D^3: A Dynamic Deadline-Driven Approach for Building Autonomous Vehicles

Anonymous Author(s)
Submission Id: 210

Abstract

Autonomous vehicles (AVs) are required to drive across a variety of challenging driving environments that impose continuously-varying deadlines and runtime-accuracy tradeoffs on their complex software pipelines. A *deadline-driven* execution of AV pipelines requires a new class of systems that enable the computation to adapt with the goal of maximizing 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 by proposing D^3 (**D**ynamic **D**eadline-**D**riven), a novel execution model for such applications that decouples the enforcement of deadlines from the computation, and allows applications to adjust their computation by modeling missed deadlines as *exceptions*. Further, we design and implement CarFlow, an open-source realization of D^3 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, our work addresses the crucial lack of AV benchmarks through our state-of-the-art open-source AV pipeline, Herbie, that works seamlessly across simulators and real vehicles. We evaluate the efficacy of D^3 and CarFlow by driving Herbie across challenging driving scenarios, and observe a 68% reduction in collisions as compared to prior execution models.

1 Introduction

The National Highway Traffic Safety Administration [5], 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 [71], (*ii*) increase traffic flow, which could free up as much as 50 minutes per person per day [69], and (*iii*) provide new employment opportunities to around 2 million people with disabilities [45]. Despite the potential benefits and investment [6], AV systems research is still in its infancy [13, 15–17, 27, 58].

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 [90] using multiple parallel processors and hardware accelerators [63]. As a result, a software system 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, 44].

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 [43, 92].

The current state-of-the-art systems for autonomous driving (e.g., Autoware [28], Cruise [90], BMW [21] etc. [51, 89]) are built atop the Robot Operating System (ROS) [73]. 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 [20] 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.

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, 48, 53, 66]. 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 [43, 92] (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

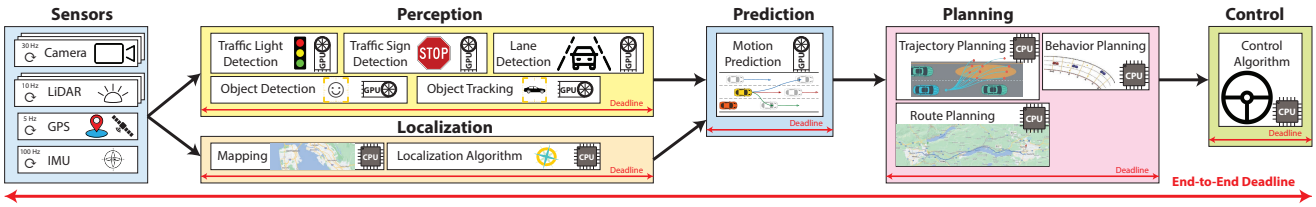


Figure 1. The architecture of a state-of-the-art AV pipeline. A modern AV uses multiple sensors to perceive the environment around it. These sensor readings are used by the *perception* module to detect other agents, and by the *localization* module to compute the location of the AV itself. The *prediction* 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.

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. Specifically, this paper makes the following two key contributions:

(**A**) We propose D^3 (**D**ynamic **D**eadline-**D**riven), an execution model for applications that interact with a continuously-evolving environment, and exhibit **C1-C2**. D^3 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* aim to meet deadlines, D^3 models missed deadlines due to **C2** as *exceptions* and allows the execution of *reactive measures*. Further, D^3 notifies downstream computation about any missed deadlines, allowing it to *eagerly* execute on incomplete input or adjust to fit in the reduced time upon arrival of the input.

(**B**) We design and implement CarFlow, a proof-of-concept realization of D^3 built specifically for AV pipelines. CarFlow 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. The system then aids the fulfillment of these deadlines by *speculatively* executing the appropriate implementation. However, in case deadlines are missed due to **C2**, CarFlow executes *exception handlers* that allow upstream components to convey intermediate results and downstream components to eagerly execute on incomplete input.

The remainder of the paper elaborates on our contributions that enable D^3 and CarFlow, 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 CarFlow to support D^3 (§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, Herbie (§7.1). Herbie works across simulators and real-vehicles, and achieves the top score in a simulated AV challenge.

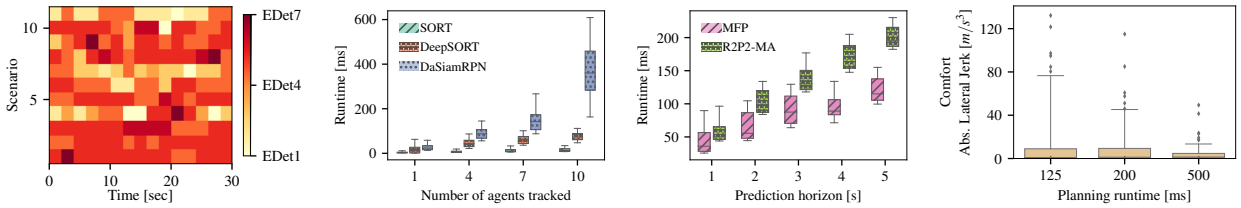
(**5**) We evaluate the efficacy of the dynamic deadline-driven execution enabled by D^3 and CarFlow by driving Herbie across 50 km of challenging driving scenarios in simulation (§7), and observe a 68% reduction in collisions as compared to the execution model of robotics platforms.

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 [8–10]. 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 [90] that execute atop several machines and accelerators [63, 68].

The *perception* module synchronizes the camera and LiDAR data streams and uses ML models to detect pedestrians, vehicles, traffic signals, and drivable regions. These detected objects along with the AV's location (computed by the *localization* module) are used by the *prediction* module to predict the trajectories of the 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 [63]. 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.



(a) Object detection. **(b)** Object tracking. **(c)** Trajectory prediction. **(d)** Trajectory planning.

Figure 2. No silver bullet. We investigate several components and show that: the best object detector varies within and across scenarios (**a**), components have runtimes that increase with an increased complexity of the environment (**b** and **c**), and allocating more time a component leads to more comfortable rides (**d**). These underscore the need for dynamically changing deadlines and components in AV pipelines.

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 a set of 12 real-world driving scenarios [4] into 2 second intervals, and plot the model with the best accuracy (adjusted by its runtime [62]) from the EfficientDet family [83], which provides multiple points in the *runtime-accuracy tradeoff* curve. Fig. 2a shows that different models perform better at different points during driving, which renders the selection of a static point on the tradeoff curve during development inadequate.

To exemplify 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* [12], 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 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 [50] 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 [94] and DaSiamRPN [96] offer high accuracy.

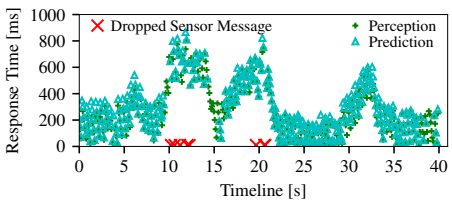


Figure 3. Response time variability. Baidu's Apollo production-grade perception and prediction suffer from response time variability.

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 into the future in order to ensure safety of the vehicle. Many prediction approaches (e.g., MFP [84] and R2P2-MA [75]) 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, 82]. To demonstrate this, we analyze sensor data from Baidu's Apollo AV [31] that drove over 108,000 miles [79, 91] (Fig. 3). Specifically, we focus on the traffic light detector [3], that relies on the map and the vehicle's location to choose between multiple cameras, and obtain bounding box proposals, which are individually refined and classified by multiple neural networks. We find that the perception response time depends on both the choice of camera and the number of traffic lights present in the environment. Apollo's p99 perception response time is 3.3× higher than the mean, which further increases the response time of the prediction downstream. Moreover, an increase in the response time keeps resources busy, thus forcing the pipeline to drop sensor messages.

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

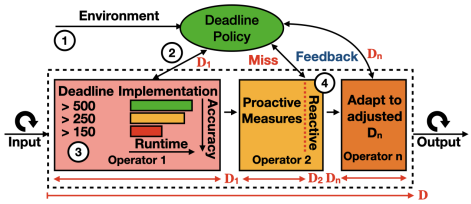


Figure 4. D^3 Model structures an application as an operator graph with a policy that decides the deadline $\mathcal{D}$ as per the environment (1), and assigns a $\mathcal{D}_i$ to each operator (2). The operators try to *proactively* meet $\mathcal{D}_i$ (3), and execute *reactive* measures otherwise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

3 D^3: *D*ynamic *D*eadline-*D*riven Execution Model

The execution of applications that interact with a continuously-evolving environment (e.g. robotics, AVs) requires a careful orchestration of their components. To enable such applications to optimize their accuracy under dynamically-varying deadlines (**C1**), we propose D^3, an execution model that decouples the management of deadlines from the computation. D^3 models missed deadlines due to **C2** as *exceptions*, allowing components to *reactively* adjust their computation.

A D^3 application is structured as a directed operator graph along with a *deadline policy* π_{DP} (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 between the input to the graph and the corresponding output (Step 1). Further, π_{DP} splits $\mathcal{D}$ across the 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 of the operators (Step 2).

While D^3 expects an operator to *proactively* meet its deadline (Operator 1 in Fig. 4; Step 3), it may be unable to achieve $\mathcal{D}_i$ due to runtime variability (**C2**). D^3 models such missed deadlines as *exceptions* and allows operators to perform *reactive measures* to quickly release output (Step 4). Moreover, it cascades the information of the missed deadline to downstream operators which can choose to either *eagerly* execute with incomplete inputs, or reason about the reduction in their available time once the upstream operator's reactive measures release output and modify their computation. Further, D^3's *feedback loop* also conveys the occurrence of such events to π_{DP}, which may choose to adjust the deadline for both downstream operators and future executions of the pipeline.

3.1 Related Execution Models

We highlight D^3's ability to maximize accuracy under *dynamic deadlines* by comparing to two key bodies of work: data-driven execution models and periodic execution models. Data-driven execution models employed by SLO-based robotics platforms (e.g., ROS [41, 73], Cyber RT [30]) provide a best-effort execution of components and lack mechanisms to reason about their variable runtimes [90]. As a result, components are unable to adjust their execution to varying deadlines

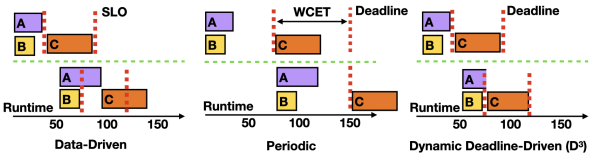


Figure 5. Timeline for execution models where C executes after it receives input from A and B. SLO-based models do not enforce deadlines and delay C's execution until all its inputs are available. Periodic models use WCET estimates to derive a periodic execution of components that is unable to maximize the runtime-accuracy tradeoff. D^3 maximizes this tradeoff by enabling components to either adjust to a constrained deadline or wait for delayed inputs.

leading them to miss their $\mathcal{D}_i$ in the presence of runtime variability (**C2**) (see Fig. 5). Moreover, since these data-driven models only execute computation upon receipt of input data, downstream components cannot initiate their computation in the absence of available inputs due to a missed upstream deadline (**C1**) or runtime variability (**C2**). However, D^3's proactive measures enable computation to meet their deadline $\mathcal{D}_i$ by either executing their computation without the missing input, or modifying their computation to fit within the reduced time.

Conversely, periodic models underpin many hard real-time systems [53, 65]. Such models use conservative worst-case execution time (WCET) estimates to derive a periodic execution of components, and thus preclude a lack of input or reduced computation time due to **C2**. However, owing to the large skew between the mean and the maximum runtime (see §2.2), these models fail to maximize the runtime-accuracy tradeoff. Moreover, periodic models attempt to meet environment-dependent deadlines (**C1**) by applying *mode changes*, which require components to either transition to SLO-based execution [35, 38, 60], or to undergo a time-consuming schedulability analysis for each possible deadline and mode transition [32, 33, 36, 74]. By contrast, D^3's event-driven nature and its proactive and reactive deadline measures enable it to forego such analysis, and adjust to the environment-dependent runtimes (**C2**) to meet dynamic deadlines (**C1**).

4 Introduction to CarFlow

We now provide an overview of the computational structure of CarFlow, a proof-of-concept instantiation of D^3, and discuss the implementation of a `Planner` (§4.2). §5 elaborates on the specific mechanisms that enable CarFlow to achieve D^3.

4.1 Computation Structure

CarFlow instantiates D^3 by modeling the application (e.g., an AV pipeline) as a directed operator graph where the operators represent the components (e.g., lane detection), and the edges are typed streams. A graph may be composed of multiple subgraphs representing the modules (e.g., perception), and has sources and sinks, where a source 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.

```
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. The `Planner` computes a trajectory using the obstacles and traffic lights, and specifies deadlines on its response time.

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., collocate 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 CarFlow (see Lst. 1).

The timestamp t is generated by the source operators, and 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 (§5.1), and $\hat{c}$ conveys application-specific information (§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, CarFlow 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 [52, 54].

4.2 CarFlow's API

We now provide an overview of the API with the help of a simplified `Planner` (Lst. 1) that receives the `Obstacles` and `TrafficLights` from perception through `DataMessages`. Upon the receipt of all the obstacles and lights for a timestamp t (notified by a `WatermarkMessage`), it finds a plan and returns a set of `Waypoints` i.e. fine-grained points on a map that characterize the desired plan. To register its input and output, the `Planner` implements the `TwoInOneOut` interface where the `ReadStreams` are typed by `Obstacles` and `TrafficLights`, and the `WriteStream` by `Waypoints`.

Further, each operator must implement the `on_msg` and `on_watermark` methods, that are invoked by CarFlow upon receipt of a `Data` and `WatermarkMessage` respectively (lines 13-21). Our `Planner` uses the `on_msg` callbacks (lines 13-17) to convert the coordinate system of each `Obstacle` and `TrafficLight`. However, in order to compute a safe plan, it requires all the obstacles and traffic lights, and hence, waits for a `WatermarkMessage` from both the upstream operators signifying the receipt of all incoming messages. The `Planner` 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` by CarFlow (line 11). However, in the presence of delays in the receipt of the `TrafficLights` (of more than 30 ms), it chooses to initiate the computation without them, and computes the plan using just the obstacles (line 6). §5.1 and §5.2 further elaborate on the specification and dynamic-variation of deadlines.

The deadlines are exposed to the callbacks in the `Context`, allowing them to employ strategies to meet deadlines and vary their computation accordingly (§5.3). Finally, the deadline specification also requires the registration of a handler to be invoked upon a missed deadline (`on_deadline`), that receives a `Context` containing information useful to mitigate the deadline miss (e.g., timestamp, deadline). For example, the handler could be used by the planner to output the previous computed plan offset from the AV's current location.

5 Achieving Dynamic End-to-end Deadlines

In this section, we discuss how CarFlow addresses the following challenges that are posed by the realization of D^3:

- CarFlow must quickly initiate the computation upon the availability of the input, and allow applications to bound their response time. In addition, CarFlow must allow applications to initiate their computation in the presence of incomplete input if upstream components miss their deadline $\mathcal{D}_i$ (§5.1).
- CarFlow must allow the *deadline policy* (π_{DP}), which dynamically varies $\mathcal{D}_i$, to meet strict latency constraints, owing to its presence on the critical path of the computation (§5.2).
- CarFlow must enable components to *proactively* output the highest-accuracy results possible within $\mathcal{D}_i$ (§5.3).
- In case of a missed $\mathcal{D}_i$, CarFlow must enable components to execute *reactive measures* to release output, and allow downstream components to begin their computation (§5.4).

5.1 Deadline Specification

To maximize the runtime-accuracy tradeoff and meet the end-to-end response time requirements, each component of a D^3

application must be able to: (*i*) initiate computation as soon as all its required inputs are available, (*ii*) bound its computation time i.e., the time between the receipt of the inputs and the generation of the output, and (*iii*) bound the time between the computation on successive inputs in the presence of runtime variability (**C2**). While (*i*) and (*ii*) ensure that a component adheres to its allocated response time, (*iii*) allows components to eagerly execute on incomplete input, thus ensuring that the end-to-end response time requirement is met.

To achieve these goals, CarFlow automatically captures fine-grained execution events from the application at the granularity of the logical time l. Specifically, for each logical time l of the timestamp t, CarFlow 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). This allows CarFlow to achieve (*i*) by automatically *synchronizing* data from multiple sources at the granularity of l, and initiating computation once all required inputs are available (denoted by the receipt of *watermarks* [7, 23, 70, 88] from the sources). In addition, CarFlow can register the completion of the computation for a logical time l once the watermark for t is generated.

CarFlow exposes these events to the applications, enabling them to specify *relative deadlines*, which limit the amount of wall-clock time that can elapse between any two events. Specifically, applications can register two boolean functions: *deadline start condition* (D_{SC}) and *deadline end condition* (D_{EC}), which return `True` to signify the beginning and end of the computation for a logical time l respectively. D_{SC} and D_{EC} are evaluated at the receipt and generation of every message. The functions receive a tuple $\left(n \in \mathbb{N}, w \in \{\texttt{True}, \texttt{False}\}\right)$, where n denotes the number of messages received or sent for that logical time, and w depicts the receipt or generation of the watermark. CarFlow maps a 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). CarFlow then ensures that D_{EC} is satisfied before the absolute deadline (Step 3 in Fig. 6).

Further, to simplify the enforcement of the response time requirements ((*ii*), (*iii*)), CarFlow provides the following two deadline abstractions that constrain a default set of events, and allow applications to specify a relative deadline $\mathcal{D}_i$:

Timestamp deadlines (lines 7-11 in Lst. 1) allow components to bound their computation time by constraining the wall-clock time elapsed between the receipt of the first message (M_t) timestamped with t and the generation of the first watermark ($W_{t'}$) timestamped with $t' \geq t$ to $\mathcal{D}_i$.

Frequency deadlines (lines 4-6 in Lst. 1) enable a precise initiation of computation in the presence of runtime variability by constraining the maximum wall-clock time that may elapse between the receipt of the watermark (W_t) timestamped with t and the first watermark ($W_{t'}$) for $t' > t$ to $\mathcal{D}_i$.

We emphasize that exposing these events enables applications to exercise a fine-grained control over these abstractions,

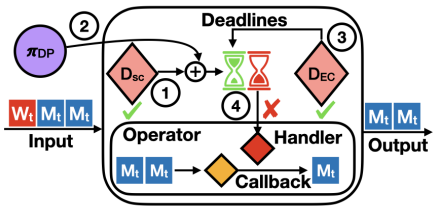


Figure 6. Environment-dependent deadlines. CarFlow evaluates D_{SC} for every message (1). If satisfied, it initiates an absolute deadline according to π_{DP} (2). Similarly, CarFlow evaluates D_{EC} upon generation of messages, and removes any satisfied deadlines (3). If a deadline is missed, CarFlow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

and provides a general model that encompasses the full spectrum of deadline constraints discussed in prior work [46]. For example, the `Planner` in Lst. 1 uses this control to define a `TimestampDeadline` constraint (lines 7-11) with a custom D_{EC}, which constrains the time between the receipt and generation of the first message timestamped with t. This constraint allows the planner to quickly release a coarse-grained plan before refining it, enabling downstream computation to begin.

5.2 Environment-Dependent Deadlines

Applications use the constructs discussed in §5.1 to specify *static deadlines* that do not evolve over time by using specific values for the time constraints (e.g., 30ms on line 6 in Lst. 1). Further, D^3 requires these constructs to support *dynamic deadlines*, which evolve according to the environment (**C1**), and are determined by a deadline policy (π_{DP}).

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. CarFlow dynamically updates absolute deadlines (see Fig. 6) by synchronizing the computation for logical time l with the corresponding relative deadline $\mathcal{D}_i$ provided by π_{DP} via the `deadlines` stream. To enable applications to meet changing deadlines (§5.3), CarFlow exposes absolute deadlines via the `Context` (§4.2).

Moreover, since π_{DP} is critical to ensuring an *accurate* and *deadline-driven* execution of the computation, its realization presents the following key design requirements to CarFlow:

Safety. The presence of π_{DP} on the critical path requires it to meet strict deadline constraints. In the event of π_{DP} missing its deadlines (due to delayed inputs), CarFlow must run a safety backup mode that performs simple maneuvers (e.g., braking or pulling over) to ensure a minimal risk condition [78].

Adaptivity. In order to reduce π_{DP}'s effect on the latency of the critical path, CarFlow 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 allocation less frequently in highways than in cities, as the environment changes infrequently.

Modularity. Modules (e.g., perception) may use local knowledge to specify policies that more efficiently split a deadline across their components (e.g., detection, tracking) than a centralized policy. Thus, CarFlow must enable the decomposition

of monolithic policies such that high-level policies provide coarser-grained deadlines to module-specific policies, which further decompose them across their components.

To achieve these goals, CarFlow executes π_{DP} as a subgraph of operators which receive information about the environment from components on the input streams. π_{DP} processes this information to compute an end-to-end deadline $\mathcal{D}$ which is decomposed into per-component deadlines $\mathcal{D}_i$, and sent to components via the output streams. Executing π_{DP} as a subgraph enables the policy to exploit CarFlow's 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 §5.3).

Moreover, π_{DP} can use CarFlow's timestamping and deadline mechanisms to achieve *adaptivity* and *safety*. 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 use CarFlow's static deadlines 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

CarFlow exposes the per-component deadline $\mathcal{D}_i$ from π_{DP} to the operators via the `Context` (lines 13, 18 in Lst. 1). The operators use $\mathcal{D}_i$ to *proactively* satisfy D_{EC} before it expires by releasing intermediate, lower-accuracy results to downstream operators using the strategies detailed below:

Executing anytime algorithms [57, 93, 95] that maximize the accuracy for a given $\mathcal{D}_i$ through iterative refinement [97], and provide a continuous runtime-accuracy tradeoff curve by monotonically increasing accuracy with increasing deadlines.

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 [56, 83] (see §2).

Executing multiple versions of components to ensure that at least one completes before $\mathcal{D}_i$ expires (similar to [81]). 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 $\mathcal{D}_i$. Alike *anytime algorithms*, components must be able to choose to release the lower-accuracy results to *speculatively execute* 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 $\mathcal{D}_i$. This strategy differs from load shedding [40, 85, 86] since AV operators can provide reduced accuracy by *amending* prior results and releasing them to eagerly execute downstream computation. For example, the `Planner` in Lst. 1 can release its last computed plan offset to the current location of the AV.

CarFlow allows applications to specify multiple implementations of components along with their *typical* runtimes as different operators, which are automatically used by CarFlow to either change the implementation, execute multiple versions or skip the execution based on $\mathcal{D}_i$. While the algorithms to *proactively* meet $\mathcal{D}_i$ are application-specific, CarFlow provides two mechanisms to enable their efficient realization:

Intermediate Results. CarFlow's novel extension of timestamps (§4.1) provides first-class support for anytime algorithms and the *speculative execution* of multiple versions. Specifically, operators annotate output messages with increasing values of $\hat{c}$ to notify downstream operators when more accurate results are available, while enabling *speculative execution* using less accurate intermediate results. CarFlow uses $\hat{c}$ to prioritize the processing of more accurate results, ensuring high accuracy in downstream operators.

For example, the `Planner` can first output a coarse-grained plan by using an anytime algorithm [57, 93], and annotate its accuracy using $t_1 = (l, \hat{c}_1)$. This allows the downstream control operator to generate vehicle commands, and refine them after a fine-grained plan is computed by the `Planner`. CarFlow automatically eschews the execution of the control operator with a coarse-grain plan tagged with t_1 in favor of the fine-grained plan tagged with $t_2 = (l, \hat{c}_2)$, where $\hat{c}_2 > \hat{c}_1$.

State Management. CarFlow achieves an efficient execution of different algorithms for successive timestamps (either via changing the implementation or executing multiple versions) by decoupling the state from the implementation of the component. Specifically, a component must register its state (e.g., `PlanningState` in Lst. 1) with CarFlow, which is automatically provided to the callbacks (lines 14, 19, 23 in Lst. 1). By assuming control of the state, CarFlow can automatically share it across operators that may implement different algorithms, and enable the *parallel* execution of multiple versions by providing each operator with a view of the state without requiring operators to synchronize updates to the state. We further discuss CarFlow's system-managed state in §5.4.

5.4 Handling Deadline Misses

Despite the proactive measures, an operator may fail to satisfy $\mathcal{D}_i$ due to **C2**. In order to ensure that a missed $\mathcal{D}_i$ does not delay downstream operators and lead to missed end-to-end deadlines, D^3 enables components to specify *reactive measures*, whose execution presents the following requirements:

Fast Invocation of the measures upon the expiration of the deadline $\mathcal{D}_i$ to unlock downstream computation and minimize the reduction of available time for downstream operators.

Access to the state of the component managed by the system (§5.3) to *quickly* compute relevant results, and ensure its consistency for executions of future timestamps.

Orchestration of the measures and the computation to enable components to eagerly execute downstream operators with lower-accuracy results while using the higher-accuracy computation for state updates. For example, a `Planner` might

miss $\mathcal{D}_i$ while running a higher-accuracy algorithm, and choose to 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.

CarFlow achieves these goals by allowing components to specify reactive measures through *deadline exception handlers* (D_{EH}) (lines 22-25 in Lst. 1). A D_{EH} is invoked if an operator misses a deadline, and completes quickly to minimize the delay of downstream operators. The system orchestrates the execution of a D_{EH} and of *proactive measures* by providing the following execution policies:

Abort. Terminates the execution of the proactive measures for the timestamp t, and requires D_{EH} to complete the computation by sending a watermark W_t committing consistent state. To achieve this, D_{EH} receives a view of the state at $t' < t$, and state changes made by the computation for timestamp t. D_{EH} uses both views to quickly release output. For example, a D_{EH} in a `Planner` could either use the state at t to output and commit the best plan found by the deadline if the operator is anytime, or amend the plan computed for t' otherwise.

Continue. Allows the execution of the proactive measures for the timestamp t to continue in parallel with the D_{EH}. The latter enables eager execution of the downstream components by releasing an output for t (M_t), while the former completes the computation for t (W_t) and ensures state consistency. To achieve this, D_{EH} receives a view of only the state at $t' < t$, and can use it to execute a fast, low-accuracy algorithm. For example, a D_{EH} in a `Planner` can amend the plan computed for t', while the proactive measure releases and saves a more-accurate plan, allowing downstream operators to use more accurate results if possible. Moreover, allowing the proactive measures to commit the state for t enables the computation at timestamps $t'' > t$ to use the high-accuracy results, and thus prevent a cascade of low-accuracy results across time.

These policies are enabled by CarFlow's *system-managed state* and release of *intermediate results* (§5.3). By decoupling the state from the execution of the proactive measures, CarFlow ensures a parallel execution of such measures with a D_{EH}. Moreover, the D_{EH} can communicate the lower accuracy of its *intermediate results* to the downstream operators, and enable these operators to refine their computation using higher-accuracy results when they become available.

Furthermore, to enable a *fast invocation* of D_{EH}, CarFlow's *system-managed state* exhibits the following two properties:

Transactional Semantics. *Abort*ing the proactive measures requires that any mutations made to the state during their execution are rolled back. To ensure this, CarFlow provides a view of the current state to such measures, and automatically commits any mutations made by them upon the successful release of the watermark for the currently executing timestamp. In the case of a missed deadline, CarFlow provides the dirty, uncommitted state to the D_{EH} along with a view of the last committed state, and allows the D_{EH} to commit the state and hence ensure its consistency for future executions.

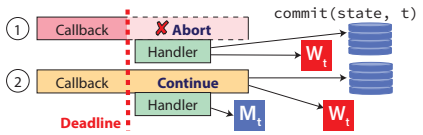


Figure 7. Handling missed deadlines. When a deadline is missed, handlers are invoked to mitigate the consequences. Callbacks which miss their deadline may **Abort** to let the handler rapidly update operator state, or **Continue** to ensure more accurate state updates.

Time-Versioning. In order to *Continue* the execution of the proactive measures along with a D_{EH}, CarFlow maintains a version of the state for each logical time l. In case a deadline is missed for $t = (l, \hat{c})$, the D_{EH} gets read access to the *committed* state for all timestamps $t' < t$ and can send M_t to start the eager execution of downstream computation. The proactive measures can continue in parallel for timestamps $t'' \geq t$ and commit state mutations by releasing W_t.

Moreover, CarFlow allows components to provide their own state abstractions by implementing a `StateT` interface that requires them to provide a custom `commit` implementation, along with a way to retrieve a view of the state at time t using `get_committed`. Thus, components can choose to customize both the time-versioning and the transactional semantics of their state using techniques such as CRDTs [77]. For example, the `Planner` could implement the `StateT` interface for `PlanningState`, instead of using the `State` provided by CarFlow In such a case, the `PlanningState` could maintain a vector of waypoints for timestamp $t = 0$, and log additions of future waypoints in its `commit` method, instead of saving the entire set of waypoints for each timestamp $t' > t$.

6 CarFlow's Implementation

CarFlow is an open-source distributed system implemented in 13k lines of Rust, whose type safety and memory semantics are essential for safety-critical applications. Further, to interact with ML frameworks [19] and enable prototyping with simulators [50], CarFlow provides a Python interface.

CarFlow's distributed nature consists of a master-worker architecture where the master manages a set of worker processes running across several machines. The master partitions the execution graph and assigns operators to workers, which are responsible for exchanging data along streams (§6.1), executing callbacks (§6.2), and managing the arming and disarming of deadlines (§6.3). We choose the master-worker architecture due to its implementation simplicity, and ensure its scalability by keeping the master off the critical path.

6.1 Communication

CarFlow's communication subsystem ensures a *rapid* transmission of messages among operators. To achieve this, each worker (executing many operators) constructs a *data plane* by initiating TCP sessions with other workers, and a *control plane* by connecting to the master. While the *data plane* manages messages sent among operators, the master uses the

control plane to assign operators to workers and synchronize operator initialization, thus minimizing execution delays.

CarFlow 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 TCP *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

To enable the execution of CarFlow operators, each worker maintains 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 creates the a view of the state (see §5.3), and inserts into the lattice a *bound callback*, consisting of the state, a `Context`, a 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 automatically commits the state upon completion.

This *execution lattice* serves as a run queue for a worker's multi-threaded runtime. A set of threads retrieve and execute the *bound callbacks*, and notify the lattice upon their completion to unlock further dependencies (e.g., callbacks with higher timestamps). We emphasize that the ordering semantics of the callbacks in a lattice enable applications to fine-tune the parallelism and state-management per operator. For example, an operator may choose to manually synchronize updates to its state, and ask CarFlow to execute all its callbacks in parallel by overriding the ordering of the callbacks.

6.3 Deadline Management

A worker's runtime also *arms* and *disarms* the deadlines specified by its assigned operators. To *arm* a deadline, the worker maintains per-stream statistics about the receipt of messages and the watermark status for each timestamp 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 wall-clock time at which it expires. These deadlines, along with their exception handlers (D_{EH}), are maintained by the worker in a priority queue ordered by the absolute deadline.

A deadline is *disarmed* either when the operator satisfies D_{EC} or misses the deadline. Similar to above, a worker maintains per-stream statistics of the transmission of messages and watermarks, and removes the deadline and the D_{EH} from the queue upon satisfaction of D_{EC}. Further, the worker polls the queue, and invokes the D_{EH} according to either the *abort* [37] or *continue* policy upon the expiration of a deadline.

7 Evaluation

We now present Herbie, our state-of-the-art AV pipeline that achieves the top score on the map track of a simulated AV challenge and drives real AVs [2]. We use Herbie to evaluate D^3 and CarFlow, and seek to answer:

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

7.1 Herbie: an AV Development Platform

The construction of Herbie 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]. In addition, we have extended the CARLA challenge [14] to construct a benchmark for AV systems that we use here in our evaluation. While a typical deployment of Herbie contains dozens of components (Fig. 1), we briefly describe a few of the components that are important to our evaluation (see [1] for an in-depth discussion). To the best of our knowledge, this is the most complete open-source pipeline with trained models for autonomous driving.

Herbie's perception module comprises of components that perceive objects, lanes, and traffic lights using multiple cameras. While Herbie provides several implementations for each component (suited for different driving environments), our experiments use EDet2 to EDet6 from the EfficientDet family [83] which enables a runtime-accuracy tradeoff (§2.1).

The Herbie planning component contains implementations of Hybrid A* [49], RRT* [57] and FOT [93, 95] that perform best under different driving scenarios [59, 61, 72]. 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 0.3 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.

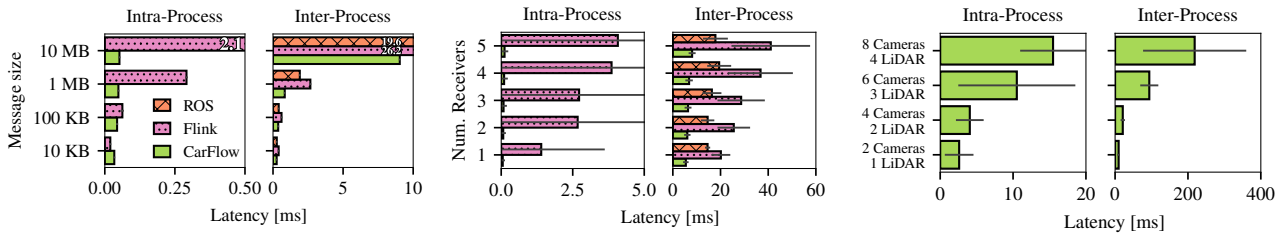
We study CarFlow's realization of D^3's *dynamic deadlines* by comparing Herbie's performance under dynamic deadlines to five static deadlines ranging from 125ms to 500ms. For each deadline, we chose the configuration that performs best.

Test Environment. All experiments were performed 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.

7.2 CarFlow's Performance vs. Other Systems

We evaluate the latency of CarFlow with respect to message size, broadcast span, and pipeline complexity. We compare against ROS, the widely used platform for AVs [21, 28, 51, 90], and Flink [39] a data-driven streaming system that is

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



(a) Message Latency **(b)** Broadcast Latency **(c)** Sensor Scaling

Figure 8. Messaging Performance. We evaluate the message latency for **(a)** varying message sizes, **(b)** operator fanout, and **(c)** pipeline sizes for intra-process and inter-process communication. In all cases, we find that CarFlow's tightly integrated design results in reduced latency.

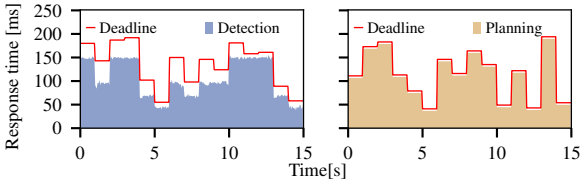


Figure 9. Meeting Deadlines. We vary the deadline every second and show how the modules respond to the new deadlines. Both the detection and planning modules adapt to meet the deadline and the more adaptive planning module is better at using its time allotment.

closest to CarFlow due to its operator-centric programming model and usage of watermarks for unlocking computation.

We measure the latency of sending messages of increasing sizes between two operators and invoking a callback upon receipt of the message. We send messages at 30Hz, the frequency at which AVs process data [2]. Fig. 8a shows the results across both intra-process and inter-process placement of the operators. CarFlow's intra-process communication latency remains constant across message sizes due to its zero-copy communication. Further, CarFlow's inter-process implementation performs 2.3× better than ROS, and 3.2× better than Flink when sending 1MB messages. We analyze the systems to attribute the latency 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.

Broadcasting the output of an operator (e.g., image calibration) to multiple operators (e.g., perception) is a common pattern in AVs. In Fig. 8b, we measure CarFlow's latency when broadcasting a typical camera image (1MB) to a varying number of operators. CarFlow sends a message to 5 operators in the same process at a median latency of 0.11 ms, 35× faster than Flink. When communicating across processes, CarFlow broadcasts to 5 operators at a median latency of 8.08 ms, which is 2.2× and 5.1× faster than ROS and Flink.

We emulate Herbie with an increasing number of sensors sending data at 10Hz. We first instrument Herbie, which has a camera and a LiDAR, and retrieve the mean size of each message type. Based on these measurements, we emulate a pipeline with an increasing number of sensors, which sends messages totalling 1.25GB/s across 60 operators. Moreover, for a worst-case estimate, we assume each component has 0 runtime. Fig. 8c compares the end-to-end latency incurred by the pipeline when executed within a process and across processes. We notice a sharp increase in the runtime of the pipeline when executed across processes with 8 cameras and 4 LiDARs. This increase is due to the time spent marshalling and unmarshalling of the messages under reduced resources as 12 out of 48 PUs are occupied by the sensors generating the data. Note that a realistic deployment of Herbie would co-locate operators in processes, and thus the worst-case latency would be similar to that observed in the intra-process graph.

Takeaway: *CarFlow's efficient implementation significantly reduces latency and scales to large pipelines.*

7.3 Efficacy of CarFlow's Deadline Mechanisms

We evaluate the latency 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-op* π_{DP} that receives data from Herbie's components and sends static deadlines to the components. We measure Herbie's response time without and with the no-op π_{DP} during a 35km drive, and we find that the policy mechanism increases the median and 90th percentile response time by 0.9ms and 2.3ms respectively.

Meeting Deadlines. We evaluate CarFlow's support for fine-grained changes in deadline allocations (§5.3) and if Herbie's components can 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 Herbie's detection and planning during a short drive. We observe that while detection meets its deadline, it fails to utilize the its entire time quota. This is because the EfficientDet [83] family provides 8 models with different runtimes, and CarFlow 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 quota because it executes an anytime algorithm [93, 95].

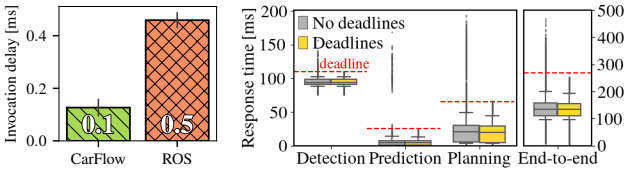


Figure 10. Impact of Deadline Exception Handlers. CarFlow supports fast invocation of handlers (left), and enables Herbie to quickly react to deadline misses (right), ensuring timely responses.

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

Takeaway: *CarFlow implements D^3 by swiftly executing π_{DP}, enabling proactive measures to meet deadlines, and rapidly taking reactive measures when deadlines are missed.*

7.4 Efficacy of the D^3 Execution Model

We evaluate the efficacy of D^3 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 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.

7.4.1 Aggregate Study. We explore if our policy adjusts the deadlines to avoid collisions during a *challenging* 50km CARLA Challenge drive [14]. 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 needs 100 days of simulation).

Fig. 11 shows results of four configurations: (*i*) a periodic execution of Herbie derived from WCET estimates (similar to Apollo [2]), (*ii*) the best data-driven configuration (similar to Autoware [28]), (*iii*) the best configuration with static deadlines, and (*iv*) Herbie with dynamic deadlines computed by our policy. Our policy reduces the number of collisions by 68% over a periodic execution, and by 26% over the best

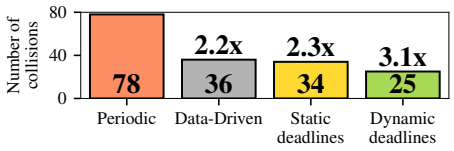


Figure 11. CarFlow Reduces Collisions. In a challenging 50km drive, the dynamic deadlines enabled by CarFlow reduce collisions by 68% over solutions using the periodic execution model.

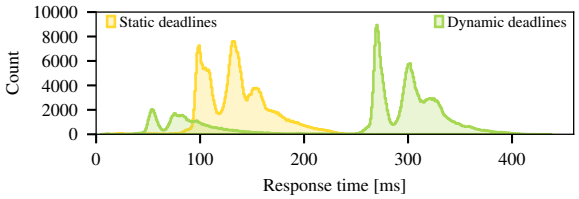


Figure 12. Response Time Histogram. The static configuration is tuned to perform the best during the drive and the variability is due to **C2**. By contrast, the dynamic configuration offers faster responses when needed, and executes more accurate computation otherwise.

configuration with static deadlines because the policy reduces the response time in challenging scenarios. Finally, we compare the end-to-end response times of Herbie with dynamic deadlines with the best configuration with static deadlines. Fig. 12 shows that in most situations Herbie with dynamic deadlines runs a slow, high-accuracy configuration, but adapts to fast configurations when the environment demands it.

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

7.4.2 In-depth Study of Scenarios We study the benefits of dynamic deadlines using two challenging scenarios that require the AV to adapt deadlines to avoid collisions.

Person Behind Truck. This scenario simulates a person illegally entering the AV's lane [3]. Because the person is occluded behind a truck until they enter the lane, the AV cannot stop in time and must perform an emergency swerving maneuver.

Traffic Jam. This scenario simulates merging into a traffic jam. We require the AV 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 set Herbie to drive at a fixed *target speed* and execute the five static configurations (see §7.1) and the dynamic configuration with 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. Note that a collision speed of 0m/s shows that Herbie avoided a collision.

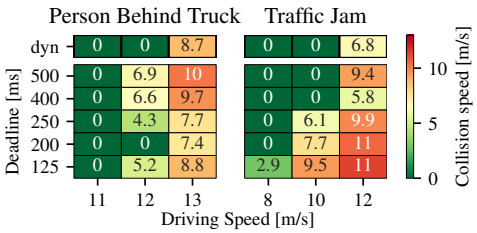


Figure 13. Versatility of Dynamic Deadlines. Using **dyn**amic deadlines, Herbie adapts to scenarios resulting in fewer collisions.

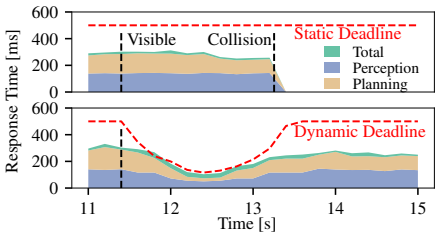


Figure 14. Adapting to Deadlines. Herbie's components adapt to meet dynamically-varying deadlines and avoid a collision.

In Fig. 13 we plot the collision speed across varying target speeds. As expected, at a target 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 pedestrian, which is first 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 motorcycle is less visible, and thus faster, less-accurate detection models perform poorly. Fig. 14 shows how Herbie adapts as the end-to-end deadline changes once the person is detected in the *person behind truck* scenario.

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

8 Related Work

AV Systems. Vendors [21, 28, 51, 89, 90] are developing AVs atop robotics platforms that provide a modular design and best-effort execution of the stages (e.g., ROS [73], ROS2 [55], CyberRT [30]). However, 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 progression of time [90]. As a result, vendors implement AVs as SLO-based best-effort systems that attempt to meet an environment-agnostic end-to-end deadline [28, 29, 90]. Vendors deploy AVs as ROS/CyberRT processes, that either execute each component to completion which may delay downstream components due to runtime variability, or run components at a fixed frequency [2, 11] which preclude adaptations to meet dynamic deadlines. In contrast, CarFlow enables the implementation of D^3 applications by offering a consistent view of time via logical times, an automatic mapping of logical time to wall-clock time with which components can reason about deadlines and available execution time, and the ability to perform reactive measures to mitigate missed deadlines.

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

Cyber-Physical Systems. Hard real-time systems conduct schedulability analyses driven by WCET estimates to guarantee that deadline constraints are met [32, 36, 48, 53, 64, 66, 87]. However, AV components preclude the accurate estimation of WCETs due to environment-dependent runtimes and large input spaces [24, 25, 82], Thus, developing AVs as such systems requires use of *conservative* WCETs to derive the periodicity of execution for each component [47]. 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, 43]. To address the former, real-time applications implement *mode changes* [26, 42, 74], which depend on WCETs to verify if transitions between modes lead to deadline misses [64, 74, 80, 87]. By contrast, *adaptive real-time systems* [35, 60, 67, 76] 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.

Streaming Systems. Flink [39], Cloud Dataflow [7], MillWheel [22], and Naiad [70] inspired elements of our design (e.g., logical time [7, 39, 70], watermarks [23, 39, 88], early results [18]). However, 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 D^3 execution model because, unlike CarFlow, they lack APIs to specify environment-dependent deadlines, and thus do not provide mechanisms to *proactively* meet these deadlines, and *reactively* adapt to deadline misses.

9 Conclusions

We highlight two key characteristics of AVs, and introduce D^3, an execution model for applications that must maximize accuracy in the presence of dynamic deadlines. We realize D^3 in CarFlow, atop which we build an AV, and find that D^3 reduces collisions by 68%. We hope that our artifacts will aid the development of safer AVs, and inspire systems research.

References

[1] Anonymized publication. To be specified on publication.

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

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

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

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

[6] Companies Have Spent Over $16 Billion on Robocars. It's A Drop in the Bucket. https://www.forbes.com/sites/bradtempleton/2020/02/18/companies-have-spent-over-16b-on-robocars--its-a-drop-in-the-bucket.

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

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

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

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

[11] Planning Loop Rate in Autoware AV. https://github.com/Autoware-AI/core_planning/blob/master/freespace_planner/src/astar_navi/astar_navi.cpp#L98.

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

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

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

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

[16] To Make Self-Driving Cars Safe, We Also Need Better Roads and Infrastructure. https://hbr.org/2018/08/to-make-self-driving-cars-safe-we-also-need-better-roads-and-infrastructure.

[17] Weather Creates Challenges For Next Generation Of Vehicles. https://www.forbes.com/sites/jimfoerster/2019/11/22/weather-creates-challenges-for-next-generation-of-vehicles/?sh=30845aa1260e.

[18] 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.

[19] 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.

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

[21] 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.

[22] 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.

[23] 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*, 2021.

[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://github.com/CPFL/Autoware-Manuals/blob/master/en/Autoware_UsersManual_v1.1.md.

[29] Baidu. Apollo 3.0 Software Architecture. https://github.com/ApolloAuto/apollo/blob/master/docs/specs/Apollo_3.0_Software_Architecture.md.

[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] 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.

[44] 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.

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

[46] 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.

[47] 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.

[48] 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.

[49] 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.

[50] 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.

[51] 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.

[52] 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.

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

[54] 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.

[55] 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.

[56] 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.

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

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

[59] 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.

[60] 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.

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

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

[63] 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.

[64] 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.

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

[66] 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.

[67] 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.

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

[69] Michele Bertoncello, and Dominik Wee. Ten Ways Autonomous Driving Could Redefine the Automotive World. https://www.mckinsey.com/industries/automotive-and-assembly/our-insights/ten-ways-autonomous-driving-could-redefine-the-automotive-world.

[70] 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.

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

[72] 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.

[73] 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.

[74] 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.

[75] 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.

[76] 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.

[77] 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.

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

[79] 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.

[80] 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.

[81] 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.

[82] 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.

[83] 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.

[84] 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.

[85] 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.

[86] 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.

[87] 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.

[88] 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.

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

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

[91] 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.

[92] 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.

[93] 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.

[94] 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.

[95] 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.

[96] 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.

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