Factora IoT

Infrastructure Guide

How our IoT platform works, explained one piece at a time, in plain language

Prepared for	Isaac Mwendwa, Founder, Tamalaki Business Network Ltd
Platform	Factora IoT (iot.factoraerp.com) and the LoRaWAN network server (lns.factoraiot.com)
Date	16 October 2026 (EAT)
Based on	A read-only check of the live server on 16 Oct 2026, about 00:45 EAT, plus our LoRaWAN plan and deployment runbook. Nothing was changed. No passwords or secret values appear in this guide.
Hardware	Vendor-neutral. No gateway or sensor brand has been chosen yet.

How to read this guide: Every technology gets the same three answers: **What it is** (with a simple comparison where it helps, but also the real mechanism), **Why we use it**, and **How it is set up in our system** (the actual settings on our server). If a word is unfamiliar, look it up in the Glossary in section 10.

Contents

- 1. The big picture
- 2. The journey of one sensor reading
- 3. Each component explained (3.1 to 3.19)
- 4. Security
- 5. Multi-tenancy: keeping factories apart
- 6. Backups and recovery
- 7. How we deploy changes
- 8. Capacity and when to scale
- 9. What is not done yet
- 10. Glossary
- Appendix A. Where things live on the server
- Appendix B. Key facts at a glance

Diagrams in this guide

- Figure 1. The big picture (section 1)
- Figure 2. The journey of one reading (section 2)
- Figure 3. How MQTT routes messages (section 3.8)
- Figure 4. Server layout: containers, networks and ports (section 3.9)
- Figure 5. One tenant per factory (section 5)
- Figure 6. Backup schedule (section 6)
- Figure 7. Deployment steps (section 7)

1. The big picture

Factora IoT lets a factory see what is happening on its floor, live: temperatures, humidity, energy use, tank levels and machine states. It stores every measurement, shows it on dashboards and raises an alert when something goes out of range.

The system has four layers. Think of them as a delivery chain:

1. **Sensors** in the factory measure something and send a tiny radio message.
2. **A gateway** in the factory hears those radio messages and forwards them over the internet.
3. **Our cloud server** receives the messages, checks them, decodes them and stores them in a database.
4. **The web app** shows the data to factory staff and managers, and raises alerts.

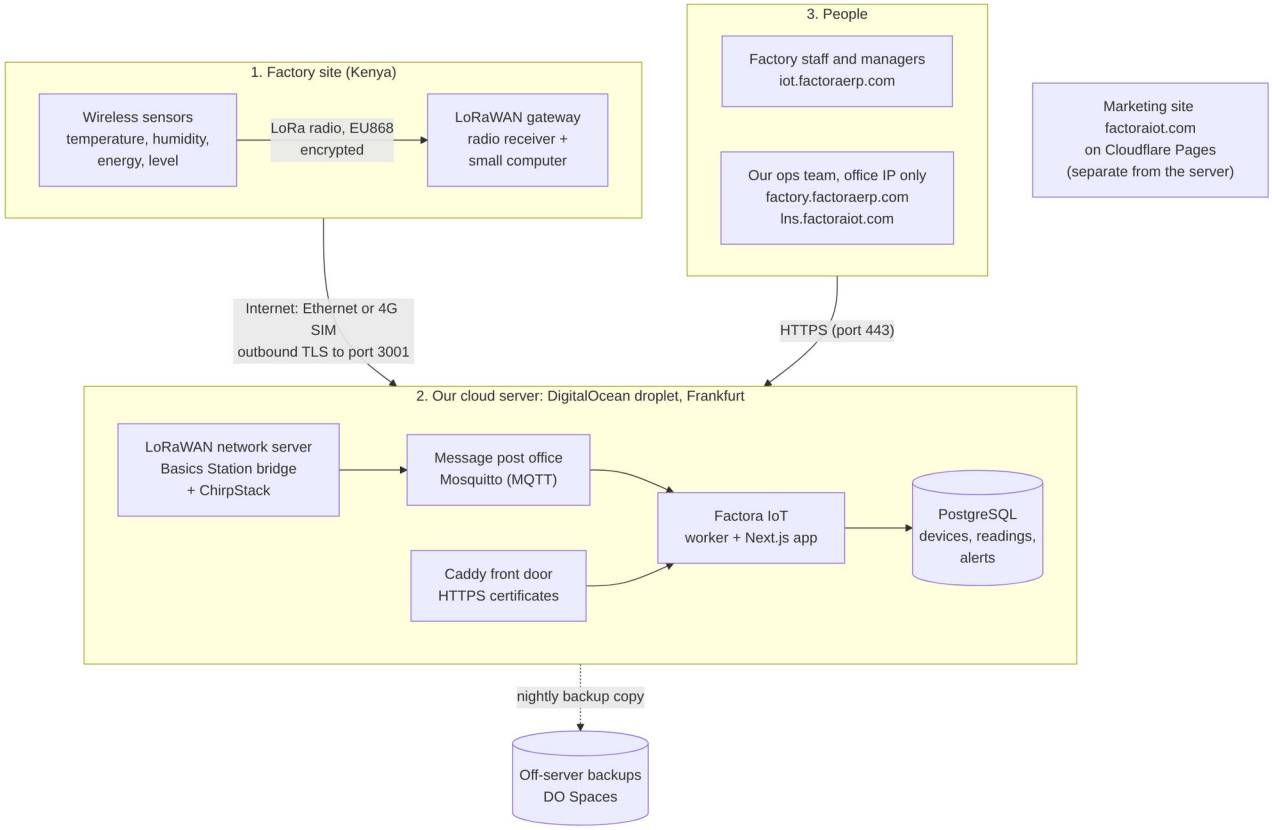

Figure 1. The big picture: from the sensor on the factory floor to the person looking at the dashboard.

The main parts at a glance

Part	What it does, in plain words	Where it runs
Sensors	Measure temperature, humidity, energy, level and similar, then transmit by radio	Factory floor
LoRaWAN gateway	Listens to all sensors in range and passes their messages to our server	Factory (one or more per site)
Basics Station bridge	The secure "receiving desk" where gateways connect (port 3001)	Our server, lorawan project
ChirpStack	The LoRaWAN network server: checks, de-duplicates, decrypts and decodes sensor messages	Our server, lorawan project
Mosquitto	The MQTT "post office" that moves messages between the parts	Our server, lorawan project
Factora worker	Background program that picks up messages, saves readings, checks alerts	Our server, factora-iot project
PostgreSQL	The database: factories, devices, sensors, readings, alerts, users	Our server (two separate databases)
Next.js web app	The website people log in to: dashboards, alerts, settings, ops console	Our server, factora-iot project

Caddy	The front door: HTTPS certificates, routing each web address, IP restrictions	Our server, factora-iot project

Our web addresses

Address	What it is	Who can open it
iot.factoraerp.com	The Factora IoT platform (factory dashboards)	Anyone with a login
factory.factoraerp.com	The ops console (our super-admin view of all factories)	Our team, from the office IP only, with 2-factor login
lns.factoraiot.com	ChirpStack admin screens (LoRaWAN network server). Gateways also connect here on port 3001	Admin screens: office IP only. Port 3001: gateways
factoraiot.com	Marketing website, hosted on Cloudflare Pages (not on our server)	Public
factoraacademy.factoraerp.com	Factora Academy (training content) served by the same app	Public

Who manages DNS: the factoraerp.com domain is managed by an external party, so any new factoraerp.com address needs a request to them. The factoraiot.com domain is in our own Cloudflare account, so we can add records ourselves (this is why the LoRaWAN address is lns.**factoraiot**.com).

Neighbours on the same server: the same droplet also runs FarmMate and the farm ERP (admin.farmmate.app, coop.farmmate.app, farm.factoraiot.com) and the FactoraERP pricing site (pricing.factoraerp.com). They are separate Docker projects with their own databases, but they share the server's CPU, memory and disk, and the same Caddy front door.

Where we are today (honest status)

Item	Status on 16 Oct 2026
Factora IoT platform	Live and healthy. 7 factories (companies) set up, all in **demo** mode, 57 devices, about 2.87 million readings (mostly from the built-in simulator), database about 1.48 GB
LoRaWAN stack	Installed and running since 15 Oct 2026 (ChirpStack 4.19.2, Basics Station bridge, Mosquitto 2.0.20, Postgres 16, Redis 7, EU868). Valid HTTPS certificate on lns.factoraiot.com
Real gateways and sensors	None registered yet. Hardware has not been chosen
Link from ChirpStack to Factora	The code is ready (MQTT connector). The live connector for the first factory is the "Day 3" task
Off-server backups	Set up 15 to 16 Oct 2026 to DigitalOcean Spaces; first scheduled nightly run is 02:30 EAT on 16 Oct

2. The journey of one sensor reading

Let us follow a single temperature reading, for example "24.5 °C in cold room 2", from the sensor to the screen. The whole trip is designed to take a few seconds (our acceptance target is under 5 seconds from radio to database).

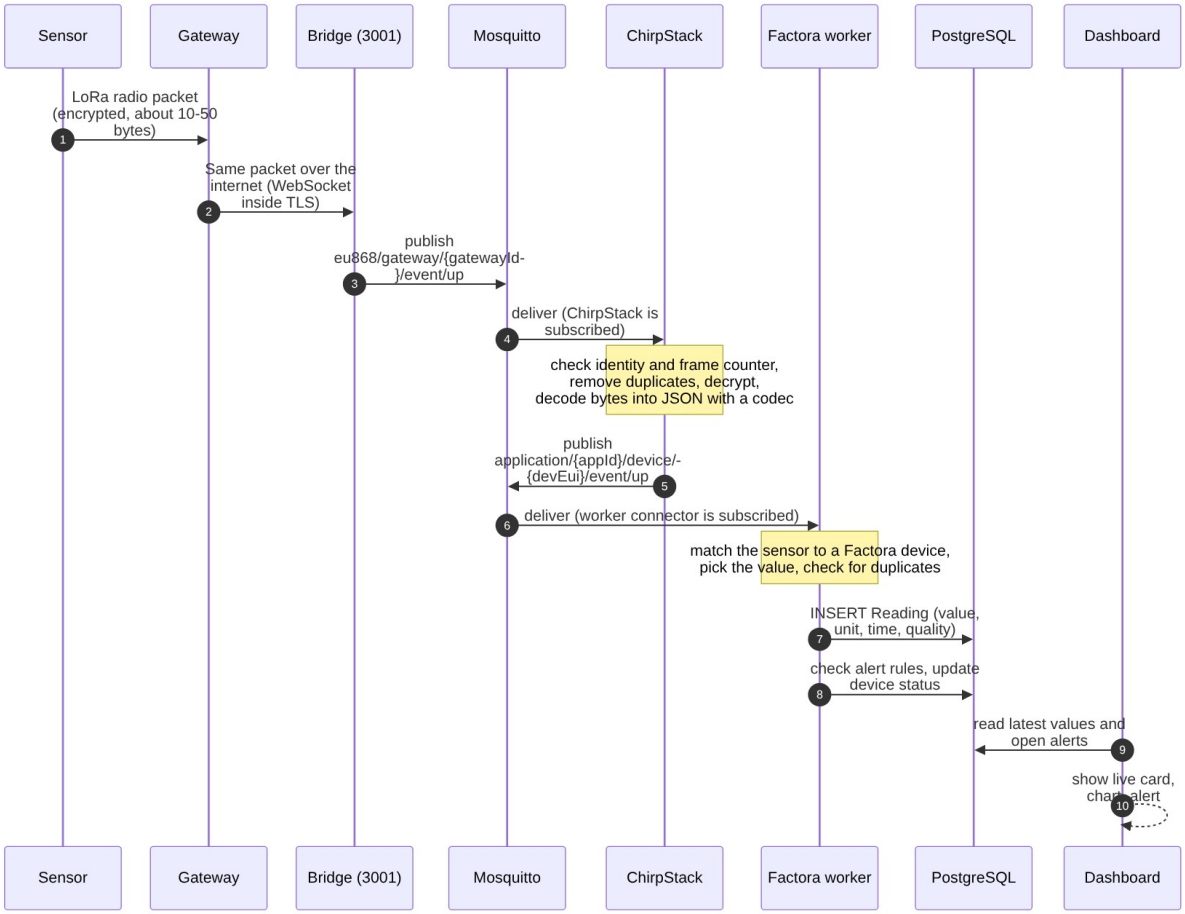

Figure 2. The journey of one reading, from radio packet to dashboard. The written steps below explain each hop.

1. The sensor measures and transmits

The sensor wakes up on a timer (for example every 10 minutes), takes a measurement and packs it into a very small message, usually 10 to 50 bytes. Before sending, it **encrypts** the data with AES-128 keys it agreed with our network server when it first joined. It then transmits by LoRa radio on one of eight channels between 867.1 and 868.5 MHz (the EU868 band). The transmission lasts from a few tens of milliseconds up to about a second, and the sensor goes back to sleep. This is why batteries can last years.

2. The gateway hears it

Any gateway in range picks up the radio signal. The gateway cannot read the content (it is encrypted). It adds a few notes of its own: the time it heard the message, the signal strength (RSSI) and the signal quality (SNR). If two gateways hear the same message, both forward it; the network server removes the duplicate later.

3. The gateway sends it over the internet

The gateway is connected to the internet by factory Ethernet or a 4G SIM. Using the **LoRa Basics Station** protocol, it keeps one long-lived, encrypted connection open to `lns.factoraiot.com` on port 3001 (a WebSocket inside TLS, written "wss"). The gateway opens this connection outwards, so the factory never has to open a port on its own firewall.

4. The bridge receives it

On our server, the **gateway bridge** container answers on port 3001. It proves it is really lns.factoraiot.com with our Let's Encrypt certificate, checks the gateway's identity, and translates the message into the internal format. It then **publishes** it to the MQTT broker (Mosquitto) on the topic `eu868/gateway/{gatewayId}/event/up`.

5. Mosquitto passes it to ChirpStack

Mosquitto is the post office. ChirpStack has **subscribed** to the gateway topics, so Mosquitto immediately delivers the message to it.

6. ChirpStack checks, decrypts and decodes

ChirpStack looks up which sensor sent it, checks the message's integrity code (so a tampered or fake message is rejected), checks the frame counter (so an old message replayed by an attacker is rejected), removes duplicates from other gateways, decrypts the data and runs a small **codec** (decoder program) for that sensor type that turns raw bytes such as `09 92` into readable JSON such as `{"temperature": 24.5}`. If the sensor needs new radio settings, ChirpStack sends a reply (a "downlink") back down the same path.

7. ChirpStack publishes the result

ChirpStack publishes a JSON "uplink event" to Mosquitto on `application/{applicationId}/device/{devEui}/event/up`. It contains the decoded values, the sensor's unique ID (DevEUI), the time, signal strength and a de-duplication ID.

8. The Factora worker picks it up

The Factora **worker** has an MQTT connector for that factory that is subscribed to that factory's application topic. It receives the event, matches the sensor to a Factora device using the connector mappings, pulls out the value (for example the field `object.temperature`) and checks it has not stored this exact message before.

9. The reading is saved in PostgreSQL

The worker writes one row to the `Reading` table: value, unit, the device, the factory, the time the event happened (`recordedAt`), the time we received it (`receivedAt`), the source and a quality flag. The device's "last seen" time and last value are updated. If the database is briefly unreachable, the reading is held in a local buffer on the server and written later.

10. Alerts are checked

Straight after saving, the alert rules for that device are checked (for example "above 30 °C" or "below 5 °C"). If a rule is broken, an `Alert` row is created with a severity. Separately, every 5 seconds the worker runs housekeeping, including spotting devices that have gone silent ("offline") and data that has gone stale.

11. The dashboard shows it

When a manager opens iot.factoraerp.com, the Next.js app reads the latest values and open alerts from the database and draws the live cards, charts and alert list.

Status: Steps 1 to 7 run inside the LoRaWAN stack that is installed today, but no real gateway or sensor is connected yet. Steps 8 to 11 are live in the Factora platform today with demo data. Wiring step 7 to step 8 (creating the first factory's MQTT connector) is the "Day 3" task in section 9.

The other way in: devices that use the internet directly

Not every device is LoRaWAN. Some meters and machine controllers speak internet protocols directly. For those, Factora IoT has more doors:

- **The HTTP ingest API** (`/api/ingest`): a device sends a reading in a web request with its own secret **ingest token**.
- **The built-in MQTT broker** inside the worker (on `127.0.0.1:1883`). A device logs in with its device code and ingest token, and may only publish to its own topic,

`factora/{factory}/devices/{deviceCode}/telemetry`, with messages up to 16 KB. Today this broker only listens inside the server (address 127.0.0.1), so it is not reachable from the internet. Opening it to outside devices would need TLS first (see section 9).

- **Other connectors** (OPC UA, Modbus, polling a web address) exist in the same connector framework for industrial equipment.

3. Each component explained

This section takes each technology one at a time, roughly in the order a reading meets them, and then the supporting pieces (containers, database, web app, security and radio rules).

3.1 The server (our DigitalOcean droplet)

In one line: One rented virtual computer in a data centre in Frankfurt runs everything on the cloud side.

What it is

A "droplet" is DigitalOcean's name for a virtual private server: a slice of a powerful physical machine that behaves like our own computer, with its own operating system, disk and public internet address. We rent it by the month.

Ours runs Ubuntu Linux 24.04 LTS with 4 virtual CPUs, 8 GB of memory and a 160 GB disk, in the fra1 (Frankfurt) region. DigitalOcean has no African region, and Frankfurt gives a typical round-trip time from Nairobi of about 130 to 180 milliseconds, which is fine for sensors that report every few minutes.

Think of it like: Renting a fully serviced office in a large building. The landlord (DigitalOcean) handles power, cooling and the building's internet; we decide what goes inside our office.

Why we use it

- It is simple and cheap for a pilot: one machine (about US$48 a month) runs the whole platform.
- We control everything on it, including the LoRaWAN network server, so there are no per-device fees.
- It has plenty of headroom today (see section 8).

How it is set up in our system

- Region fra1, 4 vCPU, 8 GB RAM, 160 GB disk, Ubuntu 24.04.5, Docker 29.8.
- Current use: about 2 GB memory used and 5.7 GB available, 21 GB of disk used (14%), load average about 0.5 (the 4 CPUs are mostly idle).
- All software runs in Docker containers (section 3.9), grouped into separate projects: factora-iot, lorawan, farmerp and factora-pricing.
- The server clock runs on UTC. Times in this guide are converted to EAT (UTC+3).

3.2 LoRaWAN (the radio network for sensors)

In one line: A way for small battery sensors to send short messages kilometres away using very little power.

What it is

LoRa ("Long Range") is a radio technique. It spreads each message across a band of frequencies in a rising "chirp" pattern, which lets a receiver pick it up even when it is weaker than the background noise. The trade-off is speed: LoRa carries only a few hundred to a few thousand bits per second. That is plenty for a temperature value, but useless for video.

LoRaWAN is the set of network rules on top of LoRa, published by the LoRa Alliance. It defines how a sensor joins the network, how messages are addressed, encrypted and counted, and how the network tells the sensor to adjust its speed and power.

Think of it like: Shouting short messages across a big field. You can be heard a long way off if you speak slowly and clearly (LoRa), and everyone follows the same etiquette about who speaks when and how to sign their messages (LoRaWAN).

Why we use it

- Long range: typically 1 to 3 km inside industrial sites and up to 10 km or more in open country, so one gateway can cover a whole factory compound.
- Very low power: sensors sleep most of the time, so batteries can last several years.
- No SIM card or Wi-Fi per sensor: only the gateway needs internet.
- It is an open standard with many manufacturers, so we are not locked to one hardware vendor.

How it is set up in our system

- **Joining (OTAA):** each sensor is registered with three values: a unique ID (**DevEUI**), an application ID (**JoinEUI**) and a secret root key (**AppKey**). On first power-up it sends a "join request"; the network server answers and both sides work out fresh **session keys**. The AppKey itself is never sent over the air.
- **Two layers of AES-128 encryption:** the *network session key* signs every message so the network can prove it is genuine and untouched; the *application session key* encrypts the actual data so even the gateway cannot read it.
- **Frame counters:** every message carries a number that goes up by one. A message with an old number is rejected, so recorded messages cannot be replayed.
- **Device class A:** most sensors are "Class A": they talk first and listen briefly afterwards. That saves the most battery.
- **Adaptive data rate (ADR):** the network server tells sensors close to a gateway to transmit faster and at lower power, which saves battery and airtime.
- In our system: ChirpStack is configured for the **EU868** region only (section 3.19), with the eight standard channels (868.1, 868.3, 868.5, 867.1, 867.3, 867.5, 867.7, 867.9 MHz) and spreading factors SF7 to SF12.

3.3 The gateway

In one line: A radio receiver with a small computer inside that listens to every sensor nearby and relays their messages to our server.

What it is

A LoRaWAN gateway has a multi-channel LoRa radio (it can listen on 8 or more channels at once), an antenna, a small Linux computer and an internet connection (Ethernet, and often a 4G modem). It does not understand or decrypt sensor data; it is a relay. It also transmits replies (downlinks) to sensors when the network server asks.

Think of it like: A switchboard operator at the factory gate who takes every sealed envelope dropped off by sensors and forwards it unopened to head office.

Why we use it

- One gateway can serve hundreds of sensors across a site.
- Sensors are not tied to a gateway. If two gateways hear a message, both forward it, which gives free redundancy.
- It moves all the internet complexity into one box per site.

How it is set up in our system

- **Not yet chosen:** we have not picked a gateway model. We will select an industrial EU868 gateway through a vendor evaluation (section 9). Requirements: Communications Authority of Kenya type approval, LoRa Basics Station support, Ethernet plus 4G (ideally dual SIM), an industrial enclosure, and power backup (UPS).
- **Internet connection:** factory Ethernet where it is reliable, with a 4G SIM as primary or fallback. Only outbound internet access is needed.

- **Connection to us:** each gateway is set to connect to `wss://lns.factoraiot.com:3001` using LoRa Basics Station (section 3.4). Each gateway is registered in ChirpStack by its unique Gateway ID (EUI) under its factory's tenant.
- Planned ChirpStack limit: up to 3 gateways per factory tenant.

3.4 LoRa Basics Station (how gateways connect to us securely)

In one line: The modern, secure way for a gateway to talk to a network server: one encrypted connection that the gateway opens outwards.

What it is

Basics Station is open-source gateway software and a protocol. The gateway first asks the server where to connect, then opens a **WebSocket** (a long-lived, two-way connection that uses the web's normal technology) wrapped in **TLS** encryption. Sensor messages travel up, and replies and configuration travel down, over that one connection.

The older alternative (the "UDP packet forwarder") sends unencrypted, unauthenticated packets. We do not open its port (1700).

Think of it like: A secure courier tunnel. Instead of posting postcards anyone can read (the old UDP method), each gateway opens a locked, private tube to our office and sends everything through it.

Why we use it

- Encrypted between gateway and server, using the same kind of certificate as a banking website.
- Each gateway proves who it is (with a per-gateway token or certificate), so a stranger cannot pretend to be our gateway.
- The gateway dials out, so no inbound port needs opening at the factory, and it works through 4G carrier networks.
- The server pushes the radio settings (EU868 channels) to the gateway, so gateways need less manual radio configuration.

How it is set up in our system

- Runs as the container **gateway-bridge-basicstation** (ChirpStack Gateway Bridge 4.1.2), the only LoRaWAN container published to the internet, on **port 3001/TCP**.
- TLS uses the Let's Encrypt certificate for lns.factoraiot.com (valid until 13 Jan 2027, renewed automatically; see section 3.16).
- Region EU868, frequency range 863 to 870 MHz.
- It translates gateway messages into MQTT and publishes them to Mosquitto under `eu868/gateway/{GatewayID}/event/{type}`, and listens for commands on `eu868/gateway/{GatewayID}/command/#`. It logs in to Mosquitto as its own user, which may only use `eu868/gateway/#` topics.
- Resource limit: 128 MB memory, half a CPU.
- Planned hardening: per-gateway client certificates (mutual TLS) from the first production gateway.

3.5 Internet backhaul (Ethernet or 4G)

In one line: "Backhaul" is simply how the gateway itself reaches the internet.

What it is

The gateway needs an ordinary internet connection. It can be the factory's wired network (Ethernet), Wi-Fi, or a mobile 4G SIM inside the gateway. Data use is small: typically a few hundred MB per month per gateway for a sensor fleet.

Think of it like: The road between the factory gate and head office. Tarmac (Ethernet) is smooth when it exists; a motorbike courier (4G) can get through anywhere.

Why we use it

- 4G avoids depending on the factory's IT network and firewall, and works where there is no cable.
- Ethernet is cheaper and steadier where the factory network is good.
- Dual SIM and a UPS keep data flowing during network or power cuts.

How it is set up in our system

- Decision per factory (pending hardware choice): Ethernet primary with 4G fallback, or 4G only.
- Only outbound access to lns.factoraiot.com port 3001 is required. No inbound ports at the factory.
- Budget estimate: about US$10 to 20 per gateway per month for 4G data.

3.6 The LoRaWAN network server (ChirpStack)

In one line: The "brain" of the LoRaWAN network: it knows every gateway and sensor, checks every message and turns it into usable data.

What it is

A network server (LNS) does the jobs the gateways cannot: it holds the list of allowed devices and their keys, handles joins, verifies and decrypts messages, removes duplicates, manages radio settings (ADR), queues downlinks, and passes decoded data to applications.

ChirpStack is a widely used open-source LNS (MIT licence, no fees). Version 4 combines the network server and application server in one program with a web admin screen and an API.

Think of it like: The sorting office at head office. It checks each envelope's seal and stamp, throws away duplicates and forgeries, opens the genuine ones with the right key and translates the contents into plain language.

Why we use it

- No licence or per-device fees, unlike managed LoRaWAN cloud services (roughly €190 a month for 1,000 devices).
- Built-in **multi-tenancy**: one ChirpStack can host many factories, each in its own tenant (section 5).
- Device profiles and **codecs** (small JavaScript decoders) let us support sensors from any manufacturer.
- It publishes clean JSON over MQTT, which our worker already understands.

How it is set up in our system

- Container **chirpstack** (image chirpstack/chirpstack:4.19.2) in the compose project **lorawan** in `/opt/lorawan`.
- Configuration: `/opt/lorawan/configuration/chirpstack/chirpstack.toml` and `region_eu868.toml`. Only the eu868 region is enabled.
- It stores devices, gateways, tenants and users in its **own** PostgreSQL 16 database, and fast-changing state (device sessions, de-duplication) in **Redis 7**.

- MQTT integration: decoded uplinks go to `application/{application_id}/device/{dev_eui}/event/{event}` as JSON. Events include `up` (data), `join`, `status` (battery) and others.
- Admin web screens and API on port 8080 **inside** Docker only. Caddy publishes them as https://lns.factoraiot.com, reachable only from the office IP. (Gateways do not use the web side; they use port 3001.)
- Resource limits: 384 MB memory, 1 CPU.
- Today: only the default admin tenant exists; no gateways or devices are registered yet.

3.7 Redis (fast short-term memory for ChirpStack)

In one line: A very fast in-memory store that ChirpStack uses for data it needs to read and change many times a second.

What it is

Redis keeps data in memory (RAM) rather than on disk, so reads and writes take a fraction of a millisecond. It can also save snapshots and an append-only log to disk, so it survives a restart.

Think of it like: A whiteboard next to the sorting desk: quick to write on and read, and photographed regularly so nothing is lost if someone bumps it.

Why we use it

- ChirpStack needs it for device sessions, frame counters, de-duplication windows and downlink queues, which change with every message.
- It keeps that churn away from the main ChirpStack database.

How it is set up in our system

- Container **redis** (redis:7-alpine) in the lorawan project, on the private lorawan network only (not reachable from outside or from Factora containers).
- Saves to disk every 5 minutes if anything changed, plus an append-only log (`--save 300 1 --appendonly yes`).
- Memory cap 96 MB inside Redis (128 MB container limit) with "noeviction", so it never silently drops session data.
- Factora IoT itself does not use Redis.

3.8 MQTT (how messages move between our parts)

In one line: We use MQTT. MQTT is a lightweight "publish and subscribe" messaging method: senders post messages to named topics on a central broker, and the broker instantly forwards each message to everyone who subscribed to that topic.

What it is

MQTT (originally "MQ Telemetry Transport") is an open standard for sending small messages between machines. It was designed for unreliable, low-bandwidth links, so it has very little overhead (a message header can be as small as 2 bytes). It runs over an ordinary network connection (TCP), optionally inside TLS. Each client keeps one connection open to the broker, so messages are pushed instantly instead of being checked for repeatedly.

Think of it like: A post office with named PO boxes. A sender drops a letter addressed to a PO box name (the topic). Anyone who has asked for copies of that PO box (a subscriber) gets a copy immediately. The sender does not need to know who the readers are, and the readers do not need to know who sent it.

The real mechanism, piece by piece:

MQTT idea	What it means	Example in our system
Broker	The central server that every client connects to. It receives every message, checks permissions and forwards copies to subscribers.	Mosquitto 2.0.20 in the lorawan project; plus a small built-in broker inside the Factora worker for direct devices
Client	Any program that connects to the broker. It can publish, subscribe, or both.	Gateway bridge, ChirpStack, the Factora worker
Topic	The "address" of a message: a text path with levels separated by slashes.	`application/7/device/a1b2c3/event/up`
Publish	Send a message to a topic.	ChirpStack publishes each decoded uplink
Subscribe	Ask the broker for every message on a topic or topic pattern.	The worker subscribes to one factory's application topics
Wildcards	`+` matches exactly one level, `#` matches everything below.	`eu868/gateway/+/event/+` = all events from all gateways
QoS 0	"At most once": sent once, no confirmation. Fastest; a message can be lost.	Gateway traffic inside the server (qos=0); acceptable because it never leaves the machine
QoS 1	"At least once": the receiver confirms; the sender retries until confirmed. A message can arrive twice, so the receiver must ignore duplicates.	Factora connectors default to QoS 1, and our ingest ignores duplicate message IDs
QoS 2	"Exactly once": a four-step handshake. Slowest; rarely needed.	Supported but not used
Login and ACL	Clients log in with a username and password; an Access Control List says which topics each user may read or write.	Gateway bridge may only use `eu868/gateway/#`; factory users will only read their own application
Keep-alive and reconnect	Clients send small "ping" packets; if the link drops, they reconnect.	Worker reconnects with increasing waits and logs each disconnect

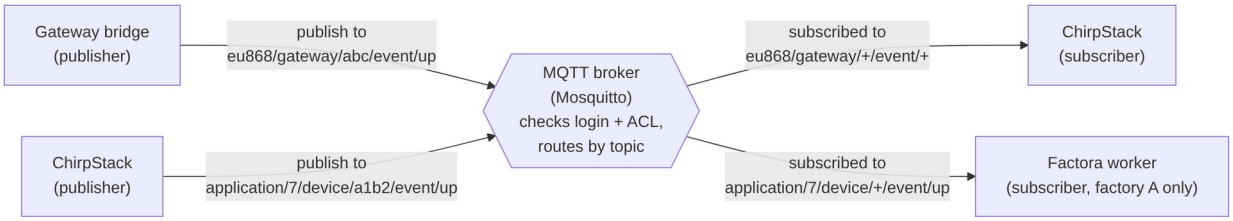

Figure 3. How MQTT routes messages. Publishers and subscribers only know the broker and the topic names.

Why we use it

- ChirpStack and the gateway bridge are built around MQTT, so it is the natural glue for LoRaWAN.
- It **decouples** the parts: ChirpStack does not need to know that Factora exists, and we can add a second consumer (for example a backup logger) without touching ChirpStack.
- It is light, fast and proven in industrial IoT, and many industrial gateways and controllers can speak it directly.
- Per-user topic permissions (ACLs) give a clean way to keep each factory's data separate.

How it is set up in our system

We actually have **two** MQTT brokers, each with a different job:

	Mosquitto (LoRaWAN broker)	Built-in Factora broker
Purpose	Moves LoRaWAN traffic: gateway bridge ↔ ChirpStack, and ChirpStack → Factora worker	Lets IP-connected devices publish readings straight into Factora
Software	Eclipse Mosquitto 2.0.20, container `lorawan-mosquitto-1`	Aedes (a Node.js broker library) running inside the worker
Address	`mosquitto:1883` on the internal Docker networks only. **Not** published to the internet	`127.0.0.1:1883` inside the app/worker container only

Login	`allow_anonymous false`; password file (hashed passwords)	Device code as username, the device's ingest token as password
Permissions	ACL file: `chirpstack` reads/writes all topics; `gateway-bridge` only `eu868/gateway/#`. Per-factory read-only users to be added (section 5)	A device may only publish to `factora/{factory}/devices/{its code}/telemetry`; max 16 KB per message
Persistence	On (stored in the `mosquitto_data` volume)	Off

- **Topics used:** gateway side `eu868/gateway/{gatewayId}/event/...` and `.../command/...`; application side `application/{applicationId}/device/{devEui}/event/up|join|status|ack`.
- **The Factora side (connector runtime):** the worker file `src/lib/connector-runtime.ts` starts one connection per enabled connector. For an MQTT connector it connects (with TLS if configured), subscribes to the configured topics at QoS 1 by default, and for each message finds the matching **mapping**, reads the value at the mapped path (for example `object.temperature`) and stores it. It reconnects with back-off, records its status (connecting, connected, disconnected, error) and protects itself with a limit of 250 messages in progress per connector ("backpressure").
- **Planned first-factory connector:** host `mqtt://mosquitto:1883` (internal network, so no TLS is needed inside the server), username `factora-<factory>`, topic `application/<appId>/device/+/event/up`. Not yet created (section 9).
- Production rule in the code: if Factora is ever pointed at an **external** broker over the internet, it must use encrypted `mqtts://` or `wss://`.

3.9 Docker and containers

In one line: Each piece of software runs in its own sealed, repeatable package called a container.

What it is

A **container image** is a frozen package that holds a program plus everything it needs (libraries, settings, operating-system files). A **container** is a running copy of that image. Containers share the server's Linux kernel, but each sees only its own files, processes and network, so they cannot interfere with each other. Docker is the tool that builds images and runs containers.

Volumes are Docker-managed folders that live outside the container, so data (for example a database) survives when the container is replaced. **Networks** are private virtual networks that let containers talk to each other by name (for example `db` or `mosquitto`) without exposing them to the internet.

Think of it like: Shipping containers. Whatever is inside, every container has the same shape, so the same crane (Docker) can load, move and stack them, and a leak in one does not soak the others.

Why we use it

- The exact same package runs on a laptop and on the server, so "it worked on my machine" problems disappear.
- Updating is simple and safe: build a new image, replace the container, keep the data volume.
- Isolation: the farm ERP, the pricing site, Factora IoT and LoRaWAN cannot step on each other's files or software versions.
- We can cap the memory and CPU each container may use.

How it is set up in our system

- 12 containers run today, all healthy: factora-iot (app, worker, db, caddy), lorawan (chirpstack, gateway-bridge-basicstation, mosquitto, postgres, redis), farmerp (app, db) and factora-pricing.
- Images: Factora IoT is built from our own `Dockerfile` on Node.js 22. The others are official images pinned to versions (postgres:16-alpine, caddy:2-alpine, chirpstack 4.19.2, gateway bridge 4.1.2, mosquitto 2.0.20, redis:7-alpine).

- Networks: `factora-iot_default` (Factora and Caddy; ChirpStack and Mosquitto also join it so Caddy and the worker can reach them), `lorawan_lorawan` (private LoRaWAN network; its Postgres and Redis are only here), `farmerp_internal`.
- Volumes: `factora_pg` (Factora database), `factora_edge` (reading buffer and worker heartbeat), `caddy_data` (certificates), plus the LoRaWAN `postgres_data`, `redis_data` and `mosquitto_data`.
- Logs are rotated (10 MB × 3 files per container) so they never fill the disk.
- Every container is set to `restart: unless-stopped`, so it comes back by itself after a crash or a server reboot.

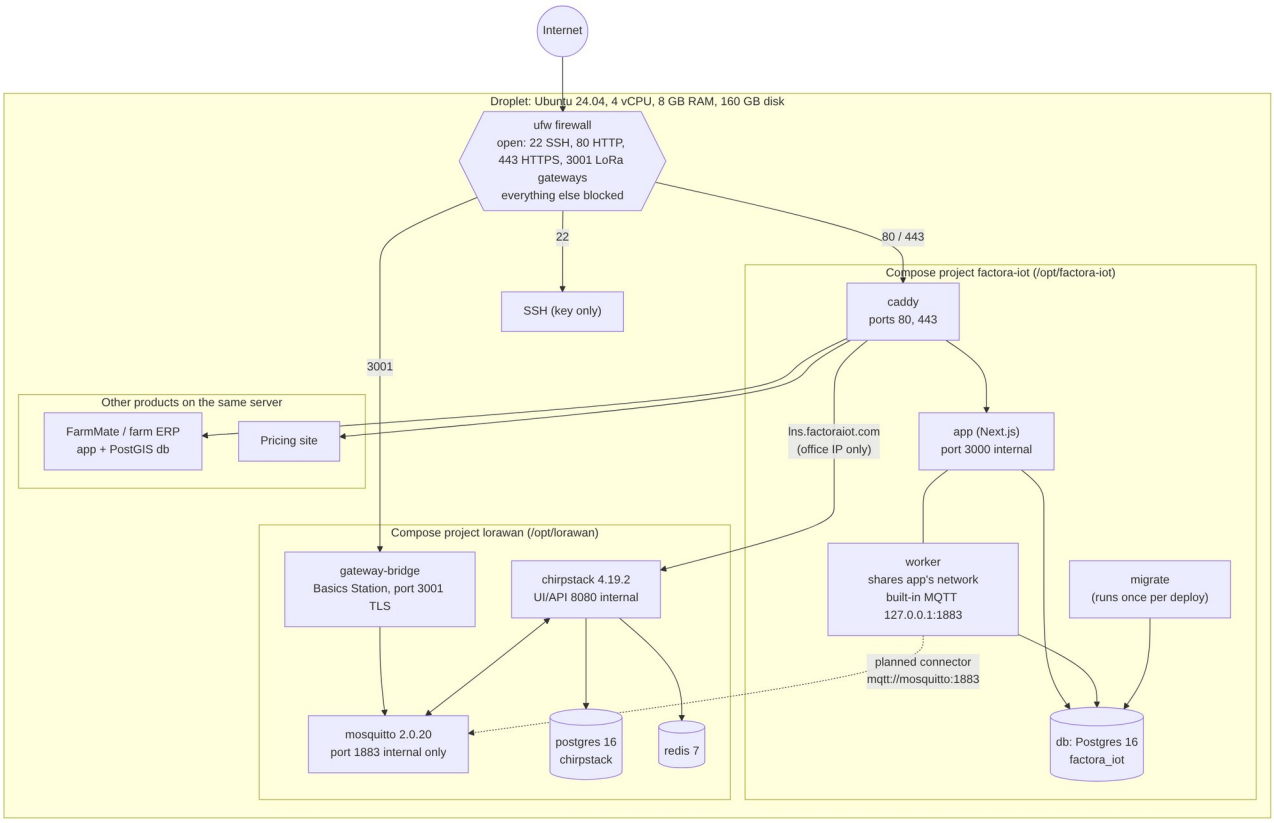

Figure 4. Server layout: the firewall, the Docker projects, the containers inside them and the ports between them.

3.10 Docker Compose

In one line: A recipe file that describes a group of containers and how they fit together, so one command starts the whole group.

What it is

Docker Compose reads a YAML file listing each service (container), its image, settings, volumes, networks, ports, health checks and start-up order. `docker compose up -d` makes reality match the file: it creates what is missing and replaces what has changed. A "project" is one such group.

Think of it like: A recipe card for a full meal: which dishes, in which order, on which plates, and which must be ready before the next starts.

Why we use it

- The whole setup is written down as a file in the code repository, not in someone's memory.
- Start-up order and health checks are automatic (for example the app waits until the database is healthy and the migrations have finished).
- Rebuilding or rolling back is one command.

How it is set up in our system

- **factora-iot** (`/opt/factora-iot/docker-compose.prod.yml`): `db` (Postgres 16), `migrate` (runs database migrations once and exits), `app` (Next.js website on port 3000, internal), `worker` (background jobs; shares the app's network so they see the same 127.0.0.1), `caddy` (ports 80 and 443).
- Start order: db healthy → migrate completes successfully → app healthy (it checks `/api/health` every 15 s) → worker and caddy start. The worker's own health check fails if its heartbeat is older than 20 seconds.
- **lorawan** (`/opt/lorawan/docker-compose.yml`): chirpstack, gateway-bridge-basicstation (port 3001), mosquitto, postgres, redis, each with memory and CPU limits (about 1.1 GB of memory in total).
- Secrets are not written in the compose files. They are read from a `.env` file next to each compose file (section 4).
- The two projects are independent: restarting LoRaWAN never restarts Factora IoT, and the reverse.

3.11 PostgreSQL (the database)

In one line: The reliable filing system where everything is stored: factories, users, devices, sensors, readings and alerts.

What it is

PostgreSQL ("Postgres") is a free, open-source relational database. Data is kept in tables of rows and columns, linked by IDs, and queried with SQL. It guarantees that a change is either fully saved or not at all (transactions), and it uses **indexes** (like a book's index) to find rows quickly among millions.

Think of it like: A very well organised filing room with an index-card system. Every reading is a card filed under its factory, device and time, so any clerk can pull "Cold room 2, last 24 hours" in a second.

Why we use it

- Proven, free and very reliable; used by banks and governments.
- Handles millions of readings with fast time-range queries through indexes.
- Standard tools for backups (`pg_dump`) and restores.

How it is set up in our system

- We run **two** separate Postgres 16 databases, so neither system can damage the other.
- **factora_iot** (container `factora-iot-db-1`, volume `factora_pg`): about **1.48 GB**, about **2.87 million readings**, **57 devices**, 7 companies. Internal port 5432 only.
- **chirpstack** (container `lorawan-postgres-1`): ChirpStack's own data (tenants, gateways, devices, keys), on the private LoRaWAN network only, with the `pg_trgm` extension ChirpStack needs.
- **Our data model in simple terms:** a **Company** is one factory customer (with a short name such as `demo-food`, timezone, currency KES and an MFA switch). It has **Areas** (sections of the plant) and **ProductionLines**. A **Device** is one measured point (for example "Cold room 2 temperature"), with a code, type, unit, secret ingest token, last value and last-seen time. Each measurement is a **Reading** row (value, unit, source, quality, event time and receipt time). **AlertRules** set limits per device (min, max, offline seconds, severity) and **Alerts** record each breach and who acknowledged or resolved it. **IndustrialConnector** and **ConnectorMapping** hold the connections to outside systems such as ChirpStack. Users are **PlantUser** (belongs to one factory) or **PlatformUser** (our staff).
- Readings are indexed by device + time and by factory + time, so a chart for one sensor over a week is quick.
- Retention: `IOT_READING_RETENTION_DAYS=0`, meaning readings are kept forever today (section 8 explains why that needs a policy).

3.12 Prisma and database migrations

In one line: Prisma is the translator between our code and the database; migrations are the numbered, step-

by-step changes to the database structure.

What it is

Prisma is a toolkit for Node.js. We describe the tables once, in `prisma/schema.prisma`, and Prisma generates safe code to read and write them, so developers write `prisma.reading.create(...)` instead of raw SQL.

A **migration** is a small SQL script that changes the structure (for example "add the columns for MFA"). Each one has a timestamped name and is applied exactly once, in order. The database records which ones have run, so every copy of the database ends up with the same structure.

Think of it like: A building's renovation log: every change to the walls is a numbered work order, done in order, and the log shows which ones are finished.

Why we use it

- Fewer mistakes: the code is checked against the real table structure.
- Database changes are reviewed in Git like any other code, and applied the same way on every server.
- Deploys are repeatable: `prisma migrate deploy` only applies what is new.

How it is set up in our system

- Schema file: `prisma/schema.prisma` (about 1,430 lines, around 60 tables).
- Migrations in `prisma/migrations/`, for example `20261008140000_tenant_user_mfa`, `20261015140000_positioning_phase1` and `20261015180000_industries_live`.
- On every deploy, the **migrate** container runs `npx prisma migrate deploy` before the app is allowed to start. If a migration fails, the new app does not start on a half-changed database.
- We always take a database backup before deploying (section 7).

3.13 The Next.js web app

In one line: The website that factory staff, managers and our ops team log in to.

What it is

Next.js is a popular framework for building websites with React (a user-interface library) and Node.js (JavaScript on the server). It builds pages on the server, so the browser gets ready-made pages quickly, and it also hosts the app's APIs (web addresses that programs call, such as `/api/ingest` or `/api/health`).

Think of it like: The shop front and the service counter: what customers see and use, while the warehouse work happens at the back (the worker).

Why we use it

- One codebase for the screens and the APIs.
- Fast pages, good on phones and slow connections.
- Large community and long-term support.

How it is set up in our system

- Container **app** (Next.js 15, React 19, Node.js 22), listening on port 3000 inside Docker only. Caddy sends web traffic to it.
- One app serves three hostnames: **iot.factoraerp.com** (factory users), **factory.factoraerp.com** (ops console) and **factoraacademy.factoraerp.com** (academy).

- Main areas: dashboards, devices, alerts, floor map, energy, OEE, maintenance, production, connectivity (connectors and gateways), audit log, settings and team, and the ops console (factories, devices, industries, partners, leads, academy).
- Health check: `/api/health`, checked every 15 seconds by Docker.
- Runs as an unprivileged user (uid 1000), not as root (section 4).

3.14 The worker (background jobs and the connector runtime)

In one line: A second copy of our code that never serves web pages; it runs continuously in the background to collect data and keep things tidy.

What it is

The worker is the same Factora image started with a different command (`scripts/worker.ts`) and the role `FACTORA_ROLE=worker`. It starts the background loops, the built-in MQTT broker and the **connector runtime**, then runs maintenance every 5 seconds.

Think of it like: The night-shift team in the warehouse: they receive deliveries around the clock, check them in, shelve them and ring the alarm if something is wrong, while the shop front deals with customers.

Why we use it

- Long-running jobs (holding MQTT connections open, polling equipment, scanning for offline devices) should not slow down web pages.
- If the website restarts, data collection keeps going, and the reverse.
- Health is easy to see: the worker writes a "heartbeat" time stamp only after a cycle that reached the database.

How it is set up in our system

- Container **worker**. It shares the app container's network (`network_mode: service:app`), so its built-in MQTT broker on 127.0.0.1:1883 is private to the app and worker.
- **Connector runtime** (`src/lib/connector-runtime.ts`): for each enabled connector it starts the right driver: **MQTT** (subscribe to topics, as in section 3.8), **gateway** (data pushed in by a gateway), or **polled** connectors (fetch a web address on a timer, with a host allowlist, an 8-second timeout, a 1 MB size limit and at most 1,000 readings per poll). Each connector runs on its own, so one failing connector does not stop the others. Status and errors are recorded, and secrets are hidden from logs.
- **Ingest pipeline:** every value, whatever its route, goes through the same steps: find the device, skip duplicates (same message ID or same time), save the Reading, update the device, check the alert rules. If the database is not reachable, readings are buffered in the `factora_edge` volume and replayed.
- **Every 5 seconds:** maintenance (offline checks, the simulator for demo factories, retention when enabled), data-staleness checks, then the heartbeat.
- Today 18 demo connectors exist (MQTT, OPC UA, Modbus), all disabled; no connector points at Mosquitto yet.

3.15 Caddy (the front door, or reverse proxy)

In one line: Caddy receives every web visit, makes sure it is encrypted, and passes it to the right container.

What it is

A **reverse proxy** sits in front of the applications. Every browser connects to it on the standard web ports (80 and 443). It reads the address the visitor asked for (for example iot.factoraerp.com), forwards the request to the right internal container, then sends the answer back. The containers themselves are never exposed.

Caddy is a modern web server and reverse proxy known for getting and renewing HTTPS certificates automatically.

Think of it like: A receptionist at the building entrance who checks visitors in, sends each to the right floor, and keeps a list of rooms that only staff from head office may enter.

Why we use it

- Automatic HTTPS for every address, with no manual certificate work.
- One place to add security headers, IP restrictions and compression for all sites.
- Apps stay hidden on internal networks.

How it is set up in our system

- Container **caddy** (caddy:2-alpine) in the factora-iot project, publishing ports **80** and **443**. Config file `/opt/factora-iot/deploy/Caddyfile` (part of the code repository, mounted read-only).
- Routes: iot.factoraerp.com and factoraacademy.factoraerp.com → Factora app; factory.factoraerp.com → Factora app **only from the office IP** (everyone else gets "403, not available from this network"); lns.factoraiot.com → ChirpStack admin, **only from the office IP**; plus the pricing, FarmMate and farm ERP addresses.
- Plain HTTP (port 80) is redirected to HTTPS. Visits to the bare server IP or unknown names are dropped, so the apps cannot be found by scanning the IP.
- Security headers on every site: HSTS (browsers must use HTTPS for a year), no-sniff, no framing (stops click-jacking), a strict referrer policy, and camera, microphone and location disabled. Server version headers are removed.
- A config change is applied with `caddy validate` then `caddy reload`, with no downtime and no rebuild.

3.16 HTTPS, TLS and certificates

In one line: The padlock in the browser: it encrypts traffic and proves the website is really ours.

What it is

TLS (Transport Layer Security) is the encryption used by HTTPS. When a browser or gateway connects, the server shows a **certificate**: a digital ID card saying "this key belongs to iot.factoraerp.com", signed by a trusted **certificate authority**. The two sides then agree on secret keys, and everything after that is encrypted.

Let's Encrypt is a free certificate authority. Its certificates last 90 days and are renewed automatically through the **ACME** protocol: Let's Encrypt checks that we control the domain (by asking for a special file over HTTP), then issues the certificate.

Think of it like: A sealed envelope with an official stamp: only the right recipient can open it, and the stamp proves who sent it.

Why we use it

- Passwords, sensor data and dashboards cannot be read or changed in transit.
- Browsers and gateways refuse to connect to an impostor.
- Free, and renewal is automatic, so certificates do not expire by accident.

How it is set up in our system

- Caddy obtains and renews certificates for all our web addresses and stores them in the `caddy_data` volume.

- The gateway bridge (port 3001) needs the **lns.factoraiot.com** certificate as files. A daily job at **06:17 EAT** (`/usr/local/bin/lorawan-cert-sync.sh`) copies Caddy's current certificate to `/opt/lorawan/certs/` (the private key with owner-only permissions) and restarts the bridge only if the certificate changed.
- Current lns.factoraiot.com certificate: issued by Let's Encrypt, valid until 13 Jan 2027 (it renews automatically about 30 days before).
- Inside the server, container-to-container traffic travels on private Docker networks and does not leave the machine.

3.17 DNS and Cloudflare

In one line: DNS is the internet's phone book: it turns a name like iot.factoraerp.com into the server's numeric address.

What it is

When someone types a web address, their device asks DNS servers for the matching IP address. The owner of a domain controls its records: an **A record** points a name to an IPv4 address, and the **nameservers** say which company hosts the records.

Cloudflare is a company that hosts DNS records (and offers a content network and website hosting). A record can be "proxied" (traffic passes through Cloudflare) or "DNS only" (Cloudflare just answers the lookup).

Think of it like: A phone book: people remember the name, the book gives the number, and if we move offices we only update the book.

Why we use it

- Friendly names that do not change even if we move servers: we only update the record.
- Cloudflare Pages hosts our marketing site cheaply and quickly, separate from the platform server.

How it is set up in our system

- **factoraerp.com**: DNS is managed by an external party. iot., factory., factoraacademy. and pricing. point to our droplet. Changes need a request to them.
- **factoraiot.com**: DNS is in our own Cloudflare account. The main site is on Cloudflare Pages. **lns.factoraiot.com** points to our droplet as **DNS only** (grey cloud), because Cloudflare's proxy would not carry the gateway port 3001. farm.factoraiot.com also points to the droplet.
- Moving the LoRaWAN server to another machine later only needs the lns record changed; gateways keep the same address.

3.18 Firewall and ports

In one line: The firewall is the gatekeeper that lets in only the four kinds of traffic we expect and blocks everything else.

What it is

A **port** is a numbered door on a server; each service listens on one (web on 443, SSH on 22 and so on). A **firewall** decides which doors can be reached from the internet. **ufw** ("uncomplicated firewall") is Ubuntu's standard firewall tool, and its default is to deny all incoming traffic.

Think of it like: A building with four guarded entrances and every other door bricked up.

Why we use it

- Fewer open doors means fewer ways in for an attacker.
- Databases, Redis, MQTT and admin APIs must never be reachable from the internet.

How it is set up in our system

- ufw is active. Allowed incoming (IPv4 and IPv6): **22** (SSH), **80** (HTTP, only to redirect to HTTPS and for certificate checks), **443** (HTTPS), **3001** (LoRa Basics Station gateways). Everything else is blocked.
- Docker note: Docker can open ports around ufw, so our rule is to publish only 80, 443 and 3001 from containers. Postgres (5432), Redis (6379), Mosquitto (1883), ChirpStack (8080) and the app (3000) are **not** published; they live only on internal Docker networks. The farm ERP's port 8080 listens on the server's own loopback address only.
- Port 1700/UDP (the old, unencrypted gateway protocol) and 8883 (MQTT over TLS for outside clients) are deliberately closed.

Port	Open to internet?	Used by
22/tcp	Yes (key login only)	SSH administration
80/tcp	Yes	Caddy: redirect to HTTPS, certificate checks
443/tcp	Yes	Caddy: all websites
3001/tcp	Yes	Gateway bridge (Basics Station, TLS)
3000, 8080	No (Docker internal)	Factora app, ChirpStack UI/API
1883	No (Docker internal / loopback)	Mosquitto, built-in Factora broker
5432, 6379	No (Docker internal)	Postgres databases, Redis

3.19 The EU868 radio band and Communications Authority type approval

In one line: LoRaWAN uses licence-free radio frequencies, but there are rules on which frequencies, how much power and how much airtime, and in Kenya the equipment must be type-approved.

What it is

Radio spectrum is regulated. LoRaWAN uses "short-range device" bands that need no individual licence. Each region has a LoRaWAN regional plan; Kenya uses **EU863-870 ("EU868")**, with three default channels at 868.1, 868.3 and 868.5 MHz, extra channels from 867.1 to 867.9 MHz, and a receive window at 869.525 MHz for replies. The rules (based on the European standard ETSI EN 300 220) limit transmit power (typically 25 mW) and **duty cycle** (typically a device may transmit at most about 1% of the time on a sub-band).

Type approval is the Communications Authority of Kenya (CA) certifying that a specific radio model meets Kenyan technical standards. Radio equipment that is not type-approved should not be imported, sold or used.

Think of it like: Public roads: anyone may drive without a special permit, but only on the right side, within speed limits, and in a roadworthy, inspected vehicle.

Why we use it

- Using the right band keeps our sensors and gateways legal and stops them interfering with other users.
- Gateways and sensors made for another region (for example US915 or AS923) would not work with our network, or would be illegal to use here.
- Type approval protects the client and us during audits and customs clearance.

How it is set up in our system

- ChirpStack has only `eu868` enabled; the gateway bridge is set to region EU868, 863 to 870 MHz.
- The 8 standard EU868 channels are configured, with spreading factors SF7 to SF12 and ADR, which keeps airtime (and duty cycle) low.

- **Hardware rule:** every gateway and sensor model we buy must be EU868 and carry CA type approval. This is a hard requirement in the hardware evaluation (section 9).

4. Security

Security is built in layers, so if one layer fails the next still protects the data. Each layer below follows the same pattern: what it is, why, and how it is set up.

4.1 Multi-factor authentication (MFA)

What it is: Logging in needs two things: something you know (a password) and something you have (a phone with an authenticator app that shows a new 6-digit code every 30 seconds). This is the **TOTP** standard (time-based one-time password): the phone and our server share a secret and both calculate the same code from the current time.

Why: A stolen password alone is not enough to get in.

How it is set up:

- The ops console requires 2-factor login (`OPS_REQUIRE_2FA=1`).
- Each factory has its own switch (`requireMfa`) that forces all its users to enrol an authenticator; a platform-wide switch can force it for every factory.
- Authenticator secrets are stored encrypted (AES-GCM). One-time **recovery codes** are shown once and stored only as hashes. A code cannot be reused within its time window.
- Passwords are stored as **scrypt** hashes (a deliberately slow one-way scramble), never in readable form. Changing a password or resetting MFA logs out all older sessions.

4.2 Roles (who can do what)

What it is: Each user has a role that decides which screens and actions they get.

Why: People see and change only what their job needs, which limits mistakes and damage.

How it is set up:

- Factory users (PlantUser) belong to exactly one factory, with roles **admin** (full control of their factory: team, devices, settings), **maintenance** (work orders, equipment, alerts) and **operator** (day-to-day views and actions). Today there are 9 admins, 7 maintenance users and 8 operators across the demo factories.
- Platform users (PlatformUser) are our staff, with roles **owner** (super admin across all factories) and **viewer** (read-only). They use the ops console.
- Important actions are written to an audit log.

4.3 Office IP allowlist

What it is: An allowlist is a list of internet addresses that may reach a site; everyone else is refused.

Why: Admin screens are the most valuable target. If only our office can even see the login page, an attacker elsewhere cannot start guessing passwords.

How it is set up:

- The ops console (factory.factoraerp.com) and the ChirpStack admin (lns.factoraiot.com) only answer requests from the **office IP**. Everyone else gets a "not available from this network" page.
- The ops console is checked **twice**: by Caddy and again inside the app.
- The two allowlists are separate settings, so one can change without the other. The lns allowlist does not affect gateways, which use port 3001.

4.4 Encryption in transit (TLS) and on the radio

What it is: Data is encrypted whenever it travels over a network we do not control.

Why: Nobody between the sensor and the screen can read or change the data.

How it is set up:

- Radio: AES-128 encryption and message signing between sensor and ChirpStack (section 3.2).
- Gateway to server: TLS on port 3001 (section 3.4). Browsers to server: HTTPS with HSTS (section 3.16).

- The code refuses unencrypted connections to any external MQTT broker in production.

4.5 Secrets kept in .env files

What it is: A **secret** is a password, key or token. We keep them out of the code, in a file called `.env` on the server that Docker reads when it starts the containers.

Why: If the code is shared or copied (for example on GitHub), no secrets go with it.

How it is set up:

- `/opt/factora-iot/.env` and `/opt/lorawan/.env` are readable only by root (permissions 600).
- They are excluded from Git, from deploy copies and from the off-server backups.
- Connector passwords and certificates (for example a factory's MQTT login) are stored **encrypted** in the database and hidden from logs.
- Secrets are never pasted in chat or email; they are entered through a secure form.
- Gap: because .env files are not in the backups, a secure offline copy of these secrets must be kept (section 9).

4.6 Non-root containers and least privilege

What it is: "Root" is the all-powerful administrator account. Programs should run with the fewest powers they need.

Why: If an attacker ever found a bug in the app, they would land as a limited user, not as an administrator.

How it is set up:

- The app and worker start as root only long enough to fix folder ownership, then switch permanently to the unprivileged `node` user (uid 1000).
- `no-new-privileges` stops them regaining powers, and all Linux "capabilities" are dropped except a small set needed for that first step.
- LoRaWAN containers have memory and CPU caps, so a runaway process cannot starve Factora IoT.

4.7 SSH keys for server access

What it is: SSH is the encrypted remote-control connection to the server. Instead of a password we use a **key pair**: a private key that stays on the administrator's computer and a public key on the server.

Why: Keys cannot be guessed like passwords, and nothing secret travels over the network.

How it is set up:

- Password login over SSH is disabled; only key login is allowed, including for root.
- Our deploy key is kept on the operator machine, never in the code or in chat.

4.8 Other built-in protections

What it is: Small safety rules in the code.

Why: They block common attacks.

How it is set up:

- Devices authenticate with their own ingest token; on the built-in broker they can only publish to their own topic, with a 16 KB limit.
- Polling connectors may only call approved hosts (this stops the server being tricked into calling internal addresses).
- Each page gets its own content security policy, and security headers are set on every site (section 3.15).

5. Multi-tenancy: keeping factories apart

In one line: Many factories share one platform and one server, but each factory can only ever see its own gateways, sensors, readings, alerts and users.

What it is: a "tenant" is one customer's private space inside a shared system. Like flats in one apartment block: shared walls, roof and water pipes, but each flat has its own lock.

Why: one shared system is far cheaper to run and update than one server per factory, as long as the separation is strict.

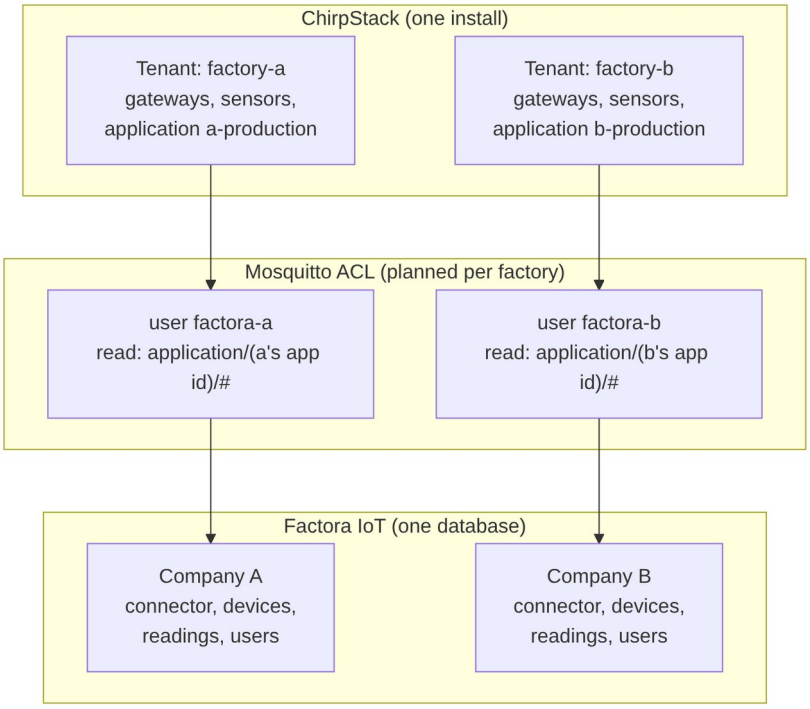

Figure 5. One tenant per factory, all the way through: ChirpStack tenant → Mosquitto user → Factora company.

How it is set up, layer by layer

Layer	How a factory is kept separate	Status
ChirpStack	One **tenant per factory** (for example `factory-<slug>`), with limits (planned: up to 3 gateways and 200 devices), and one application per factory (`<slug>-production`). Each factory's gateways and sensors live only in its tenant. A tenant-scoped API key is used for the Factora integration.	Planned; only the default admin tenant exists today
Mosquitto	One MQTT user per factory (`factora-<slug>`) whose ACL allows **reading only** `application/<that factory's app id>/#`. It cannot read another factory's topics or publish anything. The system users (`chirpstack`, `gateway-bridge`) are separate.	Planned; the ACL has the two system users today
Factora worker	One MQTT **connector per factory**, owned by that factory's company record and logging in with that factory's MQTT user. Mappings link that factory's sensors to that factory's devices only.	Code ready; first connector is the Day 3 task
Factora database	Every device, reading, alert, rule and user row carries a `companyId`. Queries are filtered by the logged-in user's company. Device codes and usernames are unique per company. Deleting a company deletes its data.	Live
Users	Factory users belong to one company only. Only our platform staff (ops console, office IP, MFA) can see across factories.	Live

The ops console (super admin)

factory.factoraerp.com is our control room. From there our team (platform owners) can create and onboard factories, manage devices, industries, partners, leads and academy content, and enter a factory's view to help them. It is reachable only from the office IP and only with MFA, and platform staff accounts are stored separately from factory accounts.

Onboarding a new factory (checklist)

5. Create the company in the ops console (name, short name, timezone, currency, MFA policy).
6. In ChirpStack: create the tenant `factory-<slug>` with limits and its application, then register its gateways (EUI and token or certificate) and sensors (DevEUI, JoinEUI, AppKey; CSV import for many).
7. Assign shared device profiles and codecs named by measurement (temperature/humidity, vibration, energy meter, level), not by brand.
8. In Mosquitto: add the user `factora-<slug>` with its password and the read-only ACL line, then reload Mosquitto.
9. In Factora IoT: create the MQTT connector for that company (host `mqtt://mosquitto:1883`, its username, topic `application/<appId>/device/+/event/up`) and map each sensor field to a Factora device.
10. Watch the first readings arrive, then hand over to the factory admin.

About half a day per factory once hardware is on site. Automating steps 2 to 5 is on the to-do list (section 9).

6. Backups and recovery

In one line: Every night the databases are copied to files on the server and to a separate storage bucket, so we can recover from a mistake or even from losing the whole server.

What it is: `pg_dump` writes a complete, consistent copy of a Postgres database into one file. `rclone` is a tool that copies files to cloud storage. **DigitalOcean Spaces** is DigitalOcean's file storage (compatible with Amazon S3), separate from our droplet.

Why: a copy on the same disk protects against mistakes (for example a bad deploy), but not against losing the server. The off-server copy covers that.

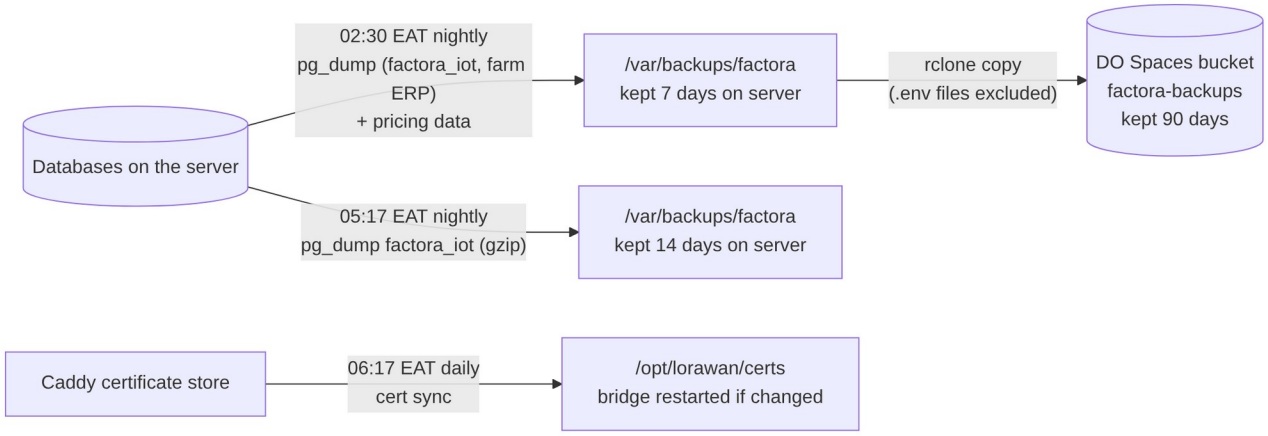

Figure 6. The nightly backup schedule (times in EAT).

How it is set up

Job	Time (EAT)	What it does	Kept for
Off-server backup (`factora-offsite-backup.sh`)	02:30 nightly	Dumps the factora_iot and farm ERP databases and the pricing site data, then copies them (and any pre-change backups) to the Spaces bucket **factora-backups** (fra1). `.env` files are excluded.	90 days in Spaces; 7 days on the server
Local Factora backup (`factora-db-backup.sh`)	05:17 nightly	Compressed SQL dump of the factora_iot database to `/var/backups/factora`	14 days on the server
Local farm ERP backup	05:27 nightly	The same for the farm ERP database	Per its own script
Certificate sync (`lorawan-cert-sync.sh`)	06:17 daily	Copies the renewed lns.factoraiot.com certificate to the gateway bridge (not a backup, but part of the nightly jobs)	n/a
Pre-change backups	Before each deploy or server change	Manual dump named `pre-...`, for example before the LoRaWAN install on 15 Oct 2026	Uploaded to Spaces by the next nightly run

The off-server job was set up and run once by hand on 16 Oct 2026 at 00:29 EAT, successfully. Its first scheduled run is 02:30 EAT the same night. The local backup folder currently holds about 7.7 GB.

How we would recover

What went wrong	How we recover	Data we could lose
A bad code deploy	Redeploy the previous Git snapshot (section 7). Restore the pre-deploy dump only if a migration damaged data.	None, or changes since the pre-deploy dump
Data deleted or corrupted by mistake	Restore the most recent dump into a temporary database (the `deploy/restore-drill.sh` script does this safely), check it, then copy back the affected data.	Up to one day
The whole server is lost	Create a new droplet, install Docker, copy the code, recreate the `.env` files from the secure offline copy, restore the databases from Spaces, and point DNS at the new server. Gateways reconnect by themselves	Up to one day; a few hours of downtime

	because they use the name lns.factoraiot.com.	
ChirpStack data lost	Restore its database dump. **Gap today:** ChirpStack's database is not yet in the nightly job (section 9). Its configuration files are in /opt/lorawan.	Gateway and device registrations, until fixed

Good practice: run a restore test at least once a month and after any backup change, so we know the backups actually work.

7. How we deploy changes

In one line: We take an exact snapshot of the tested code from GitHub, copy it to the server, rebuild the containers and let the migrate step update the database before the new version goes live.

What it is: "deploying" means putting a new version of our software live. **Git** is the system that records every code change; **GitHub** hosts our repository **FactoraERP/factora-iot**. The version that is live today is the branch **deployed-61df7be** (61df7be is the short ID of the exact commit).

Why this way: every deploy is built from one known commit, so we always know exactly which code is running, and we can go back to the previous one.

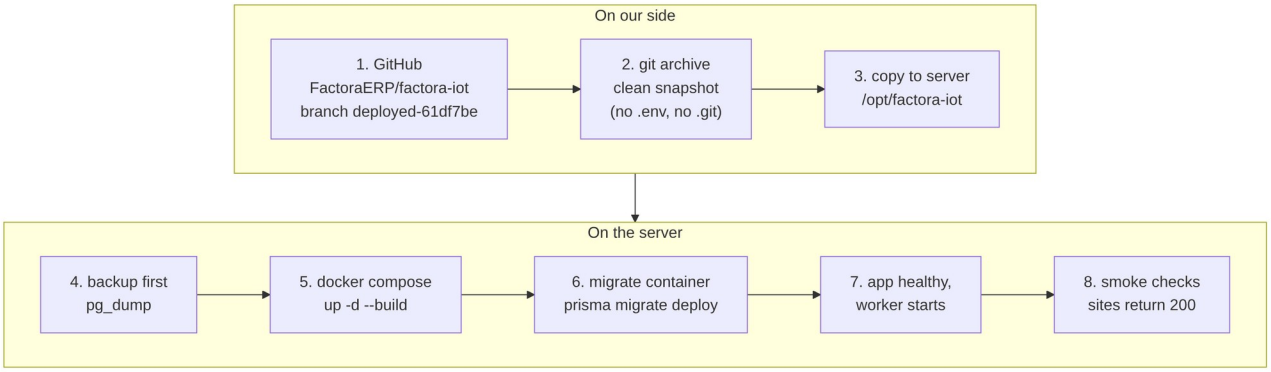

Figure 7. Deployment steps.

Step by step

11. **Snapshot:** `git archive` makes a clean copy of the chosen commit (no Git history, no `.env`, no local data).
12. **Copy** it to `/opt/factora-iot` on the server over SSH. The server's `.env` file is left untouched.
13. **Backup first:** take a `pg_dump` of the factora_iot database (named `pre-...`).
14. **Build and start:** `docker compose -f docker-compose.prod.yml up -d --build`. Docker builds a new image in stages (install packages, generate the Prisma code, build Next.js) and replaces only the containers that changed.
15. **Migrate:** the `migrate` container runs `prisma migrate deploy` and must finish successfully before the app starts.
16. **Start:** the app starts and must pass its health check; then the worker starts. Caddy keeps serving throughout.
17. **Check:** all sites return 200 (iot, factory, academy, pricing, farm) and `docker ps` shows every container healthy. Record the deployed commit as a `deployed-<commit>` branch.

Other kinds of change

- **Caddy rules** (`deploy/Caddyfile`): edit, then `caddy validate` and `caddy reload`. No rebuild and no downtime. The change is also committed to the repository so the next deploy does not overwrite it.
- **LoRaWAN stack** (`/opt/lorawan`): its own compose project with pinned image versions (ChirpStack 4.19.2 and so on). Upgrade by changing the version, backing up the ChirpStack database, then `docker compose up -d`. It never restarts Factora IoT.
- **Rollback:** redeploy the previous `deployed-...` commit the same way. For LoRaWAN, pin the previous image version and restore its database dump if needed.
- Changes to the live server are made only with the owner's approval, in a planned window.

8. Capacity and when to scale

In one line: Today's server can comfortably carry up to about 5 factories. Beyond that, or if the numbers below are crossed, we move the LoRaWAN server and later the database to their own machines.

How busy is the server today?

Resource	Total	In use (16 Oct 2026)	Comment
CPU	4 vCPU	Load average about 0.5	Roughly 10 to 15% busy; lots of headroom
Memory	8 GB (7.8 GiB)	About 2 GB used, 5.7 GB available	The LoRaWAN stack is capped at about 1.1 GB
Disk	160 GB (154 GB usable)	21 GB used (14%)	Includes about 7.7 GB of local backups
Factora database		1.48 GB, 2.87 M readings, 57 devices	About 0.5 KB per reading including indexes

What 5 factories would add

Assumption: 5 factories × about 150 sensors = 750 sensors, each reporting every 10 minutes, with 2 gateways per factory.

- Messages: 750 × 6 per hour ≈ 1.25 messages per second on average, with peaks around 10 per second. ChirpStack, Mosquitto and the worker handle this with very little CPU.
- Readings: 750 × 144 per day ≈ **108,000 rows a day**, about 39 million a year. At about 0.5 KB per row that is roughly **20 GB of database growth per year** if every reading is kept forever (more if one sensor reports several values).
- LoRaWAN stack: about 250 to 500 MB of actual memory use, within its 1.1 GB cap.
- Note: the demo simulator alone writes about 280,000 readings a day today. A retention rule for demo factories would keep the database lean.

Signals that it is time to scale

Trigger	What to do
More than 5 factories, or more than 2,000 LoRaWAN devices	Move the LoRaWAN stack to its own droplet (about US$24 a month plus backups). Change only the lns DNS record; gateways follow automatically
Available memory below 1.5 GB for days, or load average above 3	Upgrade the droplet (for example 16 GB / 8 vCPU) or move a neighbour (FarmMate, pricing) off this server
Disk above 70%, or the readings table slowing charts down	Set `IOT_READING_RETENTION_DAYS`, add hourly and daily summaries (roll-ups), or move the database to DigitalOcean Managed PostgreSQL
Customers need guaranteed uptime (an SLA)	Separate servers per role, a managed database with a standby, monitoring and alerting, and later a 2-node ChirpStack cluster

Why move the LoRaWAN server first: once several paying factories depend on it, one shared server is a single point of failure for all of them, and the LoRaWAN stack is the easiest piece to lift out.

9. What is not done yet

This is the honest to-do list between "installed" and "first real factory live".

#	Item	Why it matters	Next step
1	**Hardware not chosen**	No gateway or sensor model has been selected, and everything else depends on it.	Run the vendor evaluation: EU868, CA type approval, Basics Station support, Ethernet + 4G (dual SIM), industrial rating, UPS; get quotations
2	**Day 3: first-factory integration**	Steps 7 → 8 of the journey are not yet connected.	Set up ChirpStack (device profiles and codecs), connect a test gateway or simulator, create the first MQTT connector in Factora IoT and confirm a reading appears within 5 seconds
3	**Connector per factory**	Each factory needs a ChirpStack tenant, an application, a Mosquitto user with ACL, and a Factora connector with mappings. Today this is manual.	Do it by checklist for the pilot (section 5); later build a "lorawan" connector type that maps sensors by DevEUI, tracks gateway health and creates tenants and devices through the ChirpStack API (about 2 to 3 developer-weeks)
4	ChirpStack database not in nightly backups	Gateway and device registrations could be lost.	Add `lorawan-postgres-1` to the off-server backup script
5	Secure offline copy of secrets	.env files are deliberately not backed up, so losing the server would also lose them.	Keep them in a password manager or secure vault
6	Monitoring and alerting	We should hear about a problem before a customer does.	External uptime checks (web and port 3001), host alerts (memory, disk, unhealthy containers), "gateway silent for 10 minutes" alerts
7	Alert delivery to people	Alerts are created in the app, but SMS and email delivery to staff has not been proven live.	Approve senders and test end to end
8	Per-gateway certificates (mTLS)	Stronger gateway identity than a token.	Enable from the first production gateway
9	Reading retention policy	Readings are kept forever today; growth is about 20 GB a year at 5 factories.	Agree how long raw readings are kept and add roll-ups
10	TLS for outside MQTT devices	The built-in broker is internal only; direct IP devices would need encrypted MQTT.	Only if a factory needs it
11	Restore test schedule	A backup is only proven by a restore.	Monthly restore drill

10. Glossary

Term	Meaning
ACL (Access Control List)	A list of who may do what. In MQTT: which user may read or write which topics.
ACME	The protocol used to obtain and renew certificates automatically from Let's Encrypt.
ADR (Adaptive Data Rate)	LoRaWAN feature where the network tells each sensor the best speed and power to use.
AES-128	A strong, standard encryption method with a 128-bit key; used by LoRaWAN.
API	A set of web addresses that programs (not people) call to exchange data.
AppKey	A sensor's secret root key, used only to create session keys when it joins.
Backhaul	The gateway's own internet connection (Ethernet or 4G).
Basics Station	Modern gateway software and protocol: a TLS-encrypted WebSocket from gateway to network server.
Broker (MQTT)	The central MQTT server that receives messages and passes them to subscribers.
CA (Communications Authority of Kenya)	Kenya's communications regulator; it type-approves radio equipment.
Caddy	Our web server and reverse proxy; it handles HTTPS certificates automatically.
Certificate (TLS)	A digital ID card proving a server's identity, signed by a trusted authority.
ChirpStack	The open-source LoRaWAN network server we run.
Codec	A small program that turns a sensor's raw bytes into readable values.
Compose project	A group of containers described in one Docker Compose file.
Connector	A Factora setting plus code that pulls data from an outside system (MQTT, OPC UA, Modbus, web polling).
Container	A running, isolated package of a program and everything it needs.
Cron	The Linux scheduler that runs jobs at set times (our backups).
DevEUI	The unique, permanent ID of a LoRaWAN sensor.
DNS	The internet's phone book: it turns names into IP addresses.
Docker	The tool that builds images and runs containers.
Downlink / Uplink	A message to a sensor / a message from a sensor.
Droplet	DigitalOcean's name for a rented virtual server.
Duty cycle	The share of time a radio may transmit (about 1% in EU868).
EAT	East Africa Time, UTC+3.
.env file	A protected file on the server that holds the passwords and keys for the containers.
EU868	The 863 to 870 MHz LoRaWAN regional plan used in Kenya and Europe.
Gateway	The radio receiver at the factory that relays sensor messages to our server.
Health check	An automatic test Docker runs to see if a container is working.
HSTS	A header telling browsers to always use HTTPS for our sites.
Image (Docker)	The frozen package a container is started from.
Ingest	Taking in a reading and storing it.
Ingest token	A device's secret, used to prove it may send readings.
IP allowlist	A list of internet addresses allowed to reach a site (ours: the office IP).
JoinEUI	An ID used when a sensor joins the network.
LNS	LoRaWAN Network Server (ChirpStack in our case).
LoRa / LoRaWAN	The long-range, low-power radio technique / the network rules built on it.
MFA / 2FA / TOTP	Logging in with a password plus a time-based code from an authenticator app.
Migration	A numbered change to the database structure, applied once, in order.
Mosquitto	The open-source MQTT broker in our LoRaWAN stack.
MQTT	A lightweight publish/subscribe messaging protocol for machines.
mTLS	Mutual TLS: both sides present certificates, so the server also verifies the gateway.
Multi-tenancy	Many customers sharing one system with their data kept strictly apart.
Next.js	The web framework our app is built with.
OTAA	Over-the-air activation: the secure way a sensor joins and gets its session keys.
pg_dump	The PostgreSQL tool that writes a full database copy to a file.
Port	A numbered door on a server for one kind of service (443 = HTTPS).
PostgreSQL	Our open-source database.
Prisma	The toolkit our code uses to talk to the database and run migrations.
Publish / Subscribe	Sending a message to a topic / asking to receive a topic's messages.
QoS	MQTT delivery level: 0 at most once, 1 at least once, 2 exactly once.
rclone	A tool that copies files to cloud storage (our off-server backups).

Redis	A very fast in-memory data store used by ChirpStack.
Reverse proxy	A front server that receives all web traffic and forwards it to the right app.
RSSI / SNR	Radio signal strength / signal-to-noise ratio: measures of link quality.
Spaces	DigitalOcean's cloud file storage; it holds our off-server backups.
Spreading factor (SF7 to SF12)	LoRa speed setting: higher means longer range but slower, with more airtime.
SSH / SSH key	Encrypted remote access to the server / the key pair used instead of a password.
Tenant	One customer's private space in a shared system (one factory).
TLS / HTTPS	The standard encryption for internet connections / the web over TLS.
Topic (MQTT)	The address of an MQTT message, such as application/7/device/a1b2/event/up.
Type approval	Regulator certification that a radio model meets national standards.
ufw	Ubuntu's firewall tool.
Volume (Docker)	Storage that survives when a container is replaced.
WebSocket	A long-lived, two-way connection over web technology.
Worker	Our background program that collects data, checks alerts and runs maintenance.

Appendix A. Where things live on the server

Path or name	What is there
/opt/factora-iot	Factora IoT code: `docker-compose.prod.yml`, `Dockerfile`, `prisma/`, `src/`, `scripts/worker.ts`
/opt/factora-iot/deploy/Caddyfile	Caddy rules for all web addresses
/opt/factora-iot/.env	Factora secrets (root-only; never shared)
/opt/lorawan	LoRaWAN compose project: `docker-compose.yml`, `configuration/` (chirpstack, gateway-bridge, mosquitto), `certs/`, `initdb/`, `.env`
/var/backups/factora	Local backup files
/usr/local/bin/factora-offsite-backup.sh	Nightly off-server backup (02:30 EAT)
/usr/local/bin/factora-db-backup.sh	Nightly local Factora backup (05:17 EAT)
/usr/local/bin/lorawan-cert-sync.sh	Daily certificate copy for the gateway bridge (06:17 EAT)
/var/log/factora-offsite.log and similar	Logs of those jobs
src/lib/connector-runtime.ts	Connector runtime (MQTT, gateway and polled connectors)
src/lib/mqtt.ts	Built-in MQTT broker and device topic rules
src/lib/ingest.ts, src/lib/alerts.ts	Saving readings, de-duplication, alert checks

Appendix B. Key facts at a glance

Item	Value
Server	DigitalOcean droplet, fra1, 4 vCPU / 8 GB / 160 GB, Ubuntu 24.04.5, Docker 29.8
Factora IoT	Next.js 15 + React 19 on Node.js 22, Prisma 6, PostgreSQL 16; app + worker + migrate + db + caddy
LoRaWAN	ChirpStack 4.19.2, Gateway Bridge 4.1.2 (Basics Station, port 3001 TLS), Mosquitto 2.0.20, PostgreSQL 16, Redis 7, EU868
Open ports	22, 80, 443, 3001
Web addresses	iot.factoraerp.com, factory.factoraerp.com (office IP), lns.factoraiot.com (office IP), factoraiot.com (Cloudflare Pages)
Database	About 1.48 GB, 2.87 M readings, 57 devices, 7 demo companies
Backups	Off-server 02:30 EAT to Spaces bucket factora-backups, 90 days; local 05:17 EAT, 14 days
Code	GitHub FactoraERP/factora-iot, live branch deployed-61df7be
Capacity	Up to about 5 factories on this server