ARL-TR-10298 • MAR 2026

DEVCOM ARMY RESEARCH LABORATORY

ARL-Tack: Open-Source Software Documentation

by William Gerichs, Robert Smith, Adam Wachsman, Alex Stauff, Russell Cohen Hoffing, and Jonathan Touryan

DISTRIBUTION STATEMENT A. Approved for public release: distribution unlimited

NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.

ARL-TR-10298 ● MAR 2026

ARL-Tack: Open-Source Software Documentation

William Gerichs, Robert Smith, Adam Wachsman, and Alex Stauff
DCS Corporation

Russell Cohen Hoffing and Jonathan Touryan
DEVCOM Army Research Laboratory

DISTRIBUTION STATEMENT A. Approved for public release: distribution unlimited.

REPORT DOCUMENTATION PAGE

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED	
		START DATE	END DATE
March 2026	Technical Report	January 20, 2020	September 4, 2025

4. TITLE AND SUBTITLE
ARL-Tack: Open-Source Software Documentation

5a. CONTRACT NUMBER	5b. GRANT NUMBER	5c. PROGRAM ELEMENT NUMBER
5d. PROJECT NUMBER	**5e. TASK NUMBER**	**5f. WORK UNIT NUMBER**

6. AUTHOR(S)
William Gerichs, Robert Smith, Adam Wachsman, Alex Stauff, Russell Cohen Hoffing, and Jonathan Touryan

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
DEVCOM Army Research Laboratory ATTN: FCDD-RLA-FC Adelphi, MD 20783	ARL-TR-10298

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
DISTRIBUTION STATEMENT A. Approved for public release: distribution unlimited.

13. SUPPLEMENTARY NOTES
ORCIDs: Russell Cohen Hoffing, 0000-0002-3478-8196; Jonathan Touryan, 0000-0001-7343-7869

14. ABSTRACT
ARL has developed a suite of software tools for the unified collection of game state information, including behavioral and physiological time series data, for multiplayer human-in-the-loop simulations that use the Unreal game engine. Specifically, these tools consist of a collection of Unreal Engine plug-ins that automate the data collection processes for single- or multiplayer experiments. These tools use a database backend for near–real-time access to game or player information for third-party applications or analysis. This report provides an overview of the tools including overview, installation, customization, and data export. Detailed information about each plug-in is provided with software as a README.

15. SUBJECT TERMS
Humans in Complex Systems, human-in-the-loop simulation, human–autonomy teaming, virtual reality, VR, data collection, human performance assessment, military simulation, human factors, situational awareness, SA, Unreal Engine, UE, lab streaming layer, LSL, human sensing, eye tracking, PostgreSQL

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES
a. REPORT	b. ABSTRACT	C. THIS PAGE		
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UU	37

19a. NAME OF RESPONSIBLE PERSON	19b. PHONE NUMBER *(Include area code)*
Russell Cohen Hoffing	(520) 691-5011

STANDARD FORM 298 (REV. 5/2020)
Prescribed by ANSI Std. Z39.18

Contents

List of Figures iv

List of Tables iv

1. Introduction 1

1.1 Unreal Engine 3

1.2 Software Architecture 3

1.3 Data Export 4

2. Installation and Setup 5

2.1 Tack 6

2.2 Tack-Back 7

3. Custom Topics Exported via Generic Schema 7

4. Custom Topics Exported via Custom Schema 8

5. Conclusions 9

Appendix. Schema 10

List of Symbols, Abbreviations, and Acronyms 29

Distribution List 30

List of Figures

Figure 1. ARL-Tack implementation and workflow. 1. User develops a specific level, task, or game and includes the ARL-Tack plug-ins. 2. ARL-Tack automatically records relevant information during game play, storing it in a database for online access. 3. After the game play is complete, data is exported into structured files for analysis.. 2

Figure 2. Example database exported root and client directories. The exported data is set at a root directory (here SP10) with all data included in the data directory (here dump and XDF). 5

Figure 3. Blueprint showing the Publish Struct functionality. 8

Figure 4. Custom export files will appear in folders alongside the generic and Unreal folders. 8

List of Tables

Table A-1. unreal.export_actor_damage. 11

Table A-2. unreal.export_actor_lifetimes. 12

Table A-3. unreal.export_actor_overlaps. 12

Table A-4. unreal.export_actor_positions. 13

Table A-5. unreal.export_actors. 14

Table A-6. unreal.export_camera_positions. 17

Table A-7. unreal.export_client_snapshot. 17

Table A-8. unreal.export_client_xr_motion_controller. 18

Table A-9. unreal.export_clients. 19

Table A-10. unreal.export_input_action. 20

Table A-11. unreal.export_input_axis. 20

Table A-12. unreal.export_input_raw_axis. 21

Table A-13. unreal.export_input_raw_key. 21

Table A-14. unreal.export_match_state. 22

Table A-15. unreal.export_pawn_positions. 22

Table A-16. unreal.export_pawn_possessions. 23

Table A-17. unreal.export_time. 23

Table A-18. unreal.export_traces_left. 24

Table A-19. unreal.export_traces_main. 25

Table A-20. unreal.export_traces_raw. 26

Table A-21. unreal.export_traces_right. 27
Table A-22. unreal.export_world. 28

1. Introduction

The U.S. Army Combat Capabilities Development Command (DEVCOM) Army Research Laboratory (ARL) Tack software suite, a data collection tool for human-in-the-loop simulations, was designed to facilitate the development of next-generation Soldier-system concepts and technologies. Specifically, the purpose of ARL-Tack is to enable the rapid development of virtual environments to analyze, and potentially enhance, the performance of heterogeneous teams of human and autonomous systems. This tool was originally developed for the Tactical Awareness via Collective Knowledge (TACK) research program, which focused on technologies that use opportunistic sensing to quantify, predict, and enhance Soldier/squad shared situational awareness. However, these tools are broadly useful for any experimentation using the Unreal Engine (UE), including virtual reality (VR) tasks to assess elements of human performance. They are designed to be easy to use but require implementation within an existing UE project.

ARL-Tack uses the UE as a virtual environment platform to collect both human sensing (e.g., eye tracking) and game state data (e.g., player position), and it is specifically designed to allow multiplayer implementations. ARL-Tack is separate from the level, world, assets, or specific game mechanics of a particular task or experimental paradigm, but it can be applied to any Unreal environment—single- or multiplayer. More importantly, the ARL-Tack plug-ins use a database back end (PostgreSQL) to hold the collected data for efficient online access and therefore uses multiple server applications: Unreal, Kafka, and PostgreSQL. Data can be accessed via database queries, for real-time applications, or through automated exporting of the database after collection is complete, for post hoc analysis. Figure 1 shows the conceptualized workflow for implementation of ARL-Tack. For the complete ARL-Tack schema, including data types and descriptions, see the Appendix. By design, the ARL-Tack requires minimal installation and setup as it leverages the plug-in framework of the UE. ARL-Tack effectively abstracts and automates the collection of all relevant game state data for any Unreal-based application. In addition, ARL-Tack is designed to be used in conjunction with the lab streaming layer* (LSL), a middleware for the synchronization and collection of human-sensing time series data that supports numerous commercial human-sensing devices. Taken together, ARL-Tack plug-ins provide the following:

- Automated collection of game state data for any UE environment
    - Single player or multiplayer

* https://labstreaminglayer.readthedocs.io/info/getting_started.html

    - Actor properties, pawn positions, experimental events, ray-cast collisions, keyboard/mouse inputs, client hardware, and others
- Support for both desktop and head-mounted display (HMD) implementations
- Real-time and post hoc access to game state and human-sensing data
- Data exporting into standardized file formats, organized by clients
- Extensibility to any human-sensing devices that are LSL-compatible

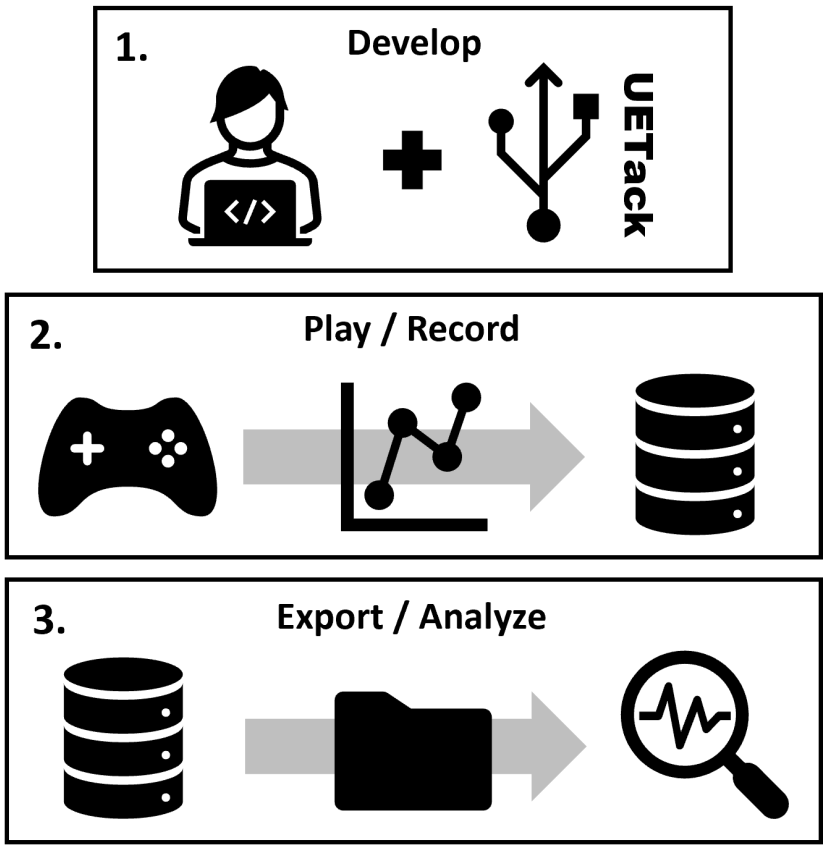

Figure 1. ARL-Tack implementation and workflow. 1. User develops a specific level, task, or game and includes the ARL-Tack plug-ins. 2. ARL-Tack automatically records relevant information during game play, storing it in a database for online access. 3. After the game play is complete, data is exported into structured files for analysis.

The next section provides a more detailed description of ARL-Tack—specifically, how ARL-Tack leverages the UE and how game state data is collected and organized. In addition, we provide a high-level description of the software (source code) and overview of the data export process. A full description of the database schema is included in the Appendix.

1.1 Unreal Engine

UE is an open-source 3D game engine, developed by Epic Games,[*] for design visualizations, cinematic rendering, and development of premier commercial video games. The ARL-Tack plug-ins use the 3D generation and rendering aspect of UE to create "Tack-ified" maps or levels with unique IDs for most object classes or actors within the map or level. ARL-Tack then captures the relevant information about the objects (size, configuration, components, texture, interactivity, etc.) in addition to how these objects change during game play (spawn time, position, orientation, and state). ARL-Tack also records the inputs from each client (i.e., each player), including the human input devices (keyboard, mouse, controller, etc.), and their effects on pawns (game-controllable objects). More importantly, ARL-Tack also captures gaze data, along with gaze vector ray tracing, when an eye-tracking device is available. Common terminology and definitions for the Unreal Game engine can be found in the link below.[†] All of this data is stored on the database server with near–real-time accessibility, via connections to the server for independent query applications (e.g., Python or Grafana).

In addition to these standardized collections, or tables in SQL, an application developer can include experiment-specific data tables that record the game mechanics and events during a particular experiment. The implementation and documentation of these data tables should follow a similar format to the standard schema (Appendix) and include auto-generated documentation by SchemaCrawler.[‡]

1.2 Software Architecture

The complete system architecture includes both ARL-Tack plug-ins as well as existing LSL tools. The UE server manages from 1 to N clients, efficiently handling multiplayer interaction including replication of actors and events across clients. The UE server generates the server time, updated during every rendered frame, linking all game events across clients. Both the server time and corresponding system time (in system clock and Coordinated Universal Time [UTC]) for each client are continuously injected into the database. The client architecture is designed to be generic, where clients can be desktop participants, HMD participants, or external autonomous agents (e.g., external AI algorithm controlling an in-game drone). Layered on top of the UE server-client architecture is the extensible LSL network (optional). Sensing devices for each client, such as the BioSemi

[*] https://www.epicgames.com/site/en-US/about

[†] https://docs.unrealengine.com/4.27/en-US/Basics/UnrealEngineTerminology/

[‡] https://www.schemacrawler.com/

Electroencephalography (EEG) System, can be streamed to LSL. ARL-Tack will also broadcast the server time and the system time of each client as LSL streams. The Lab Recorder application can then be used to capture LSL streams across all clients and devices, time-align all streams, and save data to a single Extensible Data Format (XDF) file.

Although it is possible to use the LSL timestamps as the universal clock, the recommended approach is to link the system time, or UTC, for synchronized networks of each client with the corresponding server time. The time collection contains entries from each client that include both the system-server and UTC timestamps. Although the server time updates every tick (typically ~60 Hz), system timestamps are added at a higher sampling rate to improve temporal alignment with human-sensing devices. If the LSL stream includes a system timestamp, then a direct linkage exists among the LSL signal, system, and server times. Alternatively, ARL-Tack includes an LSL stream that contains server and system time that can be cross-referenced to any LSL stream.

1.3 Data Export

A critical aspect of any data collection software tool is how the data is organized and accessed for post hoc analysis. To facilitate easy access of data across experiments, we developed a standard schema and exporting protocol. Specifically, the database can be exported into a client-centric collection of standard folders and files: Tab-Separated Values (TSV) and Java Script Object Notation (JSON). Column headers (TSV) and field names (JSON) follow the schema and can be readily imported into analysis software (e.g., Python). Data that is common across clients (e.g., actor information and positions) is exported at the root (server) level, whereas client-specific data (e.g., camera positions) is organized in client subfolders. Time series data is exported into TSV, whereas object and client data are exported into JSON. The data organization is designed to be intuitive for researchers, requiring only the schema documentation (Appendix) to interpret all data values.

After data collection is complete, files can be exported using Tack-Back (see repository README) into standard formats (JSON and TSV) that are easy to import into third-party analysis software such as MATLAB or Python. Files will be saved into either a root (server) directory or a client subdirectory (see the example in Figure 2). In the root directory, global experimental data that is common across all clients (e.g., actor information and general game events such as bomb explosions) is saved. Client information can be found in a subfolder within the root directory. Each folder in “clients” contains the data that is specific for each

individual client (e.g., mouse, keyboard, eye tracking, and camera positions). These files allow the analysis of individual behavior/performance and related physiology (e.g., eye movements). The server and every client that is used in the experiment should have a subfolder with these standard timeseries files. Additional files, containing any subset of raw or processed data, can be generated via database query, as needed, by the end user. If LSL is used in conjunction with ARL-Tack, then an additional XDF will be included that can be cross-linked with UTC and server timestamps.

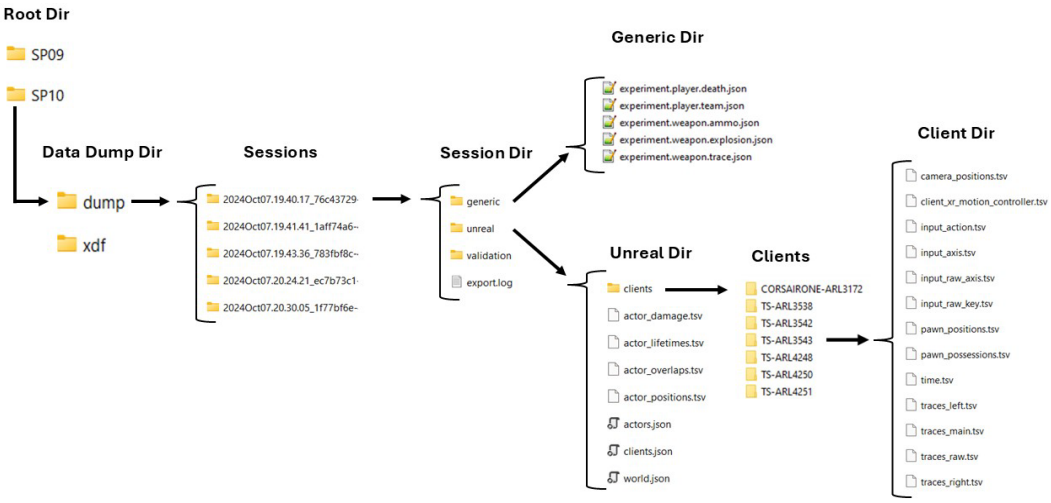

Figure 2. Example database exported root and client directories. The exported data is set at a root directory (here SP10) with all data included in the data directory (here dump and XDF).

As Figure 2 shows, every time the plug-in is started and stopped, a unique session ID is generated within a given database (here, multiple sessions were run across days and saved in the same database). Each unique session ID is exported into its own directory containing the experimental data. Within the session directory, custom folders can be generated with custom exported files (e.g., validation). The generic directory contains custom topics that have application- or experiment-specific information (e.g., game events) that are not part of the standardized SQL tables. The client directory contains all available exported data topics as specified in the schema (see the Appendix).

2. Installation and Setup

README with additional details can be found in the code repositories (links will be updated when released if placed in the ARL repositories).* Altlhough the

* https://github.com/orgs/USArmyResearchLab/repositories

complete software suite is referred to as ARL-Tack, Tack is the name given to the active data collection and storage modules.

- Tack plug-in[*]
    - Currently, Tack is compatible with UE versions 4.27 and 5.3. There are two commits that correspond with each engine version.
- Database backend[†]
    - This includes Kafka, Tack-Back, and SchemaCrawler for documentation.
- Varjo plug-in[‡]
- LSL plug-in[§]

Tack supports desktop, multiplayer, and VR game play along with eye tracking for VIVE (SRanipal), Varjo, or Tobii Pro devices.

SRanipal and Tobii Pro eye-tracking functionalities require the vendor-specific plug-ins from VIVE and Tobii. The Varjo plug-in will work without any additional plug-ins and will not interfere with the OpenXR plug-in that is available for Varjo headsets.

2.1 Tack

For installation, add the Tack and additional plug-ins to the Plug-ins folder of the Unreal project. Note that the Unreal project needs to be a C++ code project.

Once the project is compiled, Tack operation is as follows:

```
> First start tack-wsl-back
> Start Tack
> Stop Tack
> Repeat Start and Stop as much as needed
> Stop tack-wsl-back once data transmission is complete
```

Tack can be started and stopped in the editor (or in development builds with the command console), blueprints, or C++. There are project configuration settings for Tack that adjust what is saved and the Kafka network connection. Additional instructions on this can be found in the Tack README.

[*] https://gitlab.sitcore.net/VisualSearch/tack/release/arl-tack

[†] https://gitlab.sitcore.net/VisualSearch/tack/release/arl-tack-wsl-back

[‡] https://gitlab.sitcore.net/VisualSearch/tack/release/tackvarjoapi

[§] https://gitlab.sitcore.net/VisualSearch/tack/release/lsl

2.2 Tack-Back

Tack-Back contains the database and Kafka messaging service that accepts and stores game data. Tack-Back also contains code to export the stored data from the database into flat files for analysis in third-party software. Everything runs on Docker through WSL on Windows and can be run on other platforms with minor Docker modifications.

To install Tack-Back, run the *install.ps1* script with admin PowerShell and follow the command line prompts. After it has been installed into the specified directory, go to that directory and use the *start.ps1* and *stop.ps1* with PowerShell.

Web pgadmin is included and can be accessed at *localhost:80* with a web browser. The desktop version of pgadmin can be used as well with the *localhost:5432* address and user/password of "postgres."

Use *export.ps1* with PowerShell to export the PostgreSQL database into a structured scheme of flat files to the provided location. For more nuanced export, use the tack-postgres-exporter Python application directly.

3. Custom Topics Exported via Generic Schema

Custom messages can be saved as structs from Unreal to a custom topic and will be automatically exported using the generic schema. When using blueprints, the struct must be defined as an asset and the instance of the struct can be connected to the Publish Struct function from the Tack Publisher subsystem (Figure 3). The topic name is where the data will be grouped in the database and will be automatically exported to the generic folder as a JSON. These nodes will only publish after Tack has been started for that session (see above). Guidance for the C++ equivalent can be found in the Tack README.

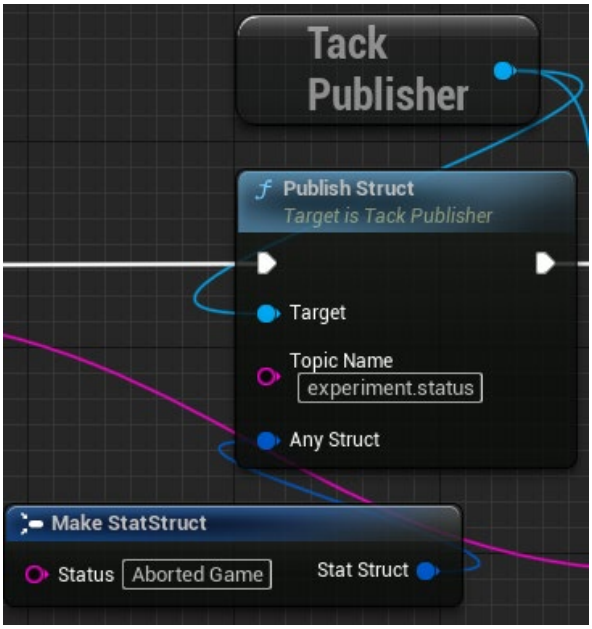

Figure 3. Blueprint showing the Publish Struct functionality.

4. Custom Topics Exported via Custom Schema

Exporting custom topics is more involved and requires SQL knowledge. The simplest way to access the data is via *pgadmin* and a query for the *tack.message* table specifying the topic name. However, this is a query and not a data export to flat files. A new schema with an export function needs to be created for each topic. For example, see the *dynamo.sql* file in the Tack-Back repo.* The export function will determine how the data will be formatted (TSV or JSON), and the exporter tool will then generate these new export files (Figure 4).

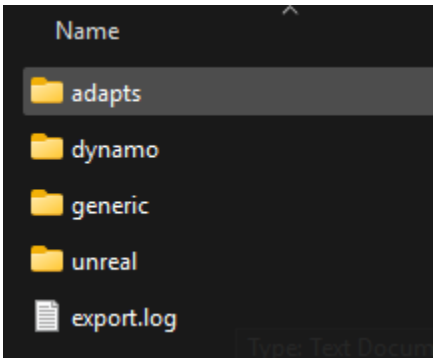

Figure 4. Custom export files will appear in folders alongside the generic and Unreal folders.

*https://gitlab.sitcore.net/VisualSearch/tack/release/arl-tack-wsl-back/-/blob/master/src/containers/postgres/docker-entrypoint-initdb.d/06-dynamo.sql

5. Conclusions

The ARL-Tack suite of software tools (described above) provides the unified collection of game state information, including behavioral and physiological time series data, for multiplayer human-in-the-loop simulations. These UE plug-ins automate the data collection and storage processes for both single- and multiplayer experiments. In addition, they provide near–real-time access to game or player information from third-party applications through standard database queries. This overview should provide potential users with an understanding of the software capabilities and how ARL-Tack can be used for research and experimentation in human performance or human–autonomy teaming.

Appendix. Schema

Tables A-1–A-22 provide the autogenerated documentation of the ARL-Tack schema (standard SQL tables) using SchemaCrawler. This schema documentation should provide the user with sufficient information (field name, meaning, data type, etc.) for both database queries and the loading of exported files. The tables are generated by SchemaCrawler* 16.26.3 and are displayed as is without editing.

Table A-1. unreal.export_actor_damage.

No longer being actively developed. Export view of Unreal damage events as time series. Ordered by utc, server_time. No longer being actively developed for ARL-Tack due to the large variety in custom damage implementations.	
session_id	uuid Id of the Tack session during which this was recorded
utc	timestamp (precision: microsecond) Kafka timestamp of when the event document was transmitted
server_time	float4(8, 8) (unit: second) Unreal server time when damage occurred
damager_actor_id	uuid Id of actor that directly caused damage
damager_name	text Name of actor that directly caused damage
damager_controller_id	uuid Id of controller responsible for causing the damage
damager_controller_name	text Name of the controller responsible for causing the damage
damager_client_id	uuid Client id linked to the controller
damager_hostname	text Hostname of client responsible for the damage
damaged_actor_id	uuid Actor id that received damage
damaged_actor_name	text Name of actor that received damage
damaged_controller_id	uuid Id of controller whose actor received damage
damaged_controller_name	text Name of the controller whose actor received damage
damaged_client_id	uuid Client id linked to damaged actor
damaged_hostname	text Hostname of the client id linked to the damaged actor
damage	float4(8, 8) Amount of damage received
is_caused_by_world	bool Whether damage has a causer or instigator. E.g. would be true for certain implementations of fall damage.
damage_name	text The name of the type of damage received
damage_class	text The class name of the damage received

* https://www.schemacrawler.com/

Table A-2. unreal.export_actor_lifetimes.

Export view listing the spawn and destruction time (if destroyed during Tack recording) of actors	
session_id	uuid Id of the Tack session during which the actor spawned
utc	timestamp (precision: microsecond): Kafka timestamp of the actor spawn time.
destroyed_utc	timestamp (precision: microsecond): Kafka timestamp of when the actor was destroyed. If not destroyed during Tack recording, left as NA
server_time	float4(8, 8) (unit: second): Unreal server time of actor spawn
destroyed_server_time	float4(8, 8) (unit: second): Unreal server time of actor destruction. If not destroyed during Tack recording, left as NA
actor_id	uuid Id of the actor
actor_name	text Name of the actor
class_name	text The name of the lowermost leaf of the actor's class hierarchy (i.e., the most specific class it is an instance of)

Table A-3. unreal.export_actor_overlaps.

Core Tack table. Time series of collision/overlap events between actors, with entering and exiting overlaps as discrete events. Ordered by utc, server_time. For more information on overlap events in Unreal, see: https://docs.unrealengine.com/4.26/en-US/InteractiveExperiences/Physics/Collision/Overview/	
session_id	Uuid Id of the Tack session during which this was recorded
utc	timestamp (precision: microsecond): Kafka timestamp of the time the collision event was transmitted
server_time	float4(8, 8) (unit: second): Unreal server time at time of collision event
overlap_type	text (enum: 'BeginOverlap', 'EndOverlap') Whether the event is the start or end of an overlap
overlapping_actor_id	uuid Id of the actor overlapping
overlapping_actor_name	text Name of the actor overlapping
overlapping_actor_component	text
overlapped_actor_id	uuid Id of the actor that was overlapped
overlapped_actor_name	text Name of the actor that was overlapped
overlapped_actor_component	text Name of the actor component that was overlapped first

Table A-4. unreal.export_actor_positions.

Core Tack table. Export view of position updates (time spent immobile does not generate rows) by moving actors as time series. Ordered by utc, server_time.	
session_id	Uuid Id of the Tack session during which this was recorded
utc	timestamp (precision: microsecond) Kafka timestamp of when the event document was transmitted
server_time	float4(8, 8) (unit: second) Unreal server time
actor_id	uuid Id of actor in motion
controller_id	uuid Id of controller possessing the actor
root_component_name	text Name of the actor component whose location is considered to be the location of the overall actor
teleport_type	text (enum: 'None', 'TeleportPhysics', or 'ResetPhysics') None when position update is not a teleport, meaning velocity reflects movement and collisions along path should occur. TeleportPhysics when collisions along path would not occur but velocity is conserved. ResetPhysics when actor is both teleported and its physics state is reset completely at destination.
location_x	float4(8, 8) (default unit: cm) Actor location at time
location_y	float4(8, 8) (default unit: cm) Actor location at time
location_z	float4(8, 8) (default unit: cm) Actor location at time
rotation_x	float4(8, 8) (unit: degrees; range: -180 to 180; positive direction: clockwise; origin: z-up and y-right): Rotation of the actor about the x axis at time, AKA pitch. Subject to animation constraints
rotation_y	float4(8, 8) (unit: degrees; range: -180 to 180; positive direction: clockwise; origin: x-left and z-up): Rotation of the actor about the y axis at time, AKA roll. Subject to animation constraints
rotation_z	float4(8, 8) (unit: degrees; range: -180 to 180; positive direction: clockwise; origin: y-up and x-right): Actor rotation about the z axis at time, AKA yaw/heading. Subject to animation constraints
scale_x	float4(8, 8) Scale factor of the actor at a given time
scale_y	float4(8, 8) Scale factor of the actor at a given time
scale_z	float4(8, 8) Scale factor of the actor at a given time
forward_x	float4(8, 8) (default unit: cm; range: -1 to 1): Unit direction vector of the actor's forward-facing vector at time
forward_y	float4(8, 8) (default unit: cm; range: -1 to 1): Unit direction vector of the actor's forward-facing vector at time
forward_z	float4(8, 8) (default unit: cm; range: -1 to 1): Unit direction vector of the actor's forward-facing vector at time
right_x	float4(8, 8) (default unit: cm; range: -1 to 1): Unit direction vector of the actor's right-facing vector at time

Table A-4. unreal.export_actor_positions (continued).

right_y	float4(8, 8) (default unit: cm; range: -1 to 1): Unit direction vector of the actor's right-facing vector at time
right_z	float4(8, 8) (default unit: cm; range: -1 to 1): Unit direction vector of the actor's right-facing vector at time
up_x	float4(8, 8) (default unit: cm; range: -1 to 1): Unit direction vector of the actor's upward-facing vector at time.
up_y	float4(8, 8) (default unit: cm; range: -1 to 1): Unit direction vector of the actor's upward-facing vector at time.
up_z	float4(8, 8) (default unit: cm; range: -1 to 1): Unit direction vector of the actor's upward-facing vector at time.
quat_x	float4(8, 8) The x component of the quaternion representing the actor's orientation with respect to the world axes
quat_y	float4(8, 8) The y component of the quaternion representing the actor's orientation with respect to the world axes
quat_z	float4(8, 8) The z component of the quaternion representing the actor's orientation with respect to the world axes
quat_w	float4(8, 8) The w component of the quaternion representing the actor's orientation with respect to the world axes

Table A-5. unreal.export_actors.

Core Tack table. Export view that is the collection of data on Tack-ified actors in the world. All actor_id columns link to this. Ordered by utc, creation_server_time. For more detailed explanation, see: https://docs.unrealengine.com/4.26/en-US/ProgrammingAndScripting/ProgrammingWithCPP/UnrealArchitecture/Actors/	
session_id	Uuid Id of the Tack session during which this was recorded
id	uuid Actor id
utc	timestamp (precision: microsecond): Kafka timestamp at which the actor document was transmitted (either during Tack startup or when spawned during Tack)
creation_server_time	float4(8, 8) (unit: second): Server time at which the actor was created
owner_id	uuid Owner actor/controller id. Primarily used for network replication
parent_actor_id	uuid Parent actor id
name	text Actor name. Not guaranteed to be unique
class_name	text The name of the lowermost leaf of the actor's class hierarchy (i.e. the most specific class it is an instance of)
mobility	text (enum: 'EComponentMobility::Static', 'EComponentMobility::Stationary', 'EComponentMobility::Movable') The mobility setting for the actor. Full explanation of states available at: https://docs.unrealengine.com/4.27/en-US/Basics/Actors/Mobility/

Table A-5. unreal.export_actors (continued).

level_name	Text Name of sublevel that the actor is assigned to. For explanation of persistent level and sub-levels see: https://docs.unrealengine.com/4.27/en-US/Basics/Levels/
is_hidden	bool Whether the actor spawned is visible on startup
is_net_startup	bool Whether the actor exists when the level starts (i.e., is not dynamically spawned)
is_in_persistent_level	bool Whether the actor exists in the persistent level, i.e., exists in all sublevels
is_collision_enabled	bool Whether the actor spawned with collision enabled
bounding_box_center_x	float4(8, 8) (default unit: cm): Centerpoint of the bounding box which encompasses the actor and all its components
bounding_box_center_y	float4(8, 8) (default unit: cm): Centerpoint of the bounding box which encompasses the actor and all its components
bounding_box_center_z	float4(8, 8) (default unit: cm): Centerpoint of the bounding box which encompasses the actor and all its components
bounding_box_extent_x	float4(8, 8) (default unit: cm): Values equal to half of the length, half of the width, and half of the height of the bounding box which encompasses the actor and all its components
bounding_box_extent_y	float4(8, 8) (default unit: cm): Values equal to half of the length, half of the width, and half of the height of the bounding box which encompasses the actor and all its components
bounding_box_extent_z	float4(8, 8) (default unit: cm): Values equal to half of the length, half of the width, and half of the height of the bounding box which encompasses the actor and all its components
collision_box_center_x	float4(8, 8) (default unit: cm): Center point of the bounding box which encompasses all collision-enabled components of the actor
collision_box_center_y	float4(8, 8) (default unit: cm): Center point of the bounding box which encompasses all collision-enabled components of the actor
collision_box_center_z	float4(8, 8) (default unit: cm): Center point of the bounding box which encompasses all collision-enabled components of the actor
collision_box_extent_x	float4(8, 8) (default unit: cm): Values equal to half of the length, half of the width, and half of the height of the bounding box which encompasses all collision-enabled components of the actor
collision_box_extent_y	float4(8, 8) (default unit: cm): Values equal to half of the length, half of the width, and half of the height of the bounding box which encompasses all collision-enabled components of the actor
collision_box_extent_z	float4(8, 8) (default unit: cm): Values equal to half of the length, half of the width, and half of the height of the bounding box which encompasses all collision-enabled components of the actor
spawn_location_x	float4(8, 8) (default unit: cm): Coordinates the actor initially spawned at.
spawn_location_y	float4(8, 8) (default unit: cm): Coordinates the actor initially spawned at.

Table A-5. unreal.export_actors (continued).

spawn_location_z	float4(8, 8) (default unit: cm): Coordinates the actor initially spawned at.
spawn_rotation_x	float4(8, 8) (unit: degrees; range: -180 to 180; increment: clockwise; origin: z-up and y-right): Rotation of the actor about the x axis at spawn, AKA pitch. Subject to animation constraints.
spawn_rotation_y	float4(8, 8) (unit: degrees; range: -180 to 180; increment: clockwise; origin: x-left and z-up): Rotation of the actor about the y axis at spawn, AKA roll. Subject to animation constraints.
spawn_rotation_z	float4(8, 8) (unit: degrees; range: -180 to 180; increment: clockwise; origin: y-up and x-right): Actor rotation about the z axis at spawn, AKA yaw/heading. Subject to animation constraints.
spawn_forward_x	float4(8, 8) (default unit: cm; range: -1 to 1): Unit direction vector of the actor's forward-facing vector at spawn.
spawn_forward_y	float4(8, 8) (default unit: cm; range: -1 to 1): Unit direction vector of the actor's forward-facing vector at spawn.
spawn_forward_z	float4(8, 8) (default unit: cm; range: -1 to 1): Unit direction vector of the actor's forward-facing vector at spawn.
spawn_right_x	float4(8, 8) (default unit: cm; range: -1 to 1): Unit direction vector of the actor's right-facing vector at spawn.
spawn_right_y	float4(8, 8) (default unit: cm; range: -1 to 1): Unit direction vector of the actor's right-facing vector at spawn.
spawn_right_z	float4(8, 8) (default unit: cm; range: -1 to 1): Unit direction vector of the actor's right-facing vector at spawn.
spawn_up_x	float4(8, 8) (default unit: cm; range: -1 to 1): Unit direction vector of the actor's upward-facing vector at spawn.
spawn_up_y	float4(8, 8) (default unit: cm; range: -1 to 1): Unit direction vector of the actor's upward-facing vector at spawn.
spawn_up_z	float4(8, 8) (default unit: cm; range: -1 to 1): Unit direction vector of the actor's upward-facing vector at spawn.
spawn_quat_x	float4(8, 8) The x component of the quaternion representing the actor's orientation with respect to the world axes at spawn
spawn_quat_y	float4(8, 8) The y component of the quaternion representing the actor's orientation with respect to the world axes at spawn
spawn_quat_z	float4(8, 8) The z component of the quaternion representing the actor's orientation with respect to the world axes at spawn
spawn_quat_w	float4(8, 8) The w component of the quaternion representing the actor's orientation with respect to the world axes at spawn
tags	_text Tags attached to the actor during scenario development.
is_child_actor	bool Whether the actor spawned as a child actor of another

Table A-6. unreal.export_camera_positions.

Core Tack table. Export view of user camera movements in the world as time series. Identical to transforms of PlayerCameraManager actors. Ordered by utc, server_time.	
session_id	Uuid Id of the Tack session during which this was recorded
utc	timestamp (precision: microsecond): Kafka timestamp of when the event document was transmitted.
server_time	float4(8, 8) (unit: second): Unreal server time
client_id	uuid Id of the client
location_x	float4(8, 8) (default unit: cm): Coordinates of the client's camera at time
location_y	float4(8, 8) (default unit: cm): Coordinates of the client's camera at time
location_z	float4(8, 8) (default unit: cm): Coordinates of the client's camera at time
forward_x	float4(8, 8) (default unit: cm; range: -1 to 1): Unit direction vector of the camera's forward vector.
forward_y	float4(8, 8) (default unit: cm; range: -1 to 1): Unit direction vector of the camera's forward vector.
forward_z	float4(8, 8) (default unit: cm; range: -1 to 1): Unit direction vector of the camera's forward vector.
quat_x	float4(8, 8) The x component of the quaternion representing the camera's orientation with respect to the world axes
quat_y	float4(8, 8) The y component of the quaternion representing the camera's orientation with respect to the world axes
quat_z	float4(8, 8) The z component of the quaternion representing the camera's orientation with respect to the world axes
quat_w	float4(8, 8) The w component of the quaternion representing the camera's orientation with respect to the world axes

Table A-7. unreal.export_client_snapshot.

Export view of Tack real-time player screen captures as a time series ordered by utc, server_time	
session_id	Uuid Id of the Tack session during which the screen capture was recorded
utc	timestamp (precision: microsecond): Kafka timestamp of when the screen capture was transmitted.
client_id	uuid Id of the client whose screen was captured
file_type	text The type of file the screen capture is encoded as
server_time	float4(8, 8) (unit: second): Unreal server time
snapshot_data	text The raw snapshot data encoded as a text string

Table A-8. unreal.export_client_xr_motion_controller.

Export view of VR/XR motion controller positioning linked to corresponding actor	
session_id	uuid
utc	timestamp
headers	jsonb
client_id	uuid
motion_controller	text
motion_source	text
actor_id	uuid
actor_name	text
component	text
tracking_status	text
location_x	float4(8, 8)
location_y	float4(8, 8)
location_z	float4(8, 8)
rotation_x	float4(8, 8)
rotation_y	float4(8, 8)
rotation_z	float4(8, 8)
quat_x	float4(8, 8)
quat_y	float4(8, 8)
quat_z	float4(8, 8)
quat_w	float4(8, 8)

Table A-9. unreal.export_clients.

Core Tack table. Export view that is a collection of information about the clients connected to the world. All client_id columns link to this. Ordered by utc, connected_server_time. For more information on the client-server model in Unreal, see: https://docs.unrealengine.com/4.26/en-US/InteractiveExperiences/Networking/Server/	
session_id	uuid Id of the Tack session during which this was recorded
id	uuid Id of the client
hostname	text Name of the client's host on the network
player_name	text Name of the client within Unreal. Not guaranteed to be unique
utc	timestamp (precision: microsecond): Kafka timestamp when the client document was transmitted
connected_server_time	float4(8, 8) (unit: second): Server time when client connected to the Unreal server
controller_id	uuid Id of the client's controller
actor_ids	json Array of all pawn actors controlled by the client at some point while Tack was running
is_client	bool Whether the client is an in-game Unreal client. Generally true except in the case of dedicated servers.
is_server	bool Whether the client is hosting the Unreal server.
display_resolution_x	int4 (unit: pixel): Width of the client's display
display_resolution_y	int4 (unit: pixel): Height of the client's display
project_name	text Name of the Unreal project being run by the client
platform_user_name	text Name of the operating system user account running the client application
app_instance_name	text Name generated when the Unreal instance initializes
app_instance_id	uuid Id generated when the Unreal instance initializes
graphics_rhi	text Graphics API being used by Unreal
engine_version	text Unreal engine version
net_mode	text (enum: 'Standalone', 'DedicatedServer', 'ListenServer', or 'Client') Specifies the network mode the client is running Unreal in. Standalone does not interact with the network but still technically runs as a server with one or more local players. DedicatedServer only hosts other players on the network without participating itself. ListenServer has a local player who is hosting the game for others on the network. Client is a player connected to a remote server with no server capabilities itself.
local_ip	inet Network IP of client
additional_vars	jsonb Array of additional/nonstandard properties of client
additional_client_info	jsonb

Table A-10. unreal.export_input_action.

Export view of all keypress events (gamepad buttons, keyboard presses, mouse clicks, etc.) registered as in-game actions as time series ordered by utc, server_time. NOTE: Multiple actions may be registered to the same input and create multiple entries per frame (server_time)	
session_id	uuid Id of the Tack session during which this was recorded
utc	timestamp (precision: microsecond): Kafka timestamp of when the event document was transmitted.
server_time	float4(8, 8) (unit: second): Unreal server time
client_id	uuid Id of the client
action_name	text Name of the action performed for the given keypress
key_name	text Name of the key used to perform the action
events	_text (enum: {IE_Pressed, IE_Released, IE_Repeat, IE_DoubleClick}) Type of input event(s) associated with the action
requires_shift	bool Whether this action requires the Shift key
requires_alt	bool Whether this action requires the Alt key
requires_ctrl	bool Whether this action requires the Ctrl key
requires_super	bool Whether this action requires the Super key

Table A-11. unreal.export_input_axis.

Export view of all axial inputs (mouse movement, analog sticks, motion controllers, etc.) as time series ordered by utc, server_time	
session_id	uuid Id of the Tack session during which this was recorded
utc	timestamp (precision: microsecond): Kafka timestamp of when the event document was transmitted.
server_time	float4(8, 8) (unit: second): Unreal server time
client_id	uuid Id of the client
axis_name	text The name of the axis moved
axis_dim	int4 The dimensions of the axis. Dictates if x (1 dim), y (2 dim), z (3 dim) are valid. (e.g. a dimension of 2 means x and y are valid but z will be null.)
value_x	float4(8, 8) X(1st dimension) Value of this axis movement
value_y	float4(8, 8) Y(2nd dimension) Value of this axis movement. (null when axis_dim < 2)
value_z	float4(8, 8) Z(3rd dimension) Value of this axis movement. (null when axis_dim < 3)

Table A-12. unreal.export_input_raw_axis.

Export view of all axial inputs (mouse movement, analog sticks, motion controllers, etc.) detected by Unreal irrespective of in-game action as time series ordered by utc, server_time	
session_id	uuid Id of the Tack session during which this was recorded
utc	timestamp (precision: microsecond): Kafka timestamp of when the event document was transmitted.
client_id	uuid Id of the client
server_time	float4(8, 8) (unit: second): Unreal server time
axis_name	text The name of the axis moved
axis_dim	int4 The dimensions of the axis. Dictates if x (1 dim), y (2 dim), z (3 dim) are valid. (e.g. a dimension of 2 means x and y are valid but z will be null.)
value_x	float4(8, 8) Value of movement along x axis
value_y	float4(8, 8) Value of movement along y axis
value_z	float4(8, 8) Value of movement along z axis
x_axis_paired_key	text Name of 'key' (for action execution purposes) linked to x axis movement
y_axis_paired_key	text Name of 'key' (for action execution purposes) linked to y axis movement
z_axis_paired_key	text Name of 'key' (for action execution purposes) linked to z axis movement

Table A-13. unreal.export_input_raw_key.

Export view of all keypresses (gamepad buttons, keyboard presses, mouse clicks, etc.) detected by Unreal irrespective of in-game action as time series ordered by utc, server_time	
session_id	uuid Id of the Tack session during which this was recorded
utc	timestamp (precision: microsecond): Kafka timestamp of when the event document was transmitted.
client_id	uuid Id of the client
server_time	float4(8, 8) (unit: second): Unreal server time
key_name	text Name of the key used to perform the action
events	_text (enum: {IE_Pressed, IE_Released, IE_Repeat, IE_DoubleClick}) Type of input event(s) associated with the action

Table A-14. unreal.export_match_state.

Core Tack table. Export view of match state changes. Useful for scenarios where there may be a preliminary 'staging' phase while Tack is running prior to actual experiment. Ordered by utc, server_time.	
session_id	uuid Id of the Tack session during which this was recorded
utc	timestamp (precision: microsecond): Kafka timestamp of the match state change
server_time	float4(8, 8) (unit: second): Unreal server time at match state change
match_state	text Name of the state the match has changed to
has_match_started	bool Flag indicating whether the experiment has started

Table A-15. unreal.export_pawn_positions.

Core Tack table. Duplicate of actor positions but selects solely for pawns controlled by either players or AI. Ordered by utc, server_time.	
session_id	uuid
utc	timestamp
server_time	float4(8, 8)
actor_id	uuid
controller_id	uuid
client_id	uuid
root_component_name	text
teleport_type	text
location_x	float4(8, 8)
location_y	float4(8, 8)
location_z	float4(8, 8)
rotation_x	float4(8, 8)
rotation_y	float4(8, 8)
rotation_z	float4(8, 8)
scale_x	float4(8, 8)
scale_y	float4(8, 8)
scale_z	float4(8, 8)
forward_x	float4(8, 8)
forward_y	float4(8, 8)
forward_z	float4(8, 8)
right_x	float4(8, 8)
right_y	float4(8, 8)
right_z	float4(8, 8)
up_x	float4(8, 8)
up_y	float4(8, 8)
up_z	float4(8, 8)
quat_x	float4(8, 8)
quat_y	float4(8, 8)
quat_z	float4(8, 8)
quat_w	float4(8, 8)

Table A-16. unreal.export_pawn_possessions.

Core Tack table. Export view of all pawn possession events by clients' controllers as time series. Ordered by utc, server_time. For more detail on controllers, see: https://docs.unrealengine.com/4.26/en-US/InteractiveExperiences/Framework/Controller/ For more detail on pawns, see: https://docs.unrealengine.com/4.26/en-US/InteractiveExperiences/Framework/Pawn/	
session_id	uuid Id of the Tack session during which this was recorded
client_id	uuid Id of the client. For AIControllers, this will be the server's client_id
utc	timestamp (precision: microsecond): Kafka timestamp of the time pawn possession document was transmitted
possession_end_utc	timestamp (precision: microsecond): Kafka timestamp of the time next pawn possession document was transmitted. On the last pawn, corresponds to session end utc.
server_time	float4(8, 8) (unit: second): Unreal server time at time of pawn possession
possession_end_server_time	float4(8, 8) (unit: second): Unreal server time when next pawn was possessed. On the last pawn, corresponds to last recorded server time for respective client from client_time.
controller_id	uuid Id of the client's controller
actor_id	uuid Id of the pawn actor being possessed.
actor_name	text Name of the pawn actor being possessed

Table A-17. unreal.export_time.

Core Tack table. Export view of recorded times and frame count by individual clients, usable to link in-game occurrences to other data streams like physiological measures. Ordered by utc, server_time, client_id.	
session_id	uuid Id of the Tack session during which this was recorded
utc	timestamp (precision: microsecond): Kafka timestamp of when the frame document was transmitted.
server_time	float4(8, 8) (unit: second): Unreal server time during frame
client_id	uuid Id of the client
system_timestamp	int8 (precision: nanosecond): System timestamp recorded during frame
frame_count	int8 Cumulative count of frames rendered by client program
delta_time	float4(8, 8) Elapsed time between previous and current frames according to Unreal
average_ms	float4(8, 8) Average delta_time across (TBA) window
average_fps	float4(8, 8) Average FPS across (TBA) window

Table A-18. unreal.export_traces_left.

Core Tack table. Export view of all left eye traces. Ordered by utc, server_time, client_id. See traces_main for more detail.	
session_id	uuid Id of the Tack session during which this was recorded
utc	timestamp (precision: microsecond): Kafka timestamp of when the trace document was transmitted.
server_time	float4(8, 8) (unit: second): Unreal server time
source_utc	int8 (precision: microsecond): Kafka timestamp corresponding to the Kafka timestamp of the hardware eyetracker data used to generate the trace.
client_id	uuid Id of the client
client_bone_name	text Name of the bone hit by the trace, in cases where the trace hits a skeletal mesh
frame_trace_number	int4 Order of traces cast during frame
is_blocking_hit	bool Whether the trace collided with an object along its path
traced_actor_id	uuid Actor id of first actor that the trace ray cast passed through, if any
traced_actor_name	text Actor name of the first actor that the trace ray cast passed through, if any. Not guaranteed to be unique
traced_actor_component	text Name of the specific actor component struck by the trace
traced_actor_bone	text
trace_start_x	float4(8, 8) (default unit: cm): Position coordinates of the trace's origin point.
trace_start_y	float4(8, 8) (default unit: cm): Position coordinates of the trace's origin point.
trace_start_z	float4(8, 8) (default unit: cm): Position coordinates of the trace's origin point.
trace_impact_x	float4(8, 8) (default unit:cm): Position coordinates of the first point where the trace made contact with an actor, if any.
trace_impact_y	float4(8, 8) (default unit:cm): Position coordinates of the first point where the trace made contact with an actor, if any.
trace_impact_z	float4(8, 8) (default unit:cm): Position coordinates of the first point where the trace made contact with an actor, if any.
trace_end_x	float4(8, 8) (default unit: cm): Position coordinates of the end of the trace.
trace_end_y	float4(8, 8) (default unit: cm): Position coordinates of the end of the trace.
trace_end_z	float4(8, 8) (default unit: cm): Position coordinates of the end of the trace.
trace_distance	float4(8, 8) (default unit: cm): Length of the distance between trace_start and trace_impact.

Table A-19. unreal.export_traces_main.

Core Tack table. Export view of all main traces averaged between left and right eye traces as a time series. Ordered by utc, server_time, client_id. For more detail on trace/raycasts in Unreal, see: https://docs.unrealengine.com/4.26/en-US/InteractiveExperiences/Tracing/Overview/.	
session_id	uuid Id of the Tack session during which this was recorded
utc	timestamp (precision: microsecond): Kafka timestamp of when the trace document was transmitted.
server_time	float4(8, 8) (unit: second): Unreal server time
source_utc	int8 (precision: microsecond): Kafka timestamp corresponding to the Kafka timestamp of the hardware eyetracker data used to generate the trace.
client_id	uuid Id of the client
client_bone_name	text Name of the bone hit by the trace, in cases where the trace hits a skeletal mesh
frame_trace_number	int4 Order of traces cast during frame
is_blocking_hit	bool Whether the trace collided with an object along its path
traced_actor_id	uuid Actor id of first actor that the trace ray cast passed through, if any
traced_actor_name	text Actor name of the first actor that the trace ray cast passed through, if any. Not guaranteed to be unique
traced_actor_component	text Name of the specific actor component struck by the trace
traced_actor_bone	text
trace_start_x	float4(8, 8) (default unit: cm): Position coordinates of the trace's origin point.
trace_start_y	float4(8, 8) (default unit: cm): Position coordinates of the trace's origin point.
trace_start_z	float4(8, 8) (default unit: cm): Position coordinates of the trace's origin point.
trace_impact_x	float4(8, 8) (default unit:cm): Position coordinates of the first point where the trace made contact with an actor, if any.
trace_impact_y	float4(8, 8) (default unit:cm): Position coordinates of the first point where the trace made contact with an actor, if any.
trace_impact_z	float4(8, 8) (default unit:cm): Position coordinates of the first point where the trace made contact with an actor, if any.
trace_end_x	float4(8, 8) (default unit: cm): Position coordinates of the end of the trace.
trace_end_y	float4(8, 8) (default unit: cm): Position coordinates of the end of the trace.
trace_end_z	float4(8, 8) (default unit: cm): Position coordinates of the end of the trace.
trace_distance	float4(8, 8) (default unit: cm): Length of the distance between trace_start and trace_impact.

Table A-20. unreal.export_traces_raw.

(deprecated) Core Tack Table. Export view of basic trace hit data and the raw eyetracker struct.	
session_id	uuid Id of the Tack session during which this was recorded
utc	timestamp (precision: microsecond): Kafka timestamp of when the trace document was transmitted.
server_time	float4(8, 8) (unit: second): Unreal server time
source_utc	int8 (precision: microsecond): Kafka timestamp corresponding to the Kafka timestamp of the hardware eyetracker data used to generate the trace.
client_id	uuid Id of the client
frame_trace_number	int4 Order of traces cast during frame
is_blocking_hit	bool Whether the trace collided with an object along its path
traced_actor_id	uuid Actor id of first actor that the trace ray cast passed through, if any
traced_actor_name	text Actor name of the first actor that the trace ray cast passed through, if any. Not guaranteed to be unique
traced_actor_component	text Name of the specific actor component struck by the trace
client_bone_name	text Name of the bone hit by the trace, in cases where the trace hits a skeletal mesh
raw	jsonb Raw eyetracker data struct. Hardware specific

Table A-21. unreal.export_traces_right.

Core Tack table. Export view of all right eye traces. Ordered by utc, server_time, client_id. See traces_main for more detail.	
session_id	uuid
	Id of the Tack session during which this was recorded
utc	timestamp
	(precision: microsecond): Kafka timestamp of when the trace document was transmitted.
server_time	float4(8, 8)
	(unit: second): Unreal server time
source_utc	int8
	(precision: microsecond): Kafka timestamp corresponding to the Kafka timestamp of the hardware eyetracker data used to generate the trace.
client_id	uuid
	Id of the client
client_bone_name	text
	Name of the bone hit by the trace, in cases where the trace hits a skeletal mesh
frame_trace_number	int4
	Order of traces cast during frame
is_blocking_hit	bool
	Whether the trace collided with an object along its path
traced_actor_id	uuid
	Actor id of first actor that the trace ray cast passed through, if any
traced_actor_name	text
	Actor name of the first actor that the trace ray cast passed through, if any. Not guaranteed to be unique
traced_actor_component	text
	Name of the specific actor component struck by the trace
traced_actor_bone	text
trace_start_x	float4(8, 8)
	(default unit: cm): Position coordinates of the trace's origin point.
trace_start_y	float4(8, 8)
	(default unit: cm): Position coordinates of the trace's origin point.
trace_start_z	float4(8, 8)
	(default unit: cm): Position coordinates of the trace's origin point.
trace_impact_x	float4(8, 8)
	(default unit:cm): Position coordinates of the first point where the trace made contact with an actor, if any.
trace_impact_y	float4(8, 8)
	(default unit:cm): Position coordinates of the first point where the trace made contact with an actor, if any.
trace_impact_z	float4(8, 8)
	(default unit:cm): Position coordinates of the first point where the trace made contact with an actor, if any.
trace_end_x	float4(8, 8)
	(default unit: cm): Position coordinates of the end of the trace.
trace_end_y	float4(8, 8)
	(default unit: cm): Position coordinates of the end of the trace.
trace_end_z	float4(8, 8)
	(default unit: cm): Position coordinates of the end of the trace.
trace_distance	float4(8, 8)
	(default unit: cm): Length of the distance between trace_start and trace_impact.

Table A-22. unreal.export_world.

Core Tack table. Export view about the characteristics of the Unreal world.	
session_id	uuid Id of the Tack session during which this was recorded
killz	float4(8, 8) Z coordinate plane at which actors automatically are destroyed
map_name	jsonb The name of the map used by the world
world_to_meters	text The conversion factor from the world coordinate system to real-world meters. e.g. 100 = world coordinates are in centimeters
levels	_text A list of the sublevels in the world

List of Symbols, Abbreviations, and Acronyms

3D	Three-Dimensional
AI	Artificial Intelligence
ARL	Army Research Laboratory
DEVCOM	U.S. Army Combat Capabilities Development Command
EEG	Electroencephalography
HMD	Head-Mounted Display
ID	Identification
JSON	Java Script Object Notation
LSL	Lab Streaming Layer
SQL	Structured Query Language
TACK	Tactical Awareness via Collective Knowledge
TSV	Tab-Separated Values
UE	Unreal Engine
UTC	Coordinated Universal Time
VR	Virtual Reality
WSL	Windows Subsystem for Linux
XDF	Extensible Data Format

1 (PDF)	DEFENSE TECHNICAL INFORMATION CTR DTIC OCA
1 (PDF)	DEVCOM ARL FCDD RLB CI TECH LIB