# ionpy Pro OS — Entwicklungs-Fahrplan & Opus-Einstieg

Erstellt: 2026-02-28
Basis: Architektur-Analyse & Konzept-Session mit Claude Sonnet
Nächster Schritt: Umsetzung mit Claude Opus

---

## Inhaltsverzeichnis

1. Kontext & Ausgangslage
2. Was in dieser Session entschieden wurde
3. Gesamtarchitektur (Zielzustand)
4. Fahrplan: Phase für Phase
5. Prompts für Claude Opus
6. Offene Fragen & Entscheidungen

---

# 1. Kontext & Ausgangslage

ionpy ist ein Python/FastAPI Backend mit Vue 3 Frontend zur Steuerung und Visualisierung von Laborgeräten. Die Architektur folgt dem Muster:

```
Transport → Framing → Device (Entity-ORM) → EventBus → WebSocket → Vue Frontend
```

**Was bereits sehr gut funktioniert:**

- Deklaratives Entity-ORM (vergleichbar Django Models)
- Dynamisches Treiber-Loading via importlib
- Transport/Framing Stack mit Factory & Registry
- GridStack-basiertes Widget-Dashboard
- uPlot Charts mit Plugin-Architektur
- Workspace-Serialisierung (JSON, pro Widget exportState())

**Was fehlt und gebaut werden muss:**

- Projekt-Konzept als übergeordnete Klammer
- Persistenz (SQLite, strukturiert)
- Device-Anlegen über die UI (aktuell nur YAML)
- Alarm-Routing (Berechnung läuft, aber nirgendwo hin)
- Session-Management (existiert, aber nicht wirklich genutzt)

# 2. Was in dieser Session entschieden wurde

## 2.1 Das Projekt-Konzept

Die drei bisher getrennten Systeme — Config (YAML), Session (Backend), Workspace (Frontend) — werden zu einem **Projekt** zusammengefasst.

Ein Projekt ist ein Experiment-Container. Alles was zu einem Messaufbau gehört, lebt zusammen:

```
projects/
└── akku-charakterisierung/
    ├── project.json       ← Metadaten, Klammer
    ├── devices.json       ← Welche Geräte, wie angeschlossen
    ├── workspace.json     ← UI-Layout (bestehendes Format)
    ├── data.db            ← EINE SQLite DB für das gesamte Projekt
    ├── scripts/           ← Automations-Scripte
    │   └── charge_test.py
    └── exports/           ← Alle Exports, egal welche Session
        └── PSU_01_session1.csv
```

## 2.2 Die JSON-Strukturen

### project.json

```
{
  "schema_version": 1,
  "id": "a3f8c2d1-4b5e-4f6a-9c2d-1e7f8b3a2c5d",
  "name": "Akku-Charakterisierung 18650",
  "description": "Kapazitätsmessung nach Lagerung bei verschiedenen
Temperaturen",
  "created": "2026-01-15T09:23:00",
  "last_opened": "2026-02-28T08:12:00",
  "author": "Marcus",
  "tags": ["battery", "18650", "characterization"],
  "active_session": "b7e2a1f4-3c8d-4e9b-a2f5-6d1c8e4b3a7f",
  "files": {
    "devices": "devices.json",
    "workspace": "workspace.json"
  },
  "notes": "Zellen wurden 30 Tage bei 10°C gelagert."
}
```

**Warum schema_version?** Spätere Format-Änderungen können automatisch migriert werden.

**Warum UUID als id?** “Copy devices from Project XY” referenziert über ID, nicht über Namen.
**Warum files-Block?** Flexibilität — theoretisch kann jedes Projekt eine andere Workspace-Datei haben.

## devices.json

```
{
  "schema_version": 1,
  "devices": [
    {
      "id": "PSU_01",
      "driver": "rd6012",
      "enabled": true,
      "alias": "Lade-PSU",
      "color": "#0ea5e9",
      "icon": "power-plug",
      "location": "Rack links, Shelf 2",
      "auto_start": true,
      "transport": {
        "type": "serial",
        "port": "COM3",
        "baudrate": 115200,
        "port_hint": {
          "vid": "0x1A86",
          "pid": "0x7523",
          "description": "USB-Serial CH340"
        }
      },
      "polling_interval_ms": 500,
      "params": {
        "modbus_address": 1
      },
      "logging": {
        "override_level": null
      }
    }
  ]
}
```

**Wichtige Entscheidungen in dieser Struktur:**

- `id` ist intern und unveränderlich (wird in DB referenziert)
- `alias` ist was der User sieht und ändern darf
- `color` und `icon` gehören zur Device-Definition, nicht zum Workspace
- `port_hint` mit VID/PID ist die Lösung für Portabilität auf anderen Rechnern
- `params` ist ein freies Objekt für treiberspezifische Konfiguration (Modbus-Adresse etc.)

# 2.3 Das Datenbank-Schema

**Entscheidung: Eine DB pro Projekt, eine Tabelle pro Device.**

Begründung:

- Eine DB → Session-übergreifende Abfragen ohne JOIN über Dateien
- Eine Tabelle pro Device → direkte Spalten statt EAV oder JSON-Blobs
- Jede Spalte = eine Entity → maximal einfache, schnelle Abfragen
- Dynamisches Schema → Spalten werden beim Treiber-Start angelegt/ergänzt

```
-- Wird beim ersten Start des Projekts angelegt
CREATE TABLE sessions (
    id          TEXT PRIMARY KEY,
    name        TEXT NOT NULL,
    created     REAL NOT NULL,
    stopped     REAL,
    STATUS      TEXT DEFAULT 'running',
    conditions  TEXT,    -- JSON blob: Temperatur, Luftfeuchte, Notizen
    notes       TEXT
);

-- Zeitmarker für Chart-Annotation und Daten-Segmentierung
CREATE TABLE markers (
    id          INTEGER PRIMARY KEY AUTOINCREMENT,
    TIMESTAMP   REAL NOT NULL,
    session_id  TEXT NOT NULL,
    device_id   TEXT,            -- NULL = gilt für alle Devices
    label       TEXT NOT NULL,   -- z.B. "Ladung Start", "Entladung Start"
    color       TEXT DEFAULT '#f59e0b',
    notes       TEXT
);

-- Alarm-Verlauf
CREATE TABLE alarms (
    id          INTEGER PRIMARY KEY AUTOINCREMENT,
    TIMESTAMP   REAL NOT NULL,
    session_id  TEXT NOT NULL,
    device_id   TEXT NOT NULL,
    entity_id   TEXT NOT NULL,
    old_state   TEXT,
    new_state   TEXT,
    VALUE       REAL,
    acknowledged INTEGER DEFAULT 0,
    ack_time    REAL,
    ack_by      TEXT
);

-- Export-Historie
CREATE TABLE exports (
```

```
    id          INTEGER PRIMARY KEY AUTOINCREMENT,
    created     REAL NOT NULL,
    session_id  TEXT,            -- NULL = ganzes Projekt
    format      TEXT,
    filename    TEXT,
    device_id   TEXT,
    entity_id   TEXT
);

-- Device-Tabellen werden DYNAMISCH angelegt:
-- CREATE TABLE device_PSU_01 (
--     id          INTEGER PRIMARY KEY AUTOINCREMENT,
--     timestamp   REAL NOT NULL,
--     session_id  TEXT NOT NULL,
--     u_out       REAL,
--     i_out       REAL,
--     p_out       REAL,
--     output      INTEGER,
--     alarm_state TEXT DEFAULT 'NORMAL'
-- );
-- CREATE INDEX idx_PSU_01 ON device_PSU_01(session_id, timestamp);
```

**Abfrage-Beispiele:**

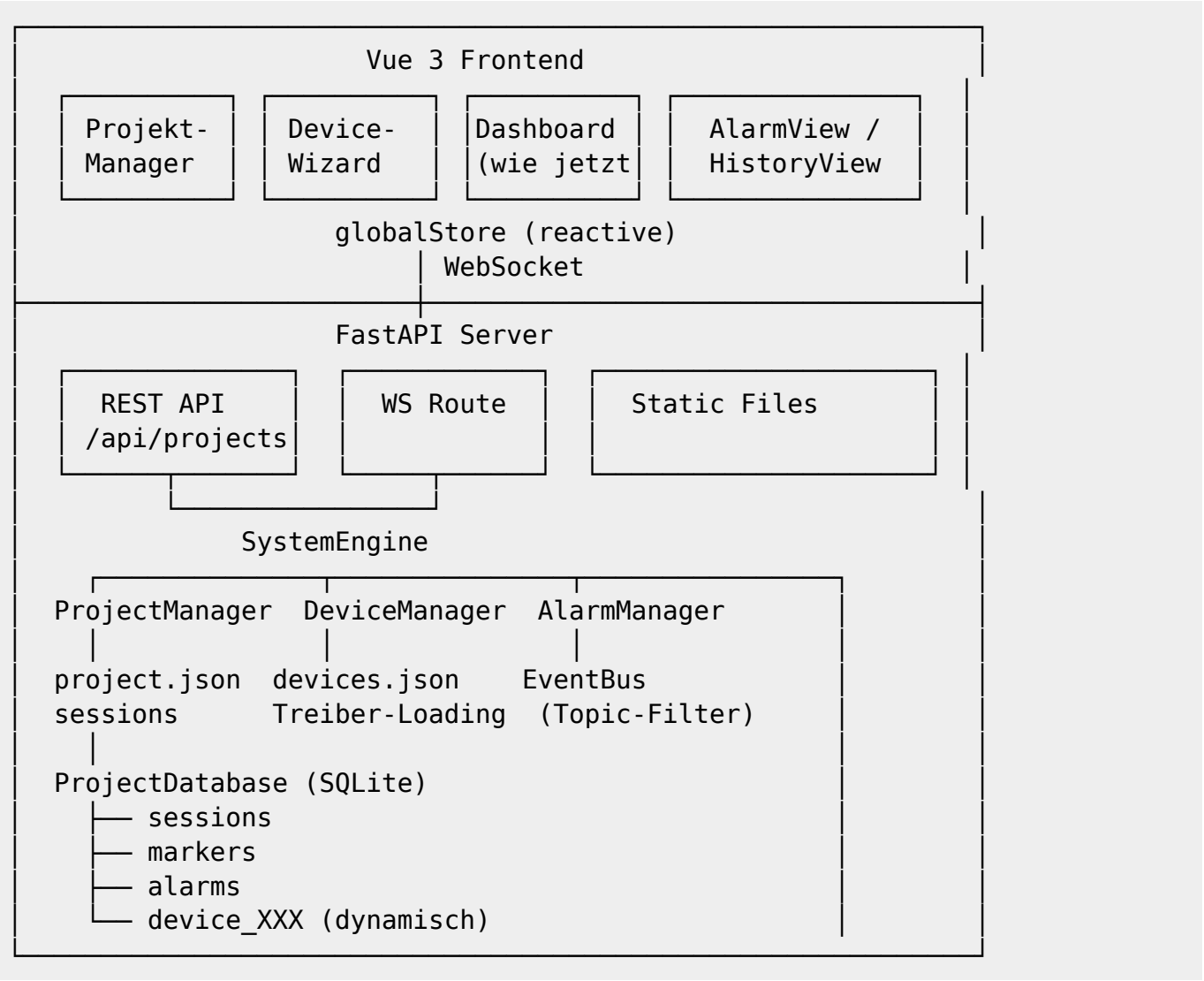
```
-- Chart: 100 Spannungswerte einer Session
SELECT TIMESTAMP, u_out, i_out
FROM device_PSU_01
WHERE session_id = 'abc123'
AND TIMESTAMP BETWEEN 1706000000 AND 1706003600
ORDER BY TIMESTAMP;

-- Session-Vergleich: Max-Spannung pro Session
SELECT session_id, MAX(u_out), MIN(u_out), AVG(i_out)
FROM device_PSU_01
GROUP BY session_id;

-- Daten zwischen zwei Markern (Lade-Segment)
SELECT d.timestamp, d.u_out, d.i_out
FROM device_PSU_01 d
WHERE d.session_id = 'abc123'
AND d.timestamp BETWEEN
    (SELECT TIMESTAMP FROM markers WHERE label='Ladung Start' AND
session_id='abc123')
    AND
    (SELECT TIMESTAMP FROM markers WHERE label='Ladung Ende' AND
session_id='abc123');
```

---

# 3. Gesamtarchitektur (Zielzustand)




---

# 4. Fahrplan: Phase für Phase

## Phase 1: SQLite Backend (Fundament)

**Warum zuerst?** Ohne Persistenz ergibt das Projekt-Konzept keinen Sinn. Die DB-Struktur muss stehen bevor der ProjectManager gebaut wird, weil `data.db` in den Projekt-Ordner gehört.

**Geschätzter Aufwand: 2 Tage**

| Aufgabe | Datei | Details |
|---|---|---|
| ProjectDatabase Klasse | `core/project_database.py` | Persistente Connection, WAL-Mode, Batch-Insert |

| Aufgabe | Datei | Details |
|---|---|---|
| ensure_device_table() | core/project_database.py | Dynamisches Schema, ALTER TABLE für neue Entities |
| Storage Worker in Engine | core/engine.py | Parallel zum Cache-Worker, Batch alle 500ms |
| Marker-API | core/project_database.py | add_marker(), get_markers() |
| History REST Endpoint | web/rest.py | GET /api/devices/{id}/history mit Zeitbereich |

**Kern-Klasse:**

```
# core/project_database.py

import aiosqlite
import asyncio
import logging
import time
import os
from typing import Optional

class ProjectDatabase:
    """
    Eine SQLite-Datenbank pro Projekt.
    Eine Tabelle pro Device, Spalten = Entities.
    Dynamisches Schema-Management.
    """

    def __init__(self, project_path: str):
        self.db_path = os.path.join(project_path, "data.db")
        self._conn: Optional[aiosqlite.Connection] = None
        self._batch: list = []
        self._flush_interval = 0.5
        self.logger = logging.getLogger("ProjectDatabase")

    async def initialize(self):
        self._conn = await aiosqlite.connect(self.db_path)
        self._conn.row_factory = aiosqlite.Row
        # WAL-Mode: bessere Concurrent-Performance, kein Full-Lock beim
Schreiben
        await self._conn.execute("PRAGMA journal_mode=WAL")
        await self._conn.execute("PRAGMA synchronous=NORMAL")
        await self._create_base_schema()
        asyncio.create_task(self._flush_loop())
        self.logger.info(f"Datenbank geöffnet: {self.db_path}")

    async def _create_base_schema(self):
        await self._conn.executescript("""
            CREATE TABLE IF NOT EXISTS sessions (
                id       TEXT PRIMARY KEY,
                name     TEXT NOT NULL,
                created  REAL NOT NULL,
```

```
                stopped  REAL,
                status   TEXT DEFAULT 'running',
                conditions TEXT,
                notes    TEXT
            );
            CREATE TABLE IF NOT EXISTS markers (
                id         INTEGER PRIMARY KEY AUTOINCREMENT,
                timestamp  REAL NOT NULL,
                session_id TEXT NOT NULL,
                device_id  TEXT,
                label      TEXT NOT NULL,
                color      TEXT DEFAULT '#f59e0b',
                notes      TEXT
            );
            CREATE TABLE IF NOT EXISTS alarms (
                id           INTEGER PRIMARY KEY AUTOINCREMENT,
                timestamp    REAL NOT NULL,
                session_id   TEXT NOT NULL,
                device_id    TEXT NOT NULL,
                entity_id    TEXT NOT NULL,
                old_state    TEXT,
                new_state    TEXT,
                value        REAL,
                acknowledged INTEGER DEFAULT 0,
                ack_time     REAL,
                ack_by       TEXT
            );
            CREATE TABLE IF NOT EXISTS exports (
                id         INTEGER PRIMARY KEY AUTOINCREMENT,
                created    REAL NOT NULL,
                session_id TEXT,
                format     TEXT,
                filename   TEXT,
                device_id  TEXT,
                entity_id  TEXT
            );
        """)
        await self._conn.commit()

    async def ensure_device_table(self, device_id: str, entities: dict):
        """
        Erstellt Device-Tabelle wenn sie nicht existiert.
        Fügt fehlende Spalten hinzu wenn Device erweitert wurde.
        """
        safe_id = device_id.replace("-", "_").replace(".", "_")
        table = f"device_{safe_id}"

        base_cols = [
            "id INTEGER PRIMARY KEY AUTOINCREMENT",
            "timestamp REAL NOT NULL",
            "session_id TEXT NOT NULL",
```

```
            "alarm_state TEXT DEFAULT 'NORMAL'"
        ]

        entity_cols = []
        for eid, entity in entities.items():
            t = entity.__class__.__name__
            if t == "NumericEntity":
                entity_cols.append(f"{eid} REAL")
            elif t == "ToggleEntity":
                entity_cols.append(f"{eid} INTEGER")
            else:
                entity_cols.append(f"{eid} TEXT")

        all_cols = base_cols + entity_cols
        await self._conn.execute(
            f"CREATE TABLE IF NOT EXISTS {table} ({', '.join(all_cols)})"
        )
        await self._conn.execute(
            f"CREATE INDEX IF NOT EXISTS idx_{safe_id} ON
{table}(session_id, timestamp)"
        )

        # Fehlende Spalten ergänzen (Treiber wurde erweitert)
        async with self._conn.execute(f"PRAGMA table_info({table})") as cur:
            existing = {row[1] async for row in cur}

        for eid, entity in entities.items():
            if eid not in existing:
                col_type = "REAL" if entity.__class__.__name__ ==
"NumericEntity" else "TEXT"
                await self._conn.execute(
                    f"ALTER TABLE {table} ADD COLUMN {eid} {col_type}"
                )
                self.logger.info(f"Neue Spalte: {table}.{eid}")

        await self._conn.commit()

    def queue_snapshot(self, device_id: str, session_id: str,
                       timestamp: float, values: dict,
                       alarm_state: str = "NORMAL"):
        """Fügt Snapshot zur Batch-Queue hinzu. Nicht-blockierend."""
        self._batch.append({
            "device_id": device_id,
            "session_id": session_id,
            "timestamp": timestamp,
            "values": values,
            "alarm_state": alarm_state
        })

    async def _flush_loop(self):
        """Schreibt alle 500ms gesammelte Snapshots in einem Batch."""
```

```
        while True:
            await asyncio.sleep(self._flush_interval)
            if not self._batch:
                continue
            batch, self._batch = self._batch, []
            try:
                for snap in batch:
                    safe_id = snap["device_id"].replace("-
","_").replace(".","_")
                    table = f"device_{safe_id}"
                    cols = ["timestamp","session_id","alarm_state"] +
list(snap["values"].keys())
                    vals = [snap["timestamp"], snap["session_id"],
snap["alarm_state"]] + list(snap["values"].values())
                    ph = ",".join(["?"]*len(cols))
                    await self._conn.execute(
                        f"INSERT INTO {table} ({','.join(cols)}) VALUES
({ph})", vals
                    )
                await self._conn.commit()
            except Exception as e:
                self.logger.error(f"Flush-Fehler: {e}")

    async def query_device(self, device_id: str, entities: list,
                           session_id: str = None,
                           start: float = None, end: float = None,
                           limit: int = None) -> list:
        """
        Liest Messwerte aus einer Device-Tabelle.
        entities: Liste der gewünschten Entity-IDs (leere Liste = alle)
        """
        safe_id = device_id.replace("-","_").replace(".","_")
        table = f"device_{safe_id}"

        cols = ", ".join(["timestamp"] + entities) if entities else "*"
        where = []
        params = []

        if session_id:
            where.append("session_id = ?")
            params.append(session_id)
        if start:
            where.append("timestamp >= ?")
            params.append(start)
        if end:
            where.append("timestamp <= ?")
            params.append(end)

        sql = f"SELECT {cols} FROM {table}"
        if where:
            sql += " WHERE " + " AND ".join(where)
```

```
        sql += " ORDER BY timestamp"
        if limit:
            sql += f" LIMIT {limit}"

        async with self._conn.execute(sql, params) as cur:
            rows = await cur.fetchall()
            return [dict(r) for r in rows]

    async def add_marker(self, session_id: str, label: str,
                         timestamp: float = None, device_id: str = None,
                         color: str = "#f59e0b", notes: str = None):
        ts = timestamp or time.time()
        await self._conn.execute(
            "INSERT INTO markers (timestamp, session_id, device_id, label,
color, notes) VALUES (?,?,?,?,?,?)",
            (ts, session_id, device_id, label, color, notes)
        )
        await self._conn.commit()

    async def close(self):
        if self._batch:
            await self._flush_loop()
        if self._conn:
            await self._conn.close()
```

---

# Phase 2: ProjectManager Backend

**Warum jetzt?** SQLite steht, jetzt kann die Klammer drumherum gebaut werden.

### Geschätzter Aufwand: 2-3 Tage

| Aufgabe | Datei | Details |
|---|---|---|
| ProjectManager Klasse | `core/project_manager.py` | Ordner, JSONs, Sessions verwalten |
| Engine umbauen | `core/engine.py` | Nimmt Project statt config_path |
| DeviceManager umbauen | `core/device_manager.py` | Liest devices.json statt YAML |
| Migrations-Script | `tools/migrate_config.py` | YAML → Projekt-Struktur konvertieren |
| REST Endpoints | `web/rest.py` | Projekt CRUD, Session Start/Stop |

**Wichtig:** Der bestehende `_spawn_device()` bleibt unverändert — er bekommt weiterhin ein Dict. Nur die Quelle ändert sich von YAML zu JSON.

### Migration von YAML zu Projekt:

```
# tools/migrate_config.py
"""
Konvertiert eine bestehende config.yaml in ein ionpy-Projekt.
Aufruf: python -m tools.migrate_config --config config.yaml --output
projects/mein-projekt
```

```
"""

import yaml
import json
import os
import uuid
from datetime import datetime

def migrate(config_path: str, output_path: str, project_name: str):
    with open(config_path, 'r') as f:
        old_config = yaml.safe_load(f)

    os.makedirs(output_path, exist_ok=True)
    os.makedirs(os.path.join(output_path, "scripts"), exist_ok=True)
    os.makedirs(os.path.join(output_path, "exports"), exist_ok=True)

    # project.json
    project = {
        "schema_version": 1,
        "id": str(uuid.uuid4()),
        "name": project_name,
        "created": datetime.now().isoformat(),
        "last_opened": datetime.now().isoformat(),
        "files": {
            "devices": "devices.json",
            "workspace": "workspace.json"
        }
    }

    # devices.json — aus YAML-Devices konvertieren
    devices = []
    for dev in old_config.get("devices", []):
        devices.append({
            "id": dev.get("id"),
            "driver": dev.get("driver"),
            "enabled": dev.get("enabled", True),
            "alias": dev.get("alias", dev.get("id")),
            "color": dev.get("color", "#0ea5e9"),
            "auto_start": dev.get("auto_start", True),
            "transport": dev.get("transport", {}),
            "polling_interval_ms": dev.get("polling_interval_ms", 1000),
            "params": {k: v for k, v in dev.items()
                       if k not in ["id","driver","enabled","alias","color",
"auto_start","transport","polling_interval_ms"]}
        })

    with open(os.path.join(output_path, "project.json"), 'w') as f:
        json.dump(project, f, indent=2)

    with open(os.path.join(output_path, "devices.json"), 'w') as f:
        json.dump({"schema_version": 1, "devices": devices}, f, indent=2)
```

```
    print(f"Projekt angelegt: {output_path}")
    print(f"Bitte workspace.json manuell in {output_path} kopieren.")
```

---

# Phase 3: Minimale Projekt-UI

**Motto: Es muss funktionieren, nicht schön sein.**

**Geschätzter Aufwand: 1 Tag**

Ein Modal beim Start das fragt “Welches Projekt?”:

```
┌──────────────────────────────────────┐
│  ionpy Pro OS — Projekt wählen       │
├──────────────────────────────────────┤
│  ► Akku-Charakterisierung            │
│    Zuletzt geöffnet: heute 08:12     │
│                                      │
│  ► LED-Treiber Test                  │
│    Zuletzt geöffnet: 15.02.2026      │
│                                      │
│  + Neues Projekt anlegen             │
└──────────────────────────────────────┘
```

Neue REST Endpoints dafür:

```
GET  /api/projects              → alle Projekte auflisten
POST /api/projects              → neues Projekt anlegen
POST /api/projects/{id}/open    → Projekt aktivieren (Engine reload)
GET  /api/projects/active       → aktuell geladenes Projekt
```

**In dieser Phase:** Devices werden noch per JSON-Editor im Browser hinzugefügt (Übergangslösung). Das ist nicht komfortabel, aber es entblockt die weitere Entwicklung.

---

# Phase 4: Device-Wizard

**Geschätzter Aufwand: 3-4 Tage**

Das ist der wichtigste UX-Schritt. Der Workflow:

```
Schritt 1: Treiber wählen
  → Dropdown aus GET /api/config/drivers (bereits vorhanden!)
  → Zeigt Beschreibung aus dem Docstring
  → Zeigt welche Transporte unterstützt werden (STACK_BLUEPRINTS)
```

```
Schritt 2: Instanz konfigurieren
  → ID, Alias, Farbe (immer gleich)
  → Transport-Typ wählen (Serial / TCP / ...)
  → Wenn Serial: Port-Dropdown aus GET /api/inventory/tools/serial
  → Wenn Serial: port_hint wird automatisch aus VID/PID gesetzt
  → Treiber-spezifische params (dynamisch aus driver.parameters)

Schritt 3: Verbindung testen
  → "Verbinden" Button → sofortiges Feedback
  → Zeigt ersten Messwert wenn erfolgreich

Schritt 4: Speichern
  → Landet in devices.json
  → Device wird live gespawnt (kein Server-Neustart nötig!)
```

**“Copy devices from Project XY”:**

```
GET /api/projects → Liste aller Projekte
→ User wählt Quell-Projekt
→ GET /api/projects/{id}/devices → devices.json des Quell-Projekts
→ Checkbox-Liste: welche Devices übernehmen?
→ POST /api/projects/active/devices/import
→ Ports müssen ggf. angepasst werden (Dialog)
```

---

# Phase 5: Quick-Wins (parallel zu jeder Phase machbar)

Diese Aufgaben sind unabhängig und können jederzeit zwischendurch erledigt werden:

| Aufgabe | Aufwand | Datei |
|---|---|---|
| Exponential Backoff beim Reconnect | 1h | devices/transport/base.py |
| Store.js State als Map (Race-Condition) | 1h | static/js/store.js |
| System-Status Endpoint (/api/system/status) | 2h | web/rest.py |
| EventBus Topic-Filtering | 2h | core/event_bus.py |
| EnumEntity + BitmaskEntity | 3h | structures/entities/logic.py |

**Exponential Backoff (fast fertig, nur einsetzen):**

```python
async def reconnect(self, max_retries: int = 10,
                    initial_delay: float = 2.0,
                    max_delay: float = 60.0,
                    backoff_factor: float = 1.5) -> bool:
    delay = initial_delay
    for attempt in range(1, max_retries + 1):
        try:
            await self.disconnect()
            if self.check_hardware_exists():
                if await self.connect():
```

```
                    self.logger.info(f"Wiederverbunden nach {attempt}
Versuch(en)")
                    return True
            except Exception as e:
                self.logger.debug(f"Reconnect #{attempt} Fehler: {e}")

            self.logger.info(f"Versuch {attempt} fehlgeschlagen. Nächster in
{delay:.1f}s")
            await asyncio.sleep(delay)
            delay = min(delay * backoff_factor, max_delay)

        return False
```

---

# Phase 6: AlarmManager

**Warum hier?** Alarm-Berechnung läuft bereits in `_evaluate_alarms()`. Die Hälfte ist schon da.

**Geschätzter Aufwand: 3-4 Tage**

Was hinzukommt:

- `core/alarm_manager.py` — AlarmEvent, Active-Alarms, History, Callbacks
- Verknüpfung mit `_evaluate_alarms()` in den Entity-Klassen
- Alarms in `ProjectDatabase.alarms` speichern
- WebSocket-Event wenn Alarm auftritt (neuer Message-Type)
- Frontend AlarmView: aktive Alarme, History, Quittierung
- Marker in DB setzen wenn Alarm ausgelöst wird (“ALARM: OVP PSU_01”)

---

# Phase 7: Session-Export

**Geschätzter Aufwand: 2 Tage**

```
GET /api/projects/active/sessions/{id}/export?format=csv&device=PSU_01
GET /api/projects/active/sessions/{id}/export?format=json
GET /api/projects/active/export?format=csv   ← ganzes Projekt
```

Export landet automatisch in `projects/xy/exports/` und wird in der `exports` Tabelle geloggt.

---

# Phase 8: Automation/Scripting

**Achtung:** Den `exec()`-basierten Ansatz aus der Roadmap-Analyse mit Vorsicht angehen.

**Empfohlene Reihenfolge:**

1. Erst: **Rule-Engine** (YAML-basiert, Condition → Action) — sicher, einfach
2. Dann: **Python-Scripting** via exec() — nur für lokale, vertrauenswürdige Umgebung

Scripts gehören in `projects/xy/scripts/` und sind Teil des Projekts.

---

# 5. Prompts für Claude Opus

Diese Prompts bauen aufeinander auf. Immer den relevanten Code als Attachment beifügen.

## Prompt 1: ProjectDatabase

```
Ich entwickle "ionpy" - ein Python/FastAPI Backend zur Laborgerätesteuerung
mit Vue 3 Frontend.

Ich habe mich entschieden ein Projekt-Konzept einzuführen. Ein Projekt ist
ein Ordner der alles enthält: devices.json (Gerätedefinitionen),
workspace.json
(UI-Layout) und data.db (SQLite für alle Messdaten).

Ich hänge dir an:
- core/engine.py (die SystemEngine)
- core/device_manager.py
- structures/entities/ (das Entity-ORM)

Bitte implementiere die Klasse ProjectDatabase in core/project_database.py
mit folgenden Anforderungen:

1. Eine SQLite-Datei pro Projekt (data.db im Projekt-Ordner)
2. Eine Tabelle pro Device - Spalten sind die Entity-IDs des Treibers
3. Methode ensure_device_table(device_id, entities_dict) die die Tabelle
   dynamisch anlegt und bei neuen Entities ALTER TABLE ADD COLUMN ausführt
4. Batch-Insert via asyncio: queue_snapshot() ist nicht-blockierend, ein
   _flush_loop() Task schreibt alle 500ms gesammelte Snapshots in einem
   executemany()-Aufruf
5. WAL-Mode und PRAGMA synchronous=NORMAL für Performance
6. Basis-Tabellen: sessions, markers, alarms, exports (Schema siehe unten)
7. Methode query_device() mit optionalem Filter auf session_id, Zeitbereich,
   entity-Liste und LIMIT
8. Methode add_marker() für Chart-Annotationen

Das Schema für die Basis-Tabellen: [sessions/markers/alarms/exports Schema
hier einfügen]
```

```
Bitte keinen neuen Connection-Handler pro Query - eine persistente
Connection
für die gesamte Laufzeit.
```

## Prompt 2: ProjectManager

```
Ich habe jetzt die ProjectDatabase (core/project_database.py - angehängt).

Bitte implementiere den ProjectManager in core/project_manager.py.

Er verwaltet:
- projects/ Ordner-Struktur (ein Unterordner pro Projekt)
- project.json (Metadaten, schema_version, UUID, name, active_session)
- devices.json (Gerätedefinitionen, ersetzt die bisherige config.yaml)
- Schnittstelle zur ProjectDatabase

Das Format von project.json und devices.json:
[JSONs hier einfügen]

Wichtige Methoden:
- list_projects() → alle Projekte im projects/ Ordner
- create_project(name, description) → legt Ordner und JSONs an
- open_project(project_id) → lädt project.json und devices.json
- add_device(device_dict) → fügt Device zu devices.json hinzu und
  validiert Pflichtfelder (id, driver, transport.type)
- remove_device(device_id) → entfernt aus devices.json
- start_session(name, conditions=None) → legt Session in DB an,
  setzt active_session in project.json
- stop_session() → setzt stopped-Timestamp und status='completed'
- get_config_for_engine() → gibt das Dict zurück das SystemEngine
  bisher als config_path erwartet hat (für Rückwärtskompatibilität)

Bestehender Code den du NICHT verändern sollst:
- device_manager._spawn_device() bekommt weiterhin ein Dict
- Alles in structures/ bleibt unverändert
- Die YAML-basierte Config soll noch als Fallback funktionieren
```

## Prompt 3: Engine umbauen

```
Ich habe jetzt ProjectManager und ProjectDatabase implementiert.
Ich hänge dir an: core/engine.py (aktueller Stand), core/project_manager.py

Bitte passe die SystemEngine in core/engine.py an:

1. __init__ nimmt jetzt optional einen project: ProjectManager Parameter
   - Wenn project übergeben: devices aus project.devices_path laden
   - Wenn nicht (Legacy): weiterhin config_path verwenden
```

```
2. Einen _storage_worker Task hinzufügen der parallel zum _cache_worker
läuft:
   - Lauscht auf den EventBus
   - Ruft project.db.queue_snapshot() auf für jeden Sample
   - Filtert WaveformSamples heraus (die werden nicht in die DB geschrieben)
   - Nur aktiv wenn eine Session läuft (session_manager.is_recording)

3. Bei engine.start(): ensure_device_table() für jedes Device aufrufen
   nachdem die Devices geladen wurden

4. engine.stop(): project.db.close() aufrufen

Wichtig: Der bestehende _cache_worker und der EventBus bleiben unverändert.
Der Storage Worker ist ein zusätzlicher Subscriber.
```

## Prompt 4: REST Endpoints für Projekt-Management

```
Ich habe jetzt ProjectManager, ProjectDatabase und die angepasste Engine.
Angehängt: web/rest.py (aktuell), core/project_manager.py

Bitte füge folgende REST Endpoints zu web/rest.py hinzu:

Projekt-Management:
GET  /api/projects              → list_projects()
POST /api/projects              → create_project(name, description)
GET  /api/projects/active       → aktuell geladenes Projekt
POST /api/projects/{id}/open    → Projekt wechseln (Engine neu
initialisieren)

Device-Management im aktiven Projekt:
GET    /api/projects/active/devices         → devices.json
POST   /api/projects/active/devices         → add_device()
DELETE /api/projects/active/devices/{id}    → remove_device()

Session-Management:
GET  /api/projects/active/sessions         → alle Sessions aus DB
POST /api/projects/active/sessions/start   → start_session(name, conditions)
POST /api/projects/active/sessions/stop    → stop_session()

Marker:
POST /api/projects/active/markers → add_marker(label, timestamp?,
device_id?, color?)

History (für Chart-Daten):
GET /api/projects/active/devices/{device_id}/history
    Query-Parameter: session_id, start (Unix-Timestamp), end, entities
(kommagetrennt), limit

Fehlerbehandlung:
```

```
- 404 wenn Projekt nicht gefunden
- 409 wenn Device-ID bereits existiert
- 400 wenn Pflichtfelder fehlen
- 500 mit aussagekräftiger Fehlermeldung bei DB-Fehlern

Alle Endpoints die das aktive Projekt benötigen sollen eine
HTTPException(503, "Kein Projekt geladen") werfen wenn kein Projekt aktiv
ist.
```

## Prompt 5: Migration und CLI-Tool

```
Bitte erstelle tools/migrate_config.py

Das Script konvertiert eine bestehende ionpy config.yaml in die neue
Projekt-Struktur.

Aufruf: python -m tools.migrate_config --config config.yaml --name "Mein
Projekt" --output projects/

Was es tun soll:
1. config.yaml einlesen
2. Projekt-Ordner anlegen (projects/{slug-des-namens}/)
3. project.json generieren (mit neuer UUID, Datum, name)
4. devices.json generieren:
   - Jeder YAML-Device wird zu einem JSON-Device
   - transport-Block bleibt wie er ist
   - Alle unbekannten Felder landen in params{}
   - port_hint wird als leeres Objekt angelegt (muss manuell befüllt werden)
5. Hinweis ausgeben: "Bitte workspace.json manuell kopieren"
6. Validierung: Warnung wenn ein Device kein 'driver' Feld hat

Das Format von devices.json und project.json ist hier definiert:
[JSONs einfügen]
```

## Prompt 6: Device-Wizard Frontend

```
Ich habe jetzt das Backend für das Projekt-Management vollständig.
Angehängt: static/index.html (komplett), die aktuellen REST Endpoints

Bitte implementiere den Device-Wizard als neues Vue 3 Widget.
Er soll als Modal über dem Dashboard erscheinen.

Schritt 1 – Treiber wählen:
- Lädt GET /api/config/drivers
- Zeigt Liste mit Treibername und Beschreibung (aus driver.description)
- Zeigt welche Transport-Typen der Treiber unterstützt (aus
STACK_BLUEPRINTS)
```

```
Schritt 2 – Instanz konfigurieren:
- Pflichtfelder: ID (Text, wird zu device_id), Alias, Farbe (Color-Picker)
- Transport-Typ Dropdown (die Keys aus STACK_BLUEPRINTS des gewählten
Treibers)
- Wenn transport.type == "serial":
    Lädt GET /api/inventory/tools/serial
    Zeigt Port-Dropdown mit Beschreibungen
- Wenn transport.type == "tcp": Host und Port Felder
- Treiber-spezifische params aus driver.parameters dynamisch als Formular-
Felder

Schritt 3 – Test:
- "Verbindung testen" Button
- POST /api/projects/active/devices mit {test: true} Flag
- Zeigt Erfolg/Fehler und wenn erfolgreich den ersten Messwert

Schritt 4 – Speichern:
- POST /api/projects/active/devices (ohne test Flag)
- Device erscheint sofort im Geräte-Menü

Style: Exakt gleicher Dark-Theme Stil wie der Rest der App
(Farben aus den CSS-Variablen). Kein zusätzliches CSS-Framework.

Wichtig: Kein <form> Tag verwenden, nur @click/@input Handler.
```

## Prompt 7: Projekt-Auswahl beim Start

```
Bitte füge dem Frontend (index.html) einen Projekt-Auswahl Dialog hinzu.

Er soll beim ersten Laden erscheinen wenn kein Projekt aktiv ist
(GET /api/projects/active gibt 503 zurück).

Das Modal zeigt:
- Liste aller Projekte (GET /api/projects)
  Pro Eintrag: Name, Beschreibung, "Zuletzt geöffnet" Datum
  Klick → POST /api/projects/{id}/open → Dashboard lädt
- Button "Neues Projekt anlegen"
  → Formular: Name (Pflicht), Beschreibung (optional)
  → POST /api/projects → öffnet das neue Projekt direkt

Das Modal kann NICHT geschlossen werden ohne ein Projekt zu wählen.
Es ist kein Widget - es ist ein Fullscreen-Overlay das den Rest der App
blockiert.

Wenn ein Projekt aktiv ist: Projekt-Name im Header anzeigen
(neben dem "ionpy PRO" Schriftzug).
```

# 6. Offene Fragen & Entscheidungen

Diese Punkte wurden in der Konzept-Session bewusst offen gelassen:

| Frage | Empfehlung | Priorität |
|---|---|---|
| Port-Portabilität: Was wenn COM3 auf neuem Rechner nicht existiert? | VID/PID in port_hint implementieren, Server sucht automatisch | Phase 4 |
| Treiber-Versionierung: Was wenn Treiber sich ändert und alte Session-Daten nicht mehr passen? | driver_version in devices_snapshot der Session speichern | Phase 2 |
| Downsampling: Wann werden historische Daten zu groß? | Retention-Policy als Hintergrund-Task, erst wenn nötig | Später |
| Automation: exec() oder Rule-Engine zuerst? | Rule-Engine (YAML-basiert) zuerst, sicherer | Phase 8 |
| Waveform-Daten in DB? | Nein, Waveforms in eigene Dateien (zu groß für SQLite-Rows) | Offen |
| Multi-User / Concurrent Access? | Aktuell kein Ziel, WAL-Mode gibt aber schon etwas Sicherheit | Nicht geplant |

---

*Erstellt im Rahmen der ionpy Architektur-Session, 2026-02-28*
*Nächster Schritt: Implementierung mit Claude Opus, beginnend mit Phase 1 (ProjectDatabase)*

From:
https://drklipper.de/ - **Dr. Klipper Wiki**

Permanent link:
**https://drklipper.de/doku.php?id=ionpy:project_rodemap**

Last update: **2026/02/28 09:36**