Well Meter Specification

WELL_METER — Indo-Gangetic Basin Well Metering System

> **Super-cheap, accurate, long-lasting IoT well metering for the Government of India**
>
> Modeled on the USGS / Tucson Water / ADWR groundwater monitoring framework. Applied at planetary scale: 30+ million tube wells across the Indo-Gangetic Basin.

Table of Contents

---

1. The Problem: Unmetered Extraction

From `IGBWP.md` — the single most important data gap:

> *"The depletion rate uncertainty is the largest source of project sizing error. A national tube well metering program is the single highest-value data investment before committing to infrastructure scale."*

Current state: - ~30 million tube wells across UP, Bihar, Punjab, Haryana, West Bengal - **Zero** are metered for volume extraction - CGWB monitors ~25,000 observation wells — static water level only, no flow - Published depletion estimates vary **5–20×** because no one knows actual pumping volumes - Political incentives actively suppress accurate reporting

What metering unlocks: - True net overdraft rate (currently 5–20 km^3/year range → need ±10%) - Spatial depletion maps at 1 km^2 resolution (currently basin-average only) - Real-time aquifer response to IGBWP recharge injection - Legal basis for extraction regulation and water pricing - Proof of IGBWP impact — the feedback loop that justifies continued investment

---

2. Inspiration: Tucson Water / USGS Model

The USGS Arizona Water Science Center, in cooperation with **Tucson Water**, has monitored water levels and aquifer compaction at wells in Avra Valley and Tucson Basin since ~1980.

What Tucson Does Right

- **Continuous automated monitoring** — pressure transducers log every 15–60 min, 24/7
- **Public data portal** — all well data online via USGS National Groundwater Monitoring Network
- **Third-party submission** — ADWR Third-Party Water Level Portal lets private owners submit data
- **USGS accuracy standards** — ±0.01 ft (±3 mm) water level, ±0.1°C temperature
- **Long deployment** — sensors run 5–10 years on a single battery pack

What India Needs Differently

Tucson Model	India Adaptation
~500 monitored wells	**30 million** tube wells
$500–2,000 per sensor	**<$15 per sensor** — 100× cheaper
Cellular/landline telemetry	LoRaWAN mesh + satellite fallback

Tucson Model	India Adaptation
English-language portal	Hindi + regional language portal
Voluntary compliance	Mandatory registration + tamper detection
USGS staff servicing	Self-service + village-level technician network

The Tucson model proves the concept. India deployment requires ground-up redesign for cost and scale.

---

3. Design Principles: Cheap, Accurate, Long-Lasting

The Three Constraints — In Order

1. Cheap — The binding constraint. At 30M wells, every $1 of unit cost = $30M total. Target: **<$15/unit installed**.

2. Long-lasting — Servicing is the hidden cost killer. A $10 sensor needing annual replacement costs more than a $50 sensor lasting 10 years. Target: **10-year deployment, zero scheduled maintenance**.

3. Accurate — Sufficient for policy: ±5 cm water level, ±1 L/min flow. Not USGS lab precision — policy precision.

Design Decisions

Decision	Rationale
Pressure transducer, not ultrasonic	Ultrasonic needs clear air column — tube wells are cased. Pressure is submersible, no moving parts, 10+ year life
No display, no buttons	Every UI element is a failure point and cost. Config via NFC tap from phone
LoRaWAN, not cellular	Cellular SIM = $2–5/month/unit = impossible at 30M scale. LoRaWAN gateway covers 15 km, amortized <$0.10/unit/year

Decision	Rationale
Lithium primary cell, not rechargeable	Rechargeable requires maintenance. Lithium AA lasts 10 years at 1 reading/15 min
Rust firmware on RISC-V	Bare-metal Rust: no OS overhead, no memory leaks, no runtime crashes. 10-year uptime without reboot
Tamper-evident seal + crypto signing	Each reading signed with device key. Tampered/replaced units detected at backend

4. Hardware Stack

Sensor Unit — Target BOM <$12

Component	Part	Cost (qty 1M)	Notes
MCU	ESP32-C3 (RISC-V, 160 MHz)	~$0.80	Wi-Fi + BLE for config; LoRa via SPI
LoRa radio	LLCC68 or SX1262	~$1.20	IN865 band, 15 km range
Pressure transducer	Ceramic capacitive, 0–10 bar	~$2.50	Submersible, no moving parts, ±0.1% FS
Flow sensor	Paddle-wheel magnetic, DN25–DN50	~$1.80	Fits standard tube well pipe sizes
Battery	2× Lithium AA (3.6V, 2400 mAh)	~$1.40	10-year life at 1 reading/15 min
PCB + enclosure	IP68 rated, UV-stabilized ABS	~$1.80	Submersible to 100 m

Component	Part	Cost (qty 1M)	Notes
NFC tag	NTAG215	~$0.15	Config + device ID
Tamper seal	Epoxy + optical fiber loop	~$0.30	Detects physical opening
Assembly + test	India manufacturing	~$1.50	Local assembly = lower cost + jobs
Total BOM		**~$11.45**	

Gateway Unit — 1 per ~15 km radius

Component	Cost	Notes
LoRaWAN gateway (8-channel)	~$80	RAK7268 or equivalent
Solar panel + battery	~$40	10W panel, 20 Ah LiFePO4
Enclosure + mounting	~$20	Pole-mount, IP65
SIM (4G data-only)	~$3/month	Gateway → cloud; 1 SIM per ~50 wells
Total gateway	**~$140 + $36/year**	Amortized: ~$0.10/well/year

Observation Well Unit (CGWB-grade, no flow meter)

For the ~25,000 CGWB observation wells — higher accuracy, no flow:

Component	Cost	Notes
Vented pressure transducer	~$45	±1 mm accuracy, desiccant vent tube
Data logger (12-bit, 1 Hz)	~$25	SD card + LoRaWAN

Component	Cost	Notes
Solar + battery	~$30	
Total	**~$100**	USGS-grade at 1/10th cost

5. Sensor Physics & Accuracy

Water Level via Pressure Transducer

```
P = ρ × g × d
d = P / (ρ × g)

where:
  P = gauge pressure (Pa)
  ρ = water density = 1000 kg/m³
  g = 9.81 m/s²

Example: P = 49,050 Pa → d = 49,050 / 9810 = 5.0 m
```

Accuracy at 0–50 m depth (0–5 bar): - Full-scale: ±0.1% FS = ±0.1 bar = ±1.0 m at 10 bar range - **Practical at 0–50 m: ±5 cm** (sufficient for trend monitoring) - Temperature compensation: ±0.02%/°C, corrected in firmware - Long-term drift: <0.1% FS/year → <0.5 m over 10 years (acceptable)

Flow Rate via Paddle-Wheel / Magnetic

```
Q = k × f × A

where:
  Q = volumetric flow rate (m³/s)
  k = calibration constant
  f = pulse frequency (Hz)
  A = pipe cross-section (m²)

DN40 pipe: A = π × (0.02)² = 0.001257 m²
At 2 m/s: Q = 0.001257 × 2 = 2.51 L/s = 151 L/min
```

Accuracy: ±2% of reading above 0.3 m/s. Below 0.3 m/s: pump not running — valid zero.

Cumulative Volume — The Key Metric

```
V_total = Σ (Q_i × Δt_i)   [m³]

Every 15 minutes (Δt = 900 s):
  V_reading = Q × 900

Annual per well:
  V_annual = Σ V_reading over 35,040 readings/year
```

This is the number India does not have for any well. This closes the IGBWP data gap.

---

6. Network Architecture

```
[Well Sensor] ──LoRa 865MHz──► [Village Gateway] ──4G──► [State Server]
[Well Sensor] ──LoRa 865MHz──► [Village Gateway]                │
[Well Sensor] ──LoRa 865MHz──► [Village Gateway]                ▼
  ...up to 2000 per gateway                          [National CGWB API]
                                                                │
[Observation Well] ──LoRa──► [Gateway]                          ▼
                                                     [Public Data Portal]
                                                     igbwp.gov.in/wells
                                                                │
                                                                ▼
                                                     [IGBWP Recharge Model]
                                                     (feeds pipeline control)
```

LoRaWAN Protocol

Parameter	Value	Notes
Frequency	865–867 MHz	India IN865 band
Spreading factor	SF9–SF12	SF12 = 15 km range, 250 bps
Payload size	11 bytes	Single LoRa frame
TX interval	Every 15 min	96 transmissions/day
Battery impact	~0.3 mAh/TX	2400 mAh / (96 × 0.3) ≈ 10 years
Gateway capacity	2,000 devices	8-channel, duty-cycle compliant

11-Byte Payload Format

```
Byte 0:    Device type flags (tube well / observation / flow-only)
Bytes 1-3: Water level in mm (24-bit unsigned, max 16,777 m)
Bytes 4-6: Cumulative volume in 10L units (24-bit, resets monthly)
Bytes 7-8: Flow rate in L/min (16-bit unsigned, max 65,535)
Byte 9:    Battery voltage × 10 (e.g., 36 = 3.6V)
Byte 10:   Status flags (tamper | sensor_fault | low_battery | pump_running)
```

HMAC-SHA256 truncated to 4 bytes appended → 15 bytes total, fits LoRa MTU.

---

7. Rust Firmware Architecture

Bare-metal Rust on ESP32-C3 (RISC-V). No RTOS, no heap allocation, no panics in production.

Firmware Cargo.toml

```toml
[package]
name = "well-meter-firmware"
version = "0.1.0"
edition = "2021"

[dependencies]
esp-hal = { version = "0.18", features = ["esp32c3"] }
esp-backtrace = { version = "0.12", features = ["esp32c3", "halt"] }
lora-phy = { version = "2", features = ["sx1262"] }
hmac = "0.12"
sha2 = { version = "0.10", default-features = false }
fixed = "1.23"

[profile.release]
opt-level = "s"
lto = true
codegen-units = 1
panic = "abort"
```

Module Structure

```
well-meter-firmware/
├── src/
│   ├── main.rs          # Entry point, sleep/wake loop
│   ├── sensor/
│   │   ├── mod.rs       # Sensor trait definitions
│   │   ├── pressure.rs  # Pressure transducer ADC driver
│   │   └── flow.rs      # Pulse-count flow meter driver
│   ├── radio/
│   │   ├── mod.rs       # LoRa abstraction
│   │   └── lorawan.rs   # LoRaWAN packet assembly + OTAA join
│   ├── crypto/
│   │   └── signing.rs   # HMAC-SHA256 payload signing
│   ├── config/
│   │   └── nfc.rs       # NFC-based device configuration
│   ├── storage/
│   │   └── flash.rs     # RTC memory for cumulative volume
│   └── payload.rs       # 11-byte payload serialization
└── Cargo.toml
```

Core Measurement Loop

```
//! main.rs — bare-metal entry point for well meter sensor
//! Table of Contents:
//!   - Deep-sleep wake → sensor read → LoRa TX → deep-sleep cycle
//!   - 15-minute interval, 10-year battery life target

#![no_std]
#![no_main]

use esp_hal::{clock::ClockControl, peripherals::Peripherals, prelude::*, rtc_cntl::Rtc};

mod sensor;
mod radio;
mod crypto;
mod config;
mod storage;
mod payload;

use sensor::{pressure::PressureSensor, flow::FlowSensor};
use radio::lorawan::LoRaWanRadio;
use payload::{WellPayload, StatusFlags};
use crypto::signing::sign_payload;
use storage::flash::CumulativeStore;

/// Measurement interval: 15 minutes = 900 seconds
const SLEEP_DURATION_US: u64 = 900 * 1_000_000;

#[entry]
```

```rust
fn main() -> ! {
    let peripherals = Peripherals::take();
    let clocks = ClockControl::max(peripherals.SYSTEM).freeze();
    let mut rtc = Rtc::new(peripherals.RTC_CNTL);

    // Initialize sensors
    let mut pressure = PressureSensor::new(peripherals.ADC1, &clocks);
    let mut flow = FlowSensor::new(peripherals.GPIO, peripherals.PCNT);

    // Read current values
    let water_level_mm = pressure.read_level_mm();
    let flow_lpm = flow.read_flow_lpm();
    let delta_volume_10l = flow.consume_delta_volume_10l();

    // Update cumulative volume in RTC-backed flash
    let mut store = CumulativeStore::load();
    store.add_volume(delta_volume_10l);
    store.save();

    // Build status flags
    let mut status = StatusFlags::default();
    let battery_mv = read_battery_mv(&peripherals);
    if battery_mv < 2800 {
        status.set(StatusFlags::LOW_BATTERY);
    }
    if flow_lpm > 0 {
        status.set(StatusFlags::PUMP_RUNNING);
    }

    // Assemble payload
    let payload = WellPayload {
        water_level_mm,
        cumulative_volume_10l: store.total(),
        flow_lpm,
        battery_mv,
        status,
    };

    // Sign and transmit
    let signed = sign_payload(&payload.serialize(), &device_key());
    let mut radio = LoRaWanRadio::new(peripherals.SPI2, &clocks);
    radio.transmit(&signed);

    // Deep sleep until next reading
    rtc.sleep_deep(SLEEP_DURATION_US);
}
```

Payload Serialization

```rust
//! payload.rs — 11-byte packed payload for LoRa transmission
//! Table of Contents:
//!   - WellPayload struct (sensor reading snapshot)
//!   - StatusFlags bitfield
//!   - serialize() → [u8; 11]

/// Sensor reading snapshot for a single 15-minute interval.
#[derive(Debug, Clone, Copy)]
pub struct WellPayload {
    /// Water level above transducer in mm (0–16,777,215)
    pub water_level_mm: u32,
    /// Cumulative extracted volume in 10-liter units (resets monthly)
    pub cumulative_volume_10l: u32,
    /// Current flow rate in liters per minute (0–65,535)
    pub flow_lpm: u16,
    /// Battery voltage in millivolts
    pub battery_mv: u16,
    /// Status bitfield
    pub status: StatusFlags,
}

/// Bitfield for device status reporting.
#[derive(Debug, Clone, Copy, Default)]
pub struct StatusFlags(pub u8);

impl StatusFlags {
    pub const TAMPER: u8       = 0b0000_0001;
    pub const SENSOR_FAULT: u8 = 0b0000_0010;
    pub const LOW_BATTERY: u8  = 0b0000_0100;
    pub const PUMP_RUNNING: u8 = 0b0000_1000;

    /// Set a flag bit.
    pub fn set(&mut self, flag: u8) { self.0 |= flag; }

    /// Check if a flag bit is set.
    pub fn is_set(&self, flag: u8) -> bool { self.0 & flag != 0 }
}

impl WellPayload {
    /// Serialize to 11-byte packed payload.
    /// Layout:
    ///   [0]    device type (0x01 = tube well + flow)
    ///   [1-3]  water_level_mm (BE 24-bit)
    ///   [4-6]  cumulative_volume_10l (BE 24-bit)
    ///   [7-8]  flow_lpm (BE 16-bit)
    ///   [9]    battery_mv / 10
    ///   [10]   status flags
    pub fn serialize(&self) -> [u8; 11] {
        let mut buf = [0u8; 11];
        buf[0] = 0x01;
        buf[1] = ((self.water_level_mm >> 16) & 0xFF) as u8;
        buf[2] = ((self.water_level_mm >> 8) & 0xFF) as u8;
        buf[3] = (self.water_level_mm & 0xFF) as u8;
        buf[4] = ((self.cumulative_volume_10l >> 16) & 0xFF) as u8;
        buf[5] = ((self.cumulative_volume_10l >> 8) & 0xFF) as u8;
        buf[6] = (self.cumulative_volume_10l & 0xFF) as u8;
        buf[7] = ((self.flow_lpm >> 8) & 0xFF) as u8;
        buf[8] = (self.flow_lpm & 0xFF) as u8;
        buf[9] = (self.battery_mv / 10) as u8;
        buf[10] = self.status.0;
        buf
    }
}
```

Pressure Sensor Driver

```
//! sensor/pressure.rs — ADC-backed pressure transducer
//! Table of Contents:
//!   - PressureSensor struct
//!   - read_level_mm() — water level in millimeters
//!   - mv_to_pressure_pa() — linear calibration
//!   - pressure_to_level_mm() — P/(ρg) conversion

use esp_hal::adc::{AdcConfig, Attenuation, ADC};

/// Pressure transducer connected to ADC1.
/// Ceramic capacitive, 0–10 bar, 0.5–2.5V output.
pub struct PressureSensor<'a> {
    adc: ADC<'a, esp_hal::peripherals::ADC1>,
    /// Output mV at 0 Pa (atmospheric reference)
    zero_mv: u16,
    /// Sensitivity: mV per Pa
    sensitivity_mv_per_pa: f32,
}

impl<'a> PressureSensor<'a> {
    pub fn new(adc1: esp_hal::peripherals::ADC1, clocks: &esp_hal::clock::Clocks) -> Self {
        let config = AdcConfig::new();
        Self {
```

```rust
            adc: ADC::new(adc1, config, clocks),
            zero_mv: 500,                // 0.5V at 0 bar
            sensitivity_mv_per_pa: 0.02, // 2V span / 100,000 Pa
        }
    }

    /// Read water level above transducer in millimeters.
    pub fn read_level_mm(&mut self) -> u32 {
        let raw_mv = self.read_raw_mv();
        let pressure_pa = self.mv_to_pressure_pa(raw_mv);
        self.pressure_to_level_mm(pressure_pa)
    }

    /// Average 16 ADC samples to reduce noise.
    fn read_raw_mv(&mut self) -> u16 {
        let sum: u32 = (0..16).map(|_| self.adc.read_blocking() as u32).sum();
        (sum / 16) as u16
    }

    /// Convert millivolts to gauge pressure in Pascals.
    fn mv_to_pressure_pa(&self, mv: u16) -> u32 {
        let delta = mv.saturating_sub(self.zero_mv) as f32;
        (delta / self.sensitivity_mv_per_pa) as u32
    }

    /// Convert pressure to water column depth: d = P / (ρg)
    /// Returns millimeters.
    fn pressure_to_level_mm(&self, pressure_pa: u32) -> u32 {
        (pressure_pa * 1000) / 9810
    }
}
```

Flow Sensor Driver

```rust
//! sensor/flow.rs — pulse-counting flow meter via PCNT hardware
//! Table of Contents:
//!   - FlowSensor struct
//!   - K_FACTOR calibration constant
//!   - read_flow_lpm() — instantaneous flow rate
//!   - consume_delta_volume_10l() — accumulated volume since last read

use esp_hal::pcnt::Pcnt;

/// Pulses per liter for DN40 paddle-wheel sensor.
/// Calibrated per manufacturing batch.
```

```rust
const K_FACTOR_PULSES_PER_LITER: u32 = 450;

/// Minimum pulses per 15-min window to register flow (noise floor).
const MIN_PULSES_PER_15MIN: u32 = 10;

pub struct FlowSensor<'a> {
    pcnt: Pcnt<'a>,
    accumulated_pulses: u32,
    last_window_pulses: u32,
}

impl<'a> FlowSensor<'a> {
    pub fn new(gpio: esp_hal::peripherals::GPIO, pcnt: esp_hal::peripherals::PCNT) -> Self
 {
        Self {
            pcnt: Pcnt::new(pcnt, gpio),
            accumulated_pulses: 0,
            last_window_pulses: 0,
        }
    }

    /// Instantaneous flow rate in liters per minute.
    pub fn read_flow_lpm(&self) -> u16 {
        if self.last_window_pulses < MIN_PULSES_PER_15MIN {
            return 0; // Pump not running — valid zero
        }
        let pulses_per_min = self.last_window_pulses / 15;
        (pulses_per_min * 1000 / K_FACTOR_PULSES_PER_LITER) as u16
    }

    /// Consume accumulated volume since last call, in 10-liter units.
    pub fn consume_delta_volume_10l(&mut self) -> u32 {
        let liters = self.accumulated_pulses * 1000 / K_FACTOR_PULSES_PER_LITER;
        self.accumulated_pulses = 0;
        liters / 10
    }
}
```

HMAC Signing

```rust
 //! crypto/signing.rs — HMAC-SHA256 payload authentication
//! Table of Contents:
//!   - sign_payload() — append 4-byte truncated HMAC to payload
//!   - DeviceKey — 32-byte key stored in eFuse

use hmac::{Hmac, Mac};
use sha2::Sha256;

type HmacSha256 = Hmac<Sha256>;

/// Sign an 11-byte payload with the device's unique key.
/// Returns 15 bytes: 11 payload + 4 HMAC truncated.
pub fn sign_payload(payload: &[u8; 11], key: &[u8; 32]) -> [u8; 15] {
    let mut mac = HmacSha256::new_from_slice(key).expect("valid key length");
    mac.update(payload);
    let result = mac.finalize().into_bytes();

    let mut signed = [0u8; 15];
    signed[..11].copy_from_slice(payload);
    signed[11..15].copy_from_slice(&result[..4]); // truncated HMAC
    signed
}
```

---

8. Rust Backend Architecture

The backend ingests LoRaWAN packets from all gateways, validates signatures, stores time-series data, and serves the public API and portal.

Backend Cargo.toml

```toml
 [package]
name = "well-meter-backend"
version = "0.1.0"
edition = "2021"

[dependencies]
tokio = { version = "1", features = ["full"] }
axum = "0.7"
sqlx = { version = "0.7", features = ["postgres", "runtime-tokio-native-tls", "chrono"] }
rumqttc = "0.24"
hmac = "0.12"
sha2 = "0.10"
hex = "0.4"
serde = { version = "1", features = ["derive"] }
serde_json = "1"
geo = "0.28"
tracing = "0.1"
tracing-subscriber = { version = "0.3", features = ["env-filter"] }
```

```
chrono = { version = "0.4", features = ["serde"] }
uuid = { version = "1", features = ["v4", "serde"] }
```

Backend Module Structure

```
well-meter-backend/
├── src/
│   ├── main.rs              # Tokio runtime, service startup
│   ├── ingest/
│   │   ├── mod.rs           # Ingest pipeline orchestration
│   │   ├── mqtt.rs          # ChirpStack MQTT subscriber
│   │   ├── decoder.rs       # 11-byte payload decoder
│   │   └── validator.rs     # HMAC signature verification
│   ├── storage/
│   │   ├── mod.rs           # Storage trait
│   │   ├── timeseries.rs    # TimescaleDB hypertable writes
│   │   └── spatial.rs       # PostGIS spatial queries
│   ├── api/
│   │   ├── mod.rs           # Axum router
│   │   ├── wells.rs         # GET /wells, GET /wells/:id
│   │   ├── basin.rs         # GET /basin/depletion-map
│   │   └── export.rs        # GET /export/csv, /export/geojson
│   ├── analysis/
│   │   ├── mod.rs           # Analysis pipeline
│   │   ├── depletion.rs     # Net overdraft per zone
│   │   ├── anomaly.rs       # Outlier / tamper detection
│   │   └── recharge.rs      # IGBWP recharge response tracking
│   └── models.rs            # Shared data types
└── Cargo.toml
```

Ingest Pipeline

```
//! ingest/decoder.rs — LoRaWAN payload decoder
//! Table of Contents:
//!   - RawLoRaPacket — raw bytes from ChirpStack MQTT
//!   - WellReading — decoded, validated sensor reading
//!   - decode() — 11-byte payload → WellReading
//!   - verify_hmac() — cryptographic tamper check

use chrono::{DateTime, Utc};
use serde::{Deserialize, Serialize};

/// Raw packet from ChirpStack MQTT broker.
#[derive(Debug, Deserialize)]
```

```rust
pub struct RawLoRaPacket {
    /// Device EUI (64-bit hex)
    pub dev_eui: String,
    /// Base64-encoded payload
    pub data: String,
    /// Gateway receive timestamp (UTC)
    pub rx_time: DateTime<Utc>,
    /// Received signal strength (dBm)
    pub rssi: i16,
    /// Signal-to-noise ratio (dB)
    pub snr: f32,
}

/// Decoded and validated sensor reading.
#[derive(Debug, Clone, Serialize)]
pub struct WellReading {
    pub device_eui: String,
    pub timestamp: DateTime<Utc>,
    pub water_level_mm: u32,
    pub cumulative_volume_10l: u32,
    pub flow_lpm: u16,
    pub battery_voltage_mv: u16,
    pub tamper_detected: bool,
    pub sensor_fault: bool,
    pub low_battery: bool,
    pub pump_running: bool,
    pub rssi: i16,
    pub snr: f32,
    pub hmac_valid: bool,
}

/// Decode 11-byte payload into a WellReading.
pub fn decode(packet: &RawLoRaPacket, payload: &[u8; 15]) -> WellReading {
    let water_level_mm =
        ((payload[1] as u32) << 16) | ((payload[2] as u32) << 8) | (payload[3] as u32);
    let cumulative_volume_10l =
        ((payload[4] as u32) << 16) | ((payload[5] as u32) << 8) | (payload[6] as u32);
    let flow_lpm = ((payload[7] as u16) << 8) | (payload[8] as u16);
    let battery_voltage_mv = (payload[9] as u16) * 10;
    let status = payload[10];

    WellReading {
        device_eui: packet.dev_eui.clone(),
        timestamp: packet.rx_time,
        water_level_mm,
        cumulative_volume_10l,
        flow_lpm,
        battery_voltage_mv,
        tamper_detected: status & 0x01 != 0,
        sensor_fault: status & 0x02 != 0,
        low_battery: status & 0x04 != 0,
        pump_running: status & 0x08 != 0,
        rssi: packet.rssi,
        snr: packet.snr,
        hmac_valid: true, // set by validator
    }
}
```

MQTT Subscriber (ChirpStack Integration)

```
 //! ingest/mqtt.rs — ChirpStack LoRaWAN network server MQTT subscriber
//! Table of Contents:
//!   - subscribe_to_chirpstack() — connect and listen for uplink events
//!   - process_uplink() — decode, validate, store each packet

use rumqttc::{AsyncClient, MqttOptions, QoS};
use tokio::sync::mpsc;

/// Subscribe to ChirpStack MQTT broker for all device uplinks.
pub async fn subscribe_to_chirpstack(
    broker_host: &str,
    broker_port: u16,
    tx: mpsc::Sender<super::decoder::WellReading>,
) -> Result<(), Box<dyn std::error::Error>> {
    let mut opts = MqttOptions::new("well-meter-ingest", broker_host, broker_port);
    opts.set_keep_alive(std::time::Duration::from_secs(30));

    let (client, mut eventloop) = AsyncClient::new(opts, 256);

    // Subscribe to all application uplinks
    client
        .subscribe("application/+/device/+/event/up", QoS::AtLeastOnce)
        .await?;

    // Process incoming messages
    loop {
        match eventloop.poll().await {
            Ok(rumqttc::Event::Incoming(rumqttc::Packet::Publish(msg))) => {
                if let Ok(packet) = serde_json::from_slice::<super::decoder::RawLoRaPacket>
 (
                    &msg.payload,
                ) {
                    // Decode payload bytes
                    let payload_bytes = base64_decode(&packet.data);
                    if payload_bytes.len() == 15 {
                        let mut buf = [0u8; 15];
                        buf.copy_from_slice(&payload_bytes);
                        let reading = super::decoder::decode(&packet, &buf);
                        let _ = tx.send(reading).await;
                    }
                }
            }
            Err(e) => {
                tracing::error!("MQTT error: {e}");
                tokio::time::sleep(std::time::Duration::from_secs(5)).await;
            }
            _ => {}
        }
    }
}
```

TimescaleDB Storage

```
//! storage/timeseries.rs — TimescaleDB hypertable for well readings
//! Table of Contents:
//!   - create_hypertable() — one-time schema setup
//!   - insert_reading() — single reading insert
//!   - batch_insert() — bulk insert for high-throughput ingest

use sqlx::PgPool;
use super::super::ingest::decoder::WellReading;

/// Create the hypertable schema (run once at startup).
pub async fn create_hypertable(pool: &PgPool) -> Result<(), sqlx::Error> {
    sqlx::query(
        r#"
        CREATE TABLE IF NOT EXISTS well_readings (
            time            TIMESTAMPTZ NOT NULL,
            device_eui      TEXT NOT NULL,
            water_level_mm  INTEGER NOT NULL,
            volume_10l      INTEGER NOT NULL,
            flow_lpm        SMALLINT NOT NULL,
            battery_mv      SMALLINT NOT NULL,
            pump_running    BOOLEAN NOT NULL,
            tamper          BOOLEAN NOT NULL,
            rssi            SMALLINT,
            snr             REAL,
            hmac_valid      BOOLEAN NOT NULL DEFAULT TRUE
        );

        -- Convert to TimescaleDB hypertable (partitioned by time)
        SELECT create_hypertable('well_readings', 'time',
            if_not_exists => TRUE,
            chunk_time_interval => INTERVAL '1 day'
        );

        -- Indexes for common queries
        CREATE INDEX IF NOT EXISTS idx_well_device_time
            ON well_readings (device_eui, time DESC);
        CREATE INDEX IF NOT EXISTS idx_well_tamper
            ON well_readings (tamper) WHERE tamper = TRUE;
        "#,
    )
    .execute(pool)
    .await?;
    Ok(())
}

/// Insert a single well reading.
pub async fn insert_reading(pool: &PgPool, r: &WellReading) -> Result<(), sqlx::Error> {
    sqlx::query(
        r#"
        INSERT INTO well_readings
            (time, device_eui, water_level_mm, volume_10l, flow_lpm,
             battery_mv, pump_running, tamper, rssi, snr, hmac_valid)
        VALUES ($1, $2, $3, $4, $5, $6, $7, $8, $9, $10, $11)
        "#,
    )
    .bind(r.timestamp)
    .bind(&r.device_eui)
    .bind(r.water_level_mm as i32)
    .bind(r.cumulative_volume_10l as i32)
    .bind(r.flow_lpm as i16)
    .bind(r.battery_voltage_mv as i16)
    .bind(r.pump_running)
    .bind(r.tamper_detected)
    .bind(r.rssi)
    .bind(r.snr)
    .bind(r.hmac_valid)
    .execute(pool)
    .await?;
    Ok(())
}
```

API Layer

```rust
//! api/wells.rs — Public REST API for well data
//! Table of Contents:
//!   - GET /api/v1/wells — list wells with pagination + spatial filter
//!   - GET /api/v1/wells/:device_eui — single well detail + history
//!   - GET /api/v1/wells/:device_eui/readings — time-series readings

use axum::{extract::{Path, Query, State}, Json, Router, routing::get};
use serde::{Deserialize, Serialize};
use sqlx::PgPool;

/// Well summary for list endpoint.
#[derive(Serialize)]
pub struct WellSummary {
    pub device_eui: String,
    pub lat: f64,
    pub lon: f64,
    pub district: String,
    pub state: String,
    pub last_water_level_m: f64,
    pub last_flow_lpm: u16,
    pub monthly_volume_m3: f64,
    pub battery_ok: bool,
    pub tamper_alert: bool,
    pub last_seen: String,
}

/// Query parameters for well list.
#[derive(Deserialize)]
pub struct WellListParams {
    pub state: Option<String>,
    pub district: Option<String>,
    pub lat_min: Option<f64>,
    pub lat_max: Option<f64>,
    pub lon_min: Option<f64>,
    pub lon_max: Option<f64>,
    pub page: Option<u32>,
    pub limit: Option<u32>,
}

/// Build the wells API router.
pub fn router() -> Router<PgPool> {
    Router::new()
        .route("/api/v1/wells", get(list_wells))
        .route("/api/v1/wells/:device_eui", get(get_well))
        .route("/api/v1/wells/:device_eui/readings", get(get_readings))
}

/// List wells with optional spatial and administrative filters.
async fn list_wells(
    State(pool): State<PgPool>,
    Query(params): Query<WellListParams>,
) -> Json<Vec<WellSummary>> {
    let limit = params.limit.unwrap_or(100).min(1000);
    let offset = params.page.unwrap_or(0) * limit;

    // Query with spatial bounding box + state/district filters
    let wells = sqlx::query_as!(
        WellSummary,
        r#"
        SELECT
            w.device_eui,
            w.lat, w.lon,
            w.district, w.state,
            COALESCE(r.water_level_mm, 0)::float / 1000.0 as "last_water_level_m!",
            COALESCE(r.flow_lpm, 0) as "last_flow_lpm!: u16",
            COALESCE(m.total_volume, 0)::float * 10.0 / 1000.0 as "monthly_volume_m3!",
            COALESCE(r.battery_mv > 2800, true) as "battery_ok!",
            COALESCE(r.tamper, false) as "tamper_alert!",
            r.time::text as "last_seen!"
        FROM well_registry w
        LEFT JOIN LATERAL (
            SELECT * FROM well_readings
            WHERE device_eui = w.device_eui
            ORDER BY time DESC LIMIT 1
        ) r ON true
        LEFT JOIN LATERAL (
            SELECT SUM(volume_10l) as total_volume
            FROM well_readings
            WHERE device_eui = w.device_eui
            AND time > date_trunc('month', now())
        ) m ON true
        WHERE ($1::text IS NULL OR w.state = $1)
          AND ($2::text IS NULL OR w.district = $2)
          AND ($3::float IS NULL OR w.lat >= $3)
          AND ($4::float IS NULL OR w.lat <= $4)
          AND ($5::float IS NULL OR w.lon >= $5)
          AND ($6::float IS NULL OR w.lon <= $6)
        ORDER BY w.state, w.district, w.device_eui
        LIMIT $7 OFFSET $8
        "#,
        params.state,
        params.district,
        params.lat_min,
        params.lat_max,
        params.lon_min,
        params.lon_max,
        limit as i64,
        offset as i64,
    )
    .fetch_all(&pool)
    .await
    .unwrap_or_default();

    Json(wells)
}
```

9. Data Model

Well Registry (PostGIS)

```sql
-- Well registration table — one row per physical well
CREATE TABLE well_registry (
    device_eui      TEXT PRIMARY KEY,           -- LoRaWAN device EUI
    well_id         TEXT UNIQUE NOT NULL,        -- Government registration ID
    owner_name      TEXT,                        -- Well owner (farmer name)
    owner_phone     TEXT,                        -- SMS notification number
    lat             DOUBLE PRECISION NOT NULL,   -- WGS84 latitude
    lon             DOUBLE PRECISION NOT NULL,   -- WGS84 longitude
    elevation_m     REAL,                        -- Ground elevation (meters ASL)
    well_depth_m    REAL,                        -- Total well depth
    pipe_diameter_mm INTEGER,                    -- DN25, DN40, DN50, etc.
    pump_hp         REAL,                        -- Pump horsepower
    state           TEXT NOT NULL,               -- e.g., 'Uttar Pradesh'
    district        TEXT NOT NULL,               -- e.g., 'Kanpur Nagar'
    block           TEXT,                        -- Administrative block
    village         TEXT,                        -- Village name
    geom            GEOMETRY(Point, 4326),       -- PostGIS point
    installed_at    TIMESTAMPTZ NOT NULL DEFAULT now(),
    device_key_hash TEXT NOT NULL,               -- SHA256 of device HMAC key
    status          TEXT NOT NULL DEFAULT 'active'  -- active | maintenance | tampered |
         decommissioned
);

-- Spatial index for bounding box queries
CREATE INDEX idx_well_geom ON well_registry USING GIST (geom);
```

```sql

-- Administrative index for state/district rollups
CREATE INDEX idx_well_admin ON well_registry (state, district, block);
```

Aggregation Views

```sql
-- District-level monthly extraction summary
CREATE MATERIALIZED VIEW district_monthly_extraction AS
SELECT
    w.state,
    w.district,
    date_trunc('month', r.time) AS month,
    COUNT(DISTINCT w.device_eui) AS active_wells,
    SUM(r.volume_10l) * 10.0 / 1e6 AS extraction_million_liters,
    AVG(r.water_level_mm) / 1000.0 AS avg_water_level_m,
    SUM(CASE WHEN r.tamper THEN 1 ELSE 0 END) AS tamper_alerts,
    SUM(CASE WHEN r.battery_mv < 2800 THEN 1 ELSE 0 END) AS low_battery_count
FROM well_registry w
JOIN well_readings r ON r.device_eui = w.device_eui
GROUP BY w.state, w.district, date_trunc('month', r.time);

-- Refresh daily via cron
-- REFRESH MATERIALIZED VIEW CONCURRENTLY district_monthly_extraction;

-- Basin-level annual overdraft estimate
CREATE VIEW basin_annual_overdraft AS
SELECT
    EXTRACT(YEAR FROM month) AS year,
    SUM(extraction_million_liters) / 1e6 AS extraction_km3,
    AVG(avg_water_level_m) AS avg_water_level_m,
    SUM(active_wells) AS total_metered_wells
FROM district_monthly_extraction
GROUP BY EXTRACT(YEAR FROM month);
```

Data Retention Policy

```sql
-- TimescaleDB automatic compression after 30 days
SELECT add_compression_policy('well_readings', INTERVAL '30 days');

-- Retain raw data for 2 years, then downsample to hourly
SELECT add_retention_policy('well_readings', INTERVAL '2 years');

-- Continuous aggregate for hourly rollups (kept indefinitely)
CREATE MATERIALIZED VIEW well_readings_hourly
```

```
WITH (timescaledb.continuous) AS
SELECT
    time_bucket('1 hour', time) AS bucket,
    device_eui,
    AVG(water_level_mm) AS avg_water_level_mm,
    MAX(volume_10l) AS max_volume_10l,
    AVG(flow_lpm) AS avg_flow_lpm,
    MIN(battery_mv) AS min_battery_mv,
    bool_or(tamper) AS any_tamper,
    bool_or(pump_running) AS any_pump_running
FROM well_readings
GROUP BY bucket, device_eui;
```

---

10. Public Data Portal

Portal: igbwp.gov.in/wells

Modeled on USGS NGWMN and ADWR Third-Party Portal. All data public by default.

Portal Features

Feature	Description	Tucson Equivalent
Interactive map	Leaflet/MapLibre GL with well markers colored by status	USGS NWISWeb mapper
Well detail page	Time-series chart (water level + flow), monthly volume, alerts	USGS hydrograph
District dashboard	Aggregated extraction, water level trends, alert counts	ADWR AMA dashboard
Basin overview	Total extraction vs. recharge, net overdraft estimate	No equivalent — new
Data export	CSV, GeoJSON, API (REST + GraphQL)	USGS waterservices API

Feature	Description	Tucson Equivalent
SMS alerts	Low battery, tamper, pump running >24h continuous	No equivalent
Language support	Hindi, Bhojpuri, Bengali, Punjabi, English	English only

API Endpoints

```
GET  /api/v1/wells                          # List wells (paginated, filterable)
GET  /api/v1/wells/:device_eui              # Single well detail
GET  /api/v1/wells/:device_eui/readings     # Time-series readings
GET  /api/v1/districts/:state/:district     # District summary
GET  /api/v1/basin/overdraft                # Basin-level overdraft estimate
GET  /api/v1/basin/depletion-map            # GeoJSON heatmap of depletion
GET  /api/v1/export/csv?state=&district=    # Bulk CSV export
GET  /api/v1/export/geojson?bbox=           # Spatial export
POST /api/v1/wells/register                 # Register new well (auth required)
POST /api/v1/wells/:id/calibrate            # Submit calibration data
```

Third-Party Submission (ADWR Model)

Following ADWR's Third-Party Water Level Portal concept — allow private well owners, NGOs, and universities to submit manual water level measurements for wells that don't have sensors yet:

```
POST /api/v1/community/submit
{
    "well_id": "UP-KAN-2024-00451",
    "lat": 26.4499,
    "lon": 80.3319,
    "water_level_m": 12.5,
    "measurement_method": "tape_drop",  // tape_drop | chalked_tape | e_tape
    "measured_by": "Ramesh Kumar",
    "phone": "+91-9876543210",
    "photo_url": "https://..."         // optional verification photo
}
```

This expands coverage to wells without sensors — critical for the initial Phase 0 data foundation before 30M sensors are deployed.

11. Deployment Strategy for India

Phase 0: Pilot (Year 1) — 10,000 Wells

Parameter	Value
Target area	Kanpur Nagar district, UP (worst GRACE-FO signal)
Wells	10,000 tube wells + 50 CGWB observation wells
Gateways	~200 (1 per 15 km radius)
Cost	~$150K sensors + $28K gateways + $50K backend = **~$230K**
Duration	6 months install, 12 months observation
Goal	Validate accuracy, battery life, network reliability, tamper rate

Phase 1: State Rollout (Years 2–4) — 5 Million Wells

Parameter	Value
Target area	All of Uttar Pradesh (~5M tube wells)
Gateways	~100,000
Cost	~$75M sensors + $14M gateways + $5M backend = **~$94M**
Manufacturing	India-based assembly (Noida/Greater Noida electronics corridor)
Installation	Village-level technician network — 1 technician per 500 wells

Parameter	Value
Goal	First state-level true extraction map; validate at scale

Phase 2: Basin-Wide (Years 4–8) — 30 Million Wells

Parameter	Value
Target area	Full IGB: UP, Bihar, Punjab, Haryana, West Bengal, MP
Gateways	~600,000
Cost	~$450M sensors + $84M gateways + $20M backend = **~$554M**
Goal	Complete IGB extraction map; close the data gap for IGBWP Phase 2

Installation Process (Per Well)

```
1. Technician arrives with sensor kit + smartphone with NFC app
2. Photograph well head, GPS auto-captured from phone
3. Measure pipe diameter (DN25/DN40/DN50 — determines flow sensor adapter)
4. Clamp flow sensor onto discharge pipe (non-invasive magnetic mount)
5. Lower pressure transducer into well on cable (pre-cut to well depth)
6. NFC tap: phone writes well_id, GPS, pipe_diameter, calibration to sensor
7. Sensor auto-joins LoRaWAN network, first reading transmitted
8. Apply tamper-evident epoxy seal over enclosure screws
9. Total time: ~20 minutes per well
```

Technician Network

Level	Count	Ratio	Role
Village technician	60,000	1 per 500 wells	Install, basic troubleshoot

Level	Count	Ratio	Role
Block supervisor	6,000	1 per 5,000 wells	Quality audit, gateway maintenance
District engineer	600	1 per 50,000 wells	Network planning, calibration
State coordinator	6	1 per state	Government liaison, data quality

Job creation: ~67,000 positions — a significant political asset for state governments.

---

12. Cost Analysis

Per-Well Lifetime Cost (10 Years)

Item	Cost	Notes
Sensor unit (BOM + assembly)	$11.45	One-time
Gateway share (amortized)	$1.00	$140 / 50 wells × 10 yr replacement
Connectivity (LoRa + 4G gateway)	$1.00	$0.10/well/year × 10 years
Installation labor	$2.00	20 min × village technician rate
Backend compute share	$0.50	Cloud hosting amortized
Total per well (10 years)	**$15.95**	
Per well per year	**$1.60**	

Basin-Wide Total Cost

Phase	Wells	Sensor Cost	Gateway Cost	Backend	Install Labor	Total
Pilot	10,000	$115K	$28K	$50K	$20K	**$213K**
UP State	5,000,000	$57M	$14M	$5M	$10M	**$86M**
Full IGB	30,000,000	$344M	$84M	$20M	$60M	**$508M**

Cost Context

Reference	Cost	Wells
WELL_METER full IGB	**$508M**	**30M wells, 10-year data**
IGBWP desalination project	$70–120B	N/A — infrastructure
India Namami Gange programme	$3B (allocated)	River cleanup
USGS NGWMN (entire USA)	~$50M/year	~850,000 wells
Single USGS pressure transducer	$500–2,000	1 well

The metering system costs <1% of the IGBWP infrastructure project and is the prerequisite that determines whether that project is $70B or $120B. The metering ROI is effectively infinite — it prevents a potential $50B sizing error.

---

13. Integration with IGBWP

How WELL_METER Feeds the Pipeline Project

```
 WELL_METER Network                    IGBWP Pipeline Control
─────────────────                    ──────────────────────

30M sensors → readings
    │
    ▼
District extraction maps ──────────→ Size the pipeline
    │                                (3 GW or 12 GW?)
    ▼
```

```
Net overdraft ± 10% ──────────────► Commit to Phase 2
        │                           infrastructure scale
        ▼
Recharge zone response ────────────► Adjust injection rates
        │                           at terminus nodes
        ▼
Seasonal pumping patterns ─────────► Optimize pipeline
        │                           seasonal operation
        ▼
Tamper / compliance data ──────────► Regulatory enforcement
                                    (extraction permits)
```

Specific Data Products for IGBWP

Data Product	Source	IGBWP Use
True net overdraft	Basin-wide extraction − GRACE-FO recharge	Pipeline sizing (595 m^3/s or different?)
Spatial depletion map	Per-district extraction + water level trends	Terminus node placement (Kanpur vs. Lucknow vs. Agra)
Recharge response	Water level rise after injection begins	Validate percolation basin effectiveness
Seasonal extraction	Monthly pumping patterns per district	Pipeline seasonal operation schedule
Compliance rate	Tamper alerts, missing readings	Regulatory enforcement effectiveness
Pump duty cycles	Pump running hours per well	Electricity subsidy reform data

The Critical Dependency

From `IGBWP.md` Phase 0:

> *"Establish true net overdraft rate ± 10%"*

WELL_METER is the only way to achieve this. GRACE-FO gives total mass change but cannot distinguish pumping from recharge. CGWB observation wells give water level but not extraction volume. Only direct metering of the 30 million tube wells closes the loop.

Timeline Alignment

IGBWP Phase	WELL_METER Phase	Data Dependency
Phase 0: Data Foundation (Yr 1–3)	Pilot: 10K wells in Kanpur (Yr 1)	First district-level extraction data
Phase 1: Proof of Concept (Yr 2–4)	UP State: 5M wells (Yr 2–4)	State-level overdraft validation
Phase 2: Pilot Plant (Yr 4–8)	Full IGB: 30M wells (Yr 4–8)	Basin-wide sizing confirmation
Phase 3: Scale-Up (Yr 8–20)	Continuous operation	Recharge response monitoring
Phase 4: Full Operation (Yr 20–100)	Continuous operation	Long-term aquifer equilibrium tracking

Appendix: Key Constants

```
//! Shared constants for WELL_METER firmware and backend
//! Table of Contents:
//!   - Physical constants (water density, gravity)
//!   - Sensor calibration defaults
//!   - Network parameters
//!   - Administrative boundaries

/// Water density at 25°C (kg/m³)
pub const WATER_DENSITY_KG_M3: f64 = 997.0;

/// Gravitational acceleration (m/s²)
pub const GRAVITY_M_S2: f64 = 9.81;

/// Standard atmospheric pressure (Pa)
pub const ATMOSPHERIC_PRESSURE_PA: f64 = 101_325.0;

/// Measurement interval (seconds)
pub const MEASUREMENT_INTERVAL_S: u64 = 900; // 15 minutes
```

```rust

/// Measurements per day
pub const MEASUREMENTS_PER_DAY: u32 = 96;

/// Measurements per year
pub const MEASUREMENTS_PER_YEAR: u32 = 35_040;

/// Low battery threshold (millivolts)
pub const LOW_BATTERY_MV: u16 = 2800;

/// LoRaWAN IN865 frequency band (Hz)
pub const LORA_FREQUENCY_HZ: u32 = 865_000_000;

/// Maximum LoRa payload with HMAC (bytes)
pub const MAX_PAYLOAD_BYTES: usize = 15;

/// HMAC truncation length (bytes)
pub const HMAC_TRUNCATED_LEN: usize = 4;

/// Default flow sensor K-factor (pulses per liter)
pub const DEFAULT_K_FACTOR: u32 = 450;

/// IGB estimated tube wells
pub const IGB_ESTIMATED_WELLS: u32 = 30_000_000;

/// IGB estimated annual pumping (m³)
pub const IGB_ESTIMATED_PUMPING_M3: f64 = 60e9;

/// IGB estimated annual recharge (m³)
pub const IGB_ESTIMATED_RECHARGE_M3: f64 = 45e9;
```

Document created: February 20, 2026 Project: WATER/Measure — Indo-Gangetic Basin Well Metering System Conceptual model: USGS / Tucson Water / ADWR groundwater monitoring Implementation: Bare-metal Rust (ESP32-C3 RISC-V) + Axum backend + TimescaleDB + PostGIS Integration: IGBWP Phase 0 data foundation prerequisite