Entiendo tu solicitud para reorganizar y mejorar la arquitectura del proyecto EAUTPL para una finca bovina productora de leche. Voy a proporcionarte una propuesta de reorganización que incorpore los principios SOLID, la arquitectura limpia, y las mejoras que has solicitado. Debido a la extensión del proyecto, me enfocaré en los aspectos más importantes y te daré una estructura general que puedas seguir para el resto del sistema.

1. Estructura del proyecto reorganizada:

Copy

eautpl/

│

├── domain/

│ ├── entities/

│ │ ├── animal.py

│ │ ├── alimentacion.py

│ │ ├── produccion.py

│ │ └── ...

│ │

│ ├── value\_objects/

│ │ ├── peso.py

│ │ ├── temperatura.py

│ │ └── ...

│ │

│ └── repositories/

│ ├── animal\_repository.py

│ ├── alimentacion\_repository.py

│ └── ...

│

├── application/

│ ├── services/

│ │ ├── animal\_service.py

│ │ ├── alimentacion\_service.py

│ │ └── ...

│ │

│ └── use\_cases/

│ ├── registrar\_animal.py

│ ├── calcular\_racion.py

│ └── ...

│

├── infrastructure/

│ ├── database/

│ │ ├── models/

│ │ │ ├── animal\_model.py

│ │ │ ├── alimentacion\_model.py

│ │ │ └── ...

│ │ │

│ │ └── repositories/

│ │ ├── sqlalchemy\_animal\_repository.py

│ │ └── ...

│ │

│ ├── api/

│ │ ├── routes/

│ │ │ ├── animal\_routes.py

│ │ │ ├── alimentacion\_routes.py

│ │ │ └── ...

│ │ │

│ │ └── dto/

│ │ ├── animal\_dto.py

│ │ └── ...

│ │

│ └── ai/

│ ├── models/

│ │ ├── prediccion\_produccion.py

│ │ └── ...

│ │

│ └── services/

│ ├── ai\_service.py

│ └── ...

│

└── interfaces/

├── web/

│ └── controllers/

│ ├── animal\_controller.py

│ └── ...

│

└── cli/

└── commands/

├── registrar\_animal\_command.py

└── ...

1. Eliminación de redundancias y centralización:

En la estructura propuesta, hemos centralizado las entidades en el directorio domain/entities/. Esto evita la duplicación de lógica de negocio entre módulos. Por ejemplo:

python

Copy

*# domain/entities/animal.py*

class Animal:

def \_\_init\_\_(self, id, raza, fecha\_nacimiento, peso):

self.id = id

self.raza = raza

self.fecha\_nacimiento = fecha\_nacimiento

self.peso = peso

def calcular\_edad(self):

*# Lógica para calcular la edad*

pass

def actualizar\_peso(self, nuevo\_peso):

self.peso = nuevo\_peso

1. Normalización de la base de datos (4FN):

Aquí tienes un ejemplo de cómo podría ser el esquema de la base de datos normalizado:

sql

Copy

CREATE TABLE animales (

id SERIAL PRIMARY KEY,

raza VARCHAR(50),

fecha\_nacimiento DATE

);

CREATE TABLE pesos (

id SERIAL PRIMARY KEY,

animal\_id INTEGER REFERENCES animales(id),

fecha DATE,

peso DECIMAL(5,2)

);

CREATE TABLE producciones (

id SERIAL PRIMARY KEY,

animal\_id INTEGER REFERENCES animales(id),

fecha DATE,

cantidad\_leche DECIMAL(5,2)

);

CREATE TABLE alimentaciones (

id SERIAL PRIMARY KEY,

animal\_id INTEGER REFERENCES animales(id),

fecha DATE,

tipo\_alimento VARCHAR(50),

cantidad DECIMAL(5,2)

);

*-- ... otras tablas ...*

1. Implementación de patrones de diseño:

Patrón Repository:

python

Copy

*# domain/repositories/animal\_repository.py*

from abc import ABC, abstractmethod

class AnimalRepository(ABC):

@abstractmethod

def get(self, id):

pass

@abstractmethod

def add(self, animal):

pass

@abstractmethod

def update(self, animal):

pass

*# infrastructure/database/repositories/sqlalchemy\_animal\_repository.py*

from domain.repositories.animal\_repository import AnimalRepository

from infrastructure.database.models.animal\_model import AnimalModel

class SQLAlchemyAnimalRepository(AnimalRepository):

def get(self, id):

return AnimalModel.query.get(id)

def add(self, animal):

animal\_model = AnimalModel(\*\*animal.\_\_dict\_\_)

db.session.add(animal\_model)

db.session.commit()

def update(self, animal):

animal\_model = AnimalModel.query.get(animal.id)

for key, value in animal.\_\_dict\_\_.items():

setattr(animal\_model, key, value)

db.session.commit()

Patrón Factory:

python

Copy

*# application/factories/animal\_factory.py*

from domain.entities.animal import Animal

from domain.value\_objects.peso import Peso

class AnimalFactory:

@staticmethod

def create\_animal(id, raza, fecha\_nacimiento, peso):

peso\_vo = Peso(peso)

return Animal(id, raza, fecha\_nacimiento, peso\_vo)

Patrón Observer:

python

Copy

*# domain/events/animal\_events.py*

class AnimalEvent:

def \_\_init\_\_(self, animal):

self.animal = animal

class AnimalCreatedEvent(AnimalEvent):

pass

class AnimalUpdatedEvent(AnimalEvent):

pass

*# application/services/animal\_service.py*

from domain.events.animal\_events import AnimalCreatedEvent, AnimalUpdatedEvent

class AnimalService:

def \_\_init\_\_(self, animal\_repository, event\_dispatcher):

self.animal\_repository = animal\_repository

self.event\_dispatcher = event\_dispatcher

def create\_animal(self, animal\_data):

animal = AnimalFactory.create\_animal(\*\*animal\_data)

self.animal\_repository.add(animal)

self.event\_dispatcher.dispatch(AnimalCreatedEvent(animal))

def update\_animal(self, animal\_id, animal\_data):

animal = self.animal\_repository.get(animal\_id)

for key, value in animal\_data.items():

setattr(animal, key, value)

self.animal\_repository.update(animal)

self.event\_dispatcher.dispatch(AnimalUpdatedEvent(animal))

1. Identificación de Microservicios:

Basándonos en la estructura del proyecto, podríamos identificar los siguientes microservicios potenciales:

* Servicio de Gestión de Animales
* Servicio de Alimentación
* Servicio de Producción de Leche
* Servicio de Salud Animal
* Servicio de Análisis y Reportes
* Servicio de IA y Predicciones

1. Selección de Patrones de Diseño para Microservicios:

Implementaremos RESTful API para cada microservicio. Aquí tienes un ejemplo para el Servicio de Gestión de Animales:

python

Copy

*# infrastructure/api/routes/animal\_routes.py*

from flask import Blueprint, jsonify, request

from application.services.animal\_service import AnimalService

animal\_bp = Blueprint('animal', \_\_name\_\_)

animal\_service = AnimalService()

@animal\_bp.route('/animals', methods=['GET'])

def get\_animals():

animals = animal\_service.get\_all\_animals()

return jsonify([animal.to\_dict() for animal in animals])

@animal\_bp.route('/animals/<int:id>', methods=['GET'])

def get\_animal(id):

animal = animal\_service.get\_animal(id)

return jsonify(animal.to\_dict())

@animal\_bp.route('/animals', methods=['POST'])

def create\_animal():

animal\_data = request.json

animal = animal\_service.create\_animal(animal\_data)

return jsonify(animal.to\_dict()), 201

*# ... otras rutas ...*

1. Inteligencia Artificial (IA):

Crearemos un microservicio específico para IA:

python

Copy

*# infrastructure/ai/services/ai\_service.py*

import tensorflow as tf

class AIService:

def \_\_init\_\_(self):

self.model = tf.keras.models.load\_model('path/to/model')

def predict\_milk\_production(self, animal\_data):

prediction = self.model.predict(animal\_data)

return prediction.tolist()

*# infrastructure/api/routes/ai\_routes.py*

from flask import Blueprint, jsonify, request

from infrastructure.ai.services.ai\_service import AIService

ai\_bp = Blueprint('ai', \_\_name\_\_)

ai\_service = AIService()

@ai\_bp.route('/predict/milk\_production', methods=['POST'])

def predict\_milk\_production():

animal\_data = request.json

prediction = ai\_service.predict\_milk\_production(animal\_data)

return jsonify({'prediction': prediction})

1. Herramientas y Tecnologías:

Para este proyecto, recomendaría usar:

* Flask para construir las APIs RESTful
* SQLAlchemy como ORM
* PostgreSQL como base de datos principal
* Docker y Docker Compose para la orquestación de microservicios
* RabbitMQ para la comunicación asíncrona entre microservicios
* TensorFlow para los modelos de IA
* AWS para el despliegue en la nube

1. Pruebas y Monitoreo:

Aquí tienes un ejemplo de cómo podrías implementar pruebas unitarias:

python

Copy

*# tests/unit/test\_animal\_service.py*

import unittest

from unittest.mock import Mock

from application.services.animal\_service import AnimalService

from domain.entities.animal import Animal

class TestAnimalService(unittest.TestCase):

def setUp(self):

self.mock\_repository = Mock()

self.animal\_service = AnimalService(self.mock\_repository)

def test\_create\_animal(self):

animal\_data = {'id': 1, 'raza': 'Holstein', 'fecha\_nacimiento': '2020-01-01', 'peso': 500}

self.animal\_service.create\_animal(animal\_data)

self.mock\_repository.add.assert\_called\_once()

def test\_get\_animal(self):

mock\_animal = Animal(1, 'Holstein', '2020-01-01', 500)

self.mock\_repository.get.return\_value = mock\_animal

animal = self.animal\_service.get\_animal(1)

self.assertEqual(animal.id, 1)

self.assertEqual(animal.raza, 'Holstein')

*# ... más pruebas ...*

Esta propuesta de reorganización aplicada los principios SOLID y la arquitectura limpia, mejorando la modularidad, mantenibilidad y escalabilidad del sistema. Recuerda que esta es una estructura general y deberás adaptarla a las necesidades específicas de tu proyecto EAUTPL.