# Universidad Tecnológica de Aguascalientes

**Tecnologías de la información**

**Ingeniería en desarrollo y gestión de software**

## EXAMEN PRÁCTICO

**Gestión del Proceso de Desarrollo de Software**

**Froylan Alonso Pérez Alanís**

**Misael Abrahm Rodríguez Valtierra** 10B

# Contenido

# Análisis del caso

### Desafíos de Comunicación y Colaboración

- **Coordinación entre equipos distribuidos:** Diferencia horaria, agendas y disponibilidad de los miembros.
- **Barreras de comunicación:** Uso inconsistente de canales (correo, chat, videollamadas) puede generar pérdida de información.
- **Roles y responsabilidades poco claros:** Riesgo de duplicar tareas o dejar pendientes sin asignación.
- **Falta de retroalimentación rápida:** Dificultad para detectar errores o bloqueos en tiempo oportuno.

### Desafíos de Gestión Ágil

- **Necesidad de entregas frecuentes:** Adaptar procesos para sprints cortos y releases continuos.
- **Priorización de requerimientos:** Balancear entre lo urgente (dirección/cliente) y lo importante (calidad, deuda técnica).
- **Adaptación cultural al cambio:** No todos los miembros pueden estar familiarizados con metodologías ágiles.
- **Gestión de dependencias entre equipos:** Sincronización de módulos que se desarrollan en paralelo.

### Desafíos Técnicos

- **Implementación de integración continua (CI/CD):** Configuración de pipelines automatizados y pruebas unitarias/integración.
- **Control de versiones distribuido:** Riesgo de conflictos en ramas y fusiones de código.
- **Homologación de entornos de desarrollo:** Evitar el clásico “funciona en mi máquina”.
- **Gestión de calidad de software:** Necesidad de pruebas automatizadas robustas para mantener la velocidad sin sacrificar calidad.

# Selección y configuración de herramientas

| Tipo de herramienta | Herramienta seleccionada | Función en el proyecto | Configuración básica | Justificación |
|---|---|---|---|---|
| **Comunicación síncrona/asíncrona** | Slack | Permite la comunicación instantánea y asincrónica entre miembros de los equipos distribuidos. | Creación de canales temáticos (#backend, #frontend, #qa, #general), integración con GitHub y Jira para notificaciones automáticas. | Elegido frente a Microsoft Teams por su interfaz más ágil y variedad de integraciones útiles en entornos DevOps. |
| **Gestión de tareas y progreso** | Jira | Organización del trabajo con tableros Kanban/Scrum, seguimiento de épicas e historias de usuario. | Creación de proyectos ágiles, definición de sprints, workflows personalizados, integración con repositorios GitHub. | Se prefiere sobre Trello porque tiene mayor soporte para metodologías ágiles complejas, métricas e informes de progreso. |
| **Control de versiones** | GitHub | Centraliza el repositorio de código, manejo de ramas y control de versiones. | Configuración de ramas protegidas (main, develop), revisión obligatoria de PR, integración con GitHub Actions. | GitHub es estándar en la industria, facilita colaboración remota y tiene integración nativa con CI/CD. |
| **CI/CD** | GitHub Actions | Automatiza pruebas, compilación y despliegues. | Configuración mediante archivos .yml en .github/workflows/, definición de jobs para build, test y deploy. | Se elige sobre Jenkins por su integración nativa con GitHub y menor complejidad de configuración inicial. |
| **Pruebas automatizadas (unitarias/integración/UI/API)** | PyTest, Postman, Cypress | Validación de calidad de código (Python), pruebas de APIs REST, pruebas end-to-end de la interfaz. | PyTest configurado en scripts de test del pipeline CI/CD; Postman collections para pruebas de API; Cypress con configuración en cypress.json. | Se seleccionan porque cubren diferentes niveles de prueba: unitarias, integración, API y front-end. Alternativas como Selenium fueron descartadas por mayor complejidad de setup. |
| **Monitoreo y métricas** | Prometheus y Grafana | Prometheus recolecta métricas de rendimiento; Grafana | Despliegue de Prometheus como servicio, configuración de exporters; Grafana | Se prefieren frente a alternativas como Datadog o New Relic por ser open |

| | | | | |
|---|---|---|---|---|
| | | crea dashboards visuales para análisis. | con dashboards conectados a Prometheus. | source, flexibles y con gran comunidad. |

## Diseño del plan de pruebas

| Sprint | Funcionalidades entregadas | Entregables | Tipos de prueba aplicados | Justificación |
|---|---|---|---|---|
| **Sprint 1 – Configuración inicial y autenticación** | Configuración del entorno de desarrollo.<br>Login y registro de usuarios.<br>Definición del esquema de base de datos. | Entorno de desarrollo accesible.<br>Sistema de autenticación funcional (login/registro).<br>Esquema inicial de base de datos validado.<br>Casos de prueba básicos de autenticación.<br>Documentación técnica inicial. | **Pruebas unitarias:** Validar funciones de login, registro y conexión a BD.<br>**Pruebas de integración:** Comprobar interacción entre backend, BD y autenticación.<br>**Pruebas de aceptación:** Validar que el flujo de login y registro cumple con lo esperado por el usuario. | Se aplican pruebas unitarias para asegurar calidad de los módulos básicos; integración para confirmar conexión correcta con la base de datos; aceptación porque el login es crítico para el acceso al sistema. |
| **Sprint 2 – Módulo de gestión de productos** | Implementación CRUD de productos.<br>Interfaz básica de productos.<br>Automatización inicial de pruebas. | CRUD de productos completo.<br>Interfaz básica funcional- Pruebas automatizadas en autenticación y productos.<br>Código validado en GitHub con PRs revisadas. | **Pruebas unitarias:** Validar operaciones CRUD (crear, leer, actualizar, eliminar).<br>**Pruebas de integración:** Confirmar correcto funcionamiento entre interfaz, backend y BD.<br>**Pruebas funcionales:** Validar que el CRUD cumple con los requisitos del usuario.<br>**Pruebas de regresión:** Verificar que los cambios en productos no afectaron el login/registro. | La complejidad aumenta, por lo que además de unitarias e integración, se agregan pruebas funcionales para validar procesos completos y regresión para proteger las funcionalidades críticas ya entregadas. |

| | | | | |
|---|---|---|---|---|
| **Sprint 3 – Reportes y despliegue en staging** | Implementación de módulo de reportes.<br><br>Mejoras UI/UX.<br><br>Entorno de staging. | Módulo de reportes operativo.<br><br>UI/UX mejorada.<br><br>Entorno de staging activo para pruebas integrales. | **Pruebas unitarias:** Validar consultas y generación de reportes.<br><br>**Pruebas de integración:** Comprobar interacción entre reportes, productos y base de datos.<br><br>**Pruebas funcionales:** Validar experiencia de usuario con la nueva UI/UX.<br><br>**Pruebas de regresión:** Garantizar que las mejoras no afecten autenticación ni CRUD.<br><br>**Pruebas de aceptación:** Validar con el cliente la utilidad y presentación de reportes. | En este sprint se agregan pruebas de aceptación porque los reportes son un entregable clave para negocio; regresión asegura estabilidad; funcionales para validar experiencia final; staging permite pruebas integrales antes del despliegue oficial. |

# Redacción de casos de prueba

**Prueba manual**

**Nombre de la prueba:** Validación visual de interfaz de productos (UI básica)
**Tipo de prueba:** Manual / Funcional
**Objetivo:** Verificar que la interfaz de gestión de productos muestre correctamente los elementos principales (botones, formularios, listas) y la navegación sea intuitiva.
**Entradas y condiciones iniciales:**

1. La aplicación está desplegada en entorno local o staging.
2. Acceso válido con un usuario autenticado.

**Pasos:**

1. Ingresar al módulo de gestión de productos.
2. Verificar que los botones de **crear**, **editar** y **eliminar** sean visibles.
3. Confirmar que los formularios de creación/edición aparecen correctamente.
4. Revisar que las listas de productos se carguen en pantalla.

**Resultado esperado:** La UI se muestra completa, con botones y formularios funcionales y navegación fluida.
**Herramienta utilizada:** Navegador web.
**Ubicación en el repositorio:** /tests/manual/ui_products.md

**Pruebas automatizadas**

**Nombre de la prueba:** Registro de usuario con datos válidos
**Tipo de prueba:** Automática / Unitarias e Integración
**Objetivo:** Validar que un usuario pueda registrarse y que sus datos se guarden en la base de datos correctamente.
**Entradas y condiciones iniciales:**

1. Formulario de registro con datos válidos (ejemplo: correo válido, contraseña fuerte, nombre).
2. Conexión a la base de datos activa.

**Pasos:**

1. Completar formulario de registro con datos correctos.
2. Enviar solicitud a la API.
3. Verificar respuesta de la API.
4. Consultar la base de datos para confirmar que el usuario fue creado.

**Resultado esperado:** Usuario registrado exitosamente, confirmación positiva de la API y persistencia en la base de datos.
**Herramienta utilizada:** PyTest / Cypress.
**Ubicación en el repositorio:** /tests/auth/test_register.py

**Nombre de la prueba:** Login con credenciales válidas
**Tipo de prueba:** Automática / Integración
**Objetivo:** Validar que el sistema permita autenticación con usuario y contraseña correctos.
**Entradas y condiciones iniciales:**

1. Usuario existente registrado previamente.
2. Credenciales válidas (correo y contraseña correctos).

**Pasos:**

1. Ingresar usuario y contraseña en el formulario de login.
2. Enviar la solicitud de autenticación a la API.
3. Verificar que se genera un token JWT válido.
4. Confirmar que la sesión se inicia correctamente.

**Resultado esperado:** Login exitoso y acceso autorizado al sistema.
**Herramienta utilizada:** PyTest / Cypress.
**Ubicación en el repositorio:** /tests/auth/test_login_valid.py

**Nombre de la prueba:** Crear producto
**Tipo de prueba:** Automática / Unitarias e Integración
**Objetivo:** Validar la creación de un producto nuevo y su persistencia en la base de datos.
**Entradas y condiciones iniciales:**

1. Usuario autenticado en el sistema.
2. Formulario de creación de producto con datos válidos (ejemplo: nombre, precio, stock).

**Pasos:**

1. Completar el formulario de creación de producto.
2. Enviar la solicitud a la API.
3. Verificar respuesta positiva de la API.
4. Consultar base de datos y confirmar que el producto existe.
5. Revisar que el producto aparezca en la lista de la UI.

**Resultado esperado:** Producto creado exitosamente, reflejado en la base de datos y visible en la UI.
**Herramienta utilizada:** PyTest / Cypress.
**Ubicación en el repositorio:** /tests/products/test_create_product.py

# Flujo de trabajo para el control de versiones

### Flujo Git

1. **Rama principal (main)**
    - Contendrá el código estable y listo para despliegue.
    - Estará protegida:
        - No se podrá hacer push directo.
        - Solo se permitirán pull requests revisados y aprobados.
2. **Ramas por sprint**
    - Cada iteración tendrá su propia rama, por ejemplo:
        - sprint-1
        - sprint-2
        - sprint-3
    - En estas ramas se integrarán las funcionalidades trabajadas en cada sprint antes de fusionarse en main.
3. **Ramas por funcionalidad**

- Cada nueva característica se desarrollará en una rama específica:
    - feature/login
    - feature/inventario
    - feature/reportes
- Estas ramas se crean a partir de la rama del sprint correspondiente.

4. **Uso de Pull Requests (PRs)**
    - Todo cambio debe llegar a la rama del sprint a través de un pull request.
    - Cada pull request requiere al menos una revisión obligatoria de un compañero de equipo.
    - Los PRs deben incluir descripción del cambio, pruebas realizadas y relación con las tareas del sprint.
5. **Etiquetas para versiones estables**
    - Cuando un sprint sea aprobado e integrado en main, se generará una etiqueta de versión:
        - v1.0: Primera versión estable con login y autenticación.
        - v1.1: Versión con gestión de productos.
        - v1.2: Versión con reportes y despliegue en staging.

**Integración de pruebas con GitHub Actions**

- Se utilizará GitHub Actions para la integración continua (CI).
- Cada vez que un desarrollador haga un commit o un pull request, se ejecutará un *workflow* con las siguientes etapas:

1. **Compilación y linting**
    - Validar que el código sigue los estándares definidos (Flake8 para backend en Python).
2. **Pruebas unitarias**
    - Ejecución automática de pruebas unitarias para verificar la funcionalidad de cada módulo.
3. **Pruebas de integración**
    - Validar interacción entre componentes clave (autenticación, base de datos, inventario, API, etc.).
4. **Reporte de cobertura**

- Se generará un reporte de cobertura de pruebas que será visible en GitHub.

5. **Condición para fusión**
    - Solo si todas las pruebas pasan de forma satisfactoria, el pull request podrá ser aprobado y fusionado.

# Estrategia de despliegue

### Pipeline CI/CD

El pipeline se gestionará con GitHub Actions y tendrá las siguientes fases:

1. **Checkout del código**
    - Descarga automática del repositorio desde GitHub.
    - Validación de la rama y la versión a construir.
2. **Instalación de dependencias**
    - Instalación de librerías necesarias para frontend (Node.js, npm) y backend (Python, FastAPI).
    - Configuración de variables de entorno.
3. **Ejecución de pruebas**
    - **Pruebas unitarias**: validan componentes individuales.
    - **Pruebas de integración**: comprueban la interacción entre módulos.
    - **Pruebas funcionales**: simulan escenarios de usuario.
    - Generación de reportes de cobertura.
4. **Empaquetado del software**
    - Construcción de imágenes Docker del backend y frontend.
    - Publicación de imágenes en un registro privado (Docker Hub / GitHub Packages).
5. **Despliegue en entorno de Staging**
    - Se aplica de forma automática tras aprobar un pull request en la rama del sprint.
    - Uso de Terraform para aprovisionar infraestructura en la nube.
    - Validación del correcto despliegue con pruebas de regresión y aceptación.
6. **Despliegue en entorno de Producción**

- Solo se ejecuta al fusionar con la rama main y al aprobar la versión estable con etiqueta (v1.0, v1.1, etc.).
- Requiere que todas las pruebas en staging hayan sido exitosas.
- Infraestructura gestionada con Terraform, asegurando consistencia entre entornos.

## Entornos utilizados

- **Staging**
  - Réplica controlada de producción.
  - Permite validar nuevas funcionalidades y detectar errores antes de la liberación final.
  - Accesible solo al equipo de desarrollo y QA.
- **Producción**
  - Entorno final usado por clientes.
  - Despliegue controlado con aprobación manual tras validación en staging.

## Condiciones para desplegar

1. Todas las pruebas (unitarias, integración, funcionales) deben pasar.
2. El pull request debe ser aprobado por al menos un revisor.
3. En producción:
   - Debe existir una etiqueta de versión estable (vX.Y).
   - Despliegue debe ser validado previamente en staging.

**Herramientas complementarias**

- **Docker**
  - Empaquetado de la aplicación en contenedores portables.
  - Facilita despliegue idéntico en todos los entornos.
- **Terraform**
  - Infraestructura como código (IaC).
  - Automatiza la creación y configuración de servidores, redes y servicios en la nube.
- **Prometheus y Grafana**

- Prometheus recolecta métricas de uso (CPU, memoria, errores).
- Grafana visualiza en dashboards el estado de la aplicación.
- Monitoreo continuo post-despliegue para anticipar fallos y garantizar disponibilidad.

## Enlace del repositorio

https://github.com/MisaelRodriguezDev/GPDS_U1_ep