NEXUS WORKSHOPS LLC

Robot Telemetry:

A Complete Field Guide to Capturing, Storing, and Analyzing FTC Robot Data

Nexus Workshops LLC | Version 1.0 | March 2026

Covers: Networks, IP, UDP, WiFi setup, Wireshark, Python data logging, robot instrumentation, triplestores, SQL, LibreOffice Calc graphing

Nexus Workshops LLC

Robot Telemetry: A Complete Field Guide

1. Why This Guide Exists

You have a robot. It has motors, sensors, a chute, a hood, an IMU, and a drive system. When something goes wrong during a match -- and things always go wrong -- you are left guessing. Was the left motor drawing too much current? Did the chute hood fail to open all the way? Did the robot lose its heading during a turn? Was the loop running at full speed or did it stall somewhere?

This guide teaches you how to answer all of those questions with data. Not guesses. Not memory. Actual numbers recorded at your loop rate from your running robot, stored in a database on your laptop, and queried in any way you can imagine.

Along the way you will learn things that go well beyond FTC robotics: how computers communicate over networks, why databases are designed the way they are, how to write SQL queries, and how to think about data systematically. These are skills that engineers, scientists, and software developers use every single day -- and they are genuinely difficult to master. Take your time with this guide. Read sections more than once. The concepts build on each other, and the payoff comes from deep understanding, not from skimming.

1.1 These Skills Matter Far Beyond This Robot

If you are planning to go to college and pursue engineering, computer science, physics, medicine, or almost any technical field, what you learn here is not a detour from that path. It is the beginning of it.

The ability to instrument a system, collect data during operation, store it efficiently, and query it intelligently is one of the most valuable skills in professional engineering. It is also one of the hardest to develop, because it requires you to think about problems from multiple levels at once: the physical system, the software, the data model, and the analysis.

Consider where these exact concepts appear in the professional world:

Field	How This Applies
Aerospace	Rockets and spacecraft send continuous telemetry streams during flight. Engineers at mission control receive thousands of data points per second from sensors across the vehicle. When something goes wrong, they query that data to reconstruct exactly what happened and in what order. The 2003 Columbia investigation, the analysis of every SpaceX anomaly, and the debugging of every Mars rover relies on exactly this: structured data capture, timestamped records, and systematic querying.
Defense	Autonomous vehicles, missile systems, and tactical robots all generate high-rate sensor streams during operation. Engineers must understand system behavior under conditions that cannot be reproduced in a lab -- high G-forces, jamming environments, electronic warfare. The data stream is the only window into what happened.
Underwater Systems	Underwater vehicles cannot be observed directly during a mission. Remotely operated vehicles and autonomous submarines log all sensor data continuously and transmit what they can over low-bandwidth acoustic links. Post-mission analysis of the recorded data is the primary debugging tool. Dynamics that seem predictable in simulation often behave completely differently underwater, and only

Field	How This Applies
	the data reveals why.
Medical Devices	Implantable devices, surgical robots, and diagnostic instruments log operational data continuously. When a device behaves unexpectedly, engineers analyze the logged data to distinguish a software fault from a sensor error from a patient-specific response. In cancer treatment, linear accelerators and proton therapy systems rely on real-time monitoring and historical data to ensure the treatment beam is hitting exactly the right target with exactly the right dose. A system that cannot be observed cannot be trusted.
Autonomous Vehicles	Every autonomous car generates terabytes of sensor data per hour of driving. The development of reliable autonomy depends entirely on the ability to find the specific moments in that data where the system behaved unexpectedly, understand why, and verify that the fix works. The underlying pattern -- log everything, query intelligently -- is the same whether you are debugging a 40kg FTC robot or a 2000kg self-driving car.

Engineers who master this skill -- who can look at a misbehaving system and find the answer in data rather than intuition -- see problems differently than their peers. They ask better questions. They validate changes instead of assuming them. They discover failure modes before they become catastrophic. That habit of mind, developed now on a robot in a school gym, carries into every technical role you will ever hold.

> **This is not easy to master.** The concepts in this guide span networking, database design, query languages, data analysis, and systems thinking. That is a lot. Take your time. Return to sections as your experience grows. The engineers who become exceptional at this are the ones who stayed curious long after they got the basic system working.

1.2 Why Standard FTC Telemetry Is Not Enough

If you have written FTC code before, you have probably used the built-in telemetry system:

```
telemetry.addLine("Left power: " + leftPower);
telemetry.addLine("Right power: " + rightPower);
telemetry.update();
```

This is useful. It shows values on the Driver Hub screen and in the FTC log. But it has significant limitations that become painful the moment you are trying to debug anything real:

- It is ephemeral. The moment the op mode ends, the data is gone. You cannot look at what the robot was doing three seconds before a failure.
- It is not queryable. You cannot ask "show me every moment where left_power exceeded right_power by more than 0.1" from the Driver Hub screen.
- It is not timestamped with precision. The FTC log has timestamps but they are not synchronized to sensor readings in a useful way for analysis.

- It is visible to the driver but not to the engineer. During a match, the driver is watching the field. Nobody is reading the telemetry screen.
- It does not persist across runs. Comparing this match to the last match is impossible.

The FTC built-in telemetry is a dashboard, not a recorder. It answers the question what is the robot doing right now. The system in this guide answers the question what did the robot do, when did it do it, and how does that compare to last time.

Both are useful. They solve different problems. The goal is not to replace the built-in telemetry -- it is to add a persistent, queryable record that you can analyze after the fact.

1.3 How Not to Let Debugging Consume Your Season

There is a real danger in building a telemetry system: it can become the project instead of a tool. You can spend three weeks perfecting a web-based dashboard that displays beautiful real-time charts, with color-coded alerts and animated field overlays, and at the end of it have a worse robot than you started with because you spent all your time on the debugging infrastructure instead of the robot.

This trap has a name in professional engineering: yak shaving. You need to fix the robot. To fix the robot you need data. To get the data you need a logger. To make the logger useful you need a better database schema. To make the schema useful you need a query tool. To make the query tool useful you need a better UI. Suddenly you are shaving a yak and the robot is still broken.

The approach in this guide is deliberately minimal. A single Python script. A two-table SQLite database. A schema that never changes. Queries you run by hand in DB Browser and paste into a spreadsheet. No web server. No GUI. No custom dashboards. No build pipeline.

That minimalism is not a limitation -- it is a feature. The entire value is in getting to the data quickly, finding the problem, and getting back to improving the robot. All within your deadline before the next meet.

There are also technical pitfalls to avoid when instrumenting your robot:

- Logging too much data makes the payload large and can increase loop time. Log the signals that matter, not every variable in your program.
- Parsing JSON and writing to a database on the robot would be too slow. Keep the database on the laptop -- the robot only sends raw UDP packets.
- Building a complex analysis pipeline before you understand your data is premature. Start with simple queries. You will discover what you actually need.
- Spending time on visualization before you have found anything interesting. A text query result that shows a stall is more useful than a beautiful chart of nothing.

The magic in this system is in the SQL queries. A pile of triples -- thousands of rows of subject, predicate, object -- looks like nothing. But a well-written SQL query can reach into that pile and surface a hidden pattern: a motor that degrades over the course of a match, a sensor that behaves differently in

the second half of autonomous, a timing gap that only appears when a specific game element is in the intake. The data was always there. The query reveals it.

That is what this guide is ultimately about. Not the Python script, not the Java class, not the database schema. Those are just scaffolding. The goal is to develop the habit of capturing data and the skill of asking it the right questions. Both are learnable. Both take practice. And both will serve you for the rest of your engineering career.

What you will need: A Windows or Linux laptop, Java, Python 3, DB Browser for SQLite (free), LibreOffice Calc (free), and a USB WiFi adapter or your laptop's built-in WiFi.

2. How Your Laptop Talks to the Robot

Before you can send or receive any data, you need to understand a few fundamentals about how computers on the same network communicate. These concepts apply everywhere -- not just in robotics -- and understanding them will help you debug connection problems when they arise.

2.1 What is a Network?

A network is a group of devices that can communicate directly with each other. Your home WiFi is a network. The school's hallway WiFi is a network. The Control Hub's built-in WiFi hotspot is also a network -- a small private one, just for your robot and the devices that connect to it.

Devices on the same network can send data to each other directly. Devices on different networks cannot communicate unless there is a router between them that knows how to forward traffic. For your robot telemetry system, everything needs to be on the same network: the Control Hub and your laptop must both be on the Control Hub's WiFi.

2.2 IP Addresses: Identity on a Network

Every device on a network has an IP address -- a number that uniquely identifies it, similar to a street address for a house. When your laptop sends a packet to the robot, it addresses it to the robot's IP address. When the robot sends telemetry back, it addresses it to your laptop's IP address.

IP addresses in the version used here (IPv4) are written as four numbers separated by dots, each between 0 and 255:

```
192.168.43.1      -- the Control Hub (always this address)
192.168.43.100    -- your laptop (assigned automatically)
192.168.43.101    -- another device, such as a Raspberry Pi
```

The Control Hub always assigns itself the address 192.168.43.1. Your laptop gets assigned an address automatically by the Control Hub when it joins the network. This automatic assignment is called DHCP. You do not configure it -- it just happens when you connect.

2.3 Subnets: What Can Talk to What

A subnet is the range of IP addresses that belong to the same network. The subnet defines which addresses can communicate directly with each other without going through a router.

The Control Hub network uses the subnet 192.168.43.0/24. The /24 means the first three numbers (192.168.43) identify the network, and the last number identifies the individual device. Any device with an address starting with 192.168.43 is on the same network and can talk directly to any other device with the same prefix.

Address	What it is	Can it reach 192.168.43.1?
192.168.43.1	Control Hub	It is the Control Hub
192.168.43.100	Your laptop on the robot WiFi	Yes -- same subnet
192.168.43.101	A Raspberry Pi on robot WiFi	Yes -- same subnet
192.168.1.50	Your laptop on home WiFi	No -- different network
10.0.0.5	School network device	No -- different network

This is the most common connection mistake: you try to run telelog.py and nothing arrives, because your laptop is connected to the school WiFi instead of the Control Hub WiFi. Always check which network your laptop is on before debugging anything else.

2.4 Ports: Which Application Receives the Data

An IP address identifies a device. A port number identifies a specific application running on that device. When your robot sends a UDP packet to your laptop on port 3000, the operating system looks at the port number and hands the packet to whichever application is listening on port 3000 -- in this case, telelog.py.

Port numbers range from 0 to 65535. Ports below 1024 are reserved for well-known services (port 80 is web traffic, port 22 is SSH). Port 3000 is in the safe zone for custom applications. Both sides just have to agree on the same number -- if you change it in telelog.py with --port, you must change it in your Java code to match.

2.5 Inside a Packet: IP and UDP Frame Structure

When your robot sends telemetry, the data travels as a packet through the network. A packet is not just your raw JSON payload -- it is wrapped in layers of header information that the network uses to route

and deliver it. You rarely need to think about these layers, but understanding them helps when you are debugging with tools like Wireshark.

A UDP telemetry packet sent from your robot looks like this from the outside in:

Layer	Header Fields	Size	Purpose
Ethernet frame	Source MAC, Destination MAC, EtherType	14 bytes	Physical delivery on the local WiFi
IP header	Source IP, Destination IP, Protocol, TTL	20 bytes	Network routing to the right device
UDP header	Source Port, Destination Port, Length, Checksum	8 bytes	Delivery to the right application
Your payload	JSON text -- your actual telemetry data	varies	The data you care about

The fields that matter most for your system:

- Source IP: 192.168.43.1 (the Control Hub). telelog.py logs this as the source.
- Destination IP: your laptop's address on the Control Hub network.
- Destination Port: 3000 (or whatever you configured). This is what telelog.py binds to.
- Payload: your JSON telemetry string, encoded as UTF-8 bytes.

Everything above the payload is added and stripped automatically by the operating system. Your Java code on the robot only provides the payload and the destination address and port. Your Python code on the laptop only sees the payload and the source address. The headers exist in the middle but are transparent to your application code.

2.6 The FTC Control Hub WiFi Network

The FTC Control Hub acts as a WiFi access point -- it creates its own wireless network, just like your home router does, except it is broadcasting from inside your robot.

When the Control Hub powers on, it broadcasts a WiFi network with an ESSID (the network name you see on your laptop) that looks something like:

```
FTC-13532-RC
```

The number in the middle is your team number, and RC stands for Robot Controller. To receive telemetry data from the robot, your laptop must be connected to this network -- not your school's WiFi, not your home network, this specific network created by the Control Hub.

> **Note:** *Only one device can act as the Driver Station while connected to the Control Hub. Your telemetry laptop is a separate listener -- it joins the same network but does not interfere with the Driver Station.*

2.7 Connecting Your Laptop

There are three practical ways to connect your laptop to the Control Hub network for telemetry.

Option A: Built-in WiFi (simplest)

Open your network settings and connect to the Control Hub's network the same way you connect to any WiFi network. No hardware required.

- Pros: No extra hardware, works immediately
- Cons: Your laptop loses internet access while connected to the robot network

Option B: USB WiFi Adapter (recommended for competition)

A USB WiFi adapter gives your laptop a second wireless network interface. Your built-in WiFi stays on the internet. The USB adapter connects to the robot network.

- Pros: Keep internet access, cleaner separation of networks
- Cons: Requires a $15-30 USB WiFi adapter -- any modern 802.11ac adapter works

Option C: Raspberry Pi as a WiFi Bridge

A Raspberry Pi connected to the Control Hub WiFi runs telelog.py locally and logs all telemetry onboard. Your laptop then connects to the Pi via a direct Ethernet cable to retrieve the database after the session.

- Pros: No laptop needed at the field; most stable data capture during matches
- Cons: Requires setting up the Pi in advance and copying the database file afterward

> **Note:** *For competition use, Option C is the most robust -- the Pi logs every match automatically with no manual steps. For day-to-day debugging at the workshop where you want immediate access to the database, Option A or B is simpler and faster to set up.*

2.8 Verifying Packets with Wireshark

When something is not working -- telelog is running but no packets arrive, or packets arrive but look wrong -- you need a way to see exactly what is on the network. Wireshark is the standard tool for this.

Wireshark is a free, open-source network packet analyzer. It captures every packet passing through your network interface and lets you inspect each one in detail: the headers, the payload, the timing, all of it. Download it at wireshark.org.

To verify your robot's telemetry is arriving at your laptop:

1. Open Wireshark
2. Select the network interface connected to the Control Hub network (look for the one with traffic activity)
3. In the filter bar at the top, type:

```
udp port 3000
```

4. Press Enter to apply the filter
5. Start your robot's op mode
6. You should see packets appearing in Wireshark every time the robot sends telemetry

Each row in Wireshark is one packet. Click on a row to expand it in the detail panel below -- you will see the Ethernet header, the IP header (with source and destination addresses), the UDP header (with port numbers), and the raw payload. You can right-click the payload section and select Copy > Bytes as Printable Text to see your JSON.

Wireshark is also invaluable for diagnosing exactly why connection is failing:

- No packets visible at all: the robot is not sending, or you are on the wrong network interface
- Packets visible in Wireshark but not in telelog: telelog is bound to the wrong port or interface
- Packets visible but payload looks wrong: encoding or JSON formatting issue in the robot code
- Packets arriving at irregular intervals: loop timing problem on the robot side

> **Note:** *Use the display filter udp.port == 3000 to see only your telemetry. Without a filter, Wireshark captures everything on the interface which can be overwhelming.*

3. Why UDP? Understanding Network Protocols

3.1 What is a Network Protocol?

A protocol is a set of rules that two computers agree to follow when communicating. Just as a conversation has rules -- you take turns, you wait for the other person to finish, you acknowledge what they said -- computers need rules for how to send and receive data.

The two most common protocols for sending data between computers are TCP and UDP. They both send data as packets across a network, but they make very different tradeoffs.

3.2 TCP: Guaranteed Delivery

TCP (Transmission Control Protocol) is the protocol your web browser uses. When you load a web page, every single byte of that page must arrive correctly. If a packet is lost along the way, TCP automatically requests it to be resent and waits until it arrives before showing you anything.

This reliability comes at a cost: time. There is a handshake when the connection is established, acknowledgments sent back for every packet received, and retransmissions when things go missing. For a web page that is fine -- you do not care if it takes half a second longer. But for a robot control loop sending data many times per second, waiting for a retransmission is unacceptable.

> **TCP tradeoff:** Every packet is guaranteed to arrive, in order, with no errors. But this requires waiting for acknowledgments, which adds latency.

3.3 UDP: Low Latency, Fire and Forget

UDP (User Datagram Protocol) works differently. When the robot sends a telemetry packet over UDP, it fires it onto the network and immediately moves on. It does not wait for an acknowledgment. It does not check whether the packet arrived. It does not resend if it was lost.

For telemetry data, this is exactly what you want. Your robot is running a control loop that sends a snapshot of state on every iteration. If one packet gets lost, that is fine -- the next one arrives moments later with fresh data. There is no point resending stale sensor readings from a previous cycle.

> **UDP tradeoff:** Packets may occasionally be dropped and there are no guarantees of arrival. But there is no waiting, no overhead, and latency is minimal. Perfect for high-rate telemetry.

Property	TCP	UDP
Delivery guarantee	Yes -- retransmits if lost	No -- fire and forget
Ordering guarantee	Yes -- arrives in order	No -- may arrive out of order
Connection required	Yes -- handshake first	No -- just send
Latency	Higher (acknowledgments needed)	Lower (no waiting)
Overhead	Higher	Minimal
Best for	Files, web pages, commands	Telemetry, video, audio, games

3.4 What a UDP Packet Looks Like

A UDP packet in your system carries a JSON payload -- a text-based format for structured data that looks like this:

```
{
    "left_motor":  { "power": 0.81, "temp": 42.1 },
```

```
    "right_motor": { "power": 0.79, "temp": 38.5 },
    "chute_hood":  { "position": 0.45 },
    "imu":         { "yaw": 12.75, "steering": -0.049 }
}
```

This text is transmitted as raw bytes over UDP. Your Python listener receives it, parses the JSON, and stores every key-value pair as a triple in the database. The entire process from the robot sending to the database row being written takes just a few milliseconds.

4. How Python Receives the Data

4.1 Getting the Software

The complete software for this guide -- telelog.py, the UdpTelemetry Java class, and sample code -- is available at:

```
https://nxgit.dev/nexus-workshops/udp-telemetry.git
```

Clone it to your laptop with:

```
git clone https://nxgit.dev/nexus-workshops/udp-telemetry.git
cd udp-telemetry
```

If you do not have Git installed, you can also download a ZIP of the repository from the same URL using the Download button on the repository page.

4.2 What telelog.py Does

telelog.py is a Python script that runs on your laptop and does exactly one job: it listens for UDP packets from the robot, parses the JSON payload, and writes the data to a SQLite database. Nothing else.

Python is well suited for this because its standard library includes everything needed for UDP networking, JSON parsing, and SQLite database access -- no external software to install. The script is deliberately minimal: one input (the network), one output (the database).

4.3 How the UDP Listener Works

At the core of the script is a socket -- a software abstraction that represents a network connection. A UDP socket in Python works like this:

```
import socket

# Create a UDP socket
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

# Bind to port 3000 on all interfaces (0.0.0.0 means any)
sock.bind(('0.0.0.0', 3000))

# Wait for a packet (blocks until one arrives)
data, addr = sock.recvfrom(65535)

# data is the raw bytes, addr is (sender_ip, sender_port)
print(f'Received {len(data)} bytes from {addr[0]}')
```

The 0.0.0.0 address means listen on all network interfaces -- whether the packet arrives over WiFi, Ethernet, or the loopback interface for local testing. Port 3000 is arbitrary; both sides just have to agree on the same number.

4.4 Running telelog.py

Open a terminal (Command Prompt or PowerShell on Windows, Terminal on Linux/macOS) in the folder containing telelog.py and run:

```
# Basic usage -- listens on port 3000, writes to telemetry.db
python telelog.py

# Tag this session for later identification
python telelog.py --tag auto_match_3

# Use a different port or database file
python telelog.py --port 3001 --db match_data.db

# Initialize (or reset) the database schema and exit
python telelog.py --init-db

# Suppress per-packet console output
python telelog.py --tag teleop_tuning --quiet
```

> **Note:** *Always run telelog.py before starting the robot op mode. The listener needs to be ready before packets start arriving.*

4.5 What You See in the Console

By default, each received packet prints one line showing the timestamp, source address, packet ID, and which subjects were in the payload:

```
[2026-03-11T18:00:01.050+00:00] src=192.168.43.1:53143 id=2 subjects=['chute_hood',
'imu', 'left_motor', 'right_motor']
[2026-03-11T18:00:01.100+00:00] src=192.168.43.1:53143 id=3 subjects=['chute_hood',
'imu', 'left_motor', 'right_motor']
```

You can leave DB Browser open and refreshing while telelog is running. Every time you press the refresh button in DB Browser you will see new rows appearing in the database.

4.6 Using a Raspberry Pi as a Dedicated Logger

Your laptop works well for logging during practice and debugging sessions. At competition, you may not want to carry your laptop onto the field or keep it open near the robot. A Raspberry Pi solves this.

A Raspberry Pi is a small, inexpensive Linux computer that can run telelog.py on its own. Mount one inside your pit cart, connect it to the Control Hub WiFi, and start telelog.py at boot. Every match is automatically logged without any manual steps. At the end of the day, connect your laptop to the Pi via its own WiFi hotspot or a direct Ethernet cable, and copy the database file for analysis.

Setup is straightforward:

7. Install Raspberry Pi OS Lite on a microSD card
8. Connect the Pi to the Control Hub WiFi network (edit /etc/wpa_supplicant/wpa_supplicant.conf)
9. Copy telelog.py to the Pi
10. Create a systemd service or cron job to run telelog.py at boot with a date-stamped tag
11. Access the database file later via SSH or by physically removing the SD card

A Pi 3 or Pi 4 is more than capable. The Pi Zero 2W works too and is small enough to hide anywhere. Any of these options costs less than $35 and logs data indefinitely on a large microSD card.

5. How Data is Stored: The Triplestore

5.1 Back to English Class: Subject, Predicate, Object

In elementary school English class, you learned to identify the parts of a sentence. Every complete sentence has at minimum a subject and a predicate. Many also have an object. Consider these examples:

Sentence	Subject	Predicate	Object
The dog chased the ball.	The dog	chased	the ball
Maria reads books.	Maria	reads	books
The motor drew 4.2 amps.	The motor	drew	4.2 amps

Sentence	Subject	Predicate	Object
The hood position is 0.85.	The hood	position	0.85
The left motor power is 0.8.	left_motor	power	0.8

Notice the last two rows. The structure of a natural English sentence maps almost perfectly onto the structure of robot telemetry data. The subject is the thing being described. The predicate is the property or attribute. The object is the value.

A triple is simply this three-part structure captured as data:

Subject -- Predicate -- Object

5.2 Your Robot's Data as Triples

Every measurement your robot sends becomes a triple. Here is what one packet from your robot looks like when broken into triples:

Subject	Predicate	Object
left_motor	power	0.8100
left_motor	temp	42.10
right_motor	power	0.7900
right_motor	temp	38.50
chute_hood	position	0.4500
imu	yaw	12.75
imu	steering	-0.0490

Every row is independent. Yet together they paint a complete picture of the robot's state at that moment. The subject tells you what component the reading belongs to. The predicate tells you what was measured. The object is the value.

5.3 Why Not Just Use a Spreadsheet-Style Table?

This is a fair question. If you were designing a database in school, you would probably be taught to create a table that looks like this:

timestamp	left_power	left_temp	right_power	right_temp	hood_pos	yaw
18:00:01.000	0.000	35.0	0.000	34.5	0.00	0.00
18:00:01.050	0.259	35.2	0.244	34.6	0.05	0.75

timestamp	left_power	left_temp	right_power	right_temp	hood_pos	yaw
18:00:01.100	0.479	35.4	0.465	34.8	0.10	1.50

This is called a fixed schema. Each column is defined in advance and every row has exactly those columns. It is clean and easy to understand. This is what you learn about in a database class, and it is the right tool for many problems.

But it has a serious problem for robot telemetry: you have to know all of your column names before you start. The moment you add a new sensor -- say, a distance sensor, a color sensor, or a second IMU -- you have to change the database schema. You add a column, update the code that creates the database, update the code that writes to it, and update every query that reads from it.

During a competition build season, robots change constantly. Sensors get added and removed. New subsystems appear. Predicates get renamed. A fixed schema fights you at every step.

Fixed schema: Great when you know your data structure in advance and it rarely changes. Fragile when the structure is still evolving.

Triplestore: The schema never changes. New data types are just new rows. The database has no opinion about what subjects or predicates you use.

5.4 Introducing Relational Databases

Before going further, it helps to understand what a relational database actually is, because everything in this guide is built on one.

A relational database stores data in tables, where each table is like a spreadsheet: rows are individual records, columns are fields, and every row in the table has the same columns. The word relational refers to the fact that tables can be related to each other through shared values -- a foreign key in one table points to a primary key in another.

Think of it this way. Suppose you have two tables:

```
packets                          triples
-------                          -------
id | ts          | tag           id | packet_id | subject    | predicate | object
---+-------------+-----          ---+-----------+------------+-----------+-------
 1 | 18:00:01.000| match3        1  |     1     | left_motor | power     | 0.81
 2 | 18:00:01.050| match3        2  |     1     | left_motor | temp      | 42.1
 3 | 18:00:01.100| match3        3  |     2     | left_motor | power     | 0.85
```

The packet_id in the triples table is a foreign key pointing back to the packets table. This means every triple knows exactly which packet it came from, and through that packet it knows the timestamp and the session tag. This is the relational part: the two tables are related by that shared id number.

When you write a query that says JOIN packets p ON t.packet_id = p.id, you are asking the database to combine those two tables, matching rows where the packet_id in triples equals the id in packets. The result is a combined view that has the timestamp alongside the triple data.

5.5 The Complete Schema

Your database has two tables and four indexes. The entire schema is defined once, at startup, and never changes no matter what data the robot sends.

```
-- One row per received UDP packet
CREATE TABLE IF NOT EXISTS packets (
    id      INTEGER PRIMARY KEY AUTOINCREMENT,
    ts      TEXT NOT NULL,
    source  TEXT,
    tag     TEXT
);

-- One row per subject-predicate-object triple
CREATE TABLE IF NOT EXISTS triples (
    id          INTEGER PRIMARY KEY AUTOINCREMENT,
    packet_id   INTEGER NOT NULL REFERENCES packets(id),
    subject     TEXT NOT NULL,
    predicate   TEXT NOT NULL,
    object      TEXT
);

-- Indexes speed up common query patterns
CREATE INDEX IF NOT EXISTS idx_packets_tag      ON packets(tag);
CREATE INDEX IF NOT EXISTS idx_triples_subject   ON triples(subject);
CREATE INDEX IF NOT EXISTS idx_triples_predicate ON triples(predicate);
CREATE INDEX IF NOT EXISTS idx_triples_packet    ON triples(packet_id);
```

That is the entire schema. Two tables. This schema can store any robot telemetry you could ever generate, regardless of what sensors you add or remove, without ever being modified.

6. Introduction to SQL

SQL (Structured Query Language) is the standard language for asking questions of a relational database. It was invented in the 1970s and is still the dominant database language in the world today. Learning SQL is one of the highest-value technical skills you can acquire early in your engineering education.

Every SQL query you will write in this guide follows the same basic pattern:

```
SELECT   what you want to see
FROM     which table
JOIN     another table (if you need columns from both)
WHERE    filter: which rows to include
ORDER BY how to sort the output;
```

6.1 The Core Keywords

Keyword	What It Does	Example
SELECT	Choose which columns to show	SELECT subject, predicate, object
FROM	Name the primary table	FROM triples t
JOIN	Merge a second table by matching ids	JOIN packets p ON p.id = t.packet_id
WHERE	Keep only rows matching a condition	WHERE t.subject = 'left_motor'
AND / OR	Combine multiple conditions	AND t.predicate = 'power'
ORDER BY	Sort the result rows	ORDER BY p.id
GROUP BY	Collapse rows by a shared value	GROUP BY p.tag
LIMIT	Cap the number of rows returned	LIMIT 20
AS	Give a column a friendlier name	t.object AS value
CAST	Convert text to a number for math	CAST(t.object AS REAL)

6.2 Understanding JOINs

The JOIN is the most important operation in relational databases. Because your data lives in two tables (packets and triples), almost every useful query needs to JOIN them. Here is the pattern used throughout this guide:

```
SELECT   p.ts, t.subject, t.predicate, t.object
FROM     triples t                          -- start with triples
JOIN     packets p ON p.id = t.packet_id   -- bring in packets where ids match
WHERE    p.tag = 'auto_match_3'
ORDER BY p.id;
```

The letters t and p after the table names are aliases -- shorthand so you do not have to type the full table name every time. p.ts means the ts column from the packets table. t.object means the object column from the triples table.

6.3 Getting Started in DB Browser for SQLite

DB Browser for SQLite is a free, open-source desktop application for viewing and querying SQLite database files. Download it from sqlitebrowser.org and install it on your laptop.

12. Open DB Browser and click Open Database
13. Navigate to your project folder and open telemetry.db
14. Click the Execute SQL tab at the top of the window
15. Type or paste a query into the upper panel
16. Press F5 or click the Run button (triangle icon)
17. Results appear in the lower panel
18. To copy all results: click in the results panel, press Ctrl+A, then Ctrl+C

> **Note:** *You can leave DB Browser open while telelog.py is running. Click the Refresh button in DB Browser (or press F5 in the Browse Data tab) to see new rows as they arrive.*

7. Queries

7.1 Discovery Queries

Run these first when you open a database session. They tell you what data you have before you start digging into specifics.

What test sessions do I have?

Lists every tag in the database with packet counts and start/end times. Your session log. Run this first every time you open the database.

```
SELECT   tag,
         COUNT(DISTINCT p.id) AS packets,
         MIN(p.ts)            AS started,
         MAX(p.ts)            AS ended
FROM     packets p
GROUP BY tag
ORDER BY started;
```

Sample output:

tag	packets	started	ended
auto_match_3	20	2026-03-	2026-03-

tag	packets	started	ended
		11T18:00:01.000Z	11T18:00:02.000Z
teleop_tuning	40	2026-03-11T18:05:14.000Z	2026-03-11T18:05:18.000Z
(null)	20	2026-03-11T17:55:00.000Z	2026-03-11T17:55:01.000Z

LibreOffice Calc: Paste directly into Calc. Sort by started to see your runs chronologically. Use this as your session index.

What subjects and predicates have I ever logged?

Discovers the complete set of signal names across the entire database. Your data dictionary. Useful when you need a reminder of how you named things.

```
SELECT DISTINCT subject, predicate
FROM   triples
ORDER BY subject, predicate;
```

Sample output:

subject	predicate
chute_hood	position
imu	steering
imu	yaw
left_motor	power
left_motor	temp
right_motor	power
right_motor	temp

LibreOffice Calc: Paste into Calc and keep it open beside your query window as a reference card.

Browse all data from one session

Returns every triple from a tagged session, joined with its timestamp. The full picture. Use LIMIT to browse a sample first.

```
SELECT   p.id, p.ts, t.subject, t.predicate, t.object
FROM     packets p
JOIN     triples t ON t.packet_id = p.id
WHERE    p.tag = 'desktop_test'
```

```
ORDER BY p.id, t.subject, t.predicate
LIMIT    50;
```

Sample output:

id	ts	subject	predicate	object
1	2026-03-11T18:00:01.000Z	chute_hood	position	0.00
1	2026-03-11T18:00:01.000Z	imu	yaw	0.00
1	2026-03-11T18:00:01.000Z	left_motor	power	0.00
1	2026-03-11T18:00:01.000Z	right_motor	power	0.00

Inspect one specific packet

Shows every triple from a single packet id. Great for zooming in on a specific moment -- for example the packet where a motor reading spiked.

```
SELECT   t.subject, t.predicate, t.object
FROM     packets p
JOIN     triples t ON t.packet_id = p.id
WHERE    p.id = 18
ORDER BY t.subject, t.predicate;
```

Sample output:

subject	predicate	object
chute_hood	position	0.85
imu	steering	-0.0490
imu	yaw	12.75
left_motor	power	0.651
left_motor	temp	38.6
right_motor	power	0.650
right_motor	temp	37.05

7.2 Time Series Queries

Time series queries track how a single signal changes over time. These are your oscilloscope view. Every one of these can be pasted into LibreOffice Calc and charted as a line graph.

Single signal over time

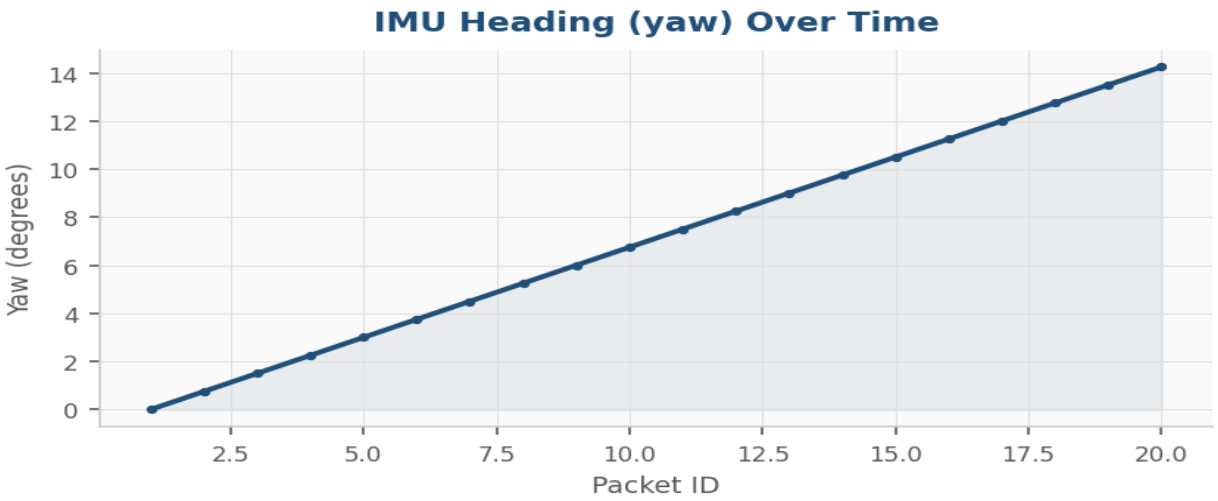
The fundamental time series query. Swap subject and predicate to graph any signal in your database. This is the query you will use most often.

```
SELECT   p.id, p.ts, CAST(t.object AS REAL) AS value
FROM     packets p
JOIN     triples t ON t.packet_id = p.id
WHERE    t.subject   = 'left_motor'
  AND    t.predicate = 'power'
  AND    p.tag       = 'desktop_test'
ORDER BY p.id;
```

Sample output:

id	ts	value
1	2026-03-11T18:00:01.000Z	0.0000
2	2026-03-11T18:00:01.050Z	0.2588
3	2026-03-11T18:00:01.100Z	0.4794
4	2026-03-11T18:00:01.150Z	0.6816
5	2026-03-11T18:00:01.200Z	0.7174

LibreOffice Calc: Select id and value columns, then Insert > Chart > Line. Set id as category axis. Each row is one loop iteration.



Two signals side by side

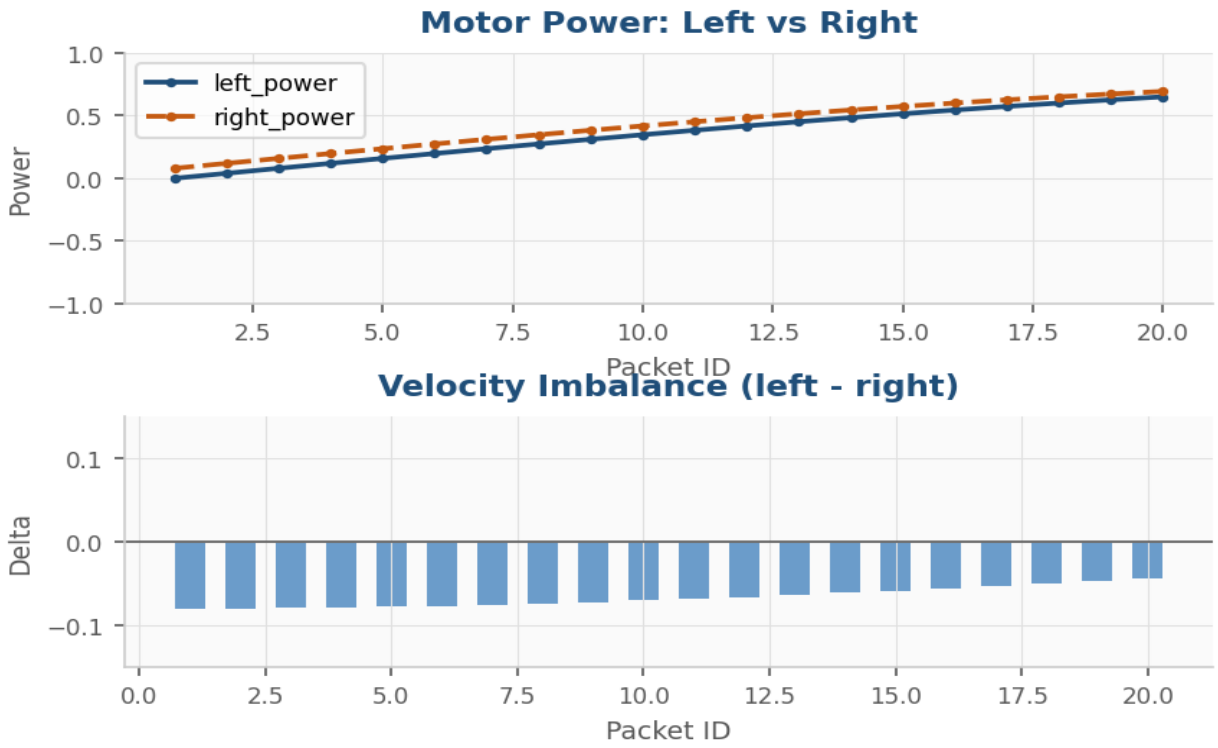
Pivots two signals into columns for direct comparison across the same packets. Add more JOIN blocks to add more signals.

```
SELECT   p.id, p.ts,
         CAST(lp.object AS REAL) AS left_power,
         CAST(rp.object AS REAL) AS right_power
FROM     packets p
JOIN     triples lp ON lp.packet_id = p.id
                   AND lp.subject   = 'left_motor'
                   AND lp.predicate = 'power'
JOIN     triples rp ON rp.packet_id = p.id
                   AND rp.subject   = 'right_motor'
                   AND rp.predicate = 'power'
WHERE    p.tag = 'desktop_test'
ORDER BY p.id;
```

Sample output:

id	ts	left_power	right_power
1	2026-03-11T18:00:01.000Z	0.000	0.000
2	2026-03-11T18:00:01.050Z	0.259	0.244
3	2026-03-11T18:00:01.100Z	0.479	0.465
4	2026-03-11T18:00:01.150Z	0.682	0.669
5	2026-03-11T18:00:01.200Z	0.717	0.706

LibreOffice Calc: Select left_power and right_power columns and chart as two series on one line chart. Imbalance between them shows up immediately as a gap between the lines.



Full robot dashboard -- all signals in one paste

Pivots your most important signals into columns. One paste into Calc gives you a complete run table ready for charting.

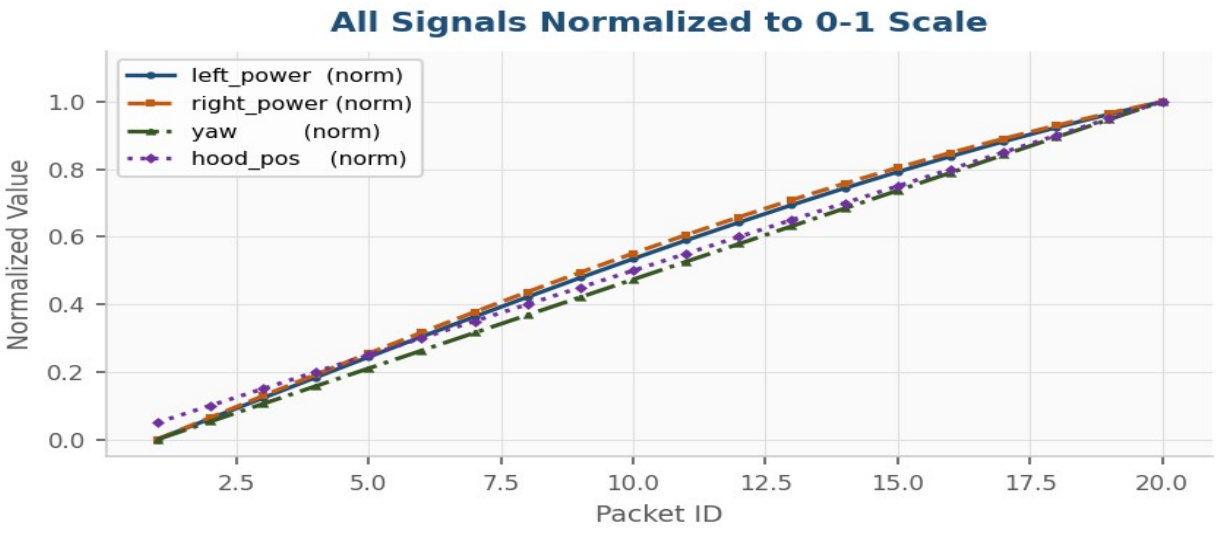
```
SELECT   p.id, p.ts,
         CAST(lp.object  AS REAL) AS left_power,
         CAST(rp.object  AS REAL) AS right_power,
         CAST(yaw.object AS REAL) AS yaw,
         CAST(hd.object  AS REAL) AS hood_pos
FROM     packets p
JOIN     triples lp  ON lp.packet_id  = p.id AND lp.subject  = 'left_motor'  AND
lp.predicate  = 'power'
JOIN     triples rp  ON rp.packet_id  = p.id AND rp.subject  = 'right_motor' AND
rp.predicate  = 'power'
JOIN     triples yaw ON yaw.packet_id = p.id AND yaw.subject = 'imu'         AND
yaw.predicate = 'yaw'
JOIN     triples hd  ON hd.packet_id  = p.id AND hd.subject  = 'chute_hood'  AND
hd.predicate  = 'position'
WHERE    p.tag = 'desktop_test'
ORDER BY p.id;
```

Sample output:

id	ts	left_power	right_power	yaw	hood_pos
1	2026-03-11T18:00:01.000Z	0.000	0.000	0.00	0.00

id	ts	left_power	right_power	yaw	hood_pos
5	2026-03-11T18:00:01.200Z	0.717	0.706	3.00	0.25
10	2026-03-11T18:00:01.450Z	0.733	0.724	6.75	0.50
15	2026-03-11T18:00:01.700Z	0.407	0.399	10.50	0.75
20	2026-03-11T18:00:01.950Z	0.651	0.640	14.25	1.00

LibreOffice Calc: Paste all columns into Calc. Create separate charts for motor balance (left vs right) and heading (yaw over time). Normalize to 0-1 scale if you want all signals on one chart.



7.3 Diagnostic Queries

These queries are for finding problems. Timing gaps, runaway values, frozen signals. Run these when something went wrong during a match and you are trying to figure out why.

Packet timing -- detect loop slowdowns

Shows the time gap between consecutive packets in milliseconds. A healthy loop shows consistent spacing throughout. Any spike means the robot loop stalled or slowed at that packet.

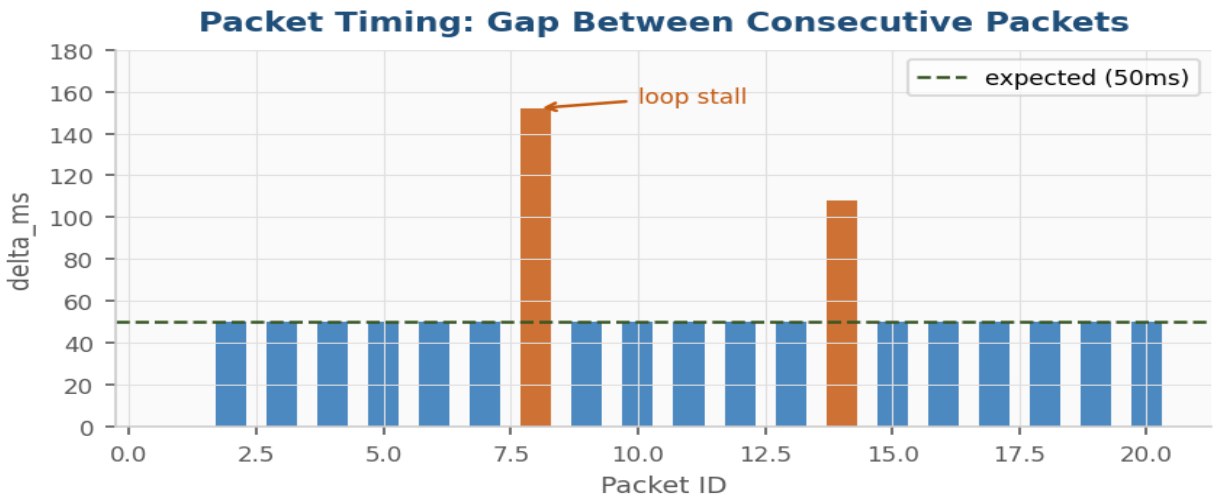
```
SELECT   p.id,
         p.ts,
         ROUND(
```

```
          (JULIANDAY(p.ts) - JULIANDAY(LAG(p.ts) OVER (ORDER BY p.id)))
          * 86400000
        ) AS delta_ms
FROM    packets p
WHERE   p.tag = 'desktop_test'
ORDER BY p.id;
```

Sample output:

id	ts	delta_ms
1	2026-03-11T18:00:01.000Z	(null)
2	2026-03-11T18:00:01.050Z	50
3	2026-03-11T18:00:01.100Z	50
8	2026-03-11T18:00:01.350Z	152
9	2026-03-11T18:00:01.400Z	50

LibreOffice Calc: Paste id and delta_ms into Calc and chart as a bar chart. Spikes jump out immediately. Packet 8 above took 152ms -- significantly longer than the surrounding packets, which signals a stall in the loop.



Find outliers -- values outside expected range

Returns any packet where a signal exceeded a threshold. Change the subject, predicate, and threshold value to match whatever you are investigating.

```
SELECT   p.id, p.ts, t.subject, t.predicate, t.object
FROM     packets p
JOIN     triples t ON t.packet_id = p.id
WHERE    t.subject              = 'left_motor'
  AND    t.predicate            = 'temp'
  AND    CAST(t.object AS REAL) > 45.0
```

```
  AND    p.tag                 = 'auto_match_3'
ORDER BY p.id;
```

Sample output:

id	ts	subject	predicate	object
18	2026-03-11T18:00:01.850Z	left_motor	temp	46.2
19	2026-03-11T18:00:01.900Z	left_motor	temp	47.1
20	2026-03-11T18:00:01.950Z	left_motor	temp	48.0

LibreOffice Calc: Empty result means the signal stayed in range the whole run. Any rows returned warrant investigation.

Find the worst value in a run

Returns the maximum (or minimum) value of any signal across a session, along with the exact packet where it occurred.

```
SELECT   p.id, p.ts, t.object AS peak_temp
FROM     packets p
JOIN     triples t ON t.packet_id = p.id
WHERE    t.subject   = 'left_motor'
  AND    t.predicate = 'temp'
  AND    p.tag       = 'auto_match_3'
ORDER BY CAST(t.object AS REAL) DESC
LIMIT    1;
```

Sample output:

id	ts	peak_temp
20	2026-03-11T18:00:01.950Z	48.0

LibreOffice Calc: Swap DESC for ASC to find the minimum value instead. Change subject and predicate to find the peak of any signal.

Rolling average -- smooth a noisy signal

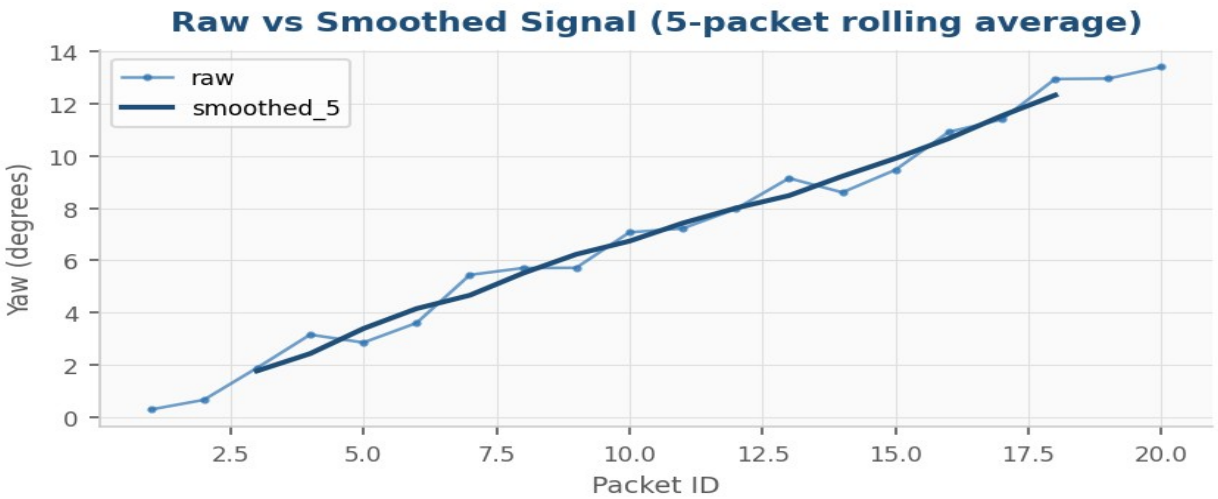
Computes a 5-packet rolling average alongside the raw value. Use this for sensors that have electrical noise or jitter. Increase the window number for more smoothing.

```
SELECT   p.id, p.ts,
         CAST(t.object AS REAL) AS raw,
         ROUND(
           AVG(CAST(t.object AS REAL))
           OVER (ORDER BY p.id ROWS BETWEEN 4 PRECEDING AND CURRENT ROW),
           4
         ) AS smoothed_5
FROM     packets p
JOIN     triples t ON t.packet_id = p.id
WHERE    t.subject   = 'imu'
  AND    t.predicate = 'yaw'
  AND    p.tag       = 'desktop_test'
ORDER BY p.id;
```

Sample output:

id	ts	raw	smoothed_5
1	2026-03-11T18:00:01.000Z	0.000	0.000
2	2026-03-11T18:00:01.050Z	0.750	0.375
3	2026-03-11T18:00:01.100Z	1.500	0.750
4	2026-03-11T18:00:01.150Z	2.250	1.125
5	2026-03-11T18:00:01.200Z	3.000	1.500

LibreOffice Calc: Chart raw and smoothed_5 as two series on the same line chart. The gap between them shows how much noise the sensor has.



Last known state -- what was the robot doing when it stopped?

Returns the final value of every signal for a tagged run. Use this after a match to see exactly what state the robot was in when time ran out.

```
SELECT   t.subject, t.predicate, t.object, p.ts
FROM     triples t
JOIN     packets p ON p.id = t.packet_id
WHERE    p.tag = 'auto_match_3'
  AND    p.id = (
    SELECT MAX(p2.id)
    FROM   packets p2
    JOIN   triples t2 ON t2.packet_id = p2.id
    WHERE  t2.subject   = t.subject
      AND  t2.predicate = t.predicate
      AND  p2.tag       = p.tag
  )
ORDER BY t.subject, t.predicate;
```

Sample output:

subject	predicate	object	ts
chute_hood	position	1.0	2026-03-11T18:00:01.950Z
imu	yaw	14.25	2026-03-11T18:00:01.950Z
left_motor	power	0.651	2026-03-11T18:00:01.950Z
left_motor	temp	38.8	2026-03-11T18:00:01.950Z
right_motor	power	0.694	2026-03-11T18:00:01.950Z
right_motor	temp	37.35	2026-03-11T18:00:01.950Z

Did a signal ever change? Detect frozen sensors

Checks whether a signal had more than one distinct value during a run. If a sensor freezes or gets stuck, this query will return only one row. A healthy sensor returns many distinct values.

```
SELECT   COUNT(DISTINCT t.object) AS distinct_values,
         MIN(t.object)            AS min_val,
         MAX(t.object)            AS max_val
FROM     packets p
JOIN     triples t ON t.packet_id = p.id
WHERE    t.subject   = 'imu'
  AND    t.predicate = 'yaw'
```

```
  AND    p.tag       = 'auto_match_3';
```

Sample output:

distinct_values	min_val	max_val
20	0.00	14.25

LibreOffice Calc: distinct_values = 1 means the sensor never changed -- almost always a bug. Change subject and predicate to check any signal.

7.4 Cross-Run Comparison

These queries compare the same signals across different tagged sessions. Use them to validate that a mechanical or code change actually produced the result you expected.

Average, min, and max per session

Computes summary statistics for one signal across every tagged run. Paste into Calc and bar-chart the avg_value column grouped by tag for an instant before/after comparison.

```
SELECT   p.tag,
         COUNT(*)                                  AS samples,
         ROUND(AVG(CAST(t.object AS REAL)), 4)  AS avg_value,
         ROUND(MIN(CAST(t.object AS REAL)), 4)  AS min_value,
         ROUND(MAX(CAST(t.object AS REAL)), 4)  AS max_value
FROM     packets p
JOIN     triples t ON t.packet_id = p.id
WHERE    t.subject   = 'left_motor'
  AND    t.predicate = 'power'
GROUP BY p.tag
ORDER BY MIN(p.ts);
```

Sample output:

tag	samples	avg_value	min_value	max_value
baseline	20	0.4712	0.0000	0.8000
after_pid_tune	20	0.5041	0.0000	0.8000
match_run_1	20	0.4833	0.0000	0.7940

LibreOffice Calc: Paste tag and avg_value into Calc, select both columns, Insert > Chart > Bar. Immediately shows whether tuning changed behavior.

Overlay two runs for direct comparison

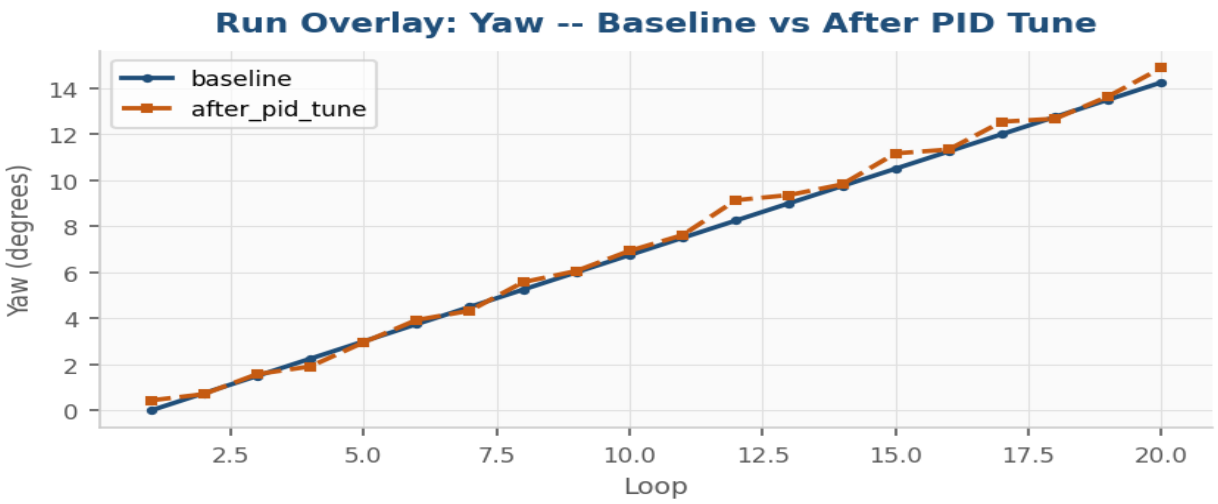
Aligns packets from two sessions by loop number so you can plot them on the same axis. Replace the two tag values with your actual session names.

```
SELECT   a.loop,
         a.value AS baseline,
         b.value AS after_tune
FROM (
  SELECT ROW_NUMBER() OVER (ORDER BY p.id) AS loop,
         CAST(t.object AS REAL)             AS value
  FROM   packets p JOIN triples t ON t.packet_id = p.id
  WHERE  t.subject = 'imu' AND t.predicate = 'yaw'
    AND  p.tag = 'baseline'
) a
JOIN (
  SELECT ROW_NUMBER() OVER (ORDER BY p.id) AS loop,
         CAST(t.object AS REAL)             AS value
  FROM   packets p JOIN triples t ON t.packet_id = p.id
  WHERE  t.subject = 'imu' AND t.predicate = 'yaw'
    AND  p.tag = 'after_pid_tune'
) b ON a.loop = b.loop
ORDER BY a.loop;
```

Sample output:

loop	baseline	after_tune
1	0.00	0.00
5	3.00	3.10
10	6.75	7.20
15	10.50	10.80
20	14.25	14.50

LibreOffice Calc: Select all three columns in Calc. Insert a line chart with loop as the X axis and both value columns as series. Two lines on one chart -- the difference between them is your tuning effect.



8. Real Engineering Problems and How Data Solves Them

Every problem described in this section is real. Teams encounter these exact situations every season. In each case, the robot appears to work fine in practice -- and then fails during a match, sometimes catastrophically, sometimes subtly. Telemetry data is what separates a team that guesses at the cause from a team that knows exactly what happened and can fix it before the next match.

8.1 The Robot That Curves When It Should Go Straight

You program your robot to drive straight for two meters during autonomous. At home, in your school's gym, it drives straight every time. At the competition, in a different building on a different floor surface, it consistently curves left -- not a lot, but enough to miss the scoring zone by half a meter.

You stare at the code. Nothing is wrong. The left and right motor commands are identical. You check the wiring. Everything looks fine. You have two matches left.

Here is what is actually happening: the left motor is slightly stronger than the right -- or the left wheel has slightly less resistance due to a worn bearing, a different tire contact patch on the new floor, or a tiny difference in gear mesh. Both motors receive the same command, but one responds differently. The robot curves.

With telemetry data, you can see this immediately. Log the commanded power and the actual encoder velocity for each motor separately:

```
SELECT   p.id, p.ts,
         CAST(lp.object AS REAL) AS left_power,
         CAST(rp.object AS REAL) AS right_power,
         CAST(lv.object AS REAL) AS left_velocity,
         CAST(rv.object AS REAL) AS right_velocity,
         CAST(lv.object AS REAL) - CAST(rv.object AS REAL) AS velocity_diff
FROM     packets p
```

```
JOIN     triples lp ON lp.packet_id = p.id AND lp.subject = 'left_motor'  AND
lp.predicate = 'power'
JOIN     triples rp ON rp.packet_id = p.id AND rp.subject = 'right_motor' AND
rp.predicate = 'power'
JOIN     triples lv ON lv.packet_id = p.id AND lv.subject = 'left_motor'  AND
lv.predicate = 'velocity'
JOIN     triples rv ON rv.packet_id = p.id AND rv.subject = 'right_motor' AND
rv.predicate = 'velocity'
WHERE    p.tag = 'auto_straight_run'
ORDER BY p.id;
```

Paste this into LibreOffice Calc and plot left_velocity and right_velocity as two lines. If they diverge -- one rising faster than the other despite identical commands -- you have found your problem. The velocity_diff column shows exactly how large the imbalance is at every moment.

Now you have something actionable: you can add a small feed-forward correction in code, implement a PID controller that uses the heading error to equalize the motors, or identify and replace the weaker motor before the next match. You went from guessing to knowing in two minutes.

8.2 The April Tag That Disappeared

Your autonomous routine uses a camera to detect April tags for positioning. It works perfectly in your build space. During the match, everything runs flawlessly for the first eight seconds -- and then the robot freezes mid-field, does nothing for three seconds, and drives into the wall.

You review the match video. At the eight-second mark, another robot drove between your robot and the April tag, blocking the camera's view for exactly three seconds. Your code waited for a tag detection that never came and had no fallback.

If you had been logging telemetry, you would see this clearly: the targets_detected count drops from 1 to 0 at packet 160, stays at 0 for 60 packets, then returns to 1. The robot's commanded velocity also drops to zero during that window -- confirming the code was waiting on the camera.

```
-- Find every packet where camera lost all targets
SELECT   p.id, p.ts, t.object AS targets_detected
FROM     packets p
JOIN     triples t ON t.packet_id = p.id
WHERE    t.subject              = 'camera'
  AND    t.predicate            = 'targets_detected'
  AND    CAST(t.object AS INTEGER) = 0
  AND    p.tag                  = 'auto_match_2'
ORDER BY p.id;
```

This data tells you exactly how long the robot was blind and when. Armed with this, you can design a fault-tolerant solution: when the camera loses targets for more than N consecutive packets, switch to dead reckoning using wheel encoders and the IMU heading. Log which mode the robot is in as a predicate called nav_mode with values like camera_guided and dead_reckoning. Now you can see in post-match analysis exactly when the fallback activated and how long it ran.

The deeper lesson: if a sensor can be occluded, obstructed, or overloaded, your code must account for it. Telemetry is how you prove the fallback is actually working and how long your dead reckoning stayed accurate before error accumulated too far.

8.3 The Color Sensor That Lied at Competition

Your robot uses a color sensor to detect whether a game element is in the intake -- red means element present, low reading means empty. In your school's workshop it is reliable. You run it a hundred times. Perfect.

At the competition venue, the robot misses elements it should detect and triggers false positives. You check the wiring, the code, the sensor position. Nothing has changed. The sensor hardware is fine. What changed is the building.

Your workshop has cool-white fluorescent lighting. The competition venue has warm LED panels at a different intensity and a completely different color temperature. The raw ambient light reading your sensor sees is different by a significant margin, and your threshold value -- which you tuned in your workshop -- is no longer valid.

This is discoverable from data. If you log the raw sensor values during every run, including at competition, the comparison is immediate:

```
-- Compare sensor behavior across environments
SELECT   p.tag,
         COUNT(*)                                  AS samples,
         ROUND(AVG(CAST(t.object AS REAL)), 1)  AS avg_reading,
         ROUND(MIN(CAST(t.object AS REAL)), 1)  AS min_reading,
         ROUND(MAX(CAST(t.object AS REAL)), 1)  AS max_reading
FROM     packets p
JOIN     triples t ON t.packet_id = p.id
WHERE    t.subject   = 'color_sensor'
  AND    t.predicate = 'ambient_light'
GROUP BY p.tag
ORDER BY MIN(p.ts);
```

Tag your sessions meaningfully: workshop_tuesday, competition_venue_practice, match_1, match_2. The query above groups them and immediately shows you the average ambient reading in each environment. If your workshop reads 420 and the competition venue reads 680, your threshold is in the wrong place for this building.

The fault-tolerant solution is not just to adjust the threshold -- it is to redesign the detection to be relative rather than absolute. Log the baseline ambient reading at startup, and compare current readings to that baseline rather than to a hard-coded number. Telemetry data from multiple venues gives you the evidence to argue for that architectural change, and data from subsequent competitions to verify it worked.

8.4 The Ball That Was Slower Than the Code

Your robot has an intake that pulls in a ball and a gate that opens to deliver it to a scoring chute. The software sequence is: motor spins, intake sensor triggers, gate opens. Simple. In testing it works every time.

During a match, the ball occasionally does not make it through the gate. Sometimes it does. The failure is intermittent, which makes it nearly impossible to debug by watching. You watch the video over and over. The gate appears to open. The ball appears to be there. Sometimes it works, sometimes it does not.

Here is what is happening: physics. The intake sensor triggers when the ball crosses its beam -- but the ball is still moving through the mechanism with momentum and deceleration. Depending on exactly how fast the intake motor is running, how much the ball has decelerated, and what angle it entered the mechanism, the ball arrives at the gate somewhere between 80 and 240 milliseconds after the sensor fires. Your gate open command fires 50 milliseconds after the sensor -- which is fast enough sometimes and not fast enough others.

The software speed was easy to measure. The physics was invisible -- until you log it. You need two signals: the moment the intake sensor fires, and the moment of confirmed delivery (perhaps a second sensor at the chute, or a current spike indicating the motor stalled against the ball at the gate). Log both:

```
-- Find the time between intake trigger and successful delivery
SELECT   a.packet_id                  AS trigger_packet,
         b.packet_id                  AS delivery_packet,
         a.ts                         AS trigger_time,
         b.ts                         AS delivery_time,
         ROUND(
           (JULIANDAY(b.ts) - JULIANDAY(a.ts)) * 86400000
         )                            AS transit_ms
FROM (
  SELECT p.id AS packet_id, p.ts
  FROM   packets p JOIN triples t ON t.packet_id = p.id
  WHERE  t.subject = 'intake' AND t.predicate = 'sensor' AND t.object = '1'
  AND    p.tag = 'intake_test'
) a
JOIN (
  SELECT p.id AS packet_id, p.ts
  FROM   packets p JOIN triples t ON t.packet_id = p.id
  WHERE  t.subject = 'chute' AND t.predicate = 'ball_confirmed' AND t.object = '1'
  AND    p.tag = 'intake_test'
) b ON b.packet_id > a.packet_id
ORDER BY a.packet_id;
```

Run this query after collecting data from twenty or thirty intake cycles and paste the transit_ms column into LibreOffice Calc. You will see the distribution. Maybe it ranges from 90ms to 280ms with a cluster around 160ms. Now you know your gate delay should be at least 280ms to guarantee delivery -- not the 50ms you had assumed. You just discovered something through measurement that no amount of code review would have revealed.

Furthermore, you can correlate transit time with intake motor power. If low motor power consistently produces longer transit times, you have a tuning insight: run the intake motor faster to tighten the delivery window and make the system more predictable.

8.5 The Motor That Could Not Lift the Load

Your arm mechanism works in testing with the practice game elements. At competition, using the official elements which are slightly heavier, the arm occasionally stalls partway through its travel. The arm motor is commanded to move to a target position but never arrives. The robot times out and moves on without completing the scoring action.

The symptom is clear in the data. When a motor is commanded to move but is mechanically overpowered, you see a characteristic pattern: power output climbs toward maximum while encoder position stops changing or changes very slowly. This is a stall signature.

```
-- Detect stall: high power commanded, minimal encoder movement
SELECT   p.id, p.ts,
         CAST(pw.object  AS REAL)    AS power,
         CAST(enc.object AS INTEGER) AS encoder_pos,
         CAST(enc.object AS INTEGER) -
           LAG(CAST(enc.object AS INTEGER)) OVER (ORDER BY p.id)
                                    AS encoder_delta
FROM     packets p
JOIN     triples pw  ON pw.packet_id  = p.id AND pw.subject  = 'arm_motor' AND
pw.predicate = 'power'
JOIN     triples enc ON enc.packet_id = p.id AND enc.subject = 'arm_motor' AND
enc.predicate = 'encoder'
WHERE    p.tag = 'auto_match_3'
ORDER BY p.id;
```

Plot power and encoder_delta as two series in LibreOffice Calc. When the arm is moving freely, encoder_delta should be large and power moderate. When it stalls, encoder_delta drops toward zero while power stays high. The exact packet where this transition occurs tells you at what arm position and under what load the motor runs out of torque.

This leads to three possible actions: increase the motor gear ratio for more torque at the cost of speed, switch to a higher-torque motor, or add a current limit to prevent the motor from burning out when stalled. All three are informed decisions based on measured evidence, not guesswork.

You can also use this pattern prospectively: log encoder_delta continuously during every run and alert (via console output or a logged flag) whenever you see the stall signature. This becomes an early warning system -- you discover mechanical wear or binding before it causes a match failure.

8.6 The Heading That Drifted and the Drive That Fought It

Field-centric drive is one of the most powerful driver aids you can build. Instead of the robot moving relative to its own nose -- where pushing the joystick forward always goes forward relative to the robot

-- field-centric drive makes pushing forward always go toward the far wall, regardless of which direction the robot is facing. The driver stops having to think about the robot's orientation and just thinks about the field.

This works beautifully when it works. When it fails, it fails in a deeply confusing way: the robot moves in the wrong direction. The driver pushes forward and the robot goes sideways. The driver pushes right and the robot goes backward. From the driver's perspective, the robot appears possessed.

The cause is almost always IMU heading drift or a single large heading error -- a hard collision, a field reset, or accumulated error from wheel slip during a fast rotation. The IMU thinks the robot is facing north when it is actually facing 45 degrees east. Every drive command is then rotated by 45 degrees in the wrong direction.

With telemetry, you can see this precisely. Log the IMU-reported heading, the field-centric corrected heading, and the raw drive commands simultaneously:

```
-- Track heading alongside drive output to find where they diverge
SELECT   p.id, p.ts,
         CAST(imu.object AS REAL)   AS imu_heading,
         CAST(fwd.object AS REAL)   AS commanded_fwd,
         CAST(str.object AS REAL)   AS commanded_strafe,
         CAST(nav.object AS TEXT)   AS nav_mode
FROM     packets p
JOIN     triples imu ON imu.packet_id = p.id AND imu.subject = 'imu'   AND
imu.predicate = 'heading'
JOIN     triples fwd ON fwd.packet_id = p.id AND fwd.subject = 'drive' AND
fwd.predicate = 'forward'
JOIN     triples str ON str.packet_id = p.id AND str.subject = 'drive' AND
str.predicate = 'strafe'
JOIN     triples nav ON nav.packet_id = p.id AND nav.subject = 'drive' AND
nav.predicate = 'nav_mode'
WHERE    p.tag = 'teleop_match_4'
ORDER BY p.id;
```

Plotting imu_heading over time reveals drift: a signal that should be stable or slowly changing instead trends upward or makes sudden jumps. Cross-referencing the timestamps with video of the match tells you exactly what physical event caused the heading error.

This data also validates your correction strategy. If you implement a heading correction that uses field landmarks -- such as a detected April tag or a known wall distance -- you can log when corrections are applied and by how much. The question becomes not just did the correction work but how often was it needed and how large were the errors it corrected. A system that needs frequent large corrections has a different mechanical or algorithmic problem than one that needs rare small ones.

Dead reckoning -- estimating position from wheel encoder deltas and heading -- accumulates error over time. Wheel slip during acceleration, differential friction, and small heading errors all compound. Log your estimated position alongside your actual scored points per match and you will discover how much position error is tolerable before autonomous routines start missing their targets. This is how you set realistic accuracy requirements for your navigation system rather than guessing.

8.7 Building Situational Awareness from Streams

Each of the problems above was diagnosed in isolation -- one signal, one failure mode. But the real power of continuous telemetry logging is that all of these signals exist simultaneously in the same database, timestamped to the same clock, and can be queried together.

Consider: during a match, your robot is executing an autonomous scoring sequence. It drives forward, approaches a target, lines up using a camera, and releases a game element. At the same moment, another robot collides with yours. The collision causes wheel slip, which corrupts the encoder-based position estimate. The camera loses the target for 200ms due to vibration. The arm motor stalls briefly against the impact force. The heading drifts 12 degrees.

Without telemetry, that sequence of events is invisible. You see: robot missed the target. You do not know whether it was the camera, the navigation, the arm, or some combination. You might tune the wrong system.

With telemetry, you can reconstruct the event timeline:

```
-- Reconstruct a multi-system event around a specific packet range
SELECT   p.id, p.ts, t.subject, t.predicate, t.object
FROM     packets p
JOIN     triples t ON t.packet_id = p.id
WHERE    p.tag  = 'auto_match_5'
  AND    p.id   BETWEEN 140 AND 180
  AND    t.subject IN ('camera', 'imu', 'left_motor', 'right_motor', 'arm_motor',
'drive')
ORDER BY p.id, t.subject, t.predicate;
```

This returns a narrow window of the match -- perhaps three seconds -- showing every system simultaneously. You can read across each packet id row and see: at packet 153, the camera lost targets, the IMU heading jumped 12 degrees, and both drive motors showed a sudden velocity drop (the impact). At packet 158, the arm motor stalled. At packet 165, camera reacquired. At packet 172, heading was still 11 degrees off from where it should have been.

That is situational awareness built from data. You can now make specific improvements: add a collision-detection threshold based on sudden velocity drops, trigger an IMU recalibration event when a large sudden heading change is detected, add a brief arm backoff when a stall is detected to prevent motor damage, and implement a camera reacquire delay before using heading data that arrived during a target-lost window.

None of these are guesses. Each is a specific response to a specific measured event. And each can be validated in the next test session by verifying in the data that the new behavior triggered correctly.

The pattern: Log everything during every run, even during practice. The database costs nothing. The data you do not collect is the data you cannot use when something goes wrong at competition.

9. Graphing in LibreOffice Calc

8.1 The Basic Workflow

The copy-paste workflow between DB Browser and LibreOffice Calc is fast once you get used to it:

19. Run a query in DB Browser
20. Click anywhere in the results panel at the bottom
21. Press Ctrl+A to select all result rows
22. Press Ctrl+C to copy
23. Switch to LibreOffice Calc and click cell A1
24. Press Ctrl+V to paste -- column headers paste as the first row
25. Select the columns you want to chart
26. Go to Insert > Chart and follow the wizard

> **Note:** *The time series queries in Section 7.2 are all designed to produce output that pastes cleanly into Calc with minimal cleanup needed.*

8.2 Chart Type Guide

Query	Recommended Chart	X Axis	Y Axis / Series
Single signal	Line	id or ts	value
Two signals compared	Line	id	both value columns
Full dashboard pivot	Line	id	each signal as a series
Packet timing	Bar	id	delta_ms (spikes obvious)
Outlier search	Scatter	id	object (mark threshold)
Rolling average	Line	id	raw and smoothed together
Per-run averages	Bar	tag	avg_value
Run overlay	Line	loop	baseline and after_tune

8.3 Normalizing Mixed-Scale Data

Some signals have very different numeric ranges: yaw might be in degrees (0-360), motor power in the range 0 to 1, and temperature in Celsius (35-50). If you want all three on the same chart axis, normalize them first.

In LibreOffice Calc, add a new column next to any raw column and enter this formula (assuming raw data is in column B, rows 2 through 21):

```
=(B2-MIN($B$2:$B$21))/(MAX($B$2:$B$21)-MIN($B$2:$B$21))
```

This scales the column to a 0-1 range: the minimum value becomes 0 and the maximum becomes 1. Drag the formula down to fill the column for all rows. Chart the normalized columns instead of the raw ones. All signals will now fit on the same axis and you can compare their shapes directly.

8.4 Adding a Reference Line

To add a threshold line to a chart -- for example showing the 45 degree temperature limit -- add a new column in Calc filled with the constant value (45.0 for every row) and include it as a series in your chart. It will appear as a flat horizontal line that you can compare your signal against.

10. Tips, Gotchas, and Common Mistakes

10.1 Always CAST When Doing Math

All values in the triples table are stored as TEXT -- even numbers. SQLite does this because the schema is designed to accept any value without knowing its type in advance. The consequence is that comparisons and math require an explicit cast:

```
-- WRONG: string comparison, '9' > '10' alphabetically
WHERE t.object > 45.0

-- CORRECT: numeric comparison
WHERE CAST(t.object AS REAL) > 45.0

-- For whole numbers
WHERE CAST(t.object AS INTEGER) > 100
```

10.2 Use Packet ID for Ordering, Not Timestamp

The packet id is an autoincrement integer that strictly reflects insertion order regardless of clock behavior. If the Pi's clock gets corrected by NTP mid-session, timestamps can jump forward or backward. The id never does. Use ORDER BY p.id for true sequence and p.ts for display and time-difference calculations.

10.3 Querying NULL Tags

Packets recorded without a --tag argument have a NULL tag in the database. To query them, you cannot use = NULL. You must use IS NULL:

```
-- WRONG: this never matches anything
WHERE p.tag = NULL

-- CORRECT
WHERE p.tag IS NULL
```

10.4 Window Functions Require SQLite 3.25 or Later

The rolling average, packet timing (LAG), and run overlay (ROW_NUMBER) queries use window functions -- a powerful SQL feature that operates across a range of rows. Window functions were added to SQLite in version 3.25, released in September 2018. Any laptop running a modern operating system has a new enough version.

To check your version in DB Browser: go to Help > About DB Browser for SQLite and look for the SQLite version number.

10.5 The --init-db Flag is Safe to Re-run

Running python telelog.py --init-db against an existing database is safe because all CREATE TABLE statements use IF NOT EXISTS. It will not delete any data. To start completely fresh, close DB Browser, manually delete the .db file, then run --init-db.

10.6 Run telelog Before the Robot

telelog.py has to be listening before packets start arriving. If you start the robot's op mode first, the early packets will be lost -- they are sent over UDP with no confirmation, so if nobody is listening, they simply disappear. Start telelog first, verify you see the startup messages, then initialize the op mode on the Driver Station.

10.7 The flat scalar subject

If your Java code sends a value without an explicit subject (using the single-argument put method), telelog assigns the source IP address as the subject. This produces a subject like 192.168.43.1:53143 in the database, which is awkward to query. The fix is simple: always use an explicit subject in your UdpTelemetry.put calls:

```
// Awkward -- source IP becomes the subject
t.put("loop_time_ms", loopTime);

// Better -- explicit subject, clean to query
t.put("robot", "loop_time_ms", loopTime);
```

11. Instrumenting Your Robot Code

The telemetry system is only as good as what you put in it. This section walks through how to add the UdpTelemetry class to your FTC robot code and what to log.

11.1 Adding UdpTelemetry to Your OpMode

The UdpTelemetry Java class handles everything: creating the socket, building the JSON payload, and sending the packet. You do not need to understand networking to use it. Here is a minimal FTC OpMode that sends telemetry:

```
import com.nexusworkshops.telemetry.UdpTelemetry;

@TeleOp(name = "TeleOp with Telemetry")
public class TeleOpWithTelemetry extends LinearOpMode {

    UdpTelemetry telem;

    @Override
    public void runOpMode() {

        // Connect to your laptop's IP on the Control Hub network
        // Use 255.255.255.255 to broadcast to all devices on the network
        telem = new UdpTelemetry("192.168.43.100", 3000);

        waitForStart();

        while (opModeIsActive()) {

            double leftPower  = -gamepad1.left_stick_y;
            double rightPower = -gamepad1.right_stick_y;

            // Drive the motors
            // ... your motor commands here ...

            // Send telemetry -- this goes out every loop iteration
            telem.put("left_motor",  "power",    leftPower)
                 .put("right_motor", "power",    rightPower)
                 .put("imu",         "heading",
imu.getRobotYawPitchRollAngles().getYaw(AngleUnit.DEGREES))
                 .put("robot",       "loop_ms",  getRuntime() * 1000)
                 .send();

        }

        telem.close();
    }
}
```

The fluent style -- chaining .put() calls before .send() -- lets you build the entire packet in one statement. Call .send() exactly once per loop iteration to send all the data as a single UDP packet, then the buffer clears automatically for the next iteration.

11.2 Using the Broadcast Address

Instead of hardcoding your laptop's IP address, you can send to the broadcast address 192.168.43.255. A broadcast packet is delivered to every device on the subnet. This means telelog.py will receive it regardless of which IP your laptop was assigned:

```
telem = new UdpTelemetry("192.168.43.255", 3000);
```

Using broadcast is more robust: you never have to update the robot code when your laptop gets a different IP address. The tradeoff is that all devices on the network receive the packet, but for a small private network this is harmless.

11.3 What to Log

The temptation is to log everything. Resist it slightly. Logging too many signals makes queries harder to read and can slow the loop if the payload gets very large. A good rule of thumb: log anything you might want to graph or correlate later, and log it with a consistent subject and predicate name.

Recommended signals to log for most robots:

Subject	Predicate	What it tells you
left_motor	power	Commanded output -- compare to velocity for stall detection
left_motor	velocity	Encoder-derived speed -- divergence from right reveals drift
left_motor	encoder	Raw encoder count -- for distance tracking and stall detection
right_motor	power	Same as above for the other side
right_motor	velocity	Same as above for the other side
right_motor	encoder	Same as above for the other side
imu	heading	Robot's orientation -- jumps reveal collisions or drift
imu	pitch	Tilt front-to-back -- detects climbing or tipping
drive	nav_mode	Active navigation strategy (auto, dead_reckoning, manual...)
camera	targets_detected	Count of visible April tags -- drops reveal occlusions
robot	loop_ms	Elapsed time -- reveals loop slowdowns
robot	op_state	Current state machine state -- where in the program is it?

Add subsystem-specific signals as your robot complexity grows. An intake mechanism might log intake_sensor, gate_state, and ball_confirmed. An arm might log arm_encoder, arm_target, and arm_power. Each one becomes a debugging tool for that subsystem.

> **Note:** *Log the op state or state machine state as a string. When you look at a time series later and see a motor doing something unexpected, knowing what state the robot was in at that moment is often the key to understanding why.*

12. Session Naming and Data Management

12.1 A Naming Convention That Pays Off

The tag you pass to telelog with --tag is how you distinguish one session from another in every query you ever write. A careless naming convention turns your database into a pile of ambiguous records. A consistent one makes cross-run analysis effortless.

A recommended convention:

```
# Format: context_YYYY_MM_DD_descriptor

python telelog.py --tag workshop_2026_03_11_motor_balance_test
python telelog.py --tag workshop_2026_03_11_intake_timing_run1
python telelog.py --tag workshop_2026_03_11_intake_timing_run2
python telelog.py --tag comp_2026_03_15_practice_match
python telelog.py --tag comp_2026_03_15_match_1
python telelog.py --tag comp_2026_03_15_match_2
```

The date prefix means the sessions list query always shows runs in chronological order without needing to read timestamps. The descriptor tells you what you were testing. Numbering variants (run1, run2) lets you compare iterations of the same test.

A few rules worth following:

- Use underscores, not spaces. Spaces require quoting in the shell.
- Use lowercase consistently. SQL matching is case-sensitive for text values -- match_1 and Match_1 are different tags.
- Be specific. test and run mean nothing when you look at this database in three months.
- Tag every session, even casual ones. An untagged session is queryable but harder to identify.

12.2 Archiving Competition Data

Your competition database is irreplaceable. It contains the only record of exactly what your robot did during official matches. Protect it.

27. Before each competition day, copy the current database to a backup:

```
copy telemetry.db telemetry_backup_2026_03_15.db
```

28. At the end of the day, copy the database to a second location -- a USB drive, cloud storage, or a team shared drive.
29. Keep competition databases permanently. Storage is cheap. Losing the record of a match where something went wrong and you need to diagnose it later is not recoverable.

12.3 Managing Database Growth

A database that logs every loop iteration during every practice session will grow. This is generally not a problem -- SQLite handles millions of rows efficiently -- but it can become slow to query if you are not selective.

Options for managing size:

- Start fresh each season: copy the previous database to an archive and initialize a new one with --init-db
- Keep separate databases for practice and competition: use --db practice.db for workshop sessions and --db competition.db for match days
- Delete old practice sessions you no longer need with a targeted DELETE query

```
-- Delete all data from a specific tag
DELETE FROM triples
WHERE packet_id IN (
  SELECT id FROM packets WHERE tag = 'workshop_2026_01_05_old_test'
);
DELETE FROM packets WHERE tag = 'workshop_2026_01_05_old_test';
```

> **Note:** *Never delete competition match data. Only delete old practice sessions that you have confirmed are no longer useful.*

13. How This Compares to FTC Dashboard

FTC Dashboard (github.com/acmerobotics/ftc-dashboard) is a widely used tool in the FTC community. If you search for FTC telemetry, you will find it quickly. It is worth understanding how it differs from this system so you can decide when to use each.

Capability	FTC Dashboard	This System (telelog)
Live data display	Yes -- real-time graphs in browser	Via console output; DB Browser requires manual refresh
Configuration variables	Yes -- adjust constants from browser UI	No -- data logging only

Capability	FTC Dashboard	This System (telelog)
Field overlay / path viz	Yes -- 2D field visualization	No
Data persistence	No -- data lives in browser session only	Yes -- SQLite database, permanent
Post-match analysis	Limited	Full SQL querying, graphing, cross-run comparison
Cross-run comparison	No	Yes -- any number of sessions in one database
Custom queries	No	Yes -- full SQL
Setup required	Gradle dependency, Android app	Python script, no install
Network protocol	WebSocket over HTTP	UDP datagrams
Best for	Live tuning during active development	Recording, analysis, diagnostics, competition data

The two tools are not competitors -- they are complementary. FTC Dashboard excels when you are actively tuning a PID controller and want to see the response curve update live. This system excels when you want to answer questions about what the robot did: during a match, across multiple matches, or in comparison to yesterday's configuration.

Many teams run both. FTC Dashboard handles real-time tuning. telelog handles recording. The data from telelog is what you analyze after the session ends, when you are trying to understand why something failed or how much a change actually improved performance.

14. Quick Reference

14.1 telelog.py Commands

Command	What It Does
python telelog.py	Listen on port 3000, write to telemetry.db
python telelog.py --tag match_3	Tag this session as match_3
python telelog.py --port 3001	Listen on a different port
python telelog.py --db myfile.db	Use a different database file
python telelog.py --quiet	No per-packet console output
python telelog.py --init-db	Create schema and exit (safe to re-run)
python telelog.py --tag run1 --quiet	Combined: tagged and silent

14.2 SQL Quick Reference

Query Goal	Key Clause to Change	Section
Session list	Nothing	7.1
Schema discovery	Nothing	7.1
Browse all data	p.tag value	7.1
Inspect one packet	p.id number	7.1
Time series for one signal	subject, predicate, tag	7.2
Two signals compared	subject/predicate in each JOIN	7.2
Full dashboard pivot	Add or remove JOIN blocks	7.2
Packet timing gaps	p.tag value	7.3
Outlier search	subject, predicate, threshold	7.3
Peak value in a run	subject, predicate, ASC or DESC	7.3
Rolling average	ROWS BETWEEN N PRECEDING	7.3
Last known state	p.tag value	7.3
Frozen sensor check	subject, predicate	7.3
Per-run statistics	subject, predicate	7.4
Run overlay	Both tag values	7.4
Motor velocity imbalance	subject names for your motors	8.1
Camera target loss events	subject/predicate for your camera	8.2
Sensor cross-environment	subject, predicate, tag per venue	8.3
Intake transit time	sensor/delivery subject names	8.4
Motor stall detection	subject for your motor	8.5
Heading drift analysis	subject names for imu and drive	8.6
Multi-system event window	packet id range, subject list	8.7

14.3 Control Hub Network Reference

Item	Value
Control Hub WiFi ESSID	FTC-[team number]-RC (e.g. FTC-13532-RC)
Control Hub IP address	192.168.43.1
Your laptop IP	192.168.43.x (assigned automatically by Control Hub)
Default telemetry port	3000 (configurable with --port)
UDP payload format	JSON text, UTF-8 encoded
Typical packet rate	Depends on your loop rate -- one packet per loop iteration