SoftMarket S.A. ha crecido rápidamente sin una guía arquitectónica ni prácticas de ingeniería consistentes. El resultado es código duplicado, deployments frágiles y un aumento del 40% en errores de producción en los últimos 3 meses. Esta sección diagnostica las causas técnicas y operativas principales que provocan esos problemas.

Problemas de arquitectura y diseño

- Código monolítico y acoplado: funcionalidades de distinta responsabilidad (facturación, clientes, pagos) mezcladas en los mismos módulos.
- Duplicación de lógica: validaciones y cálculos replicados en varios endpoints.
- Ausencia de separación de capas (presentación, dominio, persistencia), lo que dificulta pruebas y despliegues.
- Falta de patrones o contratos (interfaces), lo que aumenta el coste de cambios.

Ejemplo de código malo:

```python
# services.py (ejemplo de código problemático)
def generar_factura(order, customer):
    # validar cliente
    if not customer.get("active"):
        raise Exception("Cliente inactivo")
    # calcular total e impuestos
    total = 0
    for item in order["items"]:
        total += item["price"] * item.get("qty", 1)
    tax = total * 0.19  # hardcoded IVA
    # aplicar descuento (duplicado en otro módulo)
    if order.get("coupon") == "PROMO10":
        total -= 10
    # persistir factura y enviar email
    db.insert("invoices", {...})
    send_email(customer["email"], "Tu factura")
```

Esta función mezcla validaciones, cálculos, persistencia y notificación en un solo lugar, violando el principio de responsabilidad única (SRP).

Problemas de versionamiento y colaboración

- Commits directos en main sin PR ni revisión.
- Ausencia de branch strategy y tags semánticos.
- No hay CI que sirva de “puerta” para evitar merges defectuosos.
- Resultado: errores llegan a producción con frecuencia y sin rastreo claro.

Violaciones a principios SOLID y falta de patrones

- SRP: clases y funciones con múltiples responsabilidades.
- OCP: modificar clases existentes para añadir impuestos o descuentos.
- DIP: dependencias directas de implementaciones (ej. db.insert acoplado al servicio).
- ISP/LSP: interfaces inexistentes y substituciones peligrosas.

Riesgos para la calidad del software

La situación actual de SoftMarket compromete directamente la calidad y estabilidad del producto.

La falta de arquitectura definida, el uso inadecuado de Git y la ausencia de principios de diseño han generado un entorno difícil de mantener y escalar.

Riesgos técnicos:

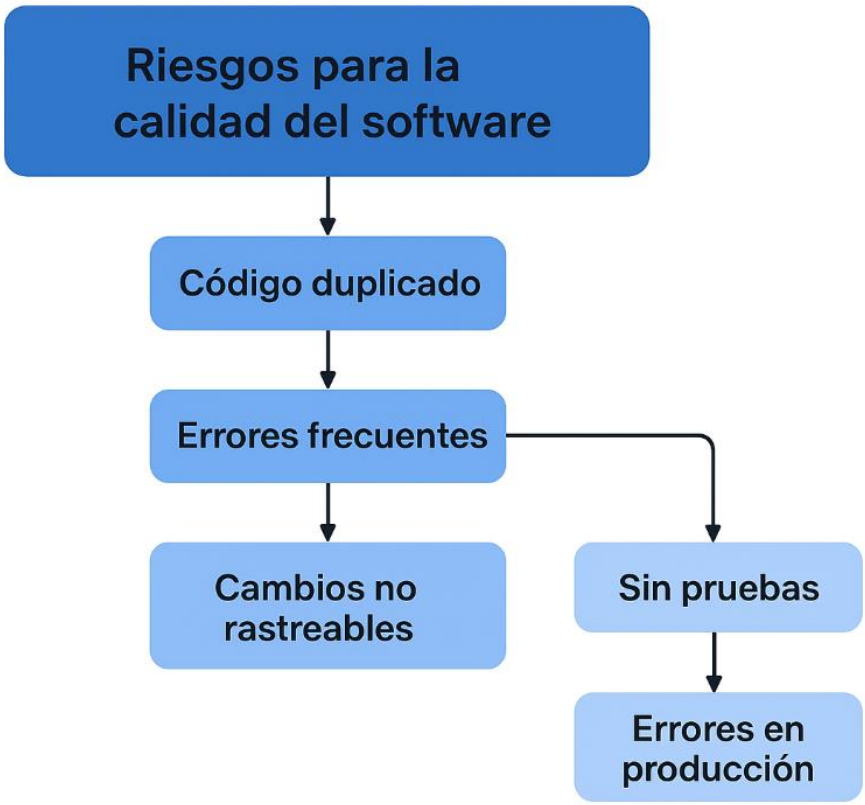
- Baja mantenibilidad: los cambios en un módulo afectan a otros, lo que incrementa el tiempo de desarrollo y la probabilidad de errores colaterales.
- Alta complejidad y duplicación: el mismo código está replicado en varios lugares, dificultando correcciones y generando inconsistencias.
- Escalabilidad limitada: al no existir una arquitectura modular, es complejo dividir el sistema en componentes independientes o reutilizables.
- Falta de trazabilidad: sin control de versiones claro, no es posible identificar fácilmente quién realizó un cambio ni cuándo.
- Ausencia de pruebas unitarias: los errores se detectan en producción en lugar de en etapas tempranas.

Riesgos operativos y de negocio:

- Aumento de errores en producción (+40% en 3 meses): afecta la confianza del cliente y la reputación de la empresa.
- Costos de mantenimiento elevados: el tiempo invertido en corregir errores supera al dedicado a desarrollar nuevas funciones.
- Retrasos en entregas: los conflictos de código y fallos al integrar nuevas funciones provocan re-trabajo y demoras.
- Dificultad para incorporar nuevos desarrolladores: la falta de documentación y estructura clara prolonga el tiempo de adaptación del personal nuevo.

Riesgo	Impacto	Consecuencia
Código duplicado	Medio	Inconsistencias y errores
Sin control de versiones	Alto	Cambios no rastreables, perdida de código
Sin pruebas unitarias	Muy alto	Errores en producción
Falta de arquitectura	Alto	Dificultad para escalar y mantener
Sin revisión de código	Medio	Los bugs pasan a producción sin control

Figura 1. Diagrama de riesgos para la calidad del software en SoftMarket



El diagrama ilustra cómo los problemas técnicos (duplicación de código, falta de control de versiones y ausencia de pruebas) se encadenan y derivan en errores en producción y mayores costos de mantenimiento.

Propuesta técnica integral

Con el fin de mejorar la calidad del código, reducir errores en producción y establecer un flujo de trabajo sostenible, se propone adoptar un conjunto de buenas prácticas centradas en tres ejes principales:

- Estrategia de versionamiento basada en GitFlow, para mejorar la colaboración, trazabilidad y control de cambios.
- Refactorización modular aplicando principios SOLID, comenzando con el módulo de facturación como caso piloto.
- Buenas prácticas complementarias, que incluyen pruebas automáticas, documentación técnica, guías de estilo y revisiones de código formales.

Estas medidas buscan sentar las bases de un proceso de desarrollo ordenado, escalable y orientado a la calidad.

Modelo GitFlow

GitFlow es una estrategia de ramas que define un flujo de trabajo estructurado para el control de versiones.

Permite mantener un entorno de desarrollo limpio, facilita el trabajo en equipo y reduce los errores al integrar nuevas funcionalidades.

Ramas principales

Rama	Proposito	Quien la usa
Main	Contiene versiones estables listas para producción	Líder técnico / CL
Develop	Rama base para desarrollo. Integra todas las nuevas funciones antes del release	Todo el equipo
Feature	Para desarrollar una funcionalidad o mejora en especifica	Desarrolladores
Release	Preparar versiones antes de pasar a produccion	QA / líder técnico
Hotfix	Corregir errores críticos en produccion	DevOps / líder técnico

Flujo de trabajo básico

1. Se crea una nueva rama desde develop para una funcionalidad:
2. 

3. Se desarrolla y se hacen commits con mensajes claros.
4. Al terminar, se abre un Pull Request hacia develop.
5. Una vez probado, se crea una rama release/x.x.x para preparar el despliegue.
6. Cuando QA valida, se hace merge a main y se crea un tag.
7. Si aparece un error crítico, se crea una rama hotfix/x.x.x desde main.

Normas de commits (Conventional Commits)

Establecer una convención clara para los mensajes de commit permite comprender fácilmente qué cambios se realizaron y por qué.

Formato estándar:

<tipo>(<área>): <breve descripción> [#ticket]

Ejemplos:

```
$ feat(facturacion): agregar cálculo de IVA al 19% [#123]
fix(api): corregir error de validación en endpoint de clientes [#77]
refactor(auth): mejorar estructura de middleware [#45]
```

Tipos recomendados:

feat → nueva característica

fix → corrección de error

refactor → mejora del código sin cambiar funcionalidad

docs → cambios en documentación

test → creación o mejora de pruebas

chore → mantenimiento o tareas menores

Revisión de código y manejo de conflictos

Todo cambio debe pasar por una revisión de código antes de ser fusionado.

Reglas mínimas:

- Ningún merge directo a main o develop.
- Se requiere al menos 1 aprobación (idealmente 2).
- CI debe aprobar (lint, tests, build).
- En caso de conflicto, el autor hace git rebase origin/develop antes del merge.

Checklist del Pull Request:

- [ ] Código formateado correctamente

- [ ] Tests ejecutados y aprobados

- [ ] Documentación actualizada

- [ ] Revisado por al menos un desarrollador

Aplicación de principios SOLID (Módulo de facturación)

Aquí es donde mostrarás cómo se mejora un módulo real con SOLID.

En este caso, puedes usar el módulo “Facturación”, refactorizado con interfaces y responsabilidades separadas.

Ejemplo de estructura:

```
facturacion/
 ├── models.py          (clase Invoice)
 ├── taxes.py           (ITaxCalculator, IvaCalculator)
 ├── discounts.py       (IDiscountStrategy, FixedDiscount)
 ├── repository.py      (IInvoiceRepository, SqlInvoiceRepository)
 ├── service.py         (InvoiceService)
 └── validator.py       (InvoiceValidator)
```

- **S (Single Responsibility):** cada clase hace una sola tarea.
- **O (Open/Closed):** se pueden agregar nuevos calculadores de impuestos sin modificar la clase principal.
- **L (Liskov):** las implementaciones pueden sustituir a la interfaz.
- **I (Interface Segregation):** interfaces pequeñas y específicas.
- **D (Dependency Inversion):** InvoiceService depende de abstracciones, no de implementaciones concretas.

Ejemplo de código:

```python
# Ejemplo refactorizado aplicando SOLID en el módulo de facturación

class ICalculator:
    def calculate(self, amount: float) -> float:
        raise NotImplementedError

class IvaCalculator(ICalculator):
    def calculate(self, amount: float) -> float:
        return amount * 0.19

class DiscountStrategy:
    def apply(self, total: float) -> float:
        return total  # implementación base

class FixedDiscount(DiscountStrategy):
    def __init__(self, amount: float):
        self.amount = amount
    def apply(self, total: float) -> float:
        return total - self.amount

class InvoiceService:
    def __init__(self, tax_calculator: ICalculator, discount: DiscountStrategy):
        self.tax_calculator = tax_calculator
        self.discount = discount

    def generate(self, subtotal: float):
        taxed = subtotal + self.tax_calculator.calculate(subtotal)
        total = self.discount.apply(taxed)
        return total
```

Este diseño aplica los principios SOLID, ya que permite extender el cálculo de impuestos o descuentos sin modificar el núcleo de la clase InvoiceService, mejorando la mantenibilidad y escalabilidad.

Buenas prácticas complementarias

Estilo de código

- Python: PEP8 + Black + Flake8
- JS: ESLint + Prettier
- Commits y branches en inglés (código más internacional).

Pruebas

- Framework: pytest
- Mínimo 70% cobertura.
- Pruebas unitarias + integración.

- CI ejecuta tests automáticamente en PR.

Documentación

- README.md con setup y flujo.
- CONTRIBUTING.md con normas de PR y commits.
- CHANGELOG.md con versiones y fechas.
- docs/ con diagramas y arquitectura.

Control de calidad / CI

Se integrará un pipeline de integración continua (CI) en GitHub Actions.

Figura 2. Modelo de versiónamiento GitFlow Propuesto para SoftMarket