SADE-SIM: A Scalable Simulation Platform for Validating City-Scale Multi-sUAS Missions

Laxminarayana Vadnala
lvadnala@nd.edu
University of Notre Dame
Notre Dame, Indiana, USA

Lucas Parzianello
lparzianello@nd.edu
University of Notre Dame
Notre Dame, Indiana, USA

Bohan Zhang
bohan.zhang.1@slu.edu
Saint Louis University
Saint Louis, Missouri, USA

Michael Murphy
mmurph51@nd.edu
University of Notre Dame
Notre Dame, Indiana, USA

Ankit Agrawal
ankit.agrawal.1@slu.edu
Saint Louis University
Saint Louis, Missouri, USA

Douglas Thain
dthain@nd.edu
University of Notre Dame
Notre Dame, Indiana, USA

Abstract

Validating autonomous small Unmanned Aerial Systems (sUAS) at city scale requires simulation frameworks that support large multi-sUAS deployments while maintaining both physical accuracy and geospatial realism. Existing systems typically support few concurrent sUAS and lack fidelity for evaluating vision-based navigation, sensor degradation in urban environments, and compliance with evolving airspace regulations. We introduce SADE-SIM: a scalable hybrid architecture that couples physical simulation with photorealistic visualization. The SADE-SIM integrates web-based mission planning, real-time control compatible with standard protocols, live visualization of video stream, and archival review of flight data. We demonstrate a case study using a 32-sUAS heterogeneous fleet across one restricted airspace zones with diverse operational profiles, where sUAS dynamically request zone entry permissions and the system evaluates mission-critical parameters before granting or denying access. SADE-SIM will enable novel validation workflows such as the evaluation of autonomous navigation systems, GPS-degraded navigation testing in metropolitan canyons, and validation of the FAA compliance protocol for dynamic airspace management. By bridging flight dynamics, photorealistic environments, and regulatory constraints, SADE-SIM advances the research computing infrastructure needed for safe deployment of autonomous aerial systems at scale.

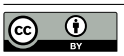
CCS Concepts

• **Computer systems organization** → **Embedded and cyber-physical systems**; • **Computing methodologies** → **Simulation evaluation**; **Simulation environments**; *Simulation tools*; • **Software and its engineering** → Virtual worlds training simulations.

Keywords

Simulation, Automated Analysis, Unmanned Aerial Vehicles, Drones

ACM Reference Format:
Laxminarayana Vadnala, Lucas Parzianello, Bohan Zhang, Michael Murphy, Ankit Agrawal, and Douglas Thain. 2026. SADE-SIM: A Scalable Simulation Platform for Validating City-Scale Multi-sUAS Missions. In *34th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE Companion '26), July 5–9, 2026, Montreal, QC, Canada.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3803437.3805540



FSE Companion '26, Montreal, QC, Canada

ACM ISBN 979-8-4007-2636-1/2026/07
https://doi.org/10.1145/3803437.3805540

1 Introduction

Autonomous small Uncrewed Aerial Systems (sUAS) are increasingly deployed in missions that span large geographic areas, including disaster response, infrastructure inspection, and public safety operations. As these systems move from prototypes to operational deployments, sUAS developers are expected to deliver repeatable, reviewable evidence that sUAS autonomy remains safe and compliant under changing environmental conditions and evolving airspace rules. Simulation testing is a primary mechanism for generating this evidence during early development phases because large-scale field testing is costly, risky, and difficult to control [2].

In practice, generating such validation evidence remains a challenge with current simulation workflows [1]. Most simulation platforms such as Flightmare [18], FlightGoggles [8], AirSim [17] are designed for performing sUAS mission validation activities on a single workstation, where rendering, physics, and software orchestration are tightly coupled within one execution node. As simulation scenarios scale to larger environments and fleets, this coupling becomes a bottleneck that restricts the coverage of various test scenarios. Developers often compensate through vertical scaling (write custom scripts or build new tools) or make test scenarios simpler by adding assumptions, both of which introduce operational overhead and reduce reproducibility. Moreover, validation artifacts such as configurations, traces, and logs are not consistently structured for systematic reuse, making regression testing across sUAS autonomy versions difficult to integrate into continuous testing workflows.

Beyond scalability, current simulation platforms rarely treat mission context as a first-class component of the validation process. Operational constraints, such as airspace restrictions and geofencing, are typically managed outside the primary simulation loop rather than natively integrated into the environment. This decoupling makes it difficult to rigorously evaluate whether sUAS autonomy software can successfully comply with airspace management controls at runtime. Consequently, existing scenario-driven workflows struggle to capture the true operational complexity these systems will face upon real-world deployment.

This paper presents a software architecture for an sUAS simulation platform, SADE-SIM, designed to address these limitations. The architecture distributes physics and environment rendering workloads, provides standardized interfaces for connecting systems under test to simulation services, supports explicit specification of airspace management constraints, and manages simulation infrastructure as an integrated service. Together, these capabilities enable structured, repeatable execution of large-scale workflows. The platform is realized as a cloud-hosted environment that allows developers to treat simulation as an on-demand service for autonomy testing. We evaluate the architecture through a case study demonstrating its ability to support complex scenarios while remaining modular and extensible for evolving validation needs.

2 Development Objectives

Drawing on our past experiences of utilizing multiple sUAS simulation tools to evaluate industrial-grade sUAS for emergency response missions, and following a design science approach, we identify the following core development objectives that a simulation platform must satisfy. These objectives reflect recurring limitations observed in practice and capture the capabilities needed to support rigorous, scalable autonomy validation.

DO1: Distribute Physics and Rendering Workload: Evaluating multi-sUAS fleets requires simulation that stays realistic and scalable as fleet size and environmental complexity increases. The architecture should distribute physics and environmental rendering workloads across multiple nodes while maintaining a consistent shared world state. The design of the simulator should also be modular so that environmental factors can be modeled and calibrated to match mission requirements. In particular, it must support integrating (1) wind-field models to assess stability under steady and gusty conditions, (2) GNSS models to assess robustness in urban settings under signal occlusion, multi-path, and dropouts and (3) Geospatial models of real-world terrain and built infrastructure to evaluate sUAS navigation and collision avoidance capabilities in realistic settings. We prioritize these factors because environment-driven disturbances and sensing failures are frequent contributors to real-world sUAS incidents [21]. Finally, the architecture should be extensible, with a clear path to incorporate additional environmental-factors such as foggy weather conditions to validate a broad range of sUAS missions in different conditions.

DO2: Custom Software Pilots: Autonomous sUAS include a mission-specific software layer responsible for high-level decision making based on sensor inputs, AI models, sUAS goals, and mission configuration. This layer directly communicates with the underlying flight controller (*e.g.*, PX4[13] or ArduPilot[19]). In this paper, we focus on testing this layer and refer to it as the Custom Software Pilot for the remainder of the paper. Accordingly, the simulation tool must support standard communication and data-exchange interfaces between Custom Software Pilots and flight controllers. This lowers the barriers for large-scale mission evaluation because developers can focus on implementing the core logic of the Custom Software Pilot and evaluate it in simulation while minimizing pilot–simulator integration overhead.

DO3: Airspace Management: Most simulation platforms prioritize modeling vehicle dynamics, which is essential for physical correctness and for comparing simulation outcomes with real-world behavior. However, autonomy validation also depends on the regulatory context in which a mission executes. From a software engineering perspective, tests that omit airspace rules provide limited evidence about whether autonomy will behave safely and remain compliant in deployment. In the autonomous driving domain, CARLA [5] captures traffic rules and right-of-way constraints through entities such as simulated traffic lights and stop signs. Similarly, sUAS validation requires explicit support for airspace constraints, including no-fly zones, and regulator-controlled zones that the autonomy must respect. The simulation platform should represent these constraints and generate reviewable evidence to enable compliance-aware validation of autonomous behavior.

DO4: Automated Simulation Infrastructure Management: End-to-end sUAS simulation workflows depend on a coordinated set of software and infrastructure components, including simulators, flight-controller stacks, communication protocol and libraries, networking, data capture, and observability. When these components are selected, configured, and orchestrated manually, testing becomes time-consuming to set up, brittle to maintain, and hard to reproduce across runs and teams. Therefore, the simulation platform should provide automated infrastructure management that configures the complete workflow from a single specification, executes runs with minimal manual intervention, persist simulation logs for debugging and analysis, and performs reliable tear-down and cleanup to reset the environment for subsequent runs.

3 SADE-SIM

3.1 Simulation Infrastructure

Figure 1 shows the conceptual architecture of SADE-SIM. The platform is designed to simulate multiple sUAS operating concurrently in a shared airspace, each behaving as an independent entity with its own control systems and decision-making capabilities. Each sUAS consists of three fundamental components: (1) a **flight controller** that manages low-level vehicle dynamics and stabilization, (2) a **software pilot** that executes high-level mission logic and autonomous behaviors, and (3) a **sensor interface** that provides the sUAS with perception of its environment. These components

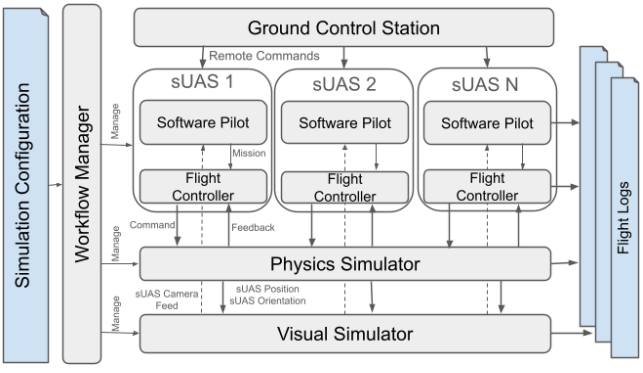


Figure 1: SADE-SIM Conceptual Architecture

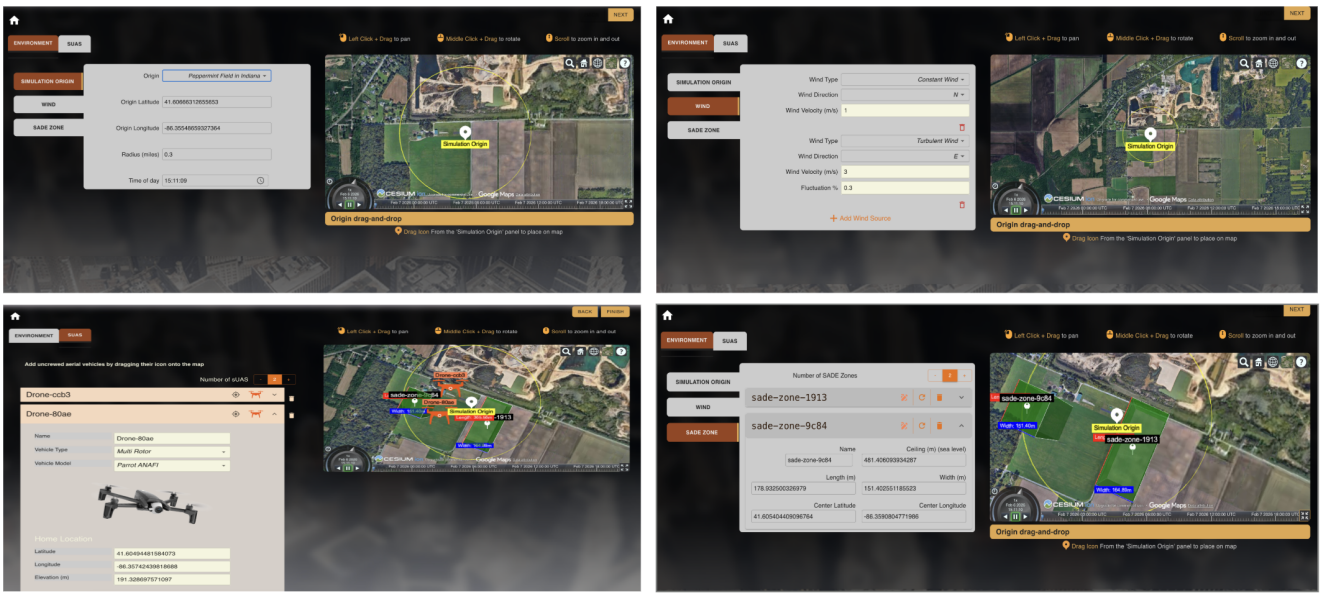


Figure 2: Example of SADE-SIM Front-End Configuration Flow

form the foundational capabilities required in a simulation platform for sUAS validation. Now, we describe the core design decisions we made to meet our development objectives.

First, our simulation infrastructure is composed of two complementary engines working in tandem. A **physics simulator** computes rigid-body dynamics, aerodynamic forces, sensor measurements, and collision detection; providing the flight controller with realistic feedback about the drone's physical state. In parallel, a **visual simulator** generates a three-dimensional rendering of the high-fidelity environment, producing camera feeds and visual data that can be consumed by onboard computer vision algorithms or human operators monitoring the simulation. This architectural decision allows us to satisfy **DO1**.

Second, the infrastructure includes a stable version of a open-source flight controller and a custom software pilot that sUAS developers can integrate into the platform for testing (discussed in more detail in Section X). We made this design choice so developers do not need to manage the complexities of flight controllers and can focus primarily on designing and evaluating their custom software pilot. These core components are also connected to a ground control station that serves as the primary interface for developer observations. These design choices aim to fulfill **DO2**

Third, a critical design choice in our infrastructure is *configuration driven simulation execution*. Each simulation run is defined by a single declarative configuration artifact that specifies all information needed to initialize the simulation, including the number and types of sUAS, their home positions, and environmental conditions. We extend this configuration with SADE Zone definitions, allowing developers to specify the airspace restrictions to be enforced during execution. This design supports **DO3** and provides greater control to automatically manage the infrastructure resources based on simulation needs.

Finally, the infrastructure includes a *simulation workflow manager* that orchestrates different components of the infrastructure based on the developer-provided simulation configuration. It provisions and scales resources according to each run's requirements, coordinates end-to-end execution (starts, monitors, and terminates runs), and maintains traceability by tracking data and metadata across all infrastructure components. This design directly addresses **DO4** and is essential for large-scale sUAS simulation, where resource demands vary across scenarios and multiple developers must reliably schedule and run experiments concurrently.

3.2 Simulation Technologies

To realize the conceptual architecture described above, SADE-SIM is implemented as two complementary infrastructure layers: a static infrastructure layer that supports simulation configuration and lifecycle management, and a dynamic infrastructure layer that provisions and executes individual simulation instances. This separation allows simulations to be reproducible, shareable, and archivable, while also enabling scalable execution across multiple concurrent runs. Figure 3 shows the technical architecture.

Static Infrastructure: This layer consists of the continuously running components needed to configure simulations and support all running workflows. The web front-end is implemented using React [14] and CesiumJS [4], allowing sUAS developers to interactively define simulation scenarios. Through this interface, as shown in Figure 2, developers can configure (1) *Simulation Origin* that maps to a real-world geo-location (2) **Environmental Conditions** such as windy conditions (3) sUAS type (4) SADE Zone for sUAS autonomy validation. These configurations are captured in a single simulation configuration document represented as a structured JSON artifact. This configuration artifact is transmitted to the back-end server

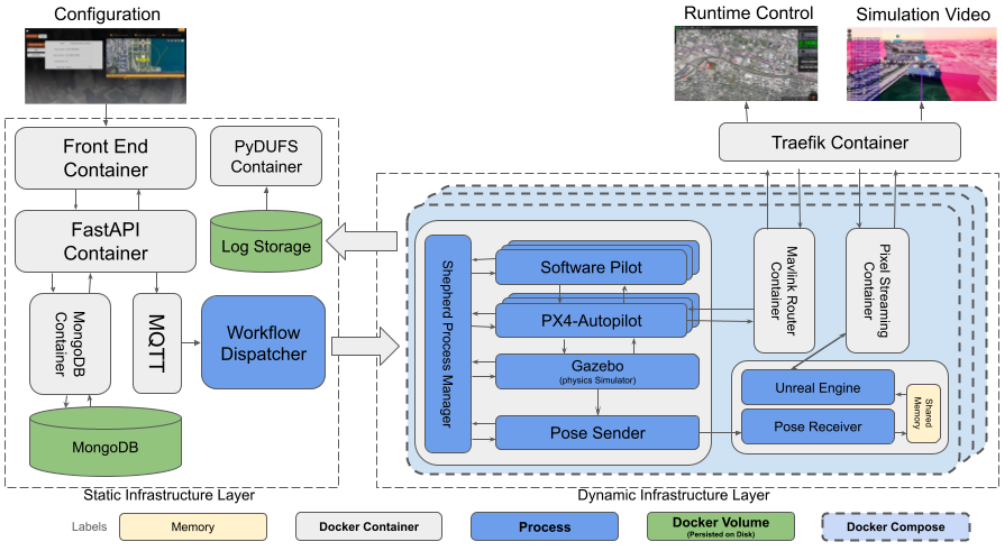


Figure 3: SADE-SIM Technical Architecture

The static infrastructure *(left) consists of the continuously running components needed to provide an interactive web front-end used to generate a simulation configuration, The* dynamic infrastructure *(right) consists of the components needed to run a single instance of a multi-sUAS simulation using PX4 flight controllers, Gazebo for physical simulation, and Unreal for visual simulation.*

(FastAPI), preserved in MongoDB, and then asynchronously communicated to the workflow dispatcher using an MQTT message broker.

Dynamic Infrastructure: This layer consists of all the components needed to run a single instance of a multi-drone simulation. The *workflow dispatcher* is a dedicated process designed to orchestrate the end-to-end simulation life cycle. It functions as the core of SADE-SIM, managing resource validation, dynamic network synthesis, and runtime state transitions. For each simulation configuration received, the workflow dispatcher creates a simulation instance by generating a Docker Compose workflow describing all the components as containers to be materialized. The workflow dispatcher starts the workflow, monitors its status, and reports its status back to the web front-end. This allows developers to continuously monitor the simulation progress (*e.g.*, Built, Started, Completed, Aborted). Due to the isolation provided by containers, multiple simulation instances may run simultaneously if sufficient computing resources are available.

Software Pilot: Each simulated sUAS consists of a Python-based automated agent that executes missions via high-level MAVLink commands *e.g.*, `takeoff`, `navigate to zone`, `return to launch`. This pilot can contain any user-provided Python code, and so may include SADE Zone authentication logic or any other advance coordination feature desired. The *flight controller* used in the PX4 autopilot run in SITL mode and send actuator signals to the physics engine, and receive sensor data back.

Physics Simulator: All simulated sUAS utilize a single instance of Gazebo [11]. This simulator runs in headless mode, calculating rigid-body dynamics, collision detection, environmental interactions, and emulating UAV sensors (IMU, GPS, altimeter). Physical environment data (*e.g.*, terrain, wind) is precomputed at the beginning of each simulated workflow, and then loaded into Gazebo at runtime. The *pose sender* consumes 1000 Hz telemetry from Gazebo, extracting position and orientation for each sUAS and camera, and sending this stream to the visual simulation for a synchronized visualization with its physical state.

Visual Simulator: The 3D simulation environment is provided by Unreal Engine 5[7] extended with plugins: each sUAS is modeled as a 3D game entity while the *player* is an observer that can switch views between any active drone or takeoff site. The *Cesium for Unreal plugin* [4] renders "digital shadow" of any real-world location using satellite maps and WGS84 coordinates. The realism aspect of the 3D environment is further discussed in Section 3.3. A *pose receiver* consumes the 1000 Hz update stream, transforming the coordinates from Gazebo's East-North-Up (ENU) frame to the East-South-Up frame used by Cesium for Unreal. It uses this stream to asynchronously update the drone positions in the visual environment. This decouples the higher simulation rate from the render frame rate (30-120 FPS) for smooth video.

Integration Services: Additional services are required to connect components and make them available to the outside world. The Unreal *pixel streaming* service [6] connects to the Unreal game, and streams fully rendered frames to end-user devices via WebRTC, enabling interaction through standard browsers. A shared *mavlink-router* [12] multiplexes all the flight controller telemetry through one port, allowing for external ground control systems (such as QGroundControl[16]) to view and control drones.

The *PyDUFS* service exposes telemetry logs through a browsable interface, so completed workflows can be analyzed offline. *Traefik* [20] is used for reverse proxying and maps semantic URLs (*e.g.*, `/simulation/123/video`) to the ports of containers running in a Docker-managed subnet. SADE-SIM can thus selectively expose data streams to authenticated users of the system without requiring reserved ports or dynamic firewall rules.

3.3 Simulation Realism

The SADE-SIM simulation environment is designed as an integration layer that coordinates multiple heterogeneous subsystems into a single, coherent execution context. Rather than treating the environment as a static backdrop or a monolithic world model, SADE-SIM composes independently generated data products, including visual, physical, environmental, and policy-related elements, into a shared interface consumed by both autonomous software pilots and human operators. This section describes how these components are integrated to maintain consistency across physics, perception, and operational control.

Geospatial reference and visual-physical consistency Both the physical simulator (Gazebo) and the visual simulator (Unreal with Cesium) use a shared WGS84 global origin. The pose receiver transforms telemetry from the physics simulator into the Cesium georeferenced frame, so vehicle state, terrain, sensor viewpoints, and SADE Zone overlays remain aligned across physics and perception. The pose receiver drives the visual scene from this stream, keeping render and physics in lockstep. Rendered imagery, onboard camera feeds, and operator-facing visualization therefore reflect the same world state used by the flight controller and Custom Software Pilot. A previously validated framework for UAV requirement validation has shown that this style of shared geospatial reference and visual-physical alignment is effective in practice [24].

Terrain integration Elevation data is derived from the Cesium-based geospatial model by scanning a 2D plane in Cesium; the result is exported in a fast-querying HDF5 format for efficient retrieval during simulation. At runtime, for each drone the elevation data is queried at the vehicle position and a unit-size ground collision plane is spawned below the drone in the physics simulator. Collision detection and altitude-dependent behaviors (*e.g.*, rooftop landing, terrain-following navigation) thus match the same geometry used for visual rendering, avoiding discrepancies between perceived and physical environment structure. The same elevation source is used for SADE Zone placement and for physics, so restricted areas and ground geometry remain consistent.

Environmental-factor models Wind and other disturbances are implemented as modular models (*cf.* DO1). The environment is represented as a voxel grid whose cells align with both the physical and the visual rendering space; each cell stores wind vector components along each direction. This voxel grid is processed by an external CFD workflow based on OpenFOAM [10] to generate spatially and temporally varying wind fields [22], which are loaded into the physical simulator and applied in the flight dynamics computation at runtime. This data-driven approach keeps the core physics and rendering engines unchanged; the same interface allows additional effects such as fog to be integrated without modifying the simulation core.

GNSS degradation An external signal propagation pipeline [23] produces location-dependent measurements that account for line-of-sight obstruction and multi-path effects in dense urban environments. The same voxel grid used for wind is reused for GNSS: each cell holds a line-of-sight count and the number of potential multi-path-inducing satellites, which are queried by any simulated GNSS sensor at the vehicle position. Those measurements are injected at the *sensor interface* to the flight controller, so autonomy and fail-safe logic can be validated under realistic positioning errors without coupling the architecture to a specific GNSS implementation.

Airspace Restrictions SADE Zone are represented in the shared geospatial frame (*cf.* DO3): volumetric overlays in the visual simulator and programmatic queries for Custom Software Pilots during mission execution. Compliance with access rules is therefore evaluated inside the simulation workflow, and operators see the same SADE Zone geometry for mission outcomes, access denials, and anomalous behaviors.

3.4 Simulation Workflow Lifecycle

The *workflow dispatcher* has the responsibility of observing the arrival of new simulation configurations, dispatching simulations, and reporting their completion status. It assigns to each newly arrived simulation a unique UUID, and determines if sufficient resources (CPU, GPU, RAM) are available to start a simulation. If sufficient resources are not available, the dispatcher waits until they become available. Transitions between simulation states (*e.g.*, Built, Started, Aborted) are persisted to the MongoDB back-end, making the current status available to users of the website.

Each simulation run dispatched is expressed as a *workflow*: a directed graph of discrete tasks and services that must be dispatched within various constraints in order to sequence the simulation correctly. Each workflow has the following stages:

Phase 1 - Artifact Generation. Several substantial data artifacts are needed to provide simulation realism: a wind vector field for the locality, terrain meshes generated from satellite data, and airspace governance data. Prior to starting any simulation components, the workflow runs the necessary data generation tools to produce datasets not already available. Simulating a fresh environment may require several minutes delay to generate the needed assets before a simulation can start. However, returning to a prior locality will reuse cached assets.

Phase 2 - Configuration Setup. A large number of network ports, unique names, and similar items must be generated to ensure that each component of the workflow is communicating with the right partner. Allowing every component to choose fixed or random ports would result in chaos. Instead, the workflow dispatcher regenerates a `ports.config` file that describes unique TCP/UDP ports for Gazebo, MAVLink, and GCS interfaces; and assigns instance IDs to each UAV. It then produces Unreal coordinate configurations, a Docker Compose file with container networks and volume mounts, and scripts to invoke each of the services.

Phase 3 - Execution and Monitoring. Finally, the dispatcher starts the produced Docker Compose workflow with all of the generated configurations. This startup process requires some care to ensure that actions are sequenced, so that unpredictable delays do not result in failures: first Gazebo starts, then flight controllers

and software pilots, followed by Unreal and the pose sender. Only once all components are initialized, are the software pilots released to begin missions simultaneously. This ordering and dispatch is accomplished using the Shepherd [9] workflow manager to describe and monitor the process. Finally, the dispatcher enters a health monitoring loop, monitoring signals from the *Shepherd* process (mission status, autopilot/Gazebo errors, UE5 health), the container workflow, and external abort signals from the front-end.

Phase 4 - Shutdown and Archival. A simulation workflow terminates either at the request of the user through the web front-end, or at the failure of any individual setup or runtime component. The dispatcher then extracts all of the component log files into persistent storage, tears down the containers and ephemeral assets, and reports the simulation completed. Logs of past runs are made available to the user via the PyDUFS service, (Future work will allow the simulation to complete automatically when certain mission objectives are reached.)

The archived logs include data from each layer of the simulation. Software pilot logs capture high level mission information, such as MAVLINK mode transitions, waypoint achievements, and interactions with airspace management services. Flight controller logs capture the sensor data such as GPS position, barometric altitude, EKF state vectors, and actuator commands in `ulog` format. Physical simulator logs capture the (simulated) ground truth of sUAS positions and attitudes, as well as those of onboard cameras mounted via gimbals. Logs from PX4 can be uploaded to PX4 Flight Review [15] for visual analysis.

4 Case Study: Emergency Response

We demonstrate SADE-SIM's ability to deploy custom software pilots into a realistic environment through 32-sUAS heterogeneous fleet scenario that exercises SADE Zone access control, autonomous decision-making, visual realism, and telemetry logging.

Scenario. To evaluate the policy-enforcement capabilities of the architecture detailed in Section 3, we simulate a future emergency response scenario at a museum in downtown Chicago. During this event, the Chicago Department of Public Safety (DPS) establishes a dynamic, high-priority SADE Zone over the area, restricting airspace access exclusively to authorized sUAS. To manage the crisis, the DPS deploys a coordinated swarm of 5 public safety sUAS for real-time crowd monitoring and response.

However, reflecting the proliferation of modern sUAS usage, the scenario introduces heterogeneous, uncoordinated traffic into the surrounding airspace. This includes commercial news agencies deploying 10 sUAS to capture live aerial broadcasts, and 17 recreational flyers attempting to penetrate the restricted zone to record unauthorized footage. Every sUAS is programmed to seek authorization prior to traversing the enforced zone. In this simulation, DPS requests are always granted, whereas requests from journalists and recreational flyers are denied. This multi-agent environment severely stresses the system's ability to dynamically authenticate requests, enforce airspace controls, and safely route authorized traffic while rejecting rogue agents.

Configuration. To implement this scenario, the end user configures the simulation environment by graphically defining the SADE Zone, pinpointing the initial deployment coordinates for the sUAS fleet, and specifying the logic for the software pilots.

The user configures the SADE Zone by drawing a bounding box over a map of the target area. These boundaries are automatically propagated to the software pilots and to the photorealistic rendering engine, where they appear as translucent pink volumetric barriers overlaid on the 3D environment for intuitive visual reference.

The user configures the 32-sUAS fleet through the web portal in three groups: *public safety* sUAS 1–5 begin in various locations around the city, are always granted clearance to enter the zone, and stay below 400' AGL unless given special clearance *journalist* sUAS 6–15 are also scattered around the city, but will not be granted clearance to the zone, and always stay under 400' AGL. *recreational* sUAS 16–32 begin in various parks, and not granted clearance, and stay below 100' AGL.

The user provides a single *software pilot* (DO2) program that encompasses all drones in the simulation. It issues navigation commands and holds responsibility to interact with airspace management services. The pilots for the public safety and journalist sUAS navigate directly toward the region of interest. The recreational sUAS follow a random path, periodically selecting random coordinates within the operational bounds and navigating toward them. This approach generates unplanned SADE Zone encounters, testing the responsiveness of the airspace management protocol.

Upon approaching a SADE Zone boundary, the software pilot transmits an entry request to the *airspace management service*. This service evaluates the request and returns one of two verdicts. *Granted:* The sUAS is authorized, a temporal lease is allocated, and the mission continues subject to zone-specific constraints (*e.g.*, maintaining a minimum altitude of 350 feet Above Ground Level). *Denied:* Access is rejected due to insufficient credentials, exhausted zone capacity, or inadequate safety margins. Upon denial, the software pilot executes a contingency maneuver, such as a Return-to-Launch (RTL), loitering in place (journalist), or redirecting to a new waypoint (recreational).

Observations. The simulation successfully validated the platform's capacity to govern complex, multi-agent urban airspace under airspace management policies.

Spatial Compliance and Trajectory Control: As illustrated in Figure 4(A), the system captured the high-fidelity flight paths of all 32 sUAS throughout the emergency event. The visual trajectory map confirms strict adherence to the geofenced policies: the red paths (Public Safety sUAS) seamlessly penetrate the SADE Zone, while the blue (Journalist) and green (Recreational) paths are visibly deflected at the boundary, validating the efficacy of the `zone authorization service`. Furthermore, Figure 4(B) demonstrates the system's interoperability with standard tools, showcasing the entire fleet being monitored and commanded in real-time via *QGroundControl*[16] without any telemetry degradation.

Visual Fidelity and 3D Rule Enforcement: The photorealistic rendering of the simulation provides critical context for these interactions. Figure 4(C–F) captures the sequential progression of an authorized public safety sUAS approaching the boundary, holding its position to request entry, and ultimately entering the SADE Zone to approach the museum. Crucially, the system enforced not just lateral boundaries, but strict vertical constraints. As shown in the altitude timeline derived from the flight control logs (Figure 4G), authorized sUAS strictly adhered to the policy mandate requiring them to fly above 350 feet within the SADE Zone. This ensured

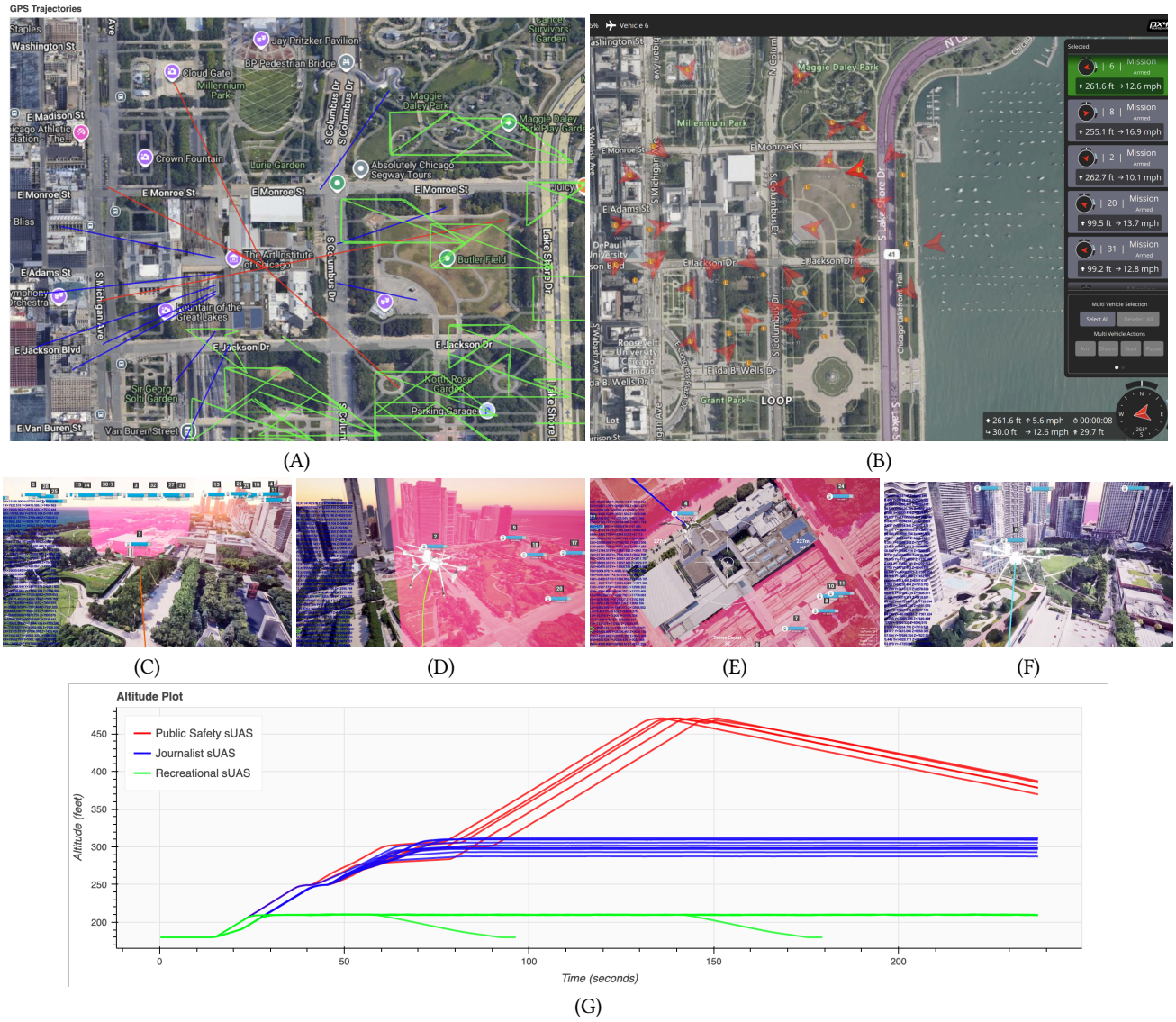


Figure 4: SADE-SIM Case Study

Case study of using SADE-SIM to conduct a simulation of 32 sUAS interacting with flight restrictions resulting from an event at a museum in downtown Chicago. (A) Flight paths of all 32 sUAS throughout the simulation. (B) Snapshot of QGroundControl[16] used to externally monitor the running simulation. (C) Unreal Engine[7] video frame of a public safety sUAS approaching the SADE Zone. (D) The sUAS holds its position to request entry to the SADE Zone. (E) The sUAS enters the zone and approaches the museum. (F) A journalist sUAS approaching the SADE Zone boundary. (G) Timeline of sUAS altitudes extracted from returned telemetry logs.

safe vertical separation from hypothetical ground operations and demonstrated the process's ability to enforce 3D volumetric rules rather than simple 2D map restrictions.

System Performance and Scalability: Beyond spatial compliance, the architecture demonstrated remarkable computational resilience. Managing 32 concurrent rigid-body physics instances alongside a unified Unreal Engine 5[7] visualization pipeline, the system maintained deterministic physics calculations at 1000 Hz and also seamlessly processed everything without inducing navigational latency or triggering collision failsafes. These outcomes confirm that SADE-SIM can effectively model large-scale urban air mobility scenarios while strictly enforcing dynamic regulatory frameworks in real-time.

5 Related Work

Researchers have developed a rich ecosystem of sUAS simulators that support software-in-the-loop and, in some cases, hardware in-the-loop simulating testing, including Gazebo [11], AirSim [17], and FlightGoggles [8]. These platforms provide core capabilities for

controlled sUAS testing, and they have been extended with richer sensor models such as LiDAR [3] and environmental effects such as wind [22] to evaluate perception and navigation under more realistic conditions. Despite this progress, large scale sUAS mission evaluation remains difficult because the design of simulation tools primarily optimize a single aspect of the simulation such as physics fidelity, environmental factors and complexity, or developer facing APIs. In contrast, SADE-SIM's architecture composes these capabilities through a workflow manager to support PX4-driven, large-scale sUAS mission validation.

Recent simulation platforms such as Flightmare [18] decouple rendering from dynamics to improve throughput and support parallel simulation. However, Flightmare's architecture is not distributed, so large scale environments and large fleets are limited by the cost of rendering many sUAS viewpoints and by physics that remains confined to a single node. Similarly, in the autonomous vehicle domain, simulators such as CARLA [5] adopt a client server design where a central server owns world state, physics, and sensor rendering, while external clients connect to control agents and stream observations. This model supports remote control and multi agent experiments, but it largely scales either by adding more compute to a single server or redesigning part of the central server. SADE-SIM follows a scale-out design that partitions rendering and physics computations across multiple nodes to improve throughput and allows development teams to conduct large scale missions by adding more machines rather than redesigning the central server.

6 Discussion and Future Work

SADE-SIM is a platform that integrates a wide variety of simulation technologies into an integrated whole for the evaluation of software pilots and multi-sUAS systems. Drawing upon available technologies and introducing new capabilities, SADE-SIM provides a "whole mission" environment suitable for evaluating city-scale multi-sUAS missions that have complex interacting needs. In this paper, we have demonstrated up to 32 sUAS in a city environment, with drones playing multiple autonomous roles, interacting with basic airspace management technologies. We plan to scale SADE-SIM to hundreds of drones.

We plan several avenues of future work to exploit the integrated realism provided by SADE-SIM: **Mission Evaluation:** Given a mission objective and a set of constraints ("avoid controlled airspace"), evaluate whether a given software pilot can execute mission without violating given constraints. Through extensive testing, software pilots can be evaluated in a variety of adverse conditions (*e.g.*, high wind) prior to real-world deployment. SADE-SIM provides an opportunity to implement and test proposed protocols and systems for airspace management in simulation.

Acknowledgments

Will Meyers of *DePaul University, Chicago* was also a contributor to this work and this work was supported by NASA grant 80NSSC23M0058, titled "A safety-aware ecosystem of interconnected and reputable sUAS."

References

[1] Ankit Agrawal, Philipp Zech, and Michael Vierhauser. 2024. Coupled requirements-driven testing of cps: from simulation to reality. In *2024 IEEE 32nd International Requirements Engineering Conference (RE)*. IEEE, 337–344.

[2] Ankit Agrawal, Bohan Zhang, Yashaswini Shivalingaiah, Michael Vierhauser, and Jane Cleland-Huang. 2023. A requirements-driven platform for validating field operations of small uncrewed aerial vehicles. In *2023 IEEE 31st International Requirements Engineering Conference (RE)*. IEEE, 29–40.

[3] Elizabeth Bondi et al. 2018. Airsim-w: a simulation environment for wildlife conservation with uavs. In *Proc. of the 1st ACM Conf. on Computing and Sustainable Societies*, 1–12.

[4] Cesium GS, Inc. 2021. Cesiumjs. https://cesium.com/platform/cesiumjs/. Accessed: 2026-02-06. (2021).

[5] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. 2017. Carla: an open urban driving simulator. In *Conference on robot learning*. PMLR, 1–16.

[6] Epic Games. 2025. Overview of Pixel Streaming in Unreal Engine. Accessed: 2025-02-08. (2025). https://dev.epicgames.com/documentation/en-us/unreal-engine/overview-of-pixel-streaming-in-unreal-engine.

[7] [SW] Epic Games, Unreal Engine 5 version 5.0, 2022. URL: https://www.unrealengine.com.

[8] Winter Guerra, Ezra Tal, Varun Murali, Gilhyun Ryou, and Sertac Karaman. 2019. Flightgoggles: photorealistic sensor simulation for perception-driven robotics using photogrammetry and virtual reality. (2019). eprint: arXiv:1905.11377.

[9] M. S. Islam and D. Thain. 2024. Shepherd: seamless integration of service workflows into task-based workflows through log monitoring. In *SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2080–2087. doi:10.1109/scw63240.2024.00260.

[10] Hrvoje Jasak. 2009. Openfoam: open source cfd in research and industry. *International Journal of Naval Architecture and Ocean Engineering*, 1, 2, 89–94. OpenFOAM version 10 used in this work.

[11] N. Koenig and A. Howard. 2004. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*. Vol. 3, 2149–2154 vol.3. doi:10.1109/IROS.2004.1389727.

[12] [SW] MAVLink Router Contributors, mavlink-router: Route MAVLink packets between endpoints version master, Feb. 13, 2024. URL: https://github.com/mavlink-router/mavlink-router.

[13] Lorenz Meier, Dominik Honegger, and Marc Pollefeys. 2015. Px4: a node-based multithreaded open source robotics framework for deeply embedded platforms. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, 6235–6240. doi:10.1109/ICRA.2015.7140074.

[14] Meta Platforms, Inc. 2013. React: a javascript library for building user interfaces. https://react.dev/. Accessed: 2026-02-06. Meta Platforms, Inc., (2013).

[15] PX4 Development Team. 2026. PX4 flight review: web-based flight log analysis tool. https://logs.px4.io/. Accessed: 2026-02-09. (2026).

[16] Cristian Ramirez-Atencia and David Camacho. 2018. Extending qgroundcontrol for automated mission planning of uavs. *Sensors*, 18, 7. doi:10.3390/s18072339.

[17] Shital Shah, Debadeepta Dey, Chris Lovett, and Ashish Kapoor. 2018. Airsim: high-fidelity visual and physical simulation for autonomous vehicles. In *Field and Service Robotics: Results of the 11th Int'l Conf.* Springer, 621–635.

[18] Yunlong Song, Selim Naji, Elia Kaufmann, Antonio Loquercio, and Davide Scaramuzza. 2021. Flightmare: a flexible quadrotor simulator. In *Conference on Robot Learning*. PMLR, 1147–1157.

[19] Danping Sun, Peng Li, Xin Xia, Di Liu, and Simone Baldi. 2024. Mrs ardupilot: an adaptive ardupilot architecture based on model reference stabilization. In *2024 IEEE Intelligent Vehicles Symposium (IV)*, 2723–2728. doi:10.1109/IV55156.2024.10588730.

[20] Traefik Labs. 2026. Traefik proxy documentation. https://doc.traefik.io/traefik/. Accessed: 2026-02-07. (2026).

[21] Michael Vierhauser, Md Nafee Al Islam, Ankit Agrawal, Jane Cleland-Huang, and James Mason. 2021. Hazard analysis for human-on-the-loop interactions in suas systems. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 8–19.

[22] Bohan Zhang and Ankit Agrawal. 2024. Dronewis: automated simulation testing of small unmanned aerial system in realistic windy conditions. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2358–2361.

[23] Bohan Zhang, Julian Gutierrez, and Ankit Agrawal. 2025. React-g: gnss signal multipath simulation framework for small uncrewed aerial systems. In *2025 IEEE/ION Position, Location and Navigation Symposium (PLANS)*. IEEE, 1170–1181.

[24] Bohan Zhang, Yashaswini Shivalingaiah, and Ankit Agrawal. 2023. Dronereq-validator: facilitating high fidelity simulation testing for uncrewed aerial systems developers. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, (Sept. 2023), 2082–2085. doi:10.1109/ASE56229.2023.00011.