

Nombres estudiantes: Betancourt Alison, Yunga Mateo
Fecha: 02/12/25
Aplicación objetivo: Byte_Click (Plataforma E-Commerce de productos Tecnológicos)

Contenido

PROYECTO 001 – SCM.....	2
1. FASE 1.....	3
1.1 CONFIGURACIÓN DEL ENTORNO Y REPOSITORIO.....	3
1.2 DEFINICIÓN DEL FLUJO DE TRABAJO	4
1.3 GESTIÓN DE ARTEFACTOS (NO CÓDIGO)	5
2. FASE 2.....	5
2.1. GESTIÓN DE CAMBIOS EN ACCIÓN	5
2.2. INTEGRACIÓN CONTINUA	6
2.3. TRAZABILIDAD Y GESTIÓN DE LINEAS BASE	7
3. FASE 3.....	7
3.1. GESTIÓN DE RELEASES (CD)	7
3.2. PLAN DE MANTENIMIENTO.....	8

PROYECTO 001 – SCM

Nombre del Proyecto: Byte_Click

Tipo: Aplicación Web E-Commerce

Descripción: Byte_Click es una plataforma de comercio electrónico que permite a los usuarios navegar por un catálogo de productos tecnológicos, agregarlos a un carrito de compras y realizar pagos seguros, incluye un panel de administración para gestionar inventario, órdenes y clientes.

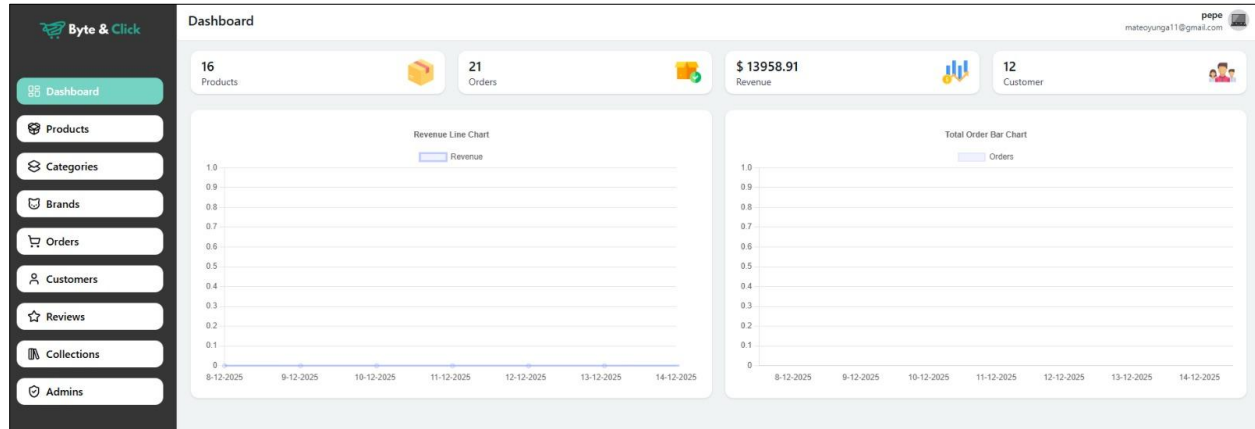


Ilustración 1. Panel de administración.

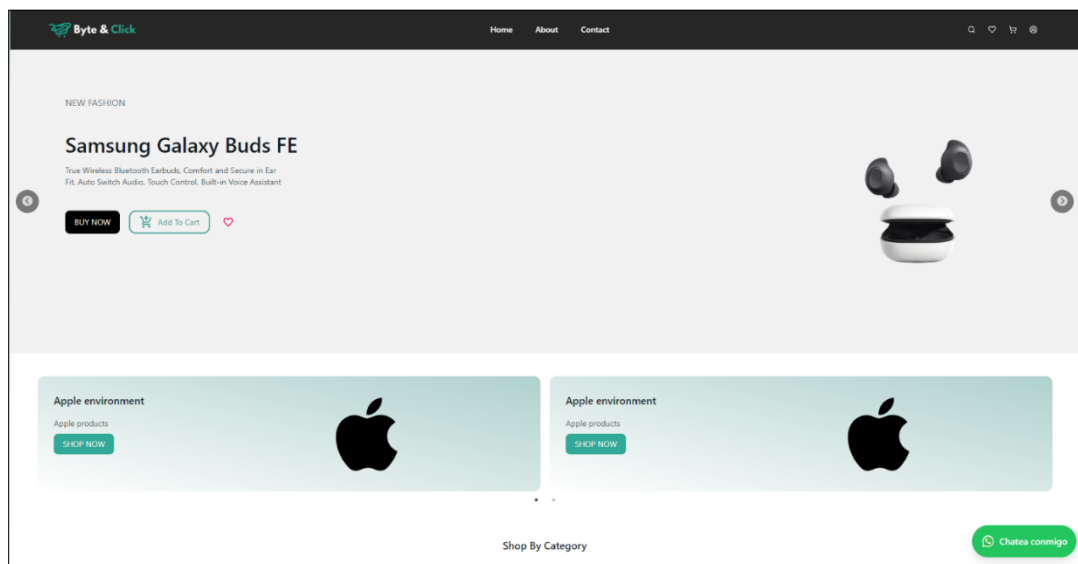


Ilustración 2. Panel de usuario.

Stack Tecnológico:

- ✦ **Frontend:** Next.js 15 (React framework con SSR)
- ✦ **Estilos:** Tailwind CSS + Material-UI

- ⤴ **Backend/Base de datos:** Firebase (Firestore + Authentication)
- ⤴ **Pagos:** Stripe
- ⤴ **Búsqueda:** Algolia (search engine)
- ⤴ **Hosting:** Vercel

Características Principales:

- ⤴ Autenticación de usuarios (login/registro)
- ⤴ Catálogo de productos con búsqueda y filtros
- ⤴ Carrito de compras persistente
- ⤴ Checkout con integración Stripe
- ⤴ Panel de administración (CRUD de productos, gestión de órdenes)
- ⤴ Sistema de reviews y ratings

1. FASE 1

1.1 CONFIGURACIÓN DEL ENTORNO Y REPOSITORIO

Herramientas Seleccionadas: Se utilizarán herramientas estándar de la industria para el desarrollo. Git será el sistema de control de versiones principal, y GitHub actuará como la plataforma de hosting centralizada para el repositorio. Para la gestión de dependencias del proyecto se empleará npm, asegurando la reproducibilidad del entorno mediante el archivo package-lock.json.

Estructura de Repositorio: Se ha decidido implementar un Monorepo único llamado Byte_Click para albergar todo el código del proyecto, esta estructura se justifica principalmente porque el frontend basado en Next.js y el backend comparten la misma base de datos y los mismos modelos de datos, lo que hace ineficiente separarlos. Dada la naturaleza de tamaño mediano del proyecto, la complejidad adicional de gestionar múltiples repositorios no se justifica, siendo el monorepo la opción que facilita el versionamiento unificado, permitiendo que una sola etiqueta englobe y lance conjuntamente las versiones completas tanto del frontend como de cualquier lógica de backend.

Configuración de la Rama Principal: La rama principal del repositorio se denominará main, y estará sujeta a estrictas reglas de protección de rama de GitHub para asegurar la calidad y estabilidad del código base, estas protecciones incluyen prohibir los pushes directos para forzar el flujo de trabajo de Pull Request, requerir al menos una aprobación en la revisión por pares, y exigir que pasen todos los status checks (incluyendo la ejecución exitosa de pruebas unitarias y análisis de linting de CI), además, se prohibirá el force push y se requerirá que la rama de origen del Pull Request esté actualizada con main antes de permitir la fusión, garantizando así que todo el código cumpla con los estándares de calidad antes de ser integrado.

Razón de las Protecciones: La aplicación rigurosa de protecciones en la rama principal es crítica debido a que Byte_Click es una plataforma de comercio electrónico que maneja transacciones financieras y datos sensibles, un cambio no revisado o defectuoso tiene riesgos directos y graves, podría llevar a la exposición de credenciales críticas, podría romper el flujo de pagos con Stripe, resultando en una pérdida económica directa, o podría introducir vulnerabilidades de seguridad serias, como inyección de código o fallos en las reglas de seguridad de Firebase o las consultas de Algolia, comprometiendo los datos de los usuarios.

Archivos Sensibles Excluidos (.gitignore): Se excluirán los archivos de configuración de entorno .env.local y .env, ya que contienen las claves API sensibles de servicios críticos como Firebase, Stripe y Algolia, que nunca deben ser subidas al repositorio público, adicionalmente, se excluirán las carpetas de salida de compilación .next/ y .vercel/, así como el directorio node_modules/ que contiene todas las dependencias del proyecto, ya que son outputs generados automáticamente que pueden ser reconstruidos en cualquier momento mediante el comando npm install.

1.2 DEFINICIÓN DEL FLUJO DE TRABAJO

Workflow: El proyecto Byte_Click adoptará el GitHub Flow como su estrategia de flujo de trabajo principal, esta elección se justifica por la naturaleza del proyecto, una aplicación web con despliegue continuo a través de Vercel que no requiere la coordinación de releases complejas, dado que se espera que el equipo de desarrollo sea pequeño, la simplicidad de GitHub Flow se prioriza sobre flujos más complejos, este modelo permite una iteración rápida, desarrollar una característica (feature), crear un Pull Request (PR), fusionar (merge) y desplegar (deploy) el mismo día, lo que se alinea perfectamente con la necesidad de agilidad y la entrega continua.

Descripción del Flujo:

1. La rama main siempre está en estado "desplegable".
2. Para toda nueva funcionalidad/bug:
 - ✦ Crear rama desde main con nombre descriptivo.
 - ✦ Desarrollar + commitear localmente.
 - ✦ Push a GitHub y abrir Pull Request.
 - ✦ Peer review + CI automático.
 - ✦ Merge a main → Auto-deploy a producción.

Nomenclatura de Ramas:

Tipo de rama	Prefijo	Ejemplo	Cuando Usar
Nueva funcionalidad	feature/	feature/algolia-product-search	Agregar capacidad nueva al sistema.
Corrección de bug	bugfix/	bugfix/cart-quantity-increment	Arreglar comportamiento incorrecto.
Mejora técnica	refactor/	refactor/firebase-queries-optimization	Mejorar código sin cambiar funcionalidad.
Documentación	docs/	docs/api-endpoints	Actualizar README, guías, comentarios.

Tabla 1. Especificaciones para nombrar las ramas.

Política de Integración:

- Prohibido hacer merge directo a main.
- Obligatoriamente todo cambio debe pasar por Pull Request.

1.3 GESTIÓN DE ARTEFACTOS (NO CÓDIGO)

La gestión de todos los artefactos de planificación y requisitos se centralizará utilizando GitHub Issues y Projects, la estrategia consiste en documentar cada requisito y funcionalidad como un GitHub Issue individual, asegurando que cada tarea esté formalizada, rastreable y enlazada al código. Para mantener la consistencia y la claridad, se aplicará una plantilla estandarizada a todos los Issues de tipo Feature Request, la cual debe especificar claramente, una descripción siguiendo el formato de historia de usuario, Como [tipo de usuario], quiero [acción] para [beneficio], los Criterios de Aceptación detallados para validar la funcionalidad, y un enlace directo al Diseño o Mockup asociado.

2. FASE 2

2.1. GESTIÓN DE CAMBIOS EN ACCIÓN

Definition of Done para PR

Un PR solo puede mergearse a main si cumple:

- ✦ Al menos 1 aprobación de peer review.
- ✦ Todos los tests unitarios y de integración pasan.
- ✦ Linting sin errores.
- ✦ Build exitoso.
- ✦ Sin conflictos con main.

- ⤴ PR referencia el Issue relacionado.

Proceso del Pull Request

1. Crear rama desde main.
2. Desarrollar y commitear con mensajes descriptivos.
3. Push a GitHub y abrir PR completando la plantilla.
4. Asignar reviewer y agregar labels apropiadas.
5. Reviewer aprueba o solicita cambios.
6. Una vez aprobado y CI verde se hace el merge.

Plantilla del Pull Request

<p><i>Descripción</i> Qué hace este PR.</p> <p><i>Issue Relacionado</i> Closes #[número].</p> <p><i>Tipo de Cambio</i> Bug fix, Nueva funcionalidad, Refactoring o Documentación.</p> <p><i>Checklist</i></p> <p><input type="checkbox"/> Código sigue guías de estilo.</p> <p><input type="checkbox"/> Self-review realizado.</p> <p><input type="checkbox"/> Documentación actualizada.</p> <p><input type="checkbox"/> Tests agregados/actualizados.</p> <p><input type="checkbox"/> Tests pasan localmente.</p>

Tabla 2. Plantilla de cómo se debe presentar un Pull Request.

2.2. INTEGRACIÓN CONTINUA

La estrategia de Integración Continua será implementada mediante GitHub Actions, utilizando un pipeline centralizado definido en el archivo `.github/workflows/ci.yml`. Este pipeline se ejecutará automáticamente en cada push a una rama de característica (feature) o a un Pull Request (PR), y es un requisito obligatorio para la fusión a la rama principal (main).

El proceso consta de cuatro etapas críticas que deben pasar exitosamente antes de que un PR pueda ser fusionado:

- ⤴ **Análisis de Código (Linting):** Se usarán ESLint para verificar el estilo de código y Prettier para garantizar el formato uniforme, el pipeline fallará si se detectan errores de estilo o formato obligatorios.

- ⬆ **Pruebas Unitarias:** Se ejecutarán pruebas exhaustivas utilizando Jest y React Testing Library, se ha establecido un mínimo de cobertura de código del 70%, y el pipeline fallará si cualquier prueba no pasa.
- ⬆ **Verificación de Compilación (Build):** Se ejecutará el comando `npm run build` para asegurar que la aplicación compile sin errores, esto es crucial para detectar problemas tempranos como imports rotos o errores de tipado incorrecto.
- ⬆ **Escaneo de Seguridad:** Se utilizará CodeQL para realizar un análisis estático del código fuente en busca de vulnerabilidades, y se ejecutará `npm audit` para verificar las dependencias contra CVE conocidos, el pipeline se bloqueará si se encuentran vulnerabilidades etiquetadas como críticas.

La regla de protección de rama de GitHub "Require status checks to pass" se aplicará estrictamente, impidiendo cualquier merge si el pipeline de CI falla en alguna de estas cuatro etapas.

2.3. TRAZABILIDAD Y GESTIÓN DE LINEAS BASE

La Trazabilidad en el desarrollo del proyecto `Byte_Click` se garantizará mediante la conexión explícita de cada elemento, desde la solicitud inicial hasta el despliegue final, esta conexión se logra utilizando la estrategia de Conventional Commits y la referencia cruzada de identificadores. El flujo clave será: Un requisito (Issue #45) es referenciado en los mensajes de los Commits ("feat: add price filter (closes #45)"), los cuales se agrupan en un Pull Request (PR #52), una vez fusionado a la rama main, este conjunto de cambios queda etiquetado y asociado a una Release específica, asegurando un historial claro y auditable de cuándo y por qué se introdujo cada cambio.

Para el versionamiento, se implementará la práctica de Conventional Commits con prefijos estandarizados, tales como:

feat: para nuevas funcionalidades

fix: para correcciones de bugs

docs: para documentación, refactor: para mejoras internas

test: para cambios en pruebas, y chore: para tareas de mantenimiento.

Finalmente, la Gestión de Línea Base se manejará a través de la creación de Tags de Git para marcar hitos de estabilidad o funcionalidad completada, los tags se utilizarán para definir versiones clave del proyecto, como `v1.0-alpha` (prototipo inicial), `v1.0-beta` (funcionalidad core completa) y `v1.0.0` (primera versión estable en producción). Se crearán tags sistemáticamente al completar un milestone importante y obligatoriamente antes de cualquier despliegue a producción, permitiendo que el equipo pueda identificar y, si es necesario, revertir a una versión estable conocida.

3. FASE 3

3.1. GESTIÓN DE RELEASES (CD)

El sistema de versiones se adhiere al Versionamiento Semántico en formato MAJOR.MINOR.PATCH, el cual guía la comunicación de cambios:

- ⬆ **MAJOR:** Indica cambios incompatibles que requieren migración.

- ✦ MINOR: Indica nuevas funcionalidades compatibles con versiones anteriores.
- ✦ PATCH: Indica correcciones de bugs.

Estrategia de Despliegue de Tres Entornos (CD): Se implementará un pipeline de Despliegue Continuo utilizando Vercel para asegurar la calidad antes de llegar al usuario final.

Notas de Versión (Release Notes): Cada release oficial irá acompañada de un documento formal de Release Notes que asegura la transparencia, este documento detallará: las nuevas funcionalidades, los bugs corregidos, los breaking changes y las instrucciones de migración.

3.2. PLAN DE MANTENIMIENTO

El mantenimiento del proyecto Byte_Click se abordará de manera estructurada y proactiva, cubriendo los cuatro tipos esenciales para garantizar la longevidad y la seguridad del sistema.

a) Mantenimiento correctivo

El objetivo es corregir defectos reportados en el entorno de producción, distinguiendo entre bugs normales y críticos.

- ✦ **Proceso Estándar:** Los defectos se reportan mediante un Issue con la etiqueta bug, se les asigna una prioridad, y se corrigen mediante una rama bugfix/descripcion que se desprende de main, la fusión se planifica para el próximo release PATCH regular.
- ✦ **Proceso de Hotfixes Críticos:** Reservado para fallos que implican riesgo económico o pérdida de datos (App caída, fallo en pagos, vulnerabilidad explotable), se crea la rama hotfix/ directamente desde main, se aplica el fix mínimo (sin refactoring), se somete a un PR expedito, se fusiona, se lanza un deploy inmediato, y se incrementa el número PATCH.

b) Mantenimiento adaptativo

El objetivo es garantizar que el sistema se adapte continuamente a los cambios en el entorno tecnológico y regulatorio externo.

✦ **Calendario de Revisiones:**

- Trimestral: Revisión de APIs de terceros (Firebase, Stripe, Algolia) para actualizar SDKs y atender anuncios de deprecation.
- Semestral: Evaluación de la compatibilidad con nuevas versiones de frameworks mayores (Next.js, React, Material-UI) y funcionalidades de Vercel.
- Anual: Auditoría completa de seguridad.
- ✦ **Proceso:** El trabajo se inicia con un Issue etiquetado como adaptation, el despliegue de estos cambios se categoriza como MINOR si solo añade compatibilidad y MAJOR si rompe la compatibilidad anterior.

c) Mantenimiento perfectivo

Se centra en mejorar la calidad, legibilidad y rendimiento del código sin añadir funcionalidad nueva al usuario final, combatiendo la deuda técnica.

⤴ **Estrategia:** Se reservará el 15% del tiempo de desarrollo en cada sprint para abordar la deuda técnica.

⤴ **Tipos de Mejoras:**

- Refactoring: Eliminar duplicación de código y mejorar la legibilidad.

- Developer Experience: Mejorar la documentación interna y agregar types más específicos (TypeScript) o configurar pre-commit hooks.

⤴ **Priorización:** Se usa la etiqueta technical-debt en los Issues, la decisión se basa en métricas objetivas.

d) Mantenimiento preventivo

El objetivo es prevenir problemas de seguridad, fallos y obsolescencia antes de que afecten a la producción.

⤴ **Herramientas Automatizadas:**

- Dependabot (GitHub): Escanea package.json diariamente, creando PRs automáticos para actualizar dependencias con vulnerabilidades.

- npm audit: Ejecutado en el pipeline de CI, bloqueando el merge si detecta vulnerabilidades críticas o altas.

- CodeQL: Ejecuta análisis estático en cada PR para detectar fallos de seguridad como hardcoded secrets o vulnerabilidades de inyección.

- Lighthouse CI: Monitorea performance, SEO y accesibilidad, alertando si las métricas caen bajo el umbral establecido.

⤴ **Políticas Preventivas:** Las actualizaciones de seguridad deben aplicarse en menos de 7 días, se utilizan backups automáticos diarios de Firestore y herramientas de Monitoring (Vercel Analytics + Firebase Crashlytics) que alertan si la tasa de errores supera el 5% o si el tiempo de respuesta excede los 3 segundos.

⤴ **Auditorías Programadas:** Se realizan revisiones mensuales de logs y patrones de errores, Penetration Testing básico trimestral, y una auditoría de seguridad profesional anual.