****

Campus Santa Fe

Implementación de redes de área amplia y servicios distribuidos

Desarrollo de aplicaciones avanzadas de ciencias computacionales

TC3003B.501 y TC3002B.502

Documentación de arquitectura y sistema

**Assessment**

Benjamín Alejandro Cruz Cervantes - A01747811

Alberto Iván Tamez González - A01026999

Bernardo Alejandro Limón Montes de Oca - A01736575

Fecha de entrega:

Lunes 09 de Junio de 2025

**ÍNDICE:**

[**1. Introducción**](#) **2**

[**2. Requisitos del sistema**](#) **2**

[2.1 Requisitos funcionales](#) 2

[2.2 Requisitos no funcionales](#) 2

[**3. Arquitectura general del sistema**](#) **3**

[Diagrama de arquitectura](#) 4

[**4. Componentes y configuración**](#) **4**

[4.1 Frontend](#) 4

[4.2 Backend (FastAPI)](#) 5

[4.3 Microservicio IA](#) 5

[4.4 Base de datos (PostgreSQL)](#) 6

[4.5 Load Balancer (NGINX)](#) 7

[**5. Justificación del diseño**](#) **8**

[**6. Tabla de tecnologías utilizadas**](#) **9**

**Arquitectura del Sistema y Diseño Funcional**

# 1. Introducción

El presente documento describe la arquitectura general y el diseño funcional del sistema de recomendación de cultivos. Esta plataforma fue desarrollada con el objetivo de brindar a agricultores y usuarios institucionales una herramienta intuitiva que, a partir de archivos *CSV* con datos relevantes, proporcione recomendaciones automatizadas mediante inteligencia artificial. La arquitectura fue diseñada bajo principios de modularidad, seguridad, escalabilidad y facilidad de mantenimiento.

La solución no solo contempla el diseño lógico del sistema, sino también su integración con infraestructura real (*OpenStack*), políticas de seguridad, y experiencia de usuario final. Además, el sistema fue concebido para poder replicarse fácilmente en otros entornos institucionales o de producción.

# 2. Requisitos del sistema

## 2.1 Requisitos funcionales

Los siguientes requisitos funcionales fueron considerados prioritarios durante el diseño e implementación del sistema:

* Registro e inicio de sesión mediante *WorkOS*, que permite múltiples métodos de autenticación incluyendo *SSO*, *Magic Link* y *OAuth*.
* Interfaz de carga de archivos *CSV* por parte del usuario final.
* Generación automática de recomendaciones con base en los datos proporcionados.
* Visualización amigable de resultados individuales por cada fila del archivo.
* Historial accesible por cada usuario para consultar sus predicciones anteriores.
* Panel administrativo para visualizar métricas globales del sistema.

## 2.2 Requisitos no funcionales

Además de las funcionalidades visibles, se contemplaron los siguientes aspectos no funcionales:

* Accesibilidad exclusiva desde la red institucional, como medida de seguridad y control.
* Modularidad, permitiendo el mantenimiento y escalamiento de componentes individuales.
* Comunicación cifrada mediante *HTTPS* en todos los puntos de entrada.
* Uso de tokens seguros (*JWT*) para proteger las sesiones aprovechando la simplicidad de *WorkOS*.
* Registro automático de todas las acciones relevantes mediante un sistema de logs en base de datos.

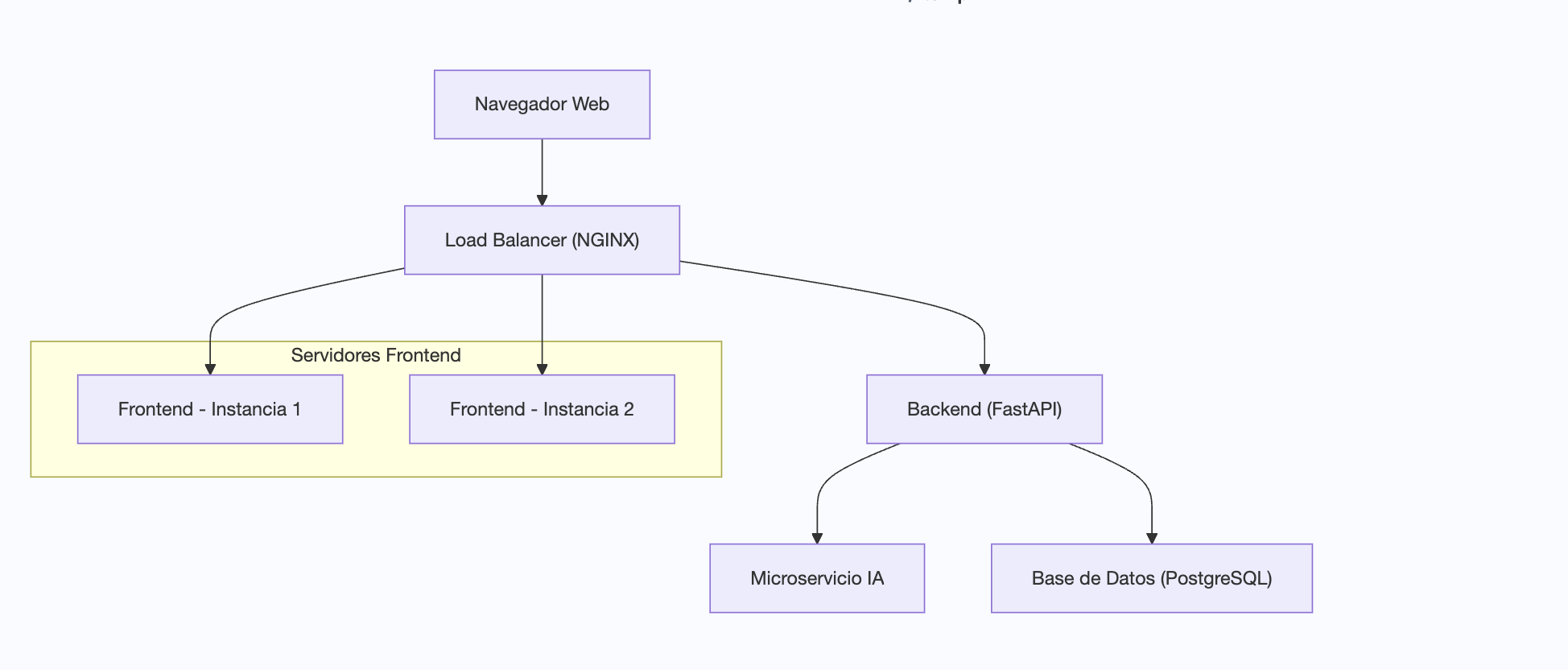
# 3. Arquitectura general del sistema

El sistema se basa en una arquitectura cliente-servidor desacoplada, que separa las responsabilidades por componente. El flujo general es el siguiente: el usuario accede al frontend para autenticarse y subir un archivo; este archivo es enviado al backend, que lo valida y reenvía al microservicio de IA para su procesamiento. Finalmente, los resultados se almacenan en la base de datos y se muestran al usuario.

Esta separación lógica permite mantener cada parte del sistema independiente, facilitando actualizaciones, cambios y pruebas sin comprometer el resto de los módulos.

* **Frontend (NextJS)**: interfaz gráfica responsiva diseñada para experiencia intuitiva.
* **Backend (FastAPI)**: orquesta las operaciones, maneja usuarios, autenticación y conexión con IA y BD.
* **Microservicio IA (FastAPI)**: especializado en recibir matrices de datos, preprocesarlas y generar una predicción por fila.
* **Base de datos (PostgreSQL)**: almacena información persistente sobre usuarios, resultados e historial de acciones.
* **Load Balancer (NGINX)**: puerta de entrada protegida que distribuye las peticiones externas hacia los servicios internos.

### Diagrama de arquitectura



# 4. Componentes y configuración

## 4.1 Frontend

El frontend fue construido con *NextJS* por su capacidad de desarrollar interfaces modernas y rápidas. Se aplicó Tailwind *CSS* para mantener consistencia visual con un diseño minimalista y adaptable.

* La navegación se estructura mediante react-router-dom, permitiendo rutas protegidas condicionales al tipo de usuario.
* El envío de archivos *CSV* al backend se realiza mediante fetch, usando cookies para mantener la sesión:

const formData = new FormData();  
formData.append("file", archivo);  
  
fetch("http://172.28.69.192:443/predict", {  
 method: "POST",  
 body: formData,  
 credentials: "include"  
});

* El sistema permite también el uso de un archivo de prueba y la paginación de resultados.

## 4.2 Backend (FastAPI)

El backend expone *APIs REST*, protege las rutas mediante *JWT* y facilita la interacción con la base de datos y el microservicio de IA.

* Al iniciar sesión, se genera un token y se guarda en una cookie *HTTP-only* para evitar accesos mediante *JavaScript*:

@app.post("/login")  
def login(user: LoginRequest):  
 token = create\_jwt(user)  
 response.set\_cookie("access\_token", token, httponly=True)  
 return {"msg": "ok"}



* Para proteger endpoints de administrador, se verifica el rol:

def verify\_admin(user=Depends(get\_current\_user)):  
 if not user.is\_admin:  
 raise HTTPException(status\_code=403)  
 return user

* Rutas principales:
  + /predict: recibe y reenvía el *CSV* a IA.
  + /history: consulta predicciones pasadas del usuario.
  + /admin/dashboard: permite visualizar estadísticas como cultivos más recomendados.

## 4.3 Microservicio IA

El microservicio especializado en predicción está desplegado como una aplicación *FastAPI* en un puerto distinto. Sólo acepta solicitudes autenticadas mediante cabecera:

@app.middleware("http")  
**async** **def** check\_api\_key(request: Request, call\_next):  
 **if** request.headers.get("x-api-key") != os.getenv("INTERNAL\_API\_KEY"):  
 **raise** HTTPException(status\_code=403)  
 **return** **await** call\_next(request)

* Recibe el *CSV*, convierte a *DataFrame*, lo normaliza si es necesario, y devuelve un JSON con predicciones por fila.
* Se entrenó un modelo con *Pytorch* con datos sintéticos para prueba de concepto.

## 4.4 Base de datos (PostgreSQL)

La base de datos sigue un diseño relacional, cuyas entidades principales son **users**, **app\_sessions**, **predictions** y **activity\_logs**.

La tabla users almacena la información del perfil del usuario, mientras que app\_sessions gestiona el ciclo de vida de las sesiones para la autenticación. Por su parte, las tablas predictions y activity\_logs registran las operaciones y la actividad que los usuarios realizan dentro de la aplicación.

CREATE TABLE users (

id SERIAL PRIMARY KEY,

workos\_user\_id VARCHAR(255) UNIQUE,

email VARCHAR(255) UNIQUE NOT NULL,

full\_name VARCHAR(255),

is\_admin BOOLEAN DEFAULT FALSE,

is\_active BOOLEAN DEFAULT TRUE,

created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP

);

CREATE TABLE app\_sessions (

id SERIAL PRIMARY KEY,

user\_id INTEGER NOT NULL,

workos\_user\_id VARCHAR(255) NOT NULL,

workos\_session\_id VARCHAR(255) UNIQUE NOT NULL,

encrypted\_refresh\_token TEXT NOT NULL,

ip\_address VARCHAR(45),

user\_agent TEXT,

issued\_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT\_TIMESTAMP,

refresh\_token\_expires\_at TIMESTAMP WITH TIME ZONE NOT NULL,

created\_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT\_TIMESTAMP,

updated\_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT\_TIMESTAMP,

CONSTRAINT fk\_user

FOREIGN KEY(user\_id)

REFERENCES users(id)

ON DELETE CASCADE

);

CREATE TABLE predictions (

id SERIAL PRIMARY KEY,

user\_id INTEGER REFERENCES users(id),

file\_name TEXT,

result TEXT,

timestamp TIMESTAMP DEFAULT now()

);

CREATE TABLE activity\_logs (

id SERIAL PRIMARY KEY,

user\_id INTEGER REFERENCES users(id),

action TEXT,

timestamp TIMESTAMP DEFAULT now()

);

CREATE INDEX idx\_app\_sessions\_user\_id ON app\_sessions(user\_id);

CREATE INDEX idx\_app\_sessions\_workos\_session\_id ON app\_sessions(workos\_session\_id);

CREATE INDEX idx\_app\_sessions\_refresh\_token\_expires\_at ON app\_sessions(refresh\_token\_expires\_at);

CREATE OR REPLACE FUNCTION update\_updated\_at\_column()

RETURNS TRIGGER AS $$

BEGIN

NEW.updated\_at = NOW();

RETURN NEW;

END;

$$ language 'plpgsql';

CREATE TRIGGER update\_app\_sessions\_updated\_at

BEFORE UPDATE ON app\_sessions

FOR EACH ROW

EXECUTE FUNCTION update\_updated\_at\_column();



## 4.5 Load Balancer (NGINX)

*NGINX* actúa como puerta de entrada. Su configuración asegura el acceso cifrado al backend:

# Define el grupo de servidores web (front-end) para balanceo de carga.

# Puertos actualizados según tu indicación.

upstream frontend\_servers {

server 172.28.69.227:3000; # Instancia bab\_front

server 172.28.69.127:3001; # Instancia bab\_front\_02

}

server {

listen 443 ssl;

server\_name 172.28.69.192;

access\_log /var/log/nginx/backend\_access.log;

ssl\_certificate /etc/nginx/ssl/selfsigned.crt;

ssl\_certificate\_key /etc/nginx/ssl/selfsigned.key;

ssl\_protocols TLSv1.2 TLSv1.3;

ssl\_ciphers HIGH:!aNULL:!MD5;

# --- Rutas de la aplicación (Locations) ---

# 1. Peticiones al Frontend (todo lo que no sea /api/)

location / {

proxy\_pass http://frontend\_servers;

proxy\_set\_header Host $host;

proxy\_set\_header X-Real-IP $remote\_addr;

proxy\_set\_header X-Forwarded-For $proxy\_add\_x\_forwarded\_for;

proxy\_set\_header X-Forwarded-Proto $scheme;

}

# 2. Peticiones a la API (todo bajo /api/)

location /api/ {

proxy\_pass http://172.28.69.182:9000;

proxy\_set\_header Host $host;

proxy\_set\_header X-Real-IP $remote\_addr;

proxy\_set\_header X-Forwarded-For $proxy\_add\_x\_forwarded\_for;

proxy\_set\_header X-Forwarded-Proto $scheme;

}

}

Gracias al uso de *PAT* en el switch, se expone el puerto 9443 en la ip del switch: **https://10.49.12.46:9943/** hacia el 443 interno del load balancer, manteniendo control de acceso a nivel de red.

# 5. Justificación del diseño

Cada decisión técnica en este sistema responde a los siguientes principios:

* **Separación de responsabilidades**: mantiene independencia entre módulos y facilita su mantenimiento.
* **Escalabilidad horizontal**: posibilidad de replicar microservicios o backend sin afectar a otros componentes.
* **Seguridad**: validación de origen, cookies *HTTP-only*, cabeceras seguras, uso de *API Keys* y control de red por *NAT*.
* **Auditoría y trazabilidad**: todas las acciones clave se registran en logs internos para análisis o depuración.
* **Entorno realista**: se optó por una infraestructura local con *OpenStack* y *Switch L3*, simulando un ambiente de producción.

# 6. Tabla de tecnologías utilizadas

| **Componente** | **Tecnología** | **Propósito** |
| --- | --- | --- |
| **Frontend** | *NextJS* | *Interfaz web responsiva* |
| **Backend** | *FastAPI (Python)* | *API REST, autenticación y lógica de negocio* |
| **IA** | *FastAPI + Pandas + Pytorch + Numpy* | *Predicción de cultivos por entrada de datos estructurada* |
| **Base de Datos** | *PostgreSQL* | *Almacenamiento persistente* |
| **Load Balancer** | *NGINX* | *Redirección segura de tráfico* |
| **Seguridad** | *JWT, API Key, HTTPS, WorkOS* | *Autenticación y protección de servicios* |
| **Infraestructura** | *OpenStack + Switch Cisco* | *Entorno controlado y replicable* |