

The Architecture of Modern GPS Tracking

From device packets to live maps, telemetry, events, automation, APIs, and fleet applications.

A GPS tracking platform is a real-time distributed data system. The map is only the final representation.

Published by Open VTS | Version 1.0

Publication information

Publisher: Open VTS

Edition: 2026 Edition

Document version: 1.0

Series: Open VTS Guide 05 of 06

Scope: A technical and operational architecture guide to modern GPS tracking platforms, using Open VTS as a concrete implementation reference where appropriate.

Editorial position

This guide explains architecture principles rather than reproducing proprietary protocol manuals or source code. Product-specific statements describe the Open VTS architecture and should be verified against the current released version when they affect deployment or purchasing decisions.

Who this guide is for

- CTOs and software architects
- GPS and telematics engineers
- Developers and system integrators
- Technical founders and product leaders
- Infrastructure and operations teams who need to understand the full data path

How to read it

The first sections are intentionally accessible to non-specialists. Technical depth increases progressively: device communication, normalization, time-series storage, state, events, realtime delivery, queues, automation, APIs, reliability, observability, and finally the canonical Open VTS architecture.

Contents

PART I GPS Tracking Is a Data System

The map is the last step · What a tracker does · Reporting logic

PART II From Device to Server

Protocols · TCP/UDP · IP/DNS/ports · Listener · Decoding

PART III From Protocol to Platform Data

Normalization · Current state · Time series · Spatial data · Application data

PART IV Turning Data Into Meaning

Vehicle state · Events · Stateful logic · Geofences/POIs/routes

PART V Realtime Systems

Live tracking · Redis · Queues/workers · Reliability/replay

PART VI Automation & Integration

Workflow · Device commands · REST APIs · Realtime APIs

PART VII Applications & User Experience

Web · Mobile · Reports · Permissions

PART VIII Scaling the Architecture

Workload · Bottlenecks · Observability

PART IX The Open VTS Architecture

End-to-end design · Separation · Follow one telemetry record · Vendor questions

PART

GPS Tracking Is a Data System

Start with the map and work backwards. A vehicle marker can exist only after positioning, communication, decoding, normalization, processing, storage, and distribution have already succeeded.

01 — The Map Is the Last Step

The most visible element of a vehicle tracking platform is usually the live map. A marker moves, a label changes from stopped to moving, and an operator sees speed, ignition, location, or an alert. This can make GPS tracking look like a mapping problem. Architecturally, it is not. The map is a client of a larger data system.

Before a marker can move, a tracker must establish a position, decide that a report should be generated, encode the information into a protocol message, reach the server over a network, enter the correct protocol endpoint, be decoded, normalized, associated with the correct vehicle, stored or cached, interpreted as state, and distributed to the application. A map failure may therefore originate far away from the map itself.

A GPS device does not send a map pin. It sends a protocol packet.

THE MAP IS THE LAST STEP

A visible vehicle marker depends on every layer before it.

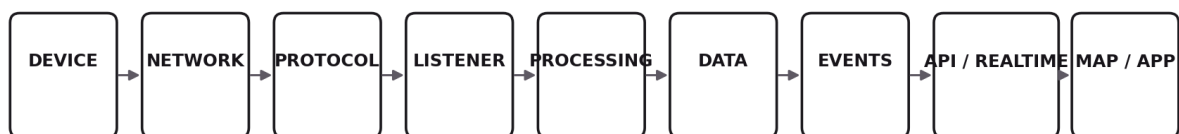


Figure 1 — The visible application is the final layer of a longer data path.

Why architecture matters to operations

This architectural view changes troubleshooting. If a vehicle is not updating, the operator should not jump immediately to the frontend. The problem may be GNSS reception, cellular connectivity, APN configuration, destination IP, a closed firewall port, the wrong protocol assignment, a decoder mismatch, downstream processing, stale current state, or application delivery. The same

view also changes platform design: each stage should have a clear responsibility and observable failure mode.

Modern GPS tracking is therefore best understood as a real-time distributed system that begins at the edge device and ends in business action. The rest of this guide follows that path in order.

02 — What a GPS Tracker Actually Does

A vehicle tracker is a compact edge computer. Its GNSS receiver determines position, but that is only one subsystem. A modem provides network connectivity. Inputs can read ignition, doors, panic buttons, or other digital states. Analog channels and external buses can carry fuel, temperature, CAN, or accessory data. Firmware decides when to create a report, what fields to include, whether to buffer data, and how to communicate with the server.

A TRACKER IS A SMALL EDGE COMPUTER

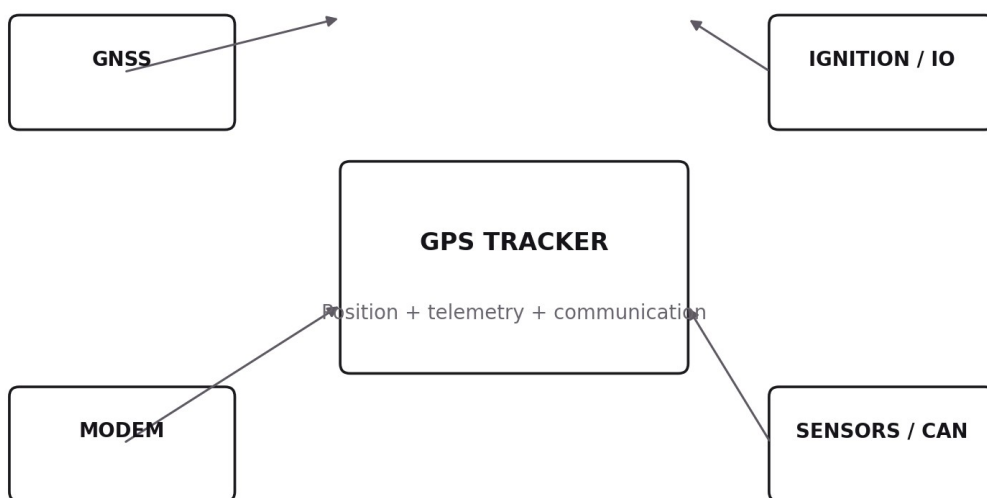


Figure 2 — A tracker combines positioning, communication, local logic, and optional vehicle/sensor inputs.

GNSS and cellular are different jobs

GNSS answers “where am I?” Cellular or another data network answers “how do I send that information to the platform?” A tracker can have a valid GNSS fix but no cellular connectivity, so the server receives nothing. It can also have network connectivity while GNSS reception is poor, causing stale or invalid positioning. These conditions should not be conflated.

Telemetry is broader than location

Depending on the hardware and protocol, a report can include timestamp, latitude, longitude, speed, heading, altitude, ignition, main power, battery, GSM quality, GNSS satellite status, odometer, input/output state, alarms, temperature, fuel, CAN values, driver identifiers, and proprietary sensor fields. A platform that handles only latitude and longitude may technically “track” a device while failing the operational use case.

03 — Reporting Logic

Trackers do not necessarily transmit at a constant rate. Reporting policy can change by motion, ignition, distance, heading change, elapsed time, or event. A common strategy is more frequent

reporting while a vehicle is moving and less frequent reporting while it is parked. Critical conditions such as SOS, power disconnect, or threshold events may trigger immediate messages.

REPORTING FREQUENCY TRADE-OFF



Reporting policy affects bandwidth, storage, event resolution, battery behavior, and server workload.

Figure 3 — Reporting policy trades time resolution against bandwidth, storage, battery behavior, and server workload.

Reporting interval is an architectural input

A fleet of 5,000 vehicles reporting every ten seconds generates a very different ingestion profile from the same fleet reporting every five minutes. The interval affects network traffic, listener concurrency, database write volume, retention growth, realtime update frequency, event evaluation, workflow volume, and downstream integration load. This is why capacity planning cannot be reduced to vehicle count alone.

Good platform architecture treats reporting frequency as a workload characteristic. Good device configuration treats it as an operational policy: frequent enough to support the business need, but not blindly aggressive.

PART

From Device to Server

The next layers translate hardware-specific communication into data the platform can understand.

04 — GPS Protocols

A GPS protocol defines how a particular device family represents information and exchanges messages with a server. It can specify login packets, message framing, device identifiers, timestamps, coordinate encoding, status bits, sensor values, event codes, checksums, acknowledgements, heartbeats, and command formats. Some protocols are binary and compact; others are textual and easy to inspect. Some are well documented; others are private or vary significantly by firmware.

CONCEPTUAL PROTOCOL PACKET



Real device protocols differ. A decoder interprets each field according to that protocol specification.

Figure 4 — Conceptual packet anatomy. Real protocols can differ substantially.

Protocol support is exact, not approximate

A manufacturer name is not enough to prove compatibility. Two models from the same vendor can use different protocols or protocol variants. A device may connect but expose only partial telemetry if its packet variation is not understood. Command behavior can also differ from telemetry behavior. The useful question is whether the exact device, protocol version, transport, required fields, events, and commands have been implemented and tested.

Protocol documentation is an asset

For organizations that buy hardware at scale, access to protocol documentation should be considered during device procurement. A low-cost device can create long-term integration cost if its protocol cannot be obtained or is changed without documentation. When a protocol specification is available, a tracking platform with a modular listener architecture can evaluate, implement, and test support without forcing a hardware replacement.

05 — TCP vs UDP

TCP and UDP are transport protocols beneath the GPS application protocol. They solve different network problems and should not be reduced to “reliable versus unreliable.” TCP provides a connection-oriented ordered byte stream with transport-level acknowledgement and retransmission behavior. UDP sends independent datagrams with lower transport overhead and no equivalent connection stream. A GPS application protocol using UDP can implement its own acknowledgements, sequence handling, or retries.

TCP AND UDP SOLVE DIFFERENT TRANSPORT PROBLEMS

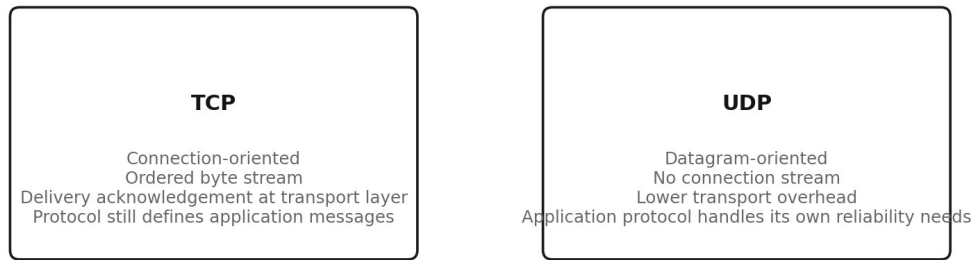


Figure 5 — TCP and UDP provide different transport semantics; the device protocol still defines application behavior.

The server must listen using the transport the device expects. A TCP-configured tracker pointed at a UDP listener will not work merely because the port number is correct. Firewall rules, NAT, cloud security groups, and load-balancing choices must also preserve the expected transport behavior.

06 — IP, DNS & Protocol Ports

A device needs a reachable destination. In practical deployments that usually means a public IP address or DNS name plus a protocol-specific TCP or UDP port. One Open VTS server can expose multiple listener ports, allowing different protocol families to reach the same infrastructure while remaining isolated at the listener level.

ONE SERVER, MULTIPLE DEVICE PROTOCOL PATHS

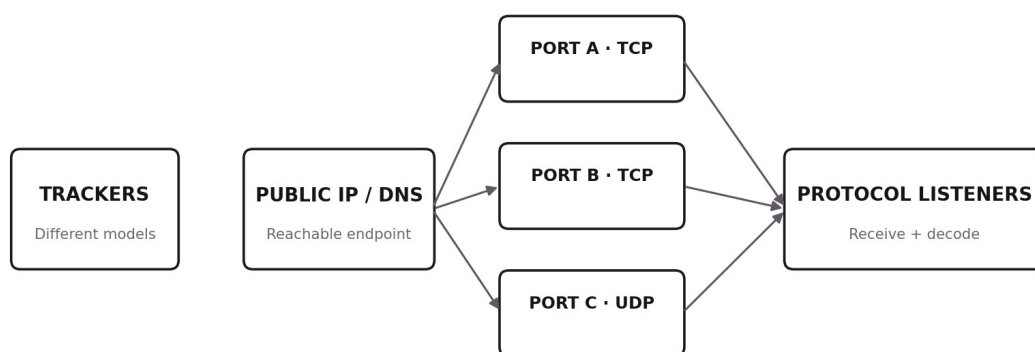


Figure 6 — Public reachability, transport, and port selection route a device to the correct listener.

Why port architecture matters

Ports provide a clean routing boundary. A Teltonika family can use one configured endpoint while a GT06-family device uses another. The listener associated with each endpoint knows how to

interpret its protocol. This simplifies debugging and avoids forcing every packet through one universal parser that must guess its format.

A supported protocol is operationally useless if the device cannot reach the listener. Network planning therefore belongs to device compatibility: public addressing, DNS, firewall rules, transport type, listener binding, and device configuration are all part of the path.

07 — The Listener

The listener is the hardware-facing edge of the software platform. Its job is to accept device traffic, manage connection or datagram behavior, determine packet boundaries, associate communication with a protocol implementation, identify the device, decode messages, send required acknowledgements, and forward normalized information into the application processing path.

A good listener isolates protocol complexity

Without a listener layer, device-specific logic tends to leak into reports, APIs, user interfaces, and business rules. A modular listener confines those differences to the ingestion boundary. Higher layers can work with normalized telemetry rather than repeatedly asking whether ignition came from a status bit, an IO field, or an event code.

Sessions and commands

Some TCP protocols keep a session open, send heartbeats, and expect commands or acknowledgements over that session. Others connect briefly, send a packet, and disconnect. A listener may therefore need connection identity, session state, pending command handling, timeout behavior, and protocol-specific acknowledgements. Bidirectional communication is a separate capability from receiving telemetry.

Failure handling begins here

Device data is time-sensitive. If the listener can receive packets but a downstream service is temporarily unavailable, the architecture should define what happens next. Open VTS includes journaling/spool-oriented mechanisms intended to preserve selected incoming traffic and replay it after downstream recovery. This improves resilience, but it should still be validated under the deployment's expected workload and failure scenarios.

08 — Protocol Decoding

A decoder converts protocol-specific bytes or text into meaningful fields. It verifies framing, reads identifiers, interprets timestamps and coordinates, converts numeric units, extracts bit fields, maps alarms, parses sensors, and may validate checksums. The result should be explicit data, not a raw packet that every later component must understand.

Decoding is also where seemingly small mistakes become large operational errors. A timezone misunderstanding can shift history. A signed/unsigned interpretation can corrupt sensor values. Swapped coordinate scaling can move a vehicle hundreds of kilometers. A wrong status-bit mapping can make ignition reporting unreliable. Protocol implementations therefore require both specification review and real-device testing.

Acknowledgement is part of the protocol

Some trackers will retransmit until they receive the correct acknowledgement. Others may treat a missing acknowledgement as a connectivity problem. The server must follow the protocol's

expected response sequence. A decoder and listener should therefore be tested as a conversation, not only as a one-way parser.

PART

From Protocol to Platform Data

Once a packet is understood, the platform must turn device-specific fields into durable, queryable, consistent operational data.

09 — Normalization

Normalization is the architectural step that gives a multi-device platform a common language. Three trackers may encode ignition differently, but reports and APIs should not need three different rules. The normalization layer maps device-specific semantics into stable platform fields and preserves device-specific extensions where they remain important.

NORMALIZATION CREATES A COMMON PLATFORM LANGUAGE

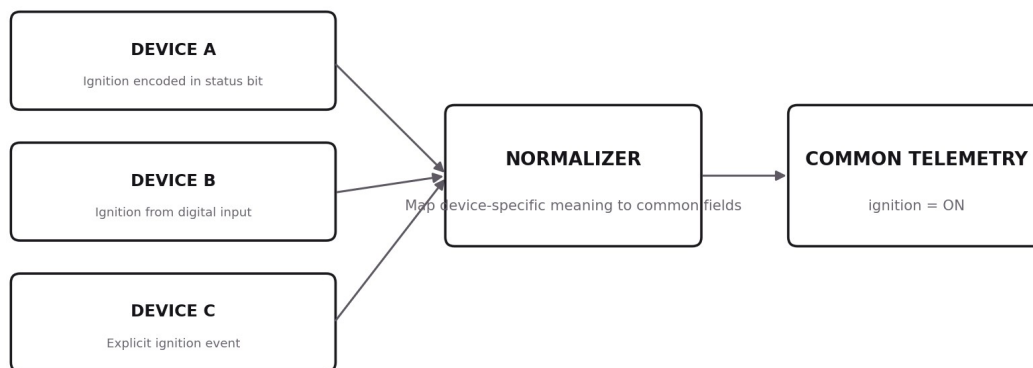


Figure 7 — Device-specific meaning is translated into common telemetry used across the platform.

Normalization enables mixed fleets. Once speed, location, ignition, event codes, and sensor values have consistent meanings, higher layers can calculate reports, evaluate rules, show current state, or expose APIs without coupling themselves to every protocol implementation.

Do not normalize away useful detail

A common model should simplify core operations without discarding meaningful device-specific information. Platforms often need both: standardized fields for common behavior and extensible attributes for specialized sensors, CAN signals, or vendor-specific events.

10 — Current State vs Historical Telemetry

Tracking systems answer two fundamentally different questions: “What is true now?” and “What happened over time?” Current state typically includes the latest known position, speed, ignition, motion, online/offline condition, and last report time. Historical telemetry consists of the ordered stream of observations used for routes, reports, events, analytics, and investigation.

CURRENT STATE AND HISTORY SERVE DIFFERENT QUESTIONS

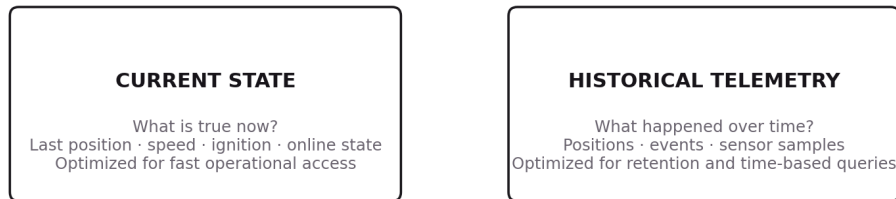


Figure 8 — Current state favors fast operational access; history favors durable time-based analysis.

Treating the historical database as the only source for every live map refresh can create unnecessary latency and load. Conversely, treating a fast in-memory current-state cache as the permanent system of record risks losing history. Modern architectures commonly separate these responsibilities while keeping them synchronized through the processing path.

11 — Time-Series Data

Telemetry is time-series data: many records for the same device, ordered by timestamp and continuously appended. This differs from ordinary application data such as a user profile or vehicle name, where updates are relatively infrequent. A single vehicle can create tens of thousands of records per day at aggressive reporting intervals, so storage and query design matter quickly.

Characteristics of telemetry storage

- high sustained write volume
- queries over time ranges
- per-vehicle filtering
- retention and cleanup
- aggregation for reports
- indexing trade-offs
- late or out-of-order packets
- continuous storage growth

Open VTS uses PostgreSQL with TimescaleDB capabilities for telemetry-oriented workloads. The value is not the product name by itself; it is the architectural recognition that time-series history deserves different storage and query treatment from ordinary business records.

Retention is part of architecture

If reporting interval doubles in frequency, storage growth can increase correspondingly before indexes, logs, journals, backups, and overhead are considered. Retention policy therefore affects infrastructure sizing, backup duration, historical report performance, and compliance. It should be a deliberate operational choice.

12 — Spatial Data

GPS systems reason about geography, not only coordinates. Geofences require point-in-polygon logic. POIs require proximity. Routes require path geometry. Distance calculations, containment, intersections, and geographic indexing are spatial operations. Open VTS uses PostGIS capabilities with PostgreSQL to support this class of geographic data and queries.

Spatial correctness also depends on coordinate systems, precision, and data quality. A point with poor GNSS accuracy can appear just outside a boundary even when the vehicle is physically near it. Operational rules should therefore consider real-world noise rather than assume every coordinate is exact.

13 — Application Data vs Telemetry Data

A modern platform contains at least two broad data families. Application/master data describes the operating model: vehicles, devices, users, drivers, SIMs, teams, permissions, geofences, routes, settings, integrations, and workflows. Telemetry data describes observations over time: positions, speeds, sensor values, events, and protocol-derived state.

Data family	Examples	Typical behavior
Application / master	Vehicle, user, driver, SIM, permission, geofence	Lower write frequency; relationship-heavy; CRUD operations
Telemetry / time series	Position, speed, ignition, sensor sample, device event	High write frequency; time-range queries; retention-driven growth
Operational / execution	Workflow runs, notifications, commands, activity	Event/execution history; troubleshooting and audit value

Separating these concepts makes system behavior easier to reason about. A vehicle name change should not require rewriting telemetry history. A telemetry retention cleanup should not delete the vehicle master record. A workflow execution history has different retention needs from live vehicle state.

PART

Turning Data Into Meaning

The platform becomes useful when normalized observations are interpreted as operational state, events, and geographic context.

14 — Vehicle State

Raw telemetry is an observation. Operational state is an interpretation of one or more observations. A packet may say speed = 0 and ignition = on. The platform may interpret the vehicle as stopped with ignition on. If no message arrives for an expected period, the vehicle may become stale or offline. If speed exceeds a policy threshold, it may enter overspeed state.

State needs time and context

“Offline” cannot be read directly from a GPS coordinate; it is inferred from the absence of expected communication. “Moving” may depend on speed, ignition, GPS jitter, and configured thresholds. “Stopped” may require persistence for a minimum duration. Architecture should keep these interpretations explicit rather than hide them inside arbitrary UI code.

15 — Events

Events turn telemetry into something operationally meaningful. A speed value becomes an overspeed event only when compared with a threshold. A coordinate becomes a geofence-entry event only when compared with a boundary and previous position/state. A temperature value becomes an alert only when it crosses a configured condition.

RAW DATA BECOMES AN EVENT THROUGH CONTEXT

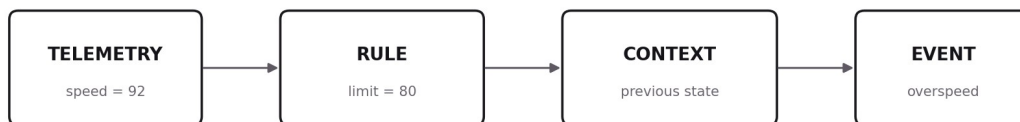


Figure 9 — Events combine telemetry, rules, and context.

Event quality matters more than event volume

A platform that creates an event on every repeated packet can overwhelm users and integrations. Good event logic distinguishes between continuing conditions and meaningful transitions. It also preserves enough context — vehicle, time, position, relevant values, previous state — to make the event actionable.

16 — State Change vs Repeated Data

Consider a vehicle traveling above an 80 km/h threshold for two minutes while reporting every ten seconds. Twelve packets may all satisfy the overspeed condition. If each packet creates a new alert, the operator receives twelve notifications for one operational episode. Stateful logic instead recognizes a transition from NORMAL to OVERSPEED, triggers once, and waits for a transition back to NORMAL.

STATE CHANGE PREVENTS ALERT FLOODS



Figure 10 — State transitions turn repeated telemetry into meaningful operational episodes.

This pattern applies beyond speed: ignition, temperature, geofence presence, route adherence, sensor alarms, connectivity, and many custom conditions. It is one reason a workflow or event engine benefits from access to previous state rather than evaluating every packet independently.

17 — Geofences, POIs & Routes

A point of interest, geofence, and route all describe geography, but they answer different operational questions. A POI identifies a meaningful location. A geofence defines an area whose inside/outside relationship can generate state or events. A route represents a planned path or sequence against which actual movement can be compared.

Concept	Architectural role	Typical questions
POI	Named point or small location reference	Which depot/customer/site is nearest?
Geofence	Boundary / area	Did the vehicle enter, exit, or remain inside?
Route	Expected path or ordered travel plan	Is the vehicle following the expected path and schedule?

The platform must combine spatial data with telemetry and time. Arrival is not only a point-in-polygon calculation; operationally it may also require dwell time, schedule context, direction, or trip assignment. This is where tracking evolves into fleet operations.

PART

Realtime Systems

Live tracking depends on low-latency state distribution, but durable history and asynchronous work still need different architectural paths.

18 — Why Live Tracking Is Realtime

A live map should not need to reload the entire history table every few seconds. After telemetry is processed, the current state can be updated and distributed to connected clients. A realtime layer pushes the relevant change — for example, new position, speed, status, or notification — to clients that are already connected.

REALTIME DISTRIBUTION

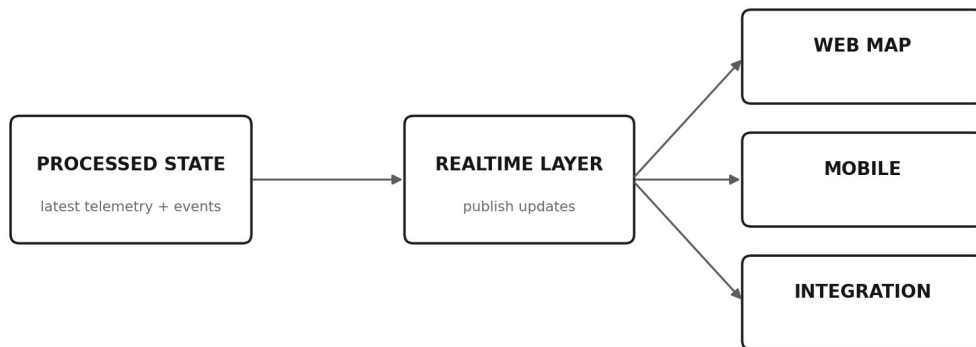


Figure 11 — Processed state can be distributed to web, mobile, and integrations without re-querying the entire history stream.

Realtime does not mean mathematically zero latency. Device reporting interval, cellular delay, server processing, network delivery, browser scheduling, and rendering all contribute. The useful engineering goal is predictable low latency with graceful reconnection and clear stale-state handling.

19 — Redis & Live State

Redis is commonly used in realtime systems because it provides fast in-memory data structures, pub/sub-style coordination, and queue-related primitives. In Open VTS it participates in live state, queueing/coordination, and other operational paths. It complements the database rather than replacing durable telemetry history.

Why separate hot state from durable history

A live map repeatedly needs the most recent value, not thousands of previous points. Keeping hot operational state in a fast layer allows the application to answer “what is happening now?” efficiently, while the database continues to preserve what happened before. The architecture should define how those layers recover and re-synchronize after restarts or failures.

20 — Queues & Workers

Not every operation should block telemetry ingestion. A notification may require an external provider. A workflow may call an HTTP service. A report can be expensive. Maintenance can scan large datasets. Queues let the ingestion path record work and continue, while dedicated workers process that work asynchronously.

QUEUES DECOUPLE HIGH-VOLUME WORK

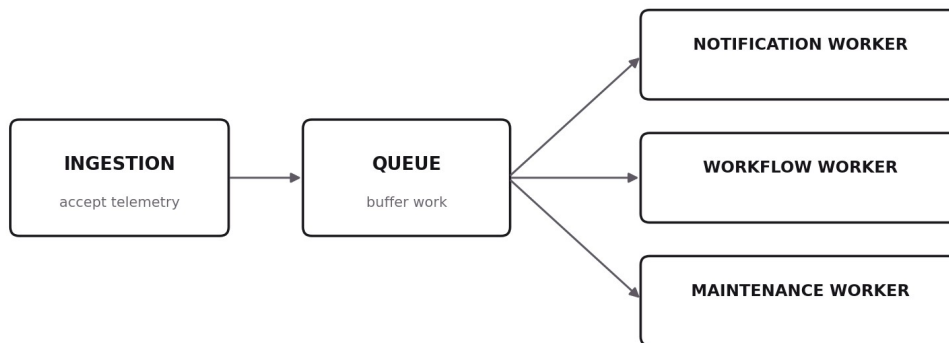


Figure 12 — Queues separate high-volume ingestion from downstream work that may be slower or failure-prone.

What queues provide

- decoupling between producers and consumers
- controlled concurrency
- retry policies
- visibility into backlog
- failure isolation
- backpressure handling
- separate worker scaling

Queues do not magically guarantee correctness. Jobs need idempotency where possible, bounded retries, dead-letter or failure visibility, and careful handling of duplicate execution. The platform should make queue health observable because a growing backlog is an early signal of downstream pressure.

21 — Reliability & Replay

The difficult reliability question is not “Does the service run?” but “What happens to data when a dependent service does not?” GPS data can be impossible to reconstruct later, so ingestion paths deserve explicit failure behavior.

Open VTS includes listener journaling/spool and replay-oriented mechanisms for selected downstream failure conditions. Conceptually, the listener can hold received work durably when the normal path cannot accept it, then replay after recovery. This creates a buffer between device communication and downstream availability.

Reliability still needs boundaries

Every buffer has capacity. Disk can fill. A device may retransmit. A server can fail before persistence. External providers can be unavailable for hours. Architecture should therefore define retention, overflow behavior, retry limits, replay ordering, duplicate handling, and operator alerts. “Resilient” is meaningful only when these boundaries are understood and tested.

PART

Automation & Integration

Once telemetry is normalized and stateful, the platform can move from observation to action and exchange data with the rest of the business.

22 — From Telemetry to Workflow

A workflow engine lets the platform evaluate operational conditions and perform actions without requiring a person to manually coordinate every response. The architectural chain is straightforward: a trigger creates work, logic evaluates context, an action executes, and the result is recorded for observability.

WORKFLOW ARCHITECTURE



Figure 13 — A workflow separates trigger, logic, action, and execution trace.

The value comes from context. A workflow may need the current vehicle, previous telemetry, a persistent state variable, assignment information, schedule context, or external data. A reliable workflow system should also make failure visible and avoid uncontrolled retries or repeated actions.

23 — Workflow Architecture in Open VTS

Open VTS includes a visual workflow architecture with trigger, logic, and action node categories. Trigger concepts include incoming data, schedules, and manual execution. Logic includes conditions, waiting, and custom code. Actions include HTTP requests, email, platform notifications, WhatsApp-related actions, report generation, supported device commands, and registered events.

The surrounding runtime maintains versions, executions, node-level results, routing, state, retries, errors, and execution history. This is important because automation should be explainable. When an external call fails or a condition takes an unexpected branch, operators need enough trace data to understand why.

Stateful workflows

For conditions such as ignition, overspeed, or sensor alarms, state can prevent repetitive actions. A workflow may remember that a vehicle is already in an alert state and trigger only when the state changes. This pattern is particularly useful when trackers report frequently.

24 — Device Commands

Telemetry is device-to-server communication. Commands reverse the direction: server to device. A platform may send configuration changes, output control, reboot requests, reporting interval changes, or protocol-specific messages — but only when the device, protocol, session, and command implementation support them.

Command delivery is protocol-specific

The platform must know the correct command format and whether the device needs an active TCP session, queued delivery, SMS fallback, acknowledgement, or other behavior. A device that reports telemetry successfully does not automatically support every remote command. Command authorization is also security-sensitive because actions may affect a physical vehicle or tracker.

For buyer evaluation, treat “supports commands” as an exact capability: which device, which command, which delivery mechanism, which acknowledgement, and which permission controls.

25 — REST APIs

REST APIs expose platform resources and actions through explicit request/response operations. They are useful when another system needs to retrieve vehicles, read telemetry, manage supported entities, perform an allowed operation, or integrate the tracking platform with ERP, TMS, CRM, BI, customer portals, and custom applications.

A useful API needs more than endpoints

- authentication and credential lifecycle
- authorization and resource scope
- stable schemas
- clear error behavior
- versioning or compatibility policy
- documentation and examples
- rate/performance expectations
- auditability for sensitive operations

Open VTS exposes a substantial documented HTTP API surface. The correct integration approach is to verify the exact operations needed by the use case rather than infer capability from the total number of internal routes.

26 — Realtime APIs

REST is excellent when a client knows when to ask. Realtime communication is better when the server needs to push changes as they happen. Live maps, notifications, vehicle status updates, and event streams are natural examples. Open VTS uses realtime Socket.IO-based patterns for areas including telemetry and notifications.

REST AND REALTIME COMPLEMENT EACH OTHER



Figure 14 — Request/response APIs and pushed realtime updates serve complementary integration patterns.

Realtime clients still need authentication, reconnection, subscription scope, duplicate handling, and stale-state recovery. A mobile app that reconnects after a network outage should be able to recover current state rather than assume it received every event while offline.

PART

Applications & User Experience

The frontend, mobile app, reports, and permission model are consumers of the architecture—not substitutes for it.

27 — The Web Application

The web application is the human-facing client of the platform. It renders maps, dashboards, reports, vehicles, users, workflows, notifications, and operational tools by calling backend services and consuming realtime updates. This separation matters because business logic and authorization should not live only in the browser.

The interface is a client of the platform architecture.

A strong frontend can make complex operations understandable, but it should not become the system of record. If another client — mobile, API, integration — performs the same operation, the backend should enforce the same rules. This is especially important for permissions and sensitive actions.

28 — Mobile Architecture

A mobile application is another client of the same platform. In Open VTS, the Flutter mobile application connects to a configured Open VTS server/API, authenticates, reads platform data, consumes realtime updates, and participates in notification flows. The mobile endpoint is configurable, which aligns with the customer-controlled deployment model.

Open-source mobile, proprietary core

The Open VTS mobile application is available with source code, while the core Open VTS platform remains proprietary self-hosted software. This distinction gives organizations flexibility at the client layer without misrepresenting the licensing model of the server platform.

From an architecture perspective, this also demonstrates the value of stable APIs: the mobile application does not need direct database access. It talks to the same controlled application boundary that other clients use.

29 — Reports

Reports are downstream products of the entire data pipeline. A distance report depends on correct coordinates, timestamps, normalization, retention, and calculation rules. An overspeed report depends on speed quality and threshold logic. A sensor report depends on protocol decoding and field mapping. If the ingestion layer loses meaning, reporting cannot reconstruct it later.

Every report begins at the tracker.

This is why report validation should begin with representative devices and known journeys. Compare timestamps, route points, ignition periods, sensor values, and event boundaries. A polished report design cannot compensate for incorrect upstream telemetry.

30 — Permissions Across the Architecture

Authorization should follow a user action from interface to resource. The frontend may hide an unavailable feature for usability, but the backend must still decide whether the authenticated identity is allowed to perform the requested operation on the specific resource.

Open VTS uses multiple role levels and scoped Team authorization concepts such as OWN, ASSIGNED, and TENANT for applicable capabilities. Architecturally the important pattern is: identity → role/capability → scope → resource. This prevents a user from gaining access merely by calling a hidden URL or API directly.

Layer	Question
Authentication	Who is making the request?
Role / capability	What action may this identity perform?
Scope	Which organizational/resource boundary applies?
Resource	Is this specific object inside that boundary?
Audit / denial	Can rejected access be observed?

PART

Scaling the Architecture

A GPS platform scales according to workload and bottlenecks, not one universal vehicle number.

31 — What Creates Scale?

Vehicle count is only the first term in the workload. Reporting interval determines packet frequency. Protocol complexity affects decoding. Sensor-heavy payloads increase processing and storage. Retention increases database size. Concurrent map users create realtime and query load. Reports create historical read pressure. Workflows and notifications amplify each telemetry record into additional work. APIs add external traffic.

A practical workload model

Think in dimensions: vehicles × reports per vehicle, plus payload complexity, retention, concurrent users, reports, automation, and integrations. Two fleets with the same vehicle count can require very different infrastructure because their reporting interval and downstream workloads differ.

This also explains why the same platform can appear fast in a small demo and struggle in production. Capacity testing must resemble the actual operating profile, not only the number of registered vehicles.

32 — Where Bottlenecks Appear

Distributed systems rarely have one permanent bottleneck. At one scale the listener may be CPU-bound. At another, disk write latency may dominate. A report workload may stress database queries while realtime delivery remains healthy. An external notification provider may throttle actions even when the Open VTS server has spare capacity.

Potential bottleneck	Typical symptom	Evidence to inspect
Network / listener	connections fail, packet delay	socket errors, ingress rate, CPU
Backend CPU / memory	processing latency, restarts	CPU, memory, GC/process health
Redis / queues	growing backlog, stale actions	queue depth, worker rate, Redis health
Database / disk	slow history/reports, write pressure	IO latency, query time, DB growth
Realtime clients	delayed map updates	socket counts, publish latency
External APIs	workflow retries/failures	HTTP latency, error rate, provider limits

Scale the bottleneck, not the assumption.

Good observability makes bottlenecks visible before operators respond by blindly adding hardware.

33 — Observability

Architecture is not production-ready if operators cannot tell whether it is healthy. Monitoring should cover infrastructure, services, queues, databases, listener traffic, telemetry freshness, and integration errors. Logs remain valuable for detail, but a human should not have to search logs manually to discover that a queue has been growing for six hours.

Useful operational signals

- CPU, RAM, disk utilization and disk latency
- backend/listener service health
- database and Redis availability
- queue depth and worker throughput
- incoming packet rate and protocol errors
- online/offline/stale vehicle trends
- API latency and error rate
- workflow/notification failure rate
- database growth and retention cleanup

Open VTS contains health and monitoring mechanisms across backend, data services, queues, memory, and listener/runtime functions. Production operators should connect those signals to their broader infrastructure monitoring and define thresholds appropriate to their workload.

PART

The Open VTS Architecture

Now assemble the concepts into one end-to-end view of the current Open VTS platform.

34 — Open VTS End-to-End Architecture

Open VTS is organized as a customer-controlled, self-hosted tracking platform with distinct hardware-ingestion, application-processing, data, asynchronous-work, API, realtime, web, and mobile responsibilities. The separation allows hardware-specific communication to remain at the listener boundary while higher-level services work with normalized fleet and application data.

OPEN VTS END-TO-END ARCHITECTURE

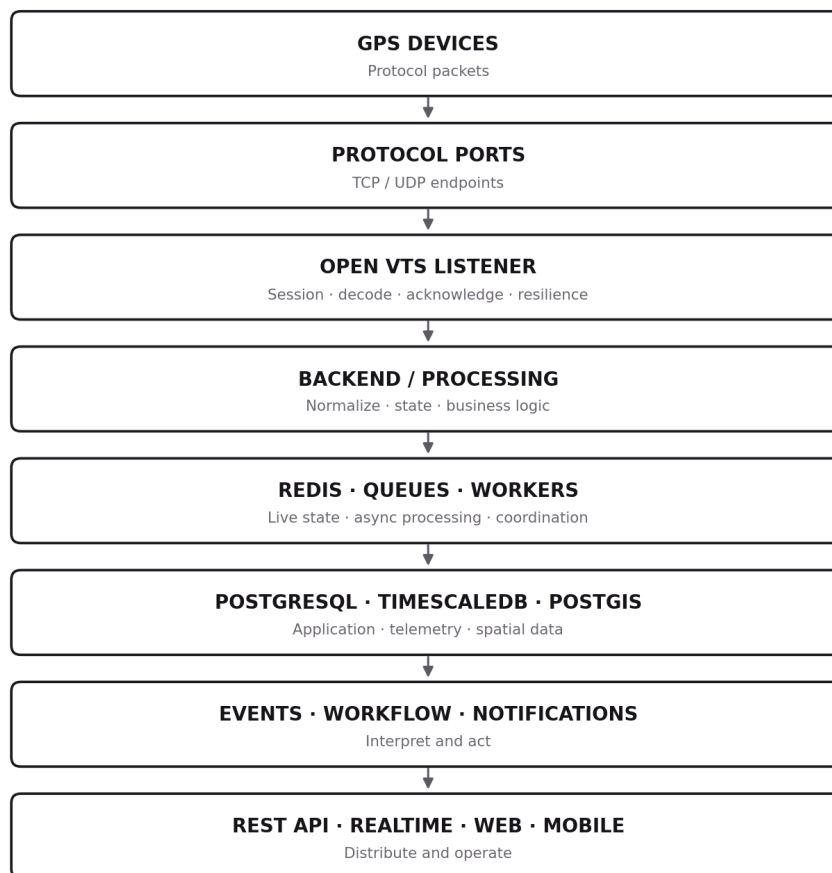


Figure 15 — Canonical Open VTS end-to-end architecture.

Major responsibilities

- GPS devices transmit protocol packets to dedicated TCP/UDP listener ports.
- The Open VTS Listener handles device communication, protocol decoding, acknowledgement, and ingestion resilience.
- Backend services normalize and process telemetry and provide the business/API layer.
- Redis, queues, and workers support current state, coordination, and asynchronous processing.

- PostgreSQL with TimescaleDB and PostGIS capabilities provides application, telemetry, time-series, and spatial data services.
- Events, notifications, reports, and workflow automation interpret and act on processed data.
- REST and realtime interfaces distribute information to the web application, open-source mobile application, and external integrations.

This diagram is intentionally conceptual. Production topology, process counts, deployment packaging, and infrastructure layout can vary by release and workload.

35 — Why the Architecture Is Separated

Protocol isolation

Device families evolve independently. Keeping protocol parsing in listener modules limits the blast radius of hardware differences and makes new documented protocols easier to implement and test.

Processing separation

Backend business logic can operate on normalized data rather than device-specific packet formats. This simplifies reports, permissions, workflows, APIs, and applications.

Asynchronous work

Queues and workers prevent slower actions such as notifications, workflows, and maintenance from unnecessarily blocking the ingestion path. They also create natural points for retries, concurrency control, and backlog observation.

Data specialization

Current operational state, historical telemetry, application records, and spatial geometry have different access patterns. Redis, PostgreSQL/TimescaleDB, and PostGIS capabilities allow those workloads to be handled according to their nature rather than forcing one storage mechanism to do everything.

Client independence

Web, mobile, and external systems consume API/realtime boundaries rather than coupling themselves directly to internal database structures. This improves security, consistency, and maintainability.

36 — Follow One Telemetry Record

Imagine Vehicle 102 reports: ignition ON, speed 62 km/h, coordinates X/Y, timestamp 10:42:15. The user eventually sees one map movement, but the platform performs a sequence of architectural steps first.

FOLLOW ONE TELEMETRY RECORD



Example: Vehicle 102 · ignition ON · 62 km/h · coordinates X/Y

Figure 16 — One telemetry record travels through multiple specialized responsibilities.

1. The tracker gathers GNSS and vehicle state and creates a protocol message.
2. The modem transmits the message to the configured Open VTS IP/DNS and protocol port.
3. The corresponding listener receives the packet and identifies the protocol/device.
4. The decoder interprets the packet and extracts timestamp, coordinates, speed, ignition, and other fields.
5. The ingestion/backend path maps those fields into normalized telemetry.
6. Current vehicle state is updated so live clients can obtain the latest operational view.
7. The telemetry record and relevant event information are persisted for history and reporting.
8. Rules or workflow triggers evaluate the new values in context; an action may be queued if required.
9. Realtime distribution publishes the updated state to authorized connected clients.
10. The web or mobile application renders the new position and operational status to the user.

If an operator understands this path, troubleshooting becomes structured. Each stage has specific evidence: device logs/configuration, network reachability, listener logs, decoder output, backend health, queue status, database records, realtime delivery, and client rendering.

37 — Architecture Questions to Ask Any GPS Vendor

Use these questions when evaluating the engineering depth of any tracking platform. The goal is not to demand one specific architecture; it is to make dependencies, responsibilities, and failure behavior explicit.

11. How does the platform identify the exact protocol used by a device?
12. Which TCP/UDP ports are mapped to our device families?
13. How are new protocols implemented and tested?
14. How are packet acknowledgements and device sessions handled?
15. What happens if a device connects but sends an unsupported packet variant?
16. How is device-specific telemetry normalized?
17. Which data represents current state and which represents history?
18. How are out-of-order or late telemetry records handled?
19. How is time-series retention managed?
20. How are geofences and spatial operations evaluated?
21. How is online/offline/stale state determined?
22. How does the event system avoid repeated alert floods?

23. Which operations are asynchronous and queue-backed?
24. How are failed queue jobs retried and observed?
25. What happens to incoming telemetry during a temporary backend outage?
26. Which data is persisted before an acknowledgement is considered durable?
27. What realtime interface is used for live maps?
28. How do realtime clients reconnect and recover state?
29. Which REST API operations are documented for customers?
30. How are API permissions and resource scopes enforced?
31. How are device commands authorized and acknowledged?
32. Can telemetry trigger stateful workflows?
33. How are workflow executions and failures traced?
34. What components depend on Redis?
35. What database technologies hold application, time-series, and spatial data?
36. How does database growth affect reports and retention?
37. Which health checks and metrics are available?
38. How is queue backlog monitored?
39. How are protocol parsing errors surfaced?
40. Which benchmark workload and environment support published scale claims?

How to use the architecture review

Do not try to answer all thirty questions in a marketing meeting. Use them as a technical due-diligence agenda. Start with the layers that create the highest operational risk for your organization: device protocols, telemetry durability, permissions, integration, or scale. Then ask the vendor to demonstrate the behavior in a test environment and record the evidence beside each answer.

For critical claims, prefer a working product, protocol/device test, API reference, architecture evidence, measured workload result, or contractual commitment over presentation language. A clear limitation is often safer than an undefined promise because it can be planned around.

Review step	Output
Architecture walkthrough	Document the major components and control boundaries.
Failure walkthrough	Ask what happens when a listener, backend, Redis, database, or external service is unavailable.
Device proof	Test representative models, protocol fields, sensors, and commands.
Data proof	Trace one telemetry record from packet to history, event, API, and application.
Scale proof	Record workload, hardware, test duration, metrics, and observed bottleneck.

Architecture Reference — The 10 Layers

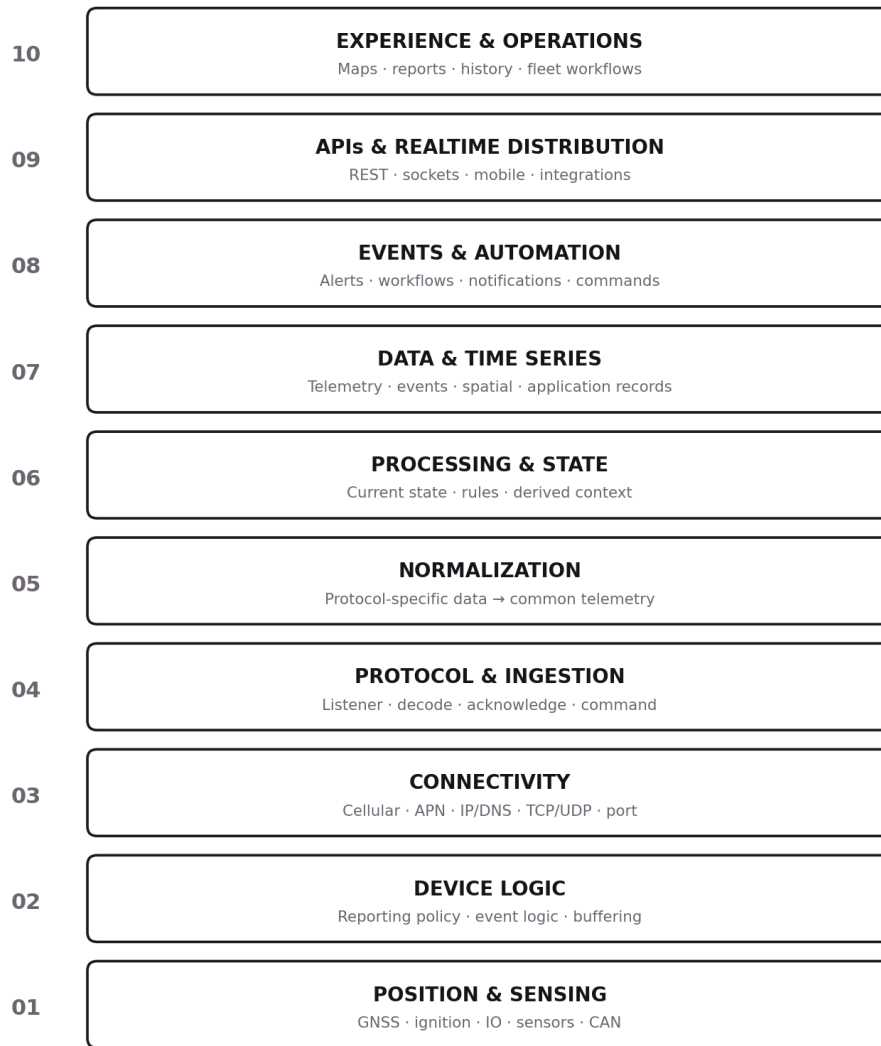


Figure 17 — The 10-Layer Modern GPS Tracking Architecture.

The ten layers provide a reusable way to reason about almost any GPS tracking platform. Product implementations differ, but every serious system must solve the underlying responsibilities: sense, communicate, understand, preserve, interpret, distribute, and present.

FROM PACKET TO DECISION



Figure 18 — From Packet to Decision: the compact data-flow reference.

Glossary

Term	Meaning
Acknowledgement (ACK)	A protocol response confirming receipt or acceptance of a message according to the device protocol.
APN	Access Point Name; a cellular-network configuration used by many trackers to reach packet data services.
API	Application Programming Interface; a controlled software boundary for reading data or performing supported operations.
Backpressure	A mechanism for slowing or buffering producers when downstream processing cannot keep up.
Decoder	Software that interprets a device protocol packet and extracts meaningful fields.
Event	An operationally meaningful occurrence derived from telemetry, rules, and context.
Geofence	A geographic boundary used for inside/outside state and related events.
GNSS	Global Navigation Satellite System; the broader satellite positioning category that includes GPS.
GPS tracker	An edge device that gathers position/vehicle information and transmits telemetry to a platform.
Ingestion	The process of receiving device data into the platform.
Listener	A hardware-facing network service that accepts and interprets supported device protocol traffic.
Normalization	Mapping protocol-specific values into a stable common telemetry model.
POI	Point of Interest; a meaningful geographic reference location.
Protocol	The message structures and communication rules used by a device family.
Queue	A buffer that stores work for asynchronous processing by workers.
Realtime	Low-latency delivery of state or events to connected clients as changes occur.
Redis	An in-memory data platform commonly used for fast state, coordination, pub/sub, and queue-related functions.
State	The platform's current interpretation of operational conditions such as moving, stopped, ignition, online, or alert state.
Telemetry	Time-stamped operational observations transmitted by devices.
Time series	Data organized as observations over time, usually with high append/write frequency.
Worker	A process that consumes queued work such as workflow actions, notifications, or maintenance.
Workflow	A defined trigger/logic/action process used to automate operational behavior.

References & Further Reading

For final publication, current versions of the following primary sources should be used where external protocol or standards statements require citation:

- IETF RFC 9293 — Transmission Control Protocol (TCP)
- IETF RFC 768 — User Datagram Protocol (UDP)
- GPS.gov — official GPS system information and performance fundamentals
- PostgreSQL official documentation
- TimescaleDB official documentation
- PostGIS official documentation
- Redis official documentation
- OWASP API Security Top 10 and OWASP ASVS for API/application security context
- Device-manufacturer protocol specifications for hardware-specific implementation and testing

Open VTS resources

- Supported Devices & Ports
- API Reference
- Deployment & Infrastructure Planning Guide
- Security resources
- Product documentation
- Mobile application repository/resources
- Current licensing and commercial terms

Any numerical Open VTS device/protocol/API/performance claim should be regenerated and verified against the current released implementation immediately before publication.

About Open VTS

Open VTS is a customer-controlled, self-hosted vehicle tracking platform for GPS and telematics businesses, fleet operators, integrators, and organizations that want to operate tracking infrastructure on environments they control. The platform brings together device-protocol listeners, telemetry processing, time-series and spatial data, live fleet operations, reports, alerts, role-based access, APIs, realtime communication, workflow automation, and web and mobile experiences.

The core Open VTS platform is proprietary software. The Open VTS mobile application is separately available with source code. Open VTS is designed to reduce the software deployment complexity traditionally associated with operating a full vehicle tracking platform while preserving the architectural control needed for serious fleet and telematics operations.

Your Infrastructure. Your Fleet Data. Your Rules.

Open VTS · Track Without Limits. Powered By You.