Objetivo Final:

Transformar Auralis de un sistema de monitoreo a una plataforma de inteligencia operacional, donde las alertas no solo informan de fallos, sino que predicen, contextualizan y previenen problemas de forma proactiva.

Fase 1: La Fundación del Motor de Reglas Avanzado

En esta fase, evolucionamos la estructura actual para soportar reglas complejas y dinámicas, pero todavía determinísticas. Es la base indispensable para todo lo que vendrá después.

Bloque 1.1: Evolución del Backend (Django Models)

- **Objetivo:** Modificar el modelo de datos para que en lugar de una AlertPolicy simple por sensor, podamos tener un "árbol de condiciones" que defina una alerta.
- **Arquitectura / Implementación:**
    1. **Nuevo Modelo Rule**: Crea un modelo Rule que será el contenedor principal de una alerta. Tendrá un nombre (ej: "Riesgo de Sobrecalentamiento en Servidores") y estará asociado a una Station o Company.
    2. **Nuevo Modelo Condition**: Este es el corazón. Cada Condition pertenece a una Rule. Sus campos serían:
        - source_sensor: El sensor físico que provee el dato.
        - metric: El tipo de dato a evaluar (ej: value, rate_of_change, std_dev_from_norm).
        - operator: El comparador (ej: >, <, ==, between).
        - threshold_type: El tipo de umbral (STATIC, TIME_BASED, CONDITIONAL).
        - value: Un campo JSON para guardar los umbrales. Si es STATIC, guarda un número. Si es TIME_BASED, guarda un objeto como {"weekday": 30, "weekend": 25}.
    3. **Nuevo Modelo RuleRelationship**: Un modelo para unir las Condition. Permitirá crear la lógica booleana. Campos:
        - rule: La regla a la que pertenece.
        - condition_a: FK a una Condition.
        - logical_operator: AND u OR.
        - condition_b: FK a otra Condition. Esto te permite encadenar condiciones: (A y B) o C.

Bloque 1.2: Creación del "Motor de Reglas" (Rule Engine)

- **Objetivo:** Desarrollar el servicio que procesa los datos entrantes contra las nuevas reglas complejas.
- **Arquitectura / Implementación:**
    - Este motor debe ser un **microservicio separado o un conjunto de tareas de Celery más robustas**. No debe correr dentro de Django. La tarea process_measurement actual se convierte en el "disparador" de este motor.
    - **Flujo de trabajo:**
        1. Llega un dato vía MQTT al mqtt_subscriber.py.
        2. Se lanza una tarea Celery: evaluate_rules(sensor_id, value).
        3. Esta tarea consulta en la base de datos todas las Condition que usan ese sensor_id.
        4. Para cada Condition, evalúa si se cumple (ej: si el umbral es TIME_BASED, primero determina cuál es el umbral activo para la hora actual).
        5. Actualiza el "estado" de esa condición en una caché rápida (como Redis).
        6. Finalmente, evalúa las Rule completas leyendo los estados de sus condiciones desde la caché. Si una regla completa se cumple, dispara el evento de alerta.

Bloque 1.3: Diseño del Constructor Visual de Reglas (UI/UX)

- **Objetivo:** Crear la interfaz para que los usuarios puedan construir estas reglas complejas sin programar.
- **Arquitectura / Implementación:**
    - Utiliza una librería de JavaScript como Rete.js, Drawflow o Blockly. Estas herramientas te dan el "lienzo" y la funcionalidad de arrastrar y soltar nodos.
    - **Proceso en el Frontend:**
        1. El usuario arrastra un "bloque de sensor" al lienzo.
        2. Arrastra un "bloque de condición" (ej: "Mayor que").
        3. Arrastra un "bloque de umbral" (ej: "Umbral Dinámico por Horario"). Al hacer clic, se abre un modal para definir los horarios y valores.
        4. Conecta los bloques.
        5. Añade operadores lógicos (AND/OR) para combinar múltiples ramas.
        6. Al "Guardar", la estructura visual se traduce a un objeto JSON.

7. Este JSON se envía a una API en tu backend (Django), que se encargará de crear y relacionar las instancias de los modelos Rule, Condition y RuleRelationship en la base de datos.

Fase 2: El Salto a la Inteligencia Artificial (ML)

Ahora que tenemos una base sólida, integramos la capacidad de aprender y predecir. Esto se construye como un servicio aparte que alimenta al Motor de Reglas.

Bloque 2.1: Arquitectura del Microservicio de IA/ML

- **Objetivo:** Diseñar un servicio independiente y escalable para entrenar y ejecutar los modelos de Machine Learning.
- **Arquitectura / Implementación:**
    - **Servidor Independiente (Recomendado):** Crea un nuevo servidor (una VM en la nube o un contenedor Docker). Instala un framework de ML como **Flask** o **FastAPI** con librerías como scikit-learn, TensorFlow/Keras o Prophet de Facebook para series temporales.
    - **¿Por qué separado?** Los modelos de ML requieren recursos (CPU/GPU) y librerías que no quieres que afecten el rendimiento de tu aplicación web principal.
    - **Comunicación:** Tu backend de Django se comunicará con este servicio a través de una **API REST**.
        - Django enviará peticiones para: POST /train_model/{sensor_id}
        - El Motor de Reglas consultará: GET /predict_anomaly/{sensor_id}?value=X o GET /predict_future/{sensor_id}.

Bloque 2.2: Umbrales Adaptativos (Detección de Anormalidades)

- **Objetivo:** Que el sistema aprenda qué es "normal" para cada sensor y alerte sobre desviaciones.
- **Implementación:**
    1. **Entrenamiento (Proceso en background):**
        - Crea una tarea periódica (ej. una vez al día) en Celery que, para cada sensor, llame a la API del servicio de ML: POST /train_model/{sensor_id}.
        - El servicio de ML recibe la petición, consulta la base de datos para obtener el historial de mediciones (ej. último mes) y entrena un modelo de detección de anomalías (como Isolation Forest o un Autoencoder).
        - Una vez entrenado, guarda el modelo en el disco del servidor de ML (ej. models/sensor_{sensor_id}.pkl).

2. **Inferencia (en tiempo real):**
    - En el **Motor de Reglas (Bloque 1.2)**, añade un nuevo metric: is_anomaly.
    - Cuando el usuario crea una regla con esta métrica, el motor, al recibir un nuevo dato, hará una llamada a la API de ML: GET /predict_anomaly/{sensor_id}?value=X.
    - El servicio de ML carga el modelo entrenado para ese sensor, predice si el valor es una anomalía y devuelve True o False.

Bloque 2.3: Sensores Virtuales Predictivos

- **Objetivo:** Crear sensores que no miden el presente, sino que predicen el futuro cercano.
- **Implementación:**
    1. **Entrenamiento (similar al anterior):**
        - Usa una tarea periódica para entrenar modelos de series temporales (como ARIMA o Prophet). El objetivo de estos modelos es, dado el historial, predecir los valores para las próximas X horas.
        - Guarda estos modelos predictivos en el servidor de ML.
    2. **Creación en la UI:**
        - Permite al usuario crear un "Sensor Virtual" y asociarlo a un sensor físico y un horizonte de predicción (ej: "predecir temperatura en 1 hora").
    3. **Ejecución y Alerta:**
        - El Motor de Reglas, a intervalos regulares (ej. cada 5 minutos), pide al servicio de ML las predicciones para todos los sensores virtuales.
        - El servicio de ML devuelve los valores predichos (ej. {"predicted_value": 28.5, "timestamp": "2025-09-12T18:00:00"}).
        - Estos valores se tratan como mediciones normales y se pueden usar en el **Constructor Visual de Reglas** para crear alertas predictivas (ej: "Alertar si la temperatura *predicha* para dentro de 1 hora supera los 30°C").

Fase 3: Ecosistema y Experiencia de Usuario Superior

Con las alertas inteligentes funcionando, nos enfocamos en la capa de interacción y automatización.

Bloque 3.1: Orquestación de Notificaciones y Acciones (Integración con n8n)

- **Objetivo:** Permitir que las alertas disparen flujos de trabajo complejos.

- **Tu idea de usar n8n es perfecta para esto.**
- **Arquitectura / Implementación:**
    1. Configura una instancia de n8n.
    2. Cuando el **Motor de Reglas** detecta que una alerta se cumple, en lugar de solo guardar un evento en la base de datos, hará una llamada a un **Webhook de n8n**.
    3. Le pasará un JSON con toda la información de la alerta (nombre, sensor, valor, etc.).
    4. **Dentro de n8n**, el usuario puede construir visualmente flujos de trabajo sin código:
        - "Si la alerta es CRITICAL, enviar SMS vía Twilio Y crear un ticket en Jira".
        - "Si la alerta es WARNING, esperar 10 minutos. Si no se resuelve, enviar un mensaje a un canal de Slack".

Bloque 3.2: Uso de IA Generativa para Contexto

- **Objetivo:** Que las alertas no solo sean números, sino que vengan con una explicación en lenguaje natural.
- **Implementación:**
    - Cuando se genere una alerta, el Motor de Reglas puede hacer una llamada a un modelo de lenguaje grande (como las APIs de Google o OpenAI).
    - **Prompt de ejemplo:** "Eres un asistente de operaciones para un sistema de monitoreo de sensores. Se ha generado una alerta llamada 'Riesgo de Sobrecalentamiento en Servidores'. La causa fue que el 'Sensor de Temperatura 1' (que normalmente opera entre 20-25°C) alcanzó los 31°C. Al mismo tiempo, el 'Sensor de Humedad' está en un 80%, lo cual es anómalo. Genera una notificación corta y clara para el técnico de turno explicando el problema y sugiriendo una acción inmediata."
    - El resultado de la IA se puede incluir en el email, SMS o ticket generado por n8n, dando un valor añadido inmenso al usuario.

Esta hoja de ruta te da un camino claro y modular. Puedes empezar por la Fase 1, que ya por sí sola mejora radicalmente Auralis, y luego ir añadiendo las capas de inteligencia y automatización a medida que avanzas. ¡Es un plan ambicioso pero totalmente factible y te posicionará a la vanguardia!