the ability to apply reactive measures to mitigate missed deadlines.

Stream processing systems such as Flink [39], Cloud Dataflow [6], MillWheel [21], and Naiad [74] also utilize the data-driven execution model. Although, these systems inspired elements of our design (e.g., logical time [6, 39, 74], watermarks [22, 39, 93], intermediate results [17]), these systems are designed for massively parallel data processing, and embed architectural and implementational decisions that make them unconducive to the development of AVs. For example, Naiad paralellizes an application by partitioning data across workers, which each execute an entire copy of the dataflow computation (AV sensor data is not partitionable). Moreover, these systems are unable to realize the D3 execution model because, unlike ERDOS, they lack APIs to specify environment-dependent deadlines and to implement proactive strategies to meet these deadlines, and to apply reactive measures when deadlines are missed.

Periodic Execution Model. Hard real-time systems conduct schedulability analyses driven by WCET estimates to guarantee that deadline constraints are met [32, 36, 49, 56, 67, 69, 92]. However, AV components preclude the accurate estimation of WCETs due to environment-dependent runtimes and large input spaces [24, 25, 87], or the non-deterministic nature of the algorithms they use [25, 43, 60, 97]. Thus, developing AVs as such systems requires use of conservative WCETs to derive the periodicity of execution for each component [48]. However, periodic executions cannot meet dynamic deadlines, and trade accuracy to ensure that components with a large gap between mean and worst-case execution time meet deadlines [25, 35, 44]. To address the former, real-time applications implement *mode changes* [26, 42, 78], which depend on WCETs to verify if transitions between modes lead to deadline misses [67, 78, 85, 92]. By contrast, *adaptive real-time systems* [35, 63, 70, 80] support the execution of components without WCET. These systems minimize deadline misses by using feedback-based policies to choose the best service level from multiple application-defined levels (similar to §5.3's changing implementations), but lack mechanisms to enforce deadlines and mitigate deadline misses.

D3 subsumes prior systems by allowing the execution of both mode changes and adaptive real-time applications. The developers of D3 applications can specify mode changes using the deadline policy (π_{DP}) and trigger them to perform graceful degradation (on deadline misses or environment changes) using D3's feedback loop (§3). Furthermore, ERDOS's use of timestamps and watermarks helps with tracking the causality of messages back to sensor data, and along with system-managed state makes ERDOS more amenable to analysis and verification than current AV platforms.

9 Conclusions

We highlight two key characteristics of AVs, and introduce D3, an execution model for applications that must maximize accuracy in the presence of dynamic deadlines. We realize D3 in ERDOS, atop which we build an AV, and find that D3 reduces collisions by 68%. We hope that our artifacts will aid the development of safer AVs, and inspire systems research.

Acknowledgements

We thank Ioan Stefanovici, our shepherd, and the anonymous reviewers whose insightful comments helped improve the paper. This research is supported by a NSF CISE Expeditions Award CCF-1730628, and by gifts from Amazon Web Services, Ant Group, Ericsson, Facebook, Futurewei, Google, Intel, Microsoft, Scotiabank, and VMware.

[3]Static vs dynamic deadlines in Pylot: https://tinyurl.com/y24p4g8d

References

[1] Apollo Planning Frequency. https://github.com/ApolloAuto/apollo/blob/master/modules/planning/common/planning_gflags.cc#L23.

[2] Apollo's Traffic Light Perception. https://github.com/ApolloAuto/apollo/blob/master/docs/specs/traffic_light.md.

[3] Argoverse. https://www.argoverse.org/.

[4] Automated Vehicles for Safety. https://www.nhtsa.gov/technology-innovation/automated-vehicles-safety.

[5] Companies Have Spent Over $16 Billion on Robocars. It's A Drop in the Bucket. https://tinyurl.com/54ydc9zk.

[6] Google Cloud Dataflow. http://cloud.google.com/dataflow/.

[7] How Does a Self-Driving Car See? https://blogs.nvidia.com/blog/2019/04/15/how-does-a-self-driving-car-see/.

[8] How Uber Self-Driving Cars See The World. https://www.therobotreport.com/how-uber-self-driving-cars-see-world/.

[9] Introducing the 5th Generation Waymo Driver. https://blog.waymo.com/2020/03/introducing-5th-generation-waymo-driver.html.

[10] Planning Loop Rate in Autoware AV. https://tinyurl.com/56m3uze2.

[11] Sight Distance Guidelines. https://mdotcf.state.mi.us/public/tands/Details_Web/mdot_sight_distance_guidelines.pdf.

[12] Snow and Ice Pose a Vexing Obstacle for Self-Driving Cars. https://www.wired.com/story/snow-ice-pose-vexing-obstacle-self-driving-cars/.

[13] The CARLA Autonomous Driving Challenge. https://leaderboard.carla.org/.

[14] The State of the Self-Driving Car Race in 2020. https://www.bloomberg.com/features/2020-self-driving-car-race/.

[15] To Make Self-Driving Cars Safe, We Also Need Better Roads and Infrastructure. https://tinyurl.com/2p8wz3t7.

[16] Weather Creates Challenges For Next Generation Of Vehicles. https://tinyurl.com/yzjwcvwa.

[17] Daniel J Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stan Zdonik. The Design of the Borealis Stream Processing Engine. In *Proceedings of the 2nd Biennial Conference on Innovative Data Systems Research (CIDR)*, pages 277–289, 2005.

[18] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A System for Large-Scale Machine Learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, November 2016.

[19] Michael Aeberhard, Thomas Kühbeck, Bernhard Seidl, M Friedl, J Thomas, and O Scheickl. Automated Driving with ROS at BMW. *ROSCon 2015 Hamburg, Germany*, 2015.

[20] Michael Aeberhard, Thomas Kühbeck, Bernhard Seidl, Martin Friedl, Julian Thomas, and Oliver Scheickl. Automated Driving with ROS at BMW. http://www.ros.org/news/2016/05/michael-aeberhard-bmw-automated-driving-with-ros-at-bmw.html.

[21] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. MillWheel: Fault-tolerant Stream Processing at Internet Scale. *Proceedings of the VLDB*, 6(11), August 2013.

[22] Tyler Akidau, Edmon Begoli, Slava Chernyak, Fabian Hueske, Kathryn Knight, Kenneth Knowles, Daniel Mills, and Dan Sotolongo. Watermarks in Stream Processing Systems: Semantics and Comparative Analysis of Apache Flink and Google Cloud Dataflow. *Proceedings of the VLDB Endowment*, 14(12), 2021.

[23] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, et al. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-scale, Unbounded, Out-of-order Data Processing. *Proceedings of the VLDB Endowment*, 8(12):1792–1803, 2015.

[24] Sergi Alcaide, Leonidas Kosmidis, Hamid Tabani, Carles Hernandez, Jaume Abella, and Francisco J Cazorla. Safety-Related Challenges and Opportunities for GPUs in the Automotive Domain. *IEEE Micro*, 38(6):46–55, 2018.

[25] Miguel Alcon, Hamid Tabani, Leonidas Kosmidis, Enrico Mezzetti, Jaume Abella, and Francisco J Cazorla. Timing of Autonomous Driving Software: Problem Analysis and Prospects for Future Solutions. In *Proceedings of the 26th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 267–280. IEEE, 2020.

[26] Luis Almeida, Sebastian Fischmeister, Madhukar Anand, and Insup Lee. A Dynamic Scheduling Approach to Designing Flexible Safety-Critical Systems. In *Proceedings of the 7th ACM & IEEE International Conference on Embedded Software*, pages 67–74, 2007.

[27] D. Anguelov. Taming The Long Tail of Autonomous Driving Challenges. https://www.youtube.com/watch?v=Q0nGo2-y0xY, 2019.

[28] Autoware. Autoware User's Manual - Document Version 1.1. https://tinyurl.com/2v2jkk9n.

[29] Baidu. Apollo 3.0 Software Architecture. https://tinyurl.com/mhd6dfka.

[30] Baidu. Apollo Cyber RT. https://github.com/ApolloAuto/apollo/tree/master/cyber.

[31] Baidu. Apollo Data Open Platform. http://data.apollo.auto/.

[32] Sanjoy Baruah. Improved Multiprocessor Global Schedulability Analysis of Sporadic DAG Task Systems. In *Proceedings of the 26th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 97–105. IEEE, 2014.

[33] Sanjoy Baruah, Vincenzo Bonifaci, and Alberto Marchetti-Spaccamela. The Global EDF Scheduling of Systems of Conditional Sporadic DAG Tasks. In *Proceedings of the 27th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 222–231. IEEE, 2015.

[34] Alex Bewley, Zongyuan Ge, Lionel Ott, Fabio Ramos, and Ben Upcroft. Simple Online and Realtime Tracking. In *Proceedings of the 23th IEEE International Conference on Image Processing (ICIP)*, pages 3464–3468, 2016.

[35] Aaron Block, Björn Brandenburg, James H Anderson, and Stephen Quint. An Adaptive Framework for Multiprocessor Real-Time System. In *Proceedings of the 20th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 23–33. IEEE, 2008.

[36] Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, Sebastian Stiller, and Andreas Wiese. Feasibility Analysis in the Sporadic DAG Task Model. In *Proceedings of the 25th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 225–233. IEEE, 2013.

[37] Sol Boucher, Anuj Kalia, David G Andersen, and Michael Kaminsky. Lightweight Preemptible Functions. In *Proceedings of the 31st USENIX Annual Technical Conference (ATC)*, pages 465–477, 2020.

[38] Giorgio C Buttazzo, Giuseppe Lipari, Marco Caccamo, and Luca Abeni. Elastic Scheduling for Flexible Workload Management. *IEEE Transactions on Computers*, 51(3):289–302, 2002.

[39] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache Flink: Stream and Batch Processing in a Single Engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[40] Don Carney, Uğur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Nesime Tatbul, Stan Zdonik, and Michael Stonebraker. Monitoring Streams - A New Class of Data Management Applications. In *Proceedings of the 28th International Conference on Very Large Databases (VLDB)*, pages 215–226, 2002.

[41] Daniel Casini, Tobias Blaß, Ingo Lütkebohle, and Björn B Brandenburg. Response-Time Analysis of ROS 2 Processing Chains Under Reservation-Based Scheduling. In *Proceedings of the 31st Euromicro Conference on Real-Time Systems (ECRTS)*, 2019.

[42] Tianyang Chen and Linh Thi Xuan Phan. SafeMC: A System for the Design and Evaluation of Mode-Change Protocols. In *Proceedings of the 24th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 105–116. IEEE, 2018.

[43] Xiaozhi Chen, Huimin Ma, Ji Wan, Bo Li, and Tian Xia. Multi-View 3D Object Detection Network for Autonomous Driving. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1907–1915, 2017.

[44] Hoon Sung Chwa, Kang G Shin, Hyeongboo Baek, and Jinkyu Lee. Physical-State-Aware Dynamic Slack Management for Mixed-Criticality Systems. In *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 129–139. IEEE, 2018.

[45] Laurene Claussmann, Marc Revilloud, Dominique Gruyer, and Sébastien Glaser. A Review of Motion Planning for Highway Autonomous Driving. *IEEE Transactions on Intelligent Transportation Systems*, 21(5):1826–1848, 2019.

[46] Henry Claypool, Amitai Bin-Nun, and Jeffrey Gerlach. Self-Driving Cars: The Impact on People with Disabilities. *Newton, MA: Ruderman Family Foundation*, 2017.

[47] B Dasarathy. Timing constraints of real-time systems: Constructs for expressing them, methods of validating them. *IEEE Transactions on Software Engineering*, (1):80–86, 1985.

[48] Dionisio De Niz, Karthik Lakshmanan, and Ragunathan Rajkumar. On the Scheduling of Mixed-Criticality Real-Time Task Sets. In *Proceedings of the 30th IEEE Real-Time Systems Symposium (RTSS)*, pages 291–300. IEEE, 2009.

[49] Patricia Derler, Thomas H Feng, Edward A Lee, Slobodan Matic, Hiren D Patel, Yang Zheo, and Jia Zou. PTIDES: A Programming Model for Distributed Real-Time Embedded Systems. Technical report, University of California, Berkeley, 2008.

[50] Dirk Thomas. Changes between ROS 1 and ROS 2. http://design.ros2.org/articles/changes.html.

[51] Dmitri Dolgov, Sebastian Thrun, Michael Montemerlo, and James Diebel. Practical Search Techniques in Path Planning for Autonomous Driving. In *Proceedings of the 1st International Symposium on Search Techniques in Artificial Intelligence and Robotics (STAIR)*, volume 1001, pages 18–80, 2008.

[52] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. CARLA: An Open Urban Driving Simulator. In *Proceedings of the 1st Conference on Robot Learning (CoRL)*, pages 1–16, 2017.

[53] Andreas Fregin, Markus Roth, Markus Braun, Sebastian Krebs, and Fabian Flohr. Building a Computer Vision Research Vehicle with ROS. http://www.ros.org/news/2018/07/roscon-2017-building-a-computer-vision-research-vehicle-with-ros----andreas-fregin.html.

[54] Yilong Geng, Shiyu Liu, Zi Yin, Ashish Naik, Balaji Prabhakar, Mendel Rosenblum, and Amin Vahdat. Exploiting a Natural Network Effect for Scalable, Fine-Grained Clock Synchronization. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, pages 81–94, 2018.

[55] Ionel Gog, Sukrit Kalra, Peter Schafhalter, Matthew A. Wright, Joseph E. Gonzalez, and Ion Stoica. Pylot: A Modular Platform for Exploring Latency-Accuracy Tradeoffs in Autonomous Vehicles. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 8806–8813. IEEE, 2021.

[56] Joël Goossens, Shelby Funk, and Sanjoy Baruah. Priority-Driven Scheduling of Periodic Task Systems on Multiprocessors. *Real-time Systems*, 25(2):187–205, 2003.

[57] Matthew P. Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert N. M. Watson, Andrew W. Moore, Steven Hand, and Jon Crowcroft. Queues don't matter when you can JUMP them! In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, May 2015.

[58] Christopher Ho, Sumanth Nirmal, Juan Pablo Samper, Serge Nikulin, Anup Pemmaiah, Dejan Pangercic, and Jan Becker. ROS2 on Autonomous Vehicles. https://roscon.ros.org/2018/presentations/ROSCon2018_ROS2onAutonomousDrivingVehicles.pdf.

[59] Jonathan Huang, Vivek Rathod, Chen Sun, Menglong Zhu, Anoop Korattikara, Alireza Fathi, Ian Fischer, Zbigniew Wojna, Yang Song, Sergio Guadarrama, and Kevin Murphy. Speed/Accuracy Trade-Offs for Modern Convolutional Object Detectors. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017.

[60] Sertac Karaman and Emilio Frazzoli. Sampling-Based Algorithms for Optimal Motion Planning. *The International Journal of Robotics Research*, 30(7):846–894, 2011.

[61] A. Karpathy. CVPR '20 - Workshop on Scalability in Autonomous Driving. https://sites.google.com/view/cvpr20-scalability/archived-talks/keynotes, 2020.

[62] Christos Katrakazas, Mohammed Quddus, Wen-Hua Chen, and Lipika Deka. Real-time Motion Planning Methods for Autonomous On-Road Driving: State-of-the-art and Future Research Directions. *Transportation Research Part C: Emerging Technologies*, 60:416–442, 2015.

[63] T-W Kuo and Aloysius K Mok. Load Adjustment in Adaptive Real-Time Systems. In *Proceedings of the 12th Real-Time Systems Symposium (RTSS)*, pages 160–161. IEEE, 1991.

[64] Steven M. LaValle. *Planning Algorithms*. Cambridge University Press, USA, 2006.

[65] Mengtian Li, Yuxiong Wang, and Deva Ramanan. Towards Streaming Perception. In *Proceedings of the European Conference on Computer Vision (ECCV)*, August 2020.

[66] Shih-Chieh Lin, Yunqi Zhang, Chang-Hong Hsu, Matt Skach, Md E. Haque, Lingjia Tang, and Jason Mars. The Architectural Implications of Autonomous Driving: Constraints and Acceleration. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 751–766, 2018.

[67] Chung Laung Liu and James W Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM (JACM)*, 20(1):46–61, 1973.

[68] Jane W.S. Liu. *Real-Time Systems*. Prentice Hall, 2000.

[69] Marten Lohstroh, Martin Schoeberl, Andrés Goens, Armin Wasicek, Christopher Gill, Marjan Sirjani, and Edward A Lee. Actors Revisited for Time-Critical Systems. In *Proceedings of the 56th ACM/IEEE Design Automation Conference (DAC)*, pages 1–4. IEEE, 2019.

[70] Chenyang Lu, John A Stankovic, Tarek F Abdelzaher, Gang Tao, Sang Hyuk Son, and Michael Marley. Performance Specifications and Metrics for Adaptive Real-Time Systems. In *Proceedings of the 21st Real-Time Systems Symposium (RTSS)*, pages 13–23. IEEE, 2000.

[71] Yuya Maruyama, Shinpei Kato, and Takuya Azumi. Exploring the performance of ros2. In *Proceedings of the 13th International Conference on Embedded Software*, pages 1–10, 2016.

[72] Matt Ranney. Self-Driving Cars As Edge Computing Devices. https://www.infoq.com/presentations/uber-atg/.

[73] Michele Bertoncello, and Dominik Wee. Ten Ways Autonomous Driving Could Redefine the Automotive World. https://tinyurl.com/2srpyv8d.

[74] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A Timely Dataflow System. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 439–455, November 2013.

[75] National Highway Traffic Safety Administration. Traffic Safety Facts (2017 Data). https://crashstats.nhtsa.dot.gov/Api/Public/ViewPublication/812687.

[76] Brian Paden, Michal Čáp, Sze Zheng Yong, Dmitry Yershov, and Emilio Frazzoli. A Survey of Motion Planning and Control Techniques for Self-Driving Urban Vehicles. *IEEE Transactions on Intelligent*

Vehicles, 1(1):33–55, 2016.

[77] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. ROS: An Open-Source Robot Operating System. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA); Workshop on Open Source Robotics*, volume 3, page 5, May 2009.

[78] Jorge Real and Alfons Crespo. Mode Change Protocols for Real-Time Systems: A Survey and a New Proposal. *Real-time systems*, 26(2):161–197, 2004.

[79] Nicholas Rhinehart, Kris M Kitani, and Paul Vernaza. R2P2: A Reparameterized Pushforward Policy for Diverse, Precise Generative Path Forecasting. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 772–788, 2018.

[80] Daniela Rosu, Karsten Schwan, Sudhakar Yalamanchili, and Rakesh Jha. On Adaptive Resource Allocation for Complex Real-Time Applications. In *Proceedings of the 18th Real-Time Systems Symposium (RTSS)*, pages 320–329. IEEE, 1997.

[81] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-Free Replicated Data Types. In *Symposium on Self-Stabilizing Systems*, pages 386–400. Springer, 2011.

[82] Society of Automotive Engineers. *Taxonomy and Definitions for Terms Related to On-Road Motor Vehicle Automated Driving Systems*. SAE International, 2018.

[83] Utkarsh Srivastava and Jennifer Widom. Flexible time management in data stream systems. In *Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 263–274, 2004.

[84] State of California Department of Motor Vehicles. Autonomous Vehicle Disengagement Reports 2018. https://www.dmv.ca.gov/portal/dmv/detail/vr/autonomous/disengagement_report_2019.

[85] Nikolay Stoimenov, Simon Perathoner, and Lothar Thiele. Reliable Mode Changes in Real-Time Systems with Fixed Priority or EDF Scheduling. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 99–104. IEEE, 2009.

[86] H Streich. Taskpair-Scheduling: An Approach for Dynamic Real-Time Systems. In *Second Workshop on Parallel and Distributed Real-Time Systems*, pages 24–31. IEEE, 1994.

[87] Hamid Tabani, Leonidas Kosmidis, Jaume Abella, Francisco J Cazorla, and Guillem Bernat. Assessing the Adherence of an Industrial Autonomous Driving Framework to ISO 26262 Software Guidelines. In *Proceedings of the 56th Annual Design Automation Conference (DAC)*, pages 1–6. IEEE, 2019.

[88] Mingxing Tan, Ruoming Pang, and Quoc V. Le. EfficientDet: Scalable and Efficient Object Detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020.

[89] Charlie Tang and Russ R Salakhutdinov. Multiple Futures Prediction. In *Proceedings of the International Conference on Advances in Neural Information Processing Systems (NeurIPS)*, pages 15398–15408, 2019.

[90] Nesime Tatbul, Ugur Çetintemel, and Stan Zdonik. Staying Fit: Efficient Load Shedding Techniques for Distributed Stream Processing. In *Proceedings of the 33rd International Conference on Very large Databases (VLDB)*, pages 159–170, 2007.

[91] Nesime Tatbul, Uğur Çetintemel, Stan Zdonik, Mitch Cherniack, and Michael Stonebraker. Load Shedding in a Data Stream Manager. In *Proceedings of the 29th International Conference on Very Large Databases (VLDB)*, pages 309–320, 2003.

[92] Ken Tindell, Alan Burns, and Andy J Wellings. Mode Changes In Priority Pre-Emptively Scheduled Systems. In *Proceedings of the 13th Real-Time Systems Symposium (RTSS)*, volume 92, pages 100–109. Citeseer, 1992.

[93] Peter A. Tucker, David Maier, Tim Sheard, and Leonidas Fegaras. Exploiting Punctuation Semantics in Continuous Data Streams. *IEEE Transactions on Knowledge and Data Engineering*, 15(3):555–568, 2003.

[94] Udacity. An Open Source Self-Driving Car. https://www.udacity.com/self-driving-car.

[95] Nicolo Valigi. Lessons Learned Building a Self-Driving Car on ROS. https://roscon.ros.org/2018/presentations/ROSCon2018_LessonsLearnedSelfDriving.pdf, 2018.

[96] Peng Wang, Xinyu Huang, Xinjing Cheng, Dingfu Zhou, Qichuan Geng, and Ruigang Yang. The Apolloscape Open Dataset for Autonomous Driving and its Application. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2019.

[97] Yan Wang, Zihang Lai, Gao Huang, Brian H Wang, Laurens Van Der Maaten, Mark Campbell, and Kilian Q Weinberger. Anytime Stereo Image Depth Estimation on Mobile Devices. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 5893–5900. IEEE, 2019.

[98] Moritz Werling, Julius Ziegler, Sören Kammel, and Sebastian Thrun. Optimal Trajectory Generation for Dynamic Street Scenarios in a Frenet Frame. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 987–993. IEEE, 2010.

[99] Nicolai Wojke, Alex Bewley, and Dietrich Paulus. Simple Online and Realtime Tracking with a Deep Association Metric. In *Proceedings of the 24th IEEE International Conference on Image Processing (ICIP)*, pages 3645–3649. IEEE, 2017.

[100] Wenda Xu, Junqing Wei, John M Dolan, Huijing Zhao, and Hongbin Zha. A Real-Time Motion Planner with Trajectory Optimization for Autonomous Vehicles. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 2061–2067. IEEE, 2012.

[101] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, pages 15–28, April 2012.

[102] Zheng Zhu, Qiang Wang, Li Bo, Wei Wu, Junjie Yan, and Weiming Hu. Distractor-Aware Siamese Networks for Visual Object Tracking. In *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018.

[103] Shlomo Zilberstein. Using Anytime Algorithms in Intelligent Systems. *AI magazine*, 17(3):73–83, 1996.

A Artifact Appendix

A.1 Artifact

We provide the following two software artifacts for reproducing the experiments discussed in the paper:

- ERDOS, the open-source distributed system for the execution of autonomous vehicle (AV) pipelines, and
- Pylot, the open-source state-of-the-art AV pipeline that works across simulators and real-vehicles.

A.2 Description & Requirements

A.2.1 How to access.

The artifacts are located at the following public repositories:

- ERDOS (DOI 10.5281/zenodo.6345345): https://github.com/erdos-project/erdos
- Pylot (DOI 0.5281/zenodo.6345352): https://github.com/erdos-project/pylot

We also publicly released the code, logs and plotting scripts for our experiments (DOI 10.5281/zenodo.6345350) at https://github.com/erdos-project/erdos-experiments.

A.2.2 Hardware dependencies.

All experiments were performed atop a machine having 2 × Xeon Gold 6226 CPUs, 128GB of RAM, and 2 × 24GB Titan-RTX GPUs, running Linux Kernel 5.3.0.

We note that executing the experiments on different hardware could significantly affect the results as any differences in end-to-end response may cause the simulation experiments to diverge from the simulations we executed (e.g., an increased response time might cause an early collision, but avoid later collisions due to the AV being at a different location). Moreover, due to limitations in the simulator's performance (simulating 1 second takes approximately 10-30 seconds in wall-clock time), running our experiments takes approximately 1 month. To address these issues, we provide logs of our experiments along with scripts for analyzing the data and generating figures in our experiments repository.

A.2.3 Software dependencies.

ERDOS's performance vs. other systems. To compare ERDOS' performance to other systems used for AV development, we used the following software:

- Docker 19.03.13
- `docker-compose` 1.29.1
- Apache Maven 3.6.0
- Python 3.6.9 with Pandas 1.1.4
- Rust nightly-2021-12-04

Efficacy of ERDOS and the D3 execution model. We provide a Docker image with ERDOS, Pylot, and the CARLA simulator configured. The image can be retrieved by running `docker pull erdosproject/pylot:v0.3.2`. Thus, these experiments only depend on NVIDIA Docker 2.6.0.

A.2.4 Benchmarks. None.

A.3 Set-up

Clone the experiments repository with

```
git clone \
  https://github.com/erdos-project/erdos-experiments.git
```

From the root of the repository, run the following to download the logs of the experiments and reproduce the figures:

```
./download_data.sh
# Verify that the experiments directory contains log data
# of our experiments.
# Install dependencies required to run the plotting
# scripts.
pip3 install -r requirements.txt
cd plotting_scripts
./generate_plots.sh
# The graphs will be generated in
# plotting_scripts/graphs.
```

Systems comparisons. Run the following to install the software dependencies for the systems experiments:

```
bash systems-experiments/scripts/install_dependencies.sh
```

AV experiments. Run the required Docker container:

```
docker pull erdosproject/pylot:v0.3.2
nvidia-docker run -itd --name pylot -p 20022:22 \
   erdosproject/pylot /bin/bash
```

To ensure that the image correctly runs on your hardware, start the simulator in the container:

```
nvidia-docker exec -i -t pylot\
   /home/erdos/workspace/pylot/scripts/run_simulator.sh
```

In another terminal, start Pylot in the container:

```
nvidia-docker exec -i -t pylot /bin/bash
cd ~/workspace/pylot/
# Execute Pylot using a Faster-RCNN object detector.
python3 pylot.py --flagfile=configs/detection.conf
```

To verify that the simulation is progressing and the AV is detecting obstacles, inspect ∼`/workspace/pylot/pylot.log`.

A.4 Evaluation workflow

A.4.1 Major Claims.

- (C1): ERDOS outperforms similar state-of-the-art execution systems and scales to large AV pipelines. This claim is proven by experiments (E1), (E2) and (E3) described in §7.2 and illustrated in Fig. 8.
- (C2): ERDOS implements D3 by swiftly executing π_{DP}, enabling proactive strategies to meet deadlines, and rapidly taking reactive measures when deadlines are missed. This claim is proven by (E4) and (E5) described in §7.3, whose results are shown in Fig. 9 and Fig. 10.
- (C3): ERDOS' implementation of D3 offers a 68% reduction in collisions compared to prior execution models. This claim is supported by (E6) and (E7) described in §7.4.1, and illustrated in Fig. 11 and Fig. 12.

- (C4): ERDOS’ instantiation of D3’s dynamic deadlines enables the AV to adapt in two opposite scenarios and avoid more collisions than any static configuration. This claim is proven by (E8) and (E9) described in §7.4.2, whose results are illustrated in Fig. 13 and Fig. 14.

A.4.2 Experiments. In the following text, we first lay out the instructions for executing the systems experiments, followed by the AV experiments.

Experiments (E1, E2, and E3): [60 human-minutes + 10 compute-hours]: To run all the system benchmark experiments, execute the following commands:

```
# Runs the ERDOS system experiments.
cd systems-experiments/erdos-experiments
bash scripts/run_all.sh
```

The results of the experiments will be stored as CSV files in systems-experiments/erdos-experiments/results/$(hostname)/

```
# Runs the Flink system experiments.
cd systems-experiments/flink-experiments
bash scripts/run_all.sh
```

The results of the experiments will be stored as CSV files in systems-experiments/flink-experiments/results/$(hostname)/

```
# Runs the ROS system experiments.
cd systems-experiments/ros-experiments
bash scripts/run_all.sh
```

The results of the experiments will be stored as CSV files in systems-experiments/ros-experiments/results/$(hostname)/

To generate §7.2 from these logs, execute the following:

```
# Script for Fig 8 (a)
cd ${ERDOS_EXPERIMENTS_HOME}/plotting_scripts
HOST=$(hostname)
python3 plot_msg_size_latency.py \
    "../eurosys_systems_experiments/$HOST/erdos/\
                msg-size-latency-intra-process.csv" \
    "../eurosys_systems_experiments/$HOST/flink/\
                msg-size-latency-intra-process.csv" \
    "../eurosys_systems_experiments/$HOST/erdos/\
                msg-size-latency-inter-process.csv" \
    "../eurosys_systems_experiments/$HOST/flink/\
                msg-size-latency-inter-process.csv" \
    "../eurosys_systems_experiments/$HOST/ros/\
                msg-size-latency-inter-process.csv"
```

```
# Script for Fig 8 (b)
cd ${ERDOS_EXPERIMENTS_HOME}/plotting_scripts
HOST=$(hostname)
python3 plot_broadcast_latency.py \
    "../eurosys_systems_experiments/$HOST/erdos/\
                broadcast-latency-intra-process.csv" \
    "../eurosys_systems_experiments/$HOST/flink/\
                broadcast-latency-intra-process.csv" \
    "../eurosys_systems_experiments/$HOST/erdos/\
                broadcast-latency-inter-process.csv" \
    "../eurosys_systems_experiments/$HOST/flink/\
                broadcast-latency-inter-process.csv" \
    "../eurosys_systems_experiments/$HOST/ros/\
                broadcast-latency-inter-process.csv"
```

```
# Script for Fig 8 (c)
cd ${ERDOS_EXPERIMENTS_HOME}/plotting_scripts
HOST=$(hostname)
python3 plot_synthetic_pipeline.py \
    "../eurosys_systems_experiments/$HOST/erdos/\
                synthetic-pipeline-intra-process.csv" \
    "../eurosys_systems_experiments/$HOST/erdos/\
                synthetic-pipeline-intra-process.csv"
```

Experiments (E4-E9): [600 human-minutes + 720 compute-hours]: Due to a lack of space to list out the complex steps required to run the experiments involving the Pylot AV pipeline and the CARLA simulator, we refer the reader to the README of the experiments repository, which provides instructions for running the experiments discussed above.

A.5 Notes on Reusability

We developed and open-sourced Pylot with a broader objective of enabling the research community to study the effects of latency and accuracy of their innovations in individual parts of the AV pipeline on its end-to-end driving behavior. To help achieve this goal, Pylot was built with the key requirements of *modularity*, *portability* and *debuggability*.

Pylot provides a “plug-and-play” architecture for the individual components of the AV pipeline and comes equipped with ground-truth implementations for various components. This allows developers to easily swap their new models and algorithms for the old ones, and compare the effects of their innovation on our extended CARLA challenge benchmark. Furthermore, developers can use ground-truth implementations for the remainder of the pipeline to isolate the effects of their components for the purposes of debuggability.

Pylot’s *pseudo-asynchronous mode* allows developers to test the effects of the runtime of their components on the end-to-end driving behavior of the AV pipeline. We hope that the feature aids component developers in realistically testing the effects of their components in simulation, and minimize the cost, time and effort required to port the components from simulation to real-world AVs.

We achieve these key goals by developing Pylot atop ERDOS. Specifically, Pylot uses D3 to implement a “plug-and-play” architecture which allows developers to modify components without cascading changes to downstream components.

We hope that our artifacts: ERDOS, Pylot, and our extended AV benchmark atop the CARLA challenge aid a broader research agenda for the development of safe AVs.