



**UNIVERSIDAD NACIONAL DE EDUCACIÓN A
DISTANCIA**

**MÁSTER UNIVERSITARIO EN INVESTIGACIÓN EN
INGENIERÍA DE SOFTWARE Y SISTEMAS
INFORMÁTICOS**

TRABAJO FIN DE MÁSTER

**REDISEÑO ARQUITECTÓNICO PARA LA MEJORA
DE LA MANTENIBILIDAD, TESTABILIDAD Y
ESCALABILIDAD EN EL DESARROLLO DEL
MÓDULO DE GESTIÓN DE RECETAS «MEALIE»**

ALEJANDRO VILLARÍN PRIETO

17 de junio de 2026

Resumen

Este trabajo presenta una refactorización arquitectónica incremental del módulo de gestión de recetas del backend de Mealie, enfocada en mejorar la mantenibilidad, testabilidad y escalabilidad sin alterar su comportamiento externo. Aunque el módulo cumple funcionalmente su finalidad, presenta alto acoplamiento con infraestructura, mezcla de responsabilidades, dependencias de HTTP en persistencia y baja testabilidad aislada, lo que dificulta su evolución.

La intervención se estructura en fases basadas en la separación entre entrada, aplicación, dominio e infraestructura, introduciendo puertos y adaptadores, extrayendo casos de uso, corrigiendo dependencias impropias entre capas y aislando la política de permisos. La evaluación compara un *estado inicial* con sucesivas iteraciones mediante métricas de acoplamiento, complejidad y testabilidad, junto con pruebas unitarias y validación funcional.

Los resultados muestran mejoras medibles en desacoplamiento, explicitud de dependencias, eliminación de dependencias HTTP en persistencia y reducción de *puntos críticos*. En conjunto, la evidencia obtenida respalda que un enfoque incremental puede producir mejoras estructurales verificables en sistemas reales en evolución.

Palabras clave: refactorización arquitectónica incremental, arquitectura hexagonal, mantenibilidad del software, testabilidad del software, separación de responsabilidades.

Abstract

This thesis presents an incremental architectural refactoring of the recipe management module of the Mealie backend, aimed at improving maintainability, testability, and scalability without altering its external behavior. Although the module meets its functional requirements, it exhibits strong coupling to infrastructure, mixed responsibilities, HTTP dependencies within the persistence layer, and limited isolated testability, which hinder its evolution.

The refactoring process is structured in phases based on the separation of input, application, domain, and infrastructure concerns. It introduces ports and adapters, extracts use cases, corrects improper dependencies between layers, and isolates permission-related policies. The evaluation compares an *initial state* with successive iterations using coupling, complexity, and testability metrics, together with unit testing and functional validation.

The results show measurable improvements in decoupling, dependency explicitness, removal of HTTP dependencies from the persistence layer, and a reduction in *architectural hotspots*. Overall, the evidence supports the conclusion that an incremental approach can produce verifiable structural improvements in real-world evolving systems.

Keywords: incremental architectural refactoring, hexagonal architecture, software maintainability, software testability, separation of concerns.

Índice general

Resumen	3
Abstract	5
Índice de figuras	9
Índice de tablas	11
Lista de acrónimos	13
1. Introducción	14
1.1. Presentación	14
1.2. Objetivos	16
1.3. Desarrollo del trabajo	18
2. Estado del arte	20
2.1. Delimitación del problema y alcance de la revisión	20
2.2. Refactorización incremental, software heredado y evolución arquitectónica	21
2.3. Modularidad arquitectónica y delimitación de responsabilidades .	23
2.4. Calidad interna y coste del cambio: mantenibilidad, testabilidad y modificabilidad	24
2.5. Degradación arquitectónica, smells y deuda técnica	25
2.6. Evaluación de la mejora arquitectónica mediante evidencia estructural y ejecutable	29
2.7. Síntesis del estado del arte y posicionamiento del estudio	31
3. Descripción de la solución	32
3.1. Planificación del trabajo	33
3.2. Herramientas, tecnologías y metodología utilizada	35
3.3. Análisis de los problemas abordados en el módulo de recetas	39
3.4. Diseño de las soluciones adoptadas	45
3.5. Implementación de la solución e integración en Mealie	50
4. Medición y evaluación de la mejora conseguida	58
4.1. Diseño de la evaluación	58
4.2. Validación por fases de la intervención	63
4.3. Evaluación final consolidada	67
4.4. Amenazas a la validez y límites	70
5. Conclusiones y trabajos futuros	72
5.1. Conclusiones del trabajo	72
5.2. Líneas de trabajo futuro	75

Apéndices	77
A. Arquitectura objetivo del módulo de recetas	77
B. Marco de medición y métricas reproducibles	82
C. Intervención de Fase 1: puertos y adaptadores	88
D. Intervención de Fase 2: creación de recetas desde web	94
E. Intervención de Fase 3: separación entre HTTP y persistencia	102
F. Intervención de Fase 4: política de permisos	109
G. Intervención de Fase 5: extracción de casos de uso	118
H. Validación detallada de la Fase 2	128
I. Validación detallada de la Fase 3	132
J. Validación detallada de la Fase 4	136
K. Validación detallada de la Fase 5	140
Bibliografía	145

Índice de figuras

1. Arquitectura conceptual objetivo del módulo de recetas.	81
2. Cambio de responsabilidad del flujo <code>CreateRecipeFromWeb</code> en la Fase 2.	97
3. Cambio de responsabilidad en la gestión de errores de consulta durante la Fase 3.	105
4. Cambio de responsabilidad en la gestión de permisos durante la Fase 4.	113
5. Cambio de responsabilidad sobre <code>RecipeService</code> durante la Fase 5. ...	126

Índice de tablas

1. Resumen de las cinco fases acumulativas de la implementación.	51
2. Métricas estructurales de acoplamiento (C1–C4).	59
3. Métricas estructurales de complejidad y puntos críticos (X1–X3).	60
4. Métricas estructurales de tamaño (S1–S2).	61
5. Métricas estructurales de testabilidad (T1–T3).	61
6. Evolución acumulada de las métricas estructurales del módulo intervenido.	67
7. Reducción estructural de <code>RecipeService</code> y redistribución de complejidad.	69

Lista de acrónimos

En esta memoria se utilizan los acrónimos y siglas recogidos a continuación.

Acrónimo	Significado
API	Interfaz de programación de aplicaciones (<i>Application Programming Interface</i>)
AST	Árbol de sintaxis abstracta (<i>Abstract Syntax Tree</i>)
ATAM	Método de análisis de compromisos arquitectónicos (<i>Architecture Tradeoff Analysis Method</i>)
ATD	Deuda técnica arquitectónica (<i>Architectural Technical Debt</i>)
CRUD	Crear, leer, actualizar y eliminar (<i>Create, Read, Update, Delete</i>)
HTML	Lenguaje de marcado de hipertexto (<i>HyperText Markup Language</i>)
HTTP	Protocolo de transferencia de hipertexto (<i>HyperText Transfer Protocol</i>)
ISO/IEC	Organización Internacional de Normalización y Comisión Electrotécnica Internacional (<i>International Organization for Standardization / International Electrotechnical Commission</i>)
JSON	Notación de objetos de JavaScript (<i>JavaScript Object Notation</i>)
LOC	Líneas de código (<i>Lines of Code</i>)
REST	Estilo arquitectónico para sistemas distribuidos basado en recursos (<i>Representational State Transfer</i>)
SQL	Lenguaje de consulta estructurado (<i>Structured Query Language</i>)
TFM	Trabajo Fin de Máster
URL	Localizador uniforme de recursos (<i>Uniform Resource Locator</i>)

1. Introducción

1.1. Presentación

1.1.1. Aplicaciones de asistencia en cocina y gestión doméstica de recetas

Las aplicaciones de asistencia en cocina agrupan un conjunto amplio de funciones: catalogación de recetas, planificación de menús, gestión de listas de compra, importación de contenidos desde fuentes externas y apoyo a la preparación de comidas en contextos domésticos o semiprofesionales. Aunque desde la perspectiva del usuario suelen presentarse como productos unificados, desde la perspectiva de ingeniería del software suelen combinar problemas heterogéneos: operaciones CRUD, reglas de negocio, integraciones con terceros, gestión de archivos multimedia y coordinación entre capas de presentación, aplicación y persistencia.

En este tipo de software, la evolución funcional suele producirse de manera incremental. Sobre una base ya existente se incorporan nuevas capacidades y, con el tiempo, el código puede acumular acoplamientos entre controladores, servicios, repositorios e integraciones. Como consecuencia, el problema principal no siempre reside en las prestaciones visibles del producto final, sino en la dificultad creciente para modificarlo con seguridad, probarlo de forma aislada y extenderlo sin que su complejidad interna aumente de manera desproporcionada.

1.1.2. Mealie como caso de estudio

Mealie funciona como una aplicación web de código abierto centrada en la gestión de recetas, la planificación de menús y la organización del hogar. Su documentación oficial la presenta como una herramienta autoalojada provista de un backend basado en una API REST y una interfaz web reactiva, diseñada para su despliegue en entornos autogestionados [43,44]. Esta naturaleza accesible y ejecutable la hace apropiada para el objetivo de este trabajo, ya que permite el acceso completo al código fuente, la intervención sobre el sistema, su ejecución local y el análisis de su comportamiento tanto original como modificado.

En lo que respecta a su uso, Mealie puede funcionar en instalaciones autoalojadas con acceso principal a través de navegador web, importación de recetas desde URL y una API documentada para integraciones externas [43,44]. Estas características no constituyen el objeto de evaluación de este trabajo; su relevancia en este contexto reside en delimitar que el caso de estudio es un sistema real, utilizable y extensible, y no un prototipo académico.

1.1.3. Delimitación precisa del alcance

Aunque Mealie incluye varios dominios funcionales, este trabajo se circunscribe de manera estricta al módulo de gestión de recetas del backend. La intervención

no aborda la planificación de comidas, la gestión de usuarios, la organización de compras, el frontend ni el rediseño transversal de todo el sistema.

Asimismo, la intervención se plantea de manera parcial y progresiva. El trabajo se estructura como una intervención arquitectónica incremental enfocada exclusivamente en aquellas partes del módulo más pertinentes para el problema definido, sin aspirar a realizar una migración exhaustiva del backend ni a estandarizar la arquitectura completa de Mealie.

Este trabajo tampoco tiene como finalidad mejorar el rendimiento del producto final, su escalado operativo, su latencia ni su comportamiento bajo carga. El eje del estudio reside en la mejora del desarrollo del software mediante el diseño arquitectónico, concretamente en lo relativo a desacoplamiento, separación de responsabilidades, mantenibilidad, testabilidad y capacidad de evolución controlada del módulo intervenido.

1.1.4. Problema de desarrollo del software abordado

El módulo de gestión de recetas de Mealie constituye un caso representativo de backend que funciona correctamente desde el punto de vista funcional, pero cuya estructura interna plantea dificultades relevantes para su evolución. El análisis del estado inicial, desarrollado posteriormente en la memoria, permite identificar cinco problemas de desarrollo principales:

1. Se observa una clara mezcla de responsabilidades entre las distintas capas. Parte de la orquestación de la lógica de negocio se encuentra alojada en los controladores HTTP, al mismo tiempo que el servicio principal centraliza decisiones de dominio, coordinación con la base de datos, administración de archivos, importación externa y reglas de autorización.
2. El módulo muestra acoplamientos directos con implementaciones concretas de infraestructura. La lógica depende de repositorios, utilidades de filesystem, scraping y servicios externos de manera poco abstraída, lo que dificulta la sustitución de colaboradores y complica la realización de pruebas aisladas.
3. Surgen puntos estructurales críticos, sobre todo alrededor de `RecipeService` y de parte de la capa de entrada HTTP. La acumulación de responsabilidades en estos puntos incrementa el coste de cualquier modificación y aumenta el riesgo de introducir errores cuando se cambia una funcionalidad concreta.
4. Las fronteras entre capas carecen en ocasiones de una definición estricta. Parte de la semántica propia de HTTP se traslada a la capa de persistencia, mientras que ciertas dependencias ligadas al contexto de ejecución se obtienen a través de un patrón de *service locator*, lo que resta claridad al diseño arquitectónico y complica la comprensión global del ensamblaje.
5. El módulo presenta una testabilidad aislada limitada. La dependencia directa de base de datos, filesystem e integraciones externas reduce la viabilidad de pruebas unitarias rápidas y obliga a validar una proporción excesiva de la funcionalidad mediante contextos más pesados o más acoplados.

Esta enumeración delimita los problemas de desarrollo que aborda el trabajo y proporciona la base sobre la que se formulan los objetivos planteados en la sección 1.2.

1.1.5. Problemas de desarrollo frente a problemas del producto final

Los problemas de desarrollo abarcan los factores que condicionan negativamente la comprensión, modificación, verificación y evolución del software. Dentro de este trabajo, esta categoría agrupa pasivos técnicos como la limitada testabilidad aislada, el elevado acoplamiento, la limitada separación de responsabilidades entre capas, la concentración lógica en puntos estructurales críticos y los impedimentos para llevar a cabo refactorizaciones continuas.

Los problemas del producto final, en cambio, son aquellos que afectan al usuario en términos de prestaciones, experiencia de uso, tiempos de respuesta, comportamiento bajo carga o cobertura funcional. Estos aspectos no constituyen la variable principal de evaluación del presente trabajo. Cuando en esta memoria se habla de mejora, se habla ante todo de mejora del desarrollo del software y de la sostenibilidad evolutiva del backend, no de incremento de rendimiento ni de optimización del comportamiento operativo de Mealie.

1.1.6. Sentido del término escalabilidad en este trabajo

En esta memoria, el término escalabilidad se utiliza para describir la capacidad del módulo de evolucionar de forma ordenada a medida que incorpora nuevas responsabilidades, adaptaciones o extensiones. Bajo esta acepción, se interpreta estrictamente como escalabilidad del desarrollo, no como escalabilidad operativa, horizontal, de infraestructura, rendimiento bajo carga o capacidad para atender más usuarios concurrentes o mayores volúmenes de tráfico.

Esta idea pone el foco en que el código pueda integrar nuevas adaptaciones, responsabilidades o extensiones sin desencadenar un aumento desmedido del acoplamiento, del tamaño de los puntos críticos, del esfuerzo de prueba o del riesgo de introducir errores. Bajo ese enfoque, la noción de escalabilidad guarda una relación directa con cualidades como la mantenibilidad, modificabilidad y control de la evolución, en sintonía con las directrices de calidad ISO/IEC 25010 [6] y la definición de arquitectura evolutiva propuesta por Ford, Parsons y Kua [4].

En consecuencia, la contribución del trabajo no consiste en demostrar que Mealie escala mejor en producción, sino en mostrar que el módulo de recetas queda mejor preparado para escalar como base de desarrollo: con dependencias más explícitas, responsabilidades mejor separadas y una mayor proporción de funcionalidad validable de forma aislada.

1.2. Objetivos

A partir del problema descrito en la sección 1.1, los objetivos del trabajo se formulan como una concreción operativa de las mejoras que se pretenden alcanzar

en el módulo de recetas. En esta sección se expresan esas líneas de mejora en forma de objetivos verificables.

1.2.1. Objetivo general

Mejorar la arquitectura interna del módulo de gestión de recetas de la aplicación Mealie mediante una estrategia de refactorización incremental, parcial y por fases, coherente con la lógica de cambio seguro sobre software existente defendida por Fowler [5] y Feathers [3].

El objetivo de esta intervención es aumentar su mantenibilidad, testabilidad, desacoplamiento y escalabilidad entendida como capacidad de evolución estructural del desarrollo del software, en el sentido recogido por el modelo de calidad ISO/IEC 25010 [6] y por la noción de arquitectura evolutiva de Ford, Parsons y Kua [4], manteniendo sin cambios el comportamiento externo de los flujos dentro del alcance del trabajo.

1.2.2. Objetivos específicos

Sobre esta base, los objetivos específicos del trabajo se orientan, dentro del alcance tratado, a dar respuesta a los problemas de desarrollo enumerados en la sección 1.1.4 y se formulan como resultados esperados de la intervención:

1. Un diagnóstico del estado inicial del módulo que caracterice y delimite los problemas de desarrollo que dificultan su evolución controlada, como punto de partida de la intervención.
2. Una arquitectura objetivo parcial, basada en límites explícitos, dependencias dirigidas al interior y separación progresiva de responsabilidades, que sirva como referencia para una migración incremental sobre código existente.
3. Una separación más clara de responsabilidades entre la entrada HTTP, la aplicación, el dominio y la infraestructura, reduciendo la mezcla de competencias y la delimitación difusa de fronteras detectadas en el estado inicial.
4. Una reducción de la dependencia directa de implementaciones concretas de infraestructura y de la concentración de responsabilidades en los puntos estructurales críticos realmente intervenidos, en particular `RecipeService` y determinados flujos de la capa de entrada HTTP.
5. Una mayor testabilidad aislada del módulo, habilitando flujos evaluables sin dependencia directa de base de datos ni del sistema de archivos real cuando el alcance de cada fase lo permita.
6. Evidencia reproducible del efecto de la intervención, mediante métricas estructurales, pruebas unitarias focalizadas y validación funcional mínima, que permita valorar los resultados anteriores sin presentar el trabajo como una mejora de rendimiento ni como un rediseño total del sistema.

1.2.3. Criterios de evaluación

El cumplimiento de estos objetivos se valorará mediante criterios observables y coherentes con el alcance del trabajo. En particular, se considerará que la intervención avanza en la dirección esperada si hace más explícitas las dependencias del módulo, mejora la separación de responsabilidades entre capas, reduce la concentración funcional en los componentes más problemáticos y aumenta la posibilidad de validar flujos de forma aislada.

Estos criterios no se interpretan como una evaluación global de la calidad del sistema ni de su rendimiento operativo. Su función es determinar si, dentro del alcance intervenido, el módulo queda mejor preparado para evolucionar con control estructural. Por ello, en el Capítulo 4 se concretan mediante métricas estructurales, pruebas unitarias focalizadas y validación funcional mínima.

1.2.4. Pregunta de investigación

A la vista de los objetivos y de los criterios expuestos, la pregunta que orienta el trabajo puede formularse del siguiente modo:

¿Puede una estrategia de refactorización arquitectónica incremental, aplicada al módulo de gestión de recetas de Mealie y dentro del alcance intervenido, mejorar su mantenibilidad, su facilidad de prueba, su desacoplamiento y su capacidad de evolución estructural sin alterar el comportamiento observable de los flujos modificados?

1.2.5. Hipótesis de trabajo

Como respuesta tentativa a esa pregunta, se plantea que una refactorización arquitectónica incremental, parcial y basada en la explicitación selectiva de dependencias, la extracción de casos de uso, la separación de políticas y la reducción de puntos críticos puede mejorar de forma medible la capacidad de evolución estructural del módulo de recetas de Mealie sin necesidad de un rediseño total del sistema. Esta hipótesis es coherente con la literatura sobre refactorización de software existente y arquitectura evolutiva [5,3,4]. Dentro del alcance realmente intervenido, ese avance debería reflejarse en un menor acoplamiento concreto, una separación más clara de responsabilidades, una mayor testabilidad aislada y una menor concentración funcional en componentes críticos, preservando al mismo tiempo el comportamiento observable validado de los flujos afectados.

1.3. Desarrollo del trabajo

1.3.1. Enfoque y tipo de trabajo

Este Trabajo de Fin de Máster se plantea como una intervención aplicada sobre un sistema de software de código abierto real y no como una propuesta

puramente teórica ni como una reescritura completa del backend. Su propósito consiste en estudiar, dentro de un alcance acotado y medible, si una estrategia de refactorización arquitectónica incremental puede mejorar la mantenibilidad, la testabilidad, el desacoplamiento y la capacidad de evolución estructural del módulo de recetas de Mealie.

Desde este punto de vista, el trabajo combina análisis arquitectónico, diseño de solución, implementación por fases y evaluación de resultados. Su valor académico no reside únicamente en proponer una arquitectura deseable, sino en justificarla, ejecutarla sobre código existente y contrastar sus efectos mediante evidencia estructural y ejecutable.

1.3.2. Estrategia de desarrollo

La estrategia seguida responde a una secuencia deliberada. En primer lugar, se delimita el problema y se fija con precisión el alcance del estudio. A continuación, se construye el marco conceptual que permite fundamentar la intervención. Sobre esa base, el trabajo se concreta en una intervención arquitectónica incremental sobre el módulo de recetas, estructurada en una serie de fases acumulativas y de riesgo controlado. Por último, los resultados obtenidos se valoran mediante criterios observables y un marco de evaluación reproducible.

Este planteamiento permite mantener la trazabilidad entre el problema detectado, las decisiones de diseño, los cambios efectivamente introducidos y la evidencia aportada al final del trabajo. En consecuencia, el desarrollo del TFM no se presenta como una sucesión informal de modificaciones sobre el código, sino como una intervención arquitectónica acotada, razonada y evaluable.

1.3.3. Estructura del documento

La memoria se organiza como una secuencia argumental que avanza del problema al marco conceptual, del marco conceptual a la intervención ejecutada y de esta a la evaluación final.

El Capítulo 1 delimita el caso de estudio, el alcance real del TFM y la naturaleza del problema abordado. Además, fija los objetivos, la pregunta de investigación, la hipótesis de trabajo y las aclaraciones conceptuales necesarias para interpretar correctamente el resto del documento.

El Capítulo 2 revisa el estado del arte de forma estrictamente vinculada a esa delimitación. Su función no es ofrecer una panorámica general de la arquitectura del software, sino fundamentar las decisiones que sostienen este trabajo: la refactorización incremental sobre software existente, la separación progresiva de responsabilidades y la evaluación de la mejora mediante métricas estructurales y evidencia ejecutable.

El Capítulo 3 traslada ese marco conceptual al caso concreto de Mealie. En él se expone la planificación del trabajo, las herramientas y tecnologías empleadas, la metodología seguida, el diagnóstico del estado inicial, la arquitectura objetivo

parcial y la implementación de las fases de intervención sobre el módulo de recetas.

El Capítulo 4 evalúa el efecto de esa intervención. Para ello articula un estado inicial reproducible, un marco de métricas estructurales, pruebas unitarias focalizadas, validación funcional mínima y una lectura acumulativa de los resultados obtenidos por fases.

El Capítulo 5 formula las conclusiones del trabajo, contrasta la hipótesis de partida y expone de forma explícita los límites del estudio, junto con las líneas de trabajo futuro más coherentes con dichos límites.

La bibliografía y los anexos completan la memoria principal. La primera sostiene el fundamento teórico y metodológico del trabajo, mientras que los segundos aportan el detalle técnico de las fases, el marco de medición y las validaciones que respaldan la intervención realizada.

2. Estado del arte

2.1. Delimitación del problema y alcance de la revisión

Este capítulo sitúa el trabajo en una línea de investigación concreta: la refactorización arquitectónica incremental de software existente. El foco no está en comparar estilos arquitectónicos en general ni en estudiar el rendimiento de aplicaciones web, sino en analizar si una intervención gradual sobre un módulo ya operativo puede mejorar atributos ligados al cambio, como la mantenibilidad, la testabilidad, el desacoplamiento y la capacidad de evolución estructural, sin alterar el comportamiento observable de los flujos intervenidos.

A partir de esta delimitación, la revisión se organiza en cinco ejes. El primero aborda la literatura sobre software heredado, refactorización incremental y evolución arquitectónica, porque ahí se explica por qué los sistemas existentes suelen evolucionar de forma más controlada mediante cambios progresivos que mediante sustituciones amplias. El segundo reúne trabajos sobre modularidad, separación de responsabilidades y dirección de dependencias, necesarios para justificar el aislamiento entre entrada HTTP, lógica de aplicación, reglas de negocio e infraestructura. El tercero se centra en la calidad interna orientada al cambio, especialmente en mantenibilidad, modificabilidad y testabilidad. El cuarto revisa la literatura sobre erosión arquitectónica, *smells* y deuda técnica, útil para interpretar el estado inicial del módulo como un problema estructural y no sólo como una suma de defectos locales. El quinto incorpora enfoques de evaluación arquitectónica y validación empírica que permitan defender las mejoras estructurales mediante evidencia medible y ejecutable.

Esta delimitación excluye las líneas de literatura que no inciden directamente en el problema arquitectónico estudiado, como las centradas en escalado operativo, despliegue, latencia, rendimiento o análisis funcionales del dominio de aplicaciones de cocina. El núcleo del problema es otro: cómo intervenir sobre código con valor funcional ya consolidado, dependencias heredadas y restricciones

reales de cambio para hacerlo más modificable y comprobable sin recurrir a una reescritura global.

La revisión cumple, por tanto, una función específica dentro del trabajo. No pretende abarcar la totalidad de la literatura sobre arquitectura software, sino reunir el marco teórico necesario para analizar una intervención localizada sobre Mealie. Desde este marco, el capítulo sostiene que una estrategia basada en un estado inicial reproducible, una intervención por fases, una arquitectura objetivo acotada y una validación apoyada en evidencia estructural y ejecutable tiene respaldo suficiente en la literatura sobre evolución y mejora arquitectónica de sistemas reales.

2.2. Refactorización incremental, software heredado y evolución arquitectónica

Un antecedente clave para comprender este problema lo ofrece la literatura sobre modularidad y cambio. Parnas planteó tempranamente que la descomposición de un sistema no debería organizarse sólo por flujo funcional, sino también por la necesidad de aislar decisiones que probablemente cambiarán con el tiempo [1]. Esta idea resulta especialmente relevante para trabajos de refactorización arquitectónica porque desplaza la atención desde la ejecución inmediata hacia el comportamiento del sistema frente al cambio. Si determinadas decisiones de infraestructura, acceso a datos o coordinación externa atraviesan el código sin barreras claras, modificar el sistema resulta más costoso y arriesgado.

La literatura de refactorización desarrolló después esa intuición en un plano más operativo. Fowler entiende la refactorización como una secuencia acotada de cambios internos que mejoran la estructura sin alterar el comportamiento observable del software [5]. El interés de este planteamiento no está sólo en legitimar la mejora interna, sino en marcar un modo de trabajo: cambios pequeños, verificables y reversibles en caso necesario. En esa lógica, refactorizar no equivale a sustituir un sistema por otro, sino a introducir mejoras controladas sobre una base que sigue en funcionamiento.

Ese marco resulta aún más pertinente cuando el código ya aporta valor. Feathers sitúa ahí el problema del software heredado: la dificultad no consiste tanto en diseñar un sistema ideal como en poder cambiar con seguridad un sistema cuyo comportamiento no se conoce por completo, con cobertura limitada de pruebas y dependencias externas que dificultan el aislamiento [3]. Desde este punto de vista, el problema del legado es también un problema de conocimiento y de coste. Cambiar resulta más difícil cuando no está claro qué debe preservarse, dónde empieza y termina cada responsabilidad o qué dependencias pueden romperse con una modificación local.

Esa misma lógica aparece en experiencias de reingeniería incremental guiadas por arquitectura. Abi-Antoun y Coelho describen un caso en el que una aplicación heredada recupera estructura mediante cambios progresivos sobre el sistema existente, sin partir de una reescritura total [36]. En este trabajo, esa

referencia resulta útil porque muestra una vía intermedia entre la corrección local y la refundación completa, que es precisamente el equilibrio que se busca en una intervención acotada sobre el módulo de recetas.

Esta lectura encaja de forma directa con Mealie. El punto de partida no es un backend nuevo, sino un módulo operativo en el que controladores, servicios, persistencia, filesystem, scraping y otros colaboradores ya están conectados. El valor funcional acumulado impide tratarlo como una pizarra en blanco. Por ello, la estrategia más razonable no es una reimplementación integral, sino una secuencia de intervenciones arquitectónicas capaces de mejorar la estructura sin provocar una ruptura transversal sobre el sistema completo.

Desde una perspectiva más amplia, la refactorización incremental forma parte del problema de la evolución arquitectónica a lo largo del tiempo. Ford, Parsons y Kua insisten en que la arquitectura de un sistema no debe entenderse como una fotografía fija, sino como una organización que necesita preservar determinados atributos mientras cambia [4]. Breivold et al. revisan la investigación sobre evolución arquitectónica y muestran que el cambio de arquitectura no es una excepción, sino una condición normal de los sistemas que siguen siendo útiles mientras cambian requisitos, tecnologías, restricciones organizativas y formas de uso [12]. En la práctica, la arquitectura se ajusta o se degrada a medida que el sistema incorpora modificaciones.

En un caso como Mealie, este marco teórico tiene una traducción directa. El módulo de recetas no es un ejemplo construido para fines académicos, sino un artefacto real, vivo y en uso [43,44]. Su estructura actual refleja una sucesión de decisiones tomadas en distintos momentos y bajo necesidades concretas, más que un diseño cerrado e inmutable. Entender la arquitectura en esos términos permite leerla como un estado transitorio y mejorable, no como un modelo terminado.

La sostenibilidad arquitectónica añade otra dimensión al problema. Koziolk plantea que una arquitectura puede evaluarse también por su capacidad para seguir siendo útil a medida que cambian el sistema y su entorno [11]. Venters et al., por su parte, sitúan la sostenibilidad del software en el terreno de las decisiones arquitectónicas de largo plazo, vinculándola con mantenibilidad, viabilidad y continuidad evolutiva [23]. En ambos casos, la idea de fondo es la misma: la arquitectura no se juzga sólo por si encaja hoy, sino por cómo soporta el cambio sin perder coherencia.

Esta perspectiva es compatible con la idea de escalabilidad del desarrollo del software que se adopta en este trabajo: la capacidad de un módulo para absorber nuevas modificaciones sin que el coste de cambio crezca de forma descontrolada. La sostenibilidad arquitectónica y la arquitectura evolutiva ofrecen así un marco adecuado para entender que mejorar la estructura interna de un módulo no es una cuestión estética, sino una forma de preservar su viabilidad futura.

El planteamiento de Ford, Parsons y Kua [4] resulta especialmente útil también por otra razón: subraya que una mejora continua sólo es defendible si cada transición se acompaña de objetivos claros, criterios de verificación y señales observables de avance. Aunque este trabajo no utilice mecanismos formales como

las *fitness functions*, sí asume esa lógica general: cada fase debe producir una mejora concreta y comprobable.

En síntesis, la literatura sobre refactorización incremental, software heredado y evolución arquitectónica desplaza la pregunta desde “qué arquitectura sería ideal desde cero” hacia “cómo cambiar una arquitectura real sin perder el control”. Esa es la pregunta que interesa aquí. El objetivo no es afirmar que el módulo de recetas alcance una arquitectura completa y definitiva al final del trabajo, sino mostrar que puede empezar a evolucionar hacia una estructura más controlable mediante intervenciones parciales, justificadas y verificables.

2.3. Modularidad arquitectónica y delimitación de responsabilidades

La segunda base del capítulo es la literatura sobre modularidad y límites arquitectónicos. El trabajo de Parnas sigue siendo central aquí no sólo por su valor histórico, sino porque su idea de descomponer un sistema según las decisiones que probablemente cambiarán con el tiempo anticipa muchas discusiones actuales sobre separación entre responsabilidades de aplicación e infraestructura [1]. Si una parte del sistema cambia por razones distintas a otra, mantener ambas decisiones en la misma unidad aumenta el acoplamiento y dificulta la evolución futura. La utilidad de esta idea para el problema estudiado es directa: el estado inicial no se reduce a clases grandes, sino a la convivencia de responsabilidades heterogéneas dentro de controladores y servicios.

Los trabajos centrados específicamente en la separación de responsabilidades aportan otra perspectiva, basada en la dirección de las dependencias y en el aislamiento del núcleo. Martin sostiene que una arquitectura equilibrada orienta sus dependencias hacia el interior del sistema, evitando que la lógica central quede acoplada a bibliotecas o componentes técnicos concretos [7]. En este trabajo, la referencia a enfoques como puertos y adaptadores, arquitectura hexagonal o clean architecture se adopta con un sentido pragmático: no como una plantilla rígida que deba imponerse al módulo completo, sino como un marco útil para razonar sobre la dirección de las dependencias, la explicitación de fronteras y el aislamiento progresivo de la lógica central [7].

Desde este punto de vista, lo relevante no es ajustar el sistema a una taxonomía arquitectónica cerrada, sino hacer visibles fronteras que en el estado inicial aparecen mezcladas. El controlador HTTP puede entenderse como adaptador de entrada; la aplicación, como coordinadora de flujos concretos; la política de negocio, como unidad aislable; y la infraestructura, como un conjunto de implementaciones que conviene consumir mediante contratos explícitos cuando eso aporte valor estructural. Esta lectura permite justificar decisiones concretas de la intervención: extraer ciertos flujos del controlador, aislar políticas de permisos, evitar que la persistencia exprese semántica HTTP y encapsular colaboraciones externas mediante abstracciones cuando su uso directo incremente el acoplamiento.

Este nivel de separación no tiene por qué ser uniforme ni total. No se trata de reconstruir todo el módulo alrededor de un dominio rico ni de sustituir todos

los flujos existentes por casos de uso independientes. La literatura respalda más bien una separación progresiva y selectiva de responsabilidades, compatible con una intervención incremental sobre código existente y con un alcance limitado al módulo de recetas.

Este planteamiento también puede valorarse desde criterios de consistencia arquitectónica. Ali et al. observan que en sistemas reales suele existir una distancia apreciable entre la arquitectura prescrita inicialmente y la que acaba materializándose en el código, y que esa distancia tiende a crecer con el tiempo [21]. Esta observación encaja con intervenciones de este tipo, que no persiguen un modelo idealizado, sino mejoras observables sobre el código real. Mantener la coherencia entre diseño y código exige acuerdos estructurales explícitos, una dirección de dependencias clara y límites verificables entre componentes.

La modularización también se ha abordado desde la perspectiva de la reestructuración automática. Bavota et al. muestran que, para mejorar la modularización del software, puede combinarse el análisis de dependencias con el estudio de qué partes del código están relacionadas con funcionalidades similares [15]. Aunque este trabajo no persigue una remodelación automática del módulo de recetas, esta línea refuerza una idea relevante: la modularidad no es sólo una preferencia estilística, sino una variable observable que condiciona cómo se comprende, se modifica y se prueba el sistema.

Por ello, la separación de responsabilidades debe entenderse aquí como un medio para reducir razones de cambio mezcladas, no como una multiplicación artificial de capas o artefactos. En el caso estudiado, esto implica intervenir sobre fronteras que en el estado inicial aparecen poco definidas: lógica HTTP dentro de flujos de negocio, coordinación de persistencia dentro de servicios demasiado amplios, uso directo de infraestructura concreta cuando sería preferible depender de contratos, o políticas de permisos mezcladas con la orquestación operativa. La literatura revisada no obliga a una única solución formal, pero sí respalda que el desacoplamiento relevante empieza por separar mejor las responsabilidades y las razones de cambio.

2.4. Calidad interna y coste del cambio: mantenibilidad, testabilidad y modificabilidad

Justificar una mejora estructural exige vincularla con atributos de calidad reconocidos. En este punto, la norma ISO/IEC 25010:2023 ofrece un marco útil para ordenar el análisis de características como mantenibilidad y modificabilidad [6]. Su utilidad en este trabajo no reside en aplicar el estándar de forma exhaustiva, sino en disponer de un vocabulario compartido para describir con precisión la calidad interna del módulo.

En un plano más operativo, Heitlager et al. proponen un modelo práctico de mantenibilidad apoyado en propiedades estructurales observables del código, lo que refuerza la idea de que los atributos ligados al cambio pueden evaluarse a partir de evidencia medible y no sólo de apreciaciones cualitativas [39].

Desde esta perspectiva, la mantenibilidad no se reduce a que el código sea legible, sino a que la estructura del módulo facilite su comprensión, la localización de responsabilidades, la modificación del comportamiento con impacto acotado y la validación de esos cambios con un coste razonable. La testabilidad tampoco se limita a poder escribir pruebas, sino que implica aislar comportamiento relevante sin necesidad de arrastrar base de datos, filesystem o integraciones externas. La modificabilidad, por su parte, no se entiende como la mera capacidad de editar archivos, sino como la posibilidad de introducir cambios sin propagar dependencias y riesgos de forma excesiva.

En este marco, el coste del cambio funciona como hilo conductor. Una arquitectura más mantenible, más testeable y más modificable es, en la práctica, una arquitectura en la que comprender, cambiar y validar el software exige menos esfuerzo y ofrece mayor control sobre los efectos de cada modificación.

La literatura empírica reciente refuerza esta idea. Bogner et al., en contextos de sistemas distribuidos, muestran que la sostenibilidad de la arquitectura afecta directamente a la operativa cotidiana y a la capacidad de mantener el ritmo de cambio [30]. Aunque el contexto técnico no sea idéntico, la idea que puede trasladarse aquí es clara: gobernar la arquitectura no es un ejercicio meramente teórico, sino una condición para sostener la evolución del software.

En la misma línea, Waseem et al. muestran desde la práctica que las decisiones de diseño, monitorización y pruebas están estrechamente ligadas a la arquitectura de sistemas distribuidos [31]. Lo trasladable al problema estudiado no es el estilo microservicios en sí, sino la idea de que la arquitectura condiciona el coste del cambio y de la validación. Cuando la estructura introduce demasiada dependencia entre piezas, probar y modificar cuesta más; cuando delimita mejor las fronteras y las colaboraciones, ambas tareas resultan más controlables.

Por eso, los indicadores más pertinentes para este trabajo no son abstractos, sino operativos: explicitud de dependencias, contaminación entre capas, concentración en puntos críticos y número de flujos que pueden aislarse de forma estricta. Estos indicadores no agotan la evaluación de la calidad del sistema, pero funcionan como aproximaciones razonables a la mantenibilidad, la modificabilidad y la testabilidad dentro del alcance de un módulo concreto.

En síntesis, esta literatura permite entender la mejora arquitectónica como una mejora en atributos internos ligados al cambio. Por ello, separar responsabilidades, reducir acoplamiento y facilitar la validación no son objetivos secundarios, sino formas concretas de avanzar hacia una arquitectura más sostenible y preparada para evolucionar.

2.5. Degradación arquitectónica, smells y deuda técnica

Si la literatura anterior explica por qué conviene mejorar la estructura de un sistema, la literatura sobre degradación arquitectónica ayuda a entender por qué esa necesidad aparece de forma recurrente en sistemas reales. Un trabajo especialmente relevante en este ámbito es el de Terra et al., que plantea re-

factorizaciones orientadas a revertir la erosión arquitectónica [13]. La noción de erosión resulta útil porque permite entender la arquitectura no como una propiedad binaria, correcta o incorrecta, sino como una estructura que puede alejarse gradualmente de los límites que la harían más controlable. Ese deterioro puede manifestarse en dependencias inadecuadas, responsabilidades mezcladas o concentración funcional en componentes que inicialmente tenían un propósito más acotado.

Macia et al. conectan esta discusión con síntomas observables en el código y muestran que ciertas anomalías pueden ser relevantes para identificar degradación arquitectónica [14]. Esta aportación es especialmente pertinente para este trabajo porque buena parte del diagnóstico del estado inicial se apoya precisamente en síntomas visibles: un servicio demasiado central, contaminación HTTP fuera de la capa de entrada, resolución implícita de dependencias mediante *service locator* y baja capacidad de prueba aislada. Su valor está en que permite sostener que estos síntomas no son simples irregularidades locales, sino posibles manifestaciones de un problema arquitectónico más profundo.

Una síntesis más reciente de Li et al. confirma y ordena este panorama a partir de un mapeo sistemático de la erosión arquitectónica, en el que identifican causas, consecuencias y contramedidas recurrentes en la literatura [41]. Su aportación resulta útil aquí porque muestra que la erosión no es una intuición vaga, sino un fenómeno que suele materializarse en violaciones de dependencia, diferencias entre la arquitectura prevista y la arquitectura finalmente implementada, y pérdida progresiva de control sobre la evolución del sistema.

Esta discusión enlaza de forma natural con el estudio de los *architecture smells*. Samarthayam et al. subrayan la necesidad de abordar las correcciones de manera explícita y advierten de las limitaciones de las medidas locales cuando los problemas estructurales están muy arraigados [19]. Mumtaz et al. amplían el panorama desde la taxonomía y muestran la heterogeneidad de los síntomas catalogados en la literatura [32]. Sharma, Singh y Spinellis completan este conjunto al analizar las relaciones jerárquicas y sistémicas entre defectos arquitectónicos y defectos de código [28]. En conjunto, estos trabajos permiten interpretar los *smells* arquitectónicos como señales de alerta sobre decisiones estructurales que pueden dificultar la evolución, más que como simples defectos aislados de implementación.

La degradación arquitectónica también se ha estudiado de forma específica en contextos de software de código abierto. Baabad et al. realizan una revisión sistemática centrada en la degradación de arquitectura en este tipo de software y muestran que la evolución continua, la diversidad de contribuciones y la presión por incorporar cambios funcionales pueden favorecer la acumulación de problemas estructurales cuando no existen mecanismos claros de control arquitectónico [27]. Esta literatura resulta especialmente cercana al caso de Mealie, porque sitúa el problema en un contexto realista: un sistema abierto, funcional y evolutivo en el que la arquitectura no se degrada por una única decisión, sino por la acumulación de cambios bajo restricciones prácticas.

La cuestión de la consistencia arquitectónica vuelve a aparecer aquí como un tema central. Ali et al. recuerdan que mantener la consistencia entre la arquitectura deseada y la arquitectura implementada es un reto práctico permanente, no una tarea de documentación que pueda darse por resuelta al inicio del proyecto [21]. Esto resulta relevante porque una intervención sobre código existente no busca imponer a posteriori un modelo ideal completo, sino corregir algunas de las divergencias que más dificultan la evolución del módulo. En ese sentido, la reducción de dependencias inadecuadas o la extracción de responsabilidades no son únicamente mejoras de diseño, sino pasos orientados a recuperar parte de la coherencia estructural perdida.

Desde otra perspectiva, la noción de deuda técnica estructural ofrece un marco útil para interpretar el problema. Kazman et al. argumentan que el deterioro relevante en sistemas grandes suele tener raíces arquitectónicas claras y no puede reducirse a defectos puntuales del código [16]. Esta lectura permite entender problemas como las dependencias indirectas mal aplicadas, la filtración de semántica HTTP hacia capas internas o la concentración de responsabilidades en un servicio principal no como simples descuidos locales, sino como manifestaciones de deuda estructural acumulada.

La investigación sobre deuda técnica arquitectónica ha abordado este fenómeno desde varias dimensiones complementarias. Martini, Bosch y Chaudron estudian su acumulación y refactorización a lo largo del tiempo, mostrando que la deuda no es estática, sino que se acumula, se desplaza y obliga a tomar decisiones bajo restricciones reales [17]. Martini y Bosch desarrollan además las ideas de deuda contagiosa, círculos viciosos e interés de la deuda, entendidos como mecanismos por los que una decisión arquitectónica problemática propaga costes y encarece modificaciones futuras [18,22]. Estas nociones ayudan a interpretar el estado inicial del módulo de recetas: no se trata sólo de que ciertas piezas sean grandes o incómodas, sino de que su forma actual puede encarecer progresivamente nuevas modificaciones y pruebas.

La literatura posterior ha ordenado todavía más este campo. Besker et al. proponen un modelo unificado y una revisión sistemática sobre deuda técnica arquitectónica que ayudan a ordenar conceptos, causas y manifestaciones [24]. Verdecchia, Malavolta y Lago abordan su identificación y muestran la necesidad de distinguirla de otros problemas estructurales sin perder capacidad de diagnóstico [25]. En esa misma línea, Verdecchia et al. introducen ATDx como propuesta de índice para medir deuda técnica arquitectónica y, más adelante, desarrollan una teoría de deuda técnica arquitectónica en sistemas intensivos en software, reforzando la idea de que la ATD puede razonarse con un marco conceptual más sólido [29,34].

Otros trabajos se centran más en la gestión, priorización y persistencia de la deuda. Martini y Bosch ofrecen herramientas para priorizar refactorizaciones teniendo en cuenta restricciones de coste y operativa [20]. Seaman et al. analizan cómo los datos de deuda técnica pueden apoyar decisiones de priorización y amortización en contextos reales [42], mientras BenIdris et al. estudian la densidad de deuda a lo largo de múltiples *releases* [37]. Perez y Laser examinan la

persistencia de estos pasivos a lo largo del ciclo de vida de la plataforma [26,38]. De Toledo et al., por su parte, aportan un caso práctico situado en microservicios en el que la deuda técnica se interpreta mediante la metáfora del principal y el interés: el principal como coste de corregirla y el interés como sobrecoste de seguir aplazándola [35].

La idea de conjunto es clara: la literatura sobre erosión, *smells* y deuda técnica arquitectónica ofrece un marco sólido para interpretar los problemas detectados en el estado inicial del módulo de recetas. Un servicio excesivamente central, dependencias mezcladas entre capas, uso de *service locator* o dificultad para probar flujos de forma aislada no son únicamente defectos locales de implementación. Son síntomas compatibles con patrones de degradación estructural que la literatura lleva tiempo estudiando y que, si no se corrigen, elevan el coste de evolución del software.

En este trabajo, estas propuestas se utilizan como marco conceptual y diagnóstico, no como instrumentos de aplicación formal. No se busca calcular el principal y el interés de la deuda, aplicar un detector automático de *smells* ni producir un índice ATDx del módulo. Su aportación es más acotada: ofrecen un lenguaje útil para interpretar el estado inicial como un conjunto de pasivos estructurales observables y para orientar la priorización de intervenciones cuyo efecto pueda seguirse y validarse dentro del experimento.

Además de aportar categorías diagnósticas, esta literatura también orienta el tipo de intervención que puede considerarse razonable ante una arquitectura degradada. Terra et al. proponen abordar las desviaciones arquitectónicas mediante refactorizaciones dirigidas específicamente a corregirlas [13]. En la misma línea, Samarthayam et al. sostienen que los *architecture smells* más arraigados no se resuelven con ajustes locales, sino mediante intervenciones estructurales de mayor alcance [19]. Esto refuerza la idea de que la extracción de responsabilidades, la corrección de dependencias o el aislamiento de políticas pueden ser respuestas justificadas cuando los síntomas detectados tienen naturaleza arquitectónica.

La literatura también ha explorado enfoques más sistemáticos y, en algunos casos, automatizados. Cortellessa y Di Pompeo estudian la sensibilidad de la refactorización arquitectónica multiobjetivo a características de configuración, y muestran que la refactorización de arquitectura puede tratarse como un problema de optimización en el que se ponderan distintos atributos y restricciones [33]. Este tipo de trabajos recuerda que la refactorización arquitectónica puede analizarse con rigor más allá de intuiciones cualitativas. Sin embargo, muchas aproximaciones de este tipo suponen modelos arquitectónicos formales, parámetros estables y objetivos cuantificables con un grado de completitud mayor que el disponible en un módulo real de un sistema de código abierto intervenido con alcance limitado.

Por ello, este trabajo se sitúa más cerca de la tradición de refactorización incremental guiada por diagnóstico y evidencia que de una optimización automática global de la arquitectura. En sistemas reales y heredados, el problema no suele consistir en encontrar matemáticamente la estructura óptima, sino en

abrir espacio para el cambio seguro dentro de restricciones prácticas: contratos existentes, dependencias ya desplegadas, flujos que deben seguir funcionando y necesidad de comprobar cada avance con un coste asumible. Feathers [3], Fowler [5] y Ford, Parsons y Kua [4] sostienen precisamente esta visión del cambio como una secuencia pragmática de movimientos pequeños pero acumulativos.

Como complemento, otra línea de literatura analiza la refactorización incremental desde la perspectiva de las restricciones que condicionan su ejecución. Trabajos como los de Martini y Bosch o Seaman et al. muestran que toda intervención sobre el código aparece entrelazada con dilemas que no son sólo técnicos: coste, validación *in situ* y nivel admisible de disrupción para el resto del sistema [20,42]. La conclusión práctica es directa: conviene evitar refundaciones globales que impidan atribuir las mejoras a cambios concretos y privilegiar intervenciones acotadas, pero suficientemente controladas como para poder medirlas.

En este contexto, la extracción de casos de uso, la introducción de puertos y adaptadores, la corrección de direcciones de dependencia o la externalización de políticas no deben entenderse como recetas universales, sino como respuestas distintas a problemas también distintos. La literatura revisada respalda esa lectura: la refactorización arquitectónica eficaz no consiste en aplicar un patrón de forma indiscriminada, sino en escoger intervenciones compatibles con el tipo de acoplamiento, degradación o deuda que se quiere corregir.

2.6. Evaluación de la mejora arquitectónica mediante evidencia estructural y ejecutable

Una vez justificada la intervención, surge la cuestión de cómo evaluar si realmente mejora la arquitectura. La literatura de evaluación arquitectónica ofrece aquí referencias relevantes. Kazman et al. propusieron el análisis basado en escenarios para estudiar la arquitectura del software desde la perspectiva de los cambios y de los atributos de calidad, enfatizando que el valor de una arquitectura puede discutirse a partir de situaciones concretas de evolución y uso [9]. ATAM, por su parte, formalizó una metodología de evaluación arquitectónica centrada en *trade-offs* y escenarios, mostrando que la evaluación de una arquitectura puede ser sistemática sin reducirse a métricas de rendimiento [10].

Esta tradición resulta pertinente para el problema analizado porque vincula las decisiones arquitectónicas con atributos de calidad evaluables, como la capacidad de evolucionar sin deterioro estructural, y los separa con claridad de variables operativas como el rendimiento o la concurrencia. En este trabajo no se aplica ATAM en sentido estricto ni se reproduce una evaluación sistémica completa, pero sí se adopta su lógica general: razonar sobre la arquitectura frente a escenarios de cambio, atributos de calidad y *trade-offs*, adaptando esa perspectiva a una escala más limitada, centrada en un único módulo, un estado inicial reproducible, medición estructural y evidencia ejecutable ligada a las fases concretas de intervención.

En el plano de la validación ejecutable, Muccini et al. exploran la regresión basada en arquitectura y subrayan que los cambios estructurales también pueden

evaluarse atendiendo al esfuerzo de retesteo y a la preservación de la conformidad entre arquitectura e implementación [40]. Aunque el presente trabajo no utiliza lenguajes formales de descripción arquitectónica, sí comparte esa intuición: una mejora estructural defendible exige combinar razonamiento arquitectónico y comprobación de que los flujos intervenidos siguen comportándose como se espera.

La evaluación de la sostenibilidad arquitectónica refuerza esta misma orientación. Koziolok propone considerar la sostenibilidad de la arquitectura como una dimensión evaluable, lo que encaja con un análisis centrado en si la estructura soporta mejor el cambio futuro [11]. En un trabajo de alcance local y experimental, esta pregunta debe trasladarse a un plano operativo: determinar si la arquitectura queda en mejores condiciones para absorber cambios posteriores dentro del alcance intervenido.

También conviene señalar que una evaluación apoyada sólo en observaciones generales y sin respaldo empírico tangible resulta poco defendible. En este sentido, las propuestas centradas en medir rasgos internos del código ofrecen un referente útil. McCabe formaliza una vía analítica para razonar sobre la complejidad estructural de funciones y procedimientos [8], y Chidamber y Kemerer aportan un marco extendido para describir métricas estructurales en diseño orientado a objetos [2]. Este trabajo no adopta estas formulaciones como instrumentos literales de evaluación, pero sí retoma su idea de fondo: hacer explícitas algunas dimensiones internas del código permite razonar sobre la mejora arquitectónica con más objetividad que la mera intuición.

No obstante, la mejora arquitectónica tampoco puede deducirse sólo de números estructurales. La literatura sobre refactorización recuerda que la mejora del diseño sólo es defendible si se preserva el comportamiento relevante del software [5,3]. Por ello, la evidencia estructural debe combinarse con evidencia ejecutable. En un estudio de este alcance, esa combinación se concreta en métricas estructurales sobre el mismo módulo antes y después de la intervención, pruebas unitarias centradas en nuevas unidades aislables y validación funcional mínima sobre los flujos HTTP o de integración afectados.

La literatura empírica reciente sobre arquitectura y pruebas refuerza esta relación. Waseem et al. muestran que diseño y estrategia de validación están estrechamente relacionados, y que ciertas decisiones arquitectónicas pueden facilitar o dificultar en la práctica la prueba del sistema [31]. En contextos distintos, Bogner et al. subrayan que la capacidad de cambio exige mecanismos de aseguramiento y no puede darse por supuesta [30]. La idea útil para este tipo de intervención es clara: la calidad arquitectónica tiene que mostrarse, no sólo afirmarse. En un módulo concreto, eso implica observar mejor desacoplamiento y menor concentración estructural, pero también demostrar que ciertos flujos pueden comprobarse ahora con más aislamiento y menos dependencia de infraestructura real.

Con este planteamiento, la validación del trabajo evita dos extremos poco productivos: el discurso puramente cualitativo, difícil de contrastar, y la reduc-

ción del análisis a un conjunto de cifras sin interpretación arquitectónica. La combinación de métricas estructurales y pruebas ejecutables permite sostener afirmaciones de mejora con una base más sólida y dentro de un alcance bien delimitado.

Esta forma de evaluación debe leerse, no obstante, con cautela. La literatura sobre deuda técnica y *smells* recuerda que introducir nuevas abstracciones no garantiza por sí solo una mejora arquitectónica: puede desplazar responsabilidades sin aclararlas, multiplicar artefactos sin ganar cohesión o generar un desacomplamiento sólo nominal [24,34]. Del mismo modo, la refactorización incremental suele producir mejoras parciales, no una resolución total de la arquitectura. Esto obliga a interpretar los resultados con precisión: una fase puede mejorar una dirección de dependencias concreta sin eliminar todos los puntos críticos, o extraer un caso de uso aislable sin reformar toda la capa HTTP [18,17].

También existe el riesgo de desplazar la deuda en lugar de reducirla. Como advierten algunos autores al analizar el coste y el interés de la deuda, sanear focos aislados sin un marco corrector más amplio puede arrastrar costes adicionales que afloren más adelante [22]. A ello se suma un límite de observabilidad: no siempre existen definiciones plenamente estables ni una métrica única para cada fenómeno estructural [32,27]. Por ello, resulta poco riguroso afirmar que la arquitectura mejora por el simple descenso de un único indicador. Lo más defendible es hablar de una convergencia de evidencias: menos contaminación entre capas, mayor explicitud de dependencias, más flujos unitariamente aislables, menor concentración funcional en puntos críticos concretos y preservación del comportamiento observable relevante.

La literatura empírica sobre la capacidad de cambio en microservicios añade una advertencia complementaria. Bogner et al. y Waseem et al. muestran que explicitar fronteras no elimina por sí solo la complejidad; a veces la redistribuye y exige nuevas disciplinas de validación y observación [30,31]. En un contexto como el aquí analizado, esta idea ayuda a no presentar la introducción de puertos, adaptadores o casos de uso como una mejora automática. Sólo puede hablarse de mejora si la intervención reduce de forma neta el coste de cambio y aumenta la controlabilidad del módulo dentro del alcance intervenido. Por eso, el enfoque iterativo adoptado en este trabajo renuncia a proclamar transformaciones totales del sistema y se limita a defender avances locales, medibles y revisables.

2.7. Síntesis del estado del arte y posicionamiento del estudio

A partir del marco anterior, el posicionamiento del estudio puede expresarse con precisión. El trabajo se sitúa en la intersección entre la refactorización incremental de software existente, la separación arquitectónica apoyada en límites explícitos, el diagnóstico de degradación y deuda arquitectónica, y la evaluación empírica de mejoras estructurales en un sistema real de código abierto. Su contribución se concreta en intervenir un módulo real de Mealie mediante una secuencia de cambios arquitectónicos de alcance acotado y evaluar si dichos cambios mejoran su capacidad de evolución estructural.

Este posicionamiento diferencia el trabajo de otras líneas cercanas. Aunque se apoya en parte de la literatura sobre capacidad de cambio y deuda en sistemas distribuidos, no plantea un estudio sobre microservicios como objeto principal. Aunque toma de la evaluación arquitectónica la idea de razonar en términos de calidad, escenarios de cambio y *trade-offs*, no reproduce una evaluación completa de arquitectura de sistema al estilo ATAM. En todos los casos, el foco se mantiene en el coste estructural del cambio dentro del módulo de recetas.

El caso Mealie aporta valor metodológico precisamente porque permite trasladar ese marco conceptual a un artefacto real. Las referencias primarias del proyecto muestran que se trata de un sistema de código abierto, ejecutable y funcional, con backend HTTP y funcionalidades relevantes para el módulo de recetas [43,44]. La literatura aporta el marco para interpretar la intervención, mientras que el sistema real aporta el objeto sobre el que dicha intervención puede aplicarse y evaluarse. Esta combinación sitúa el estudio en un punto intermedio: más concreto que una revisión puramente conceptual y más sistemático que una refactorización oportunista sobre código legado.

La conclusión general del estado del arte respalda la hipótesis del trabajo. Existe una base suficiente para sostener que, en software existente, la mejora arquitectónica razonable suele pasar por refactorización incremental; que la separación progresiva de responsabilidades y dependencias es una estrategia coherente para reducir acoplamiento y mejorar la testabilidad; que fenómenos como erosión, *smells* y deuda técnica arquitectónica ofrecen un marco pertinente para interpretar los problemas del estado inicial; y que una mejora arquitectónica puede evaluarse de forma defendible combinando evidencia estructural y ejecutable.

A partir de este recorrido, el siguiente paso es trasladar el marco teórico al plano operativo del trabajo. El Capítulo 3 retoma estas ideas para presentar el diseño concreto de la intervención, las decisiones arquitectónicas adoptadas y la secuencia de fases sobre el código del módulo, mientras que el Capítulo 4 las contrasta mediante evidencia estructural y ejecutable.

3. Descripción de la solución

Una vez fijado el problema en el Capítulo 1 y construido el marco conceptual en el Capítulo 2, el presente capítulo traslada ambos a una solución concreta sobre el módulo de recetas de Mealie. La exposición sigue un orden deliberado: primero la planificación del trabajo, después las herramientas y la metodología empleadas, a continuación el diagnóstico del estado inicial, después el diseño de la solución y, por último, su implementación e integración por fases.

3.1. Planificación del trabajo

3.1.1. Función metodológica de la planificación

La planificación de este trabajo tuvo una función metodológica central: ordenar la intervención arquitectónica sobre software existente de forma que el alcance, la compatibilidad funcional y la atribución de resultados permanecieran controlados. Dado que el problema abordado no consistía en añadir prestaciones visibles, sino en mejorar la estructura interna de un módulo ya operativo, la planificación debía asegurar desde el inicio la relación entre diagnóstico, diseño, implementación y evaluación.

Desde esta perspectiva, la planificación atendió tres necesidades metodológicas. La primera fue formular el problema con suficiente amplitud para situar el TFM dentro de la mejora arquitectónica de software existente, pero con un alcance defendible dentro de su escala. La segunda fue intervenir un sistema real de código abierto sin convertir el trabajo en una reescritura del backend de Mealie. La tercera fue ordenar el desarrollo de forma que cada decisión pudiera vincularse después con evidencias observables y con un efecto atribuible dentro del alcance evaluado.

Por ello, la planificación se presenta aquí en términos metodológicos y de secuencia de intervención, más que como un cronograma detallado.

3.1.2. De la planificación estratégica a la delimitación del alcance

En una primera etapa, el trabajo se organizó alrededor de grandes bloques de actividad: preparación del entorno y análisis del código, delimitación del problema técnico, diseño de la solución, implementación, obtención de evidencias y consolidación documental. Esta formulación inicial no pretendía fijar todavía una secuencia detallada de decisiones arquitectónicas, sino proporcionar una estructura de avance mientras el conocimiento del caso se construía mediante el análisis directo del sistema.

A medida que avanzó el análisis del backend y se precisó mejor el problema de investigación, esa planificación fue concretándose de manera progresiva. El alcance quedó finalmente delimitado al módulo de gestión de recetas, descartando otros ámbitos del sistema cuya incorporación habría ampliado innecesariamente la intervención sin reforzar la pregunta central del TFM. Lejos de alterar la dirección original del trabajo, esta concreción reforzó su consistencia: la mejora arquitectónica de software existente seguía siendo el núcleo del estudio, pero ahora sobre una unidad funcional identificable, con fronteras suficientes y con problemas estructurales que podían observarse y validarse con más control.

La consecuencia metodológica de esta delimitación fue igualmente clara. La planificación dejó de apoyarse principalmente en bloques amplios de actividad y pasó a organizarse mediante fases arquitectónicas acumulativas. El cambio no afectó al sentido general del trabajo, sino al grado de precisión con el que ese sentido podía ejecutarse y evaluarse.

3.1.3. Estructura metodológica adoptada

Una vez concretado el alcance, el trabajo quedó organizado en una secuencia metodológica estable, apoyada en cuatro bloques estrechamente relacionados.

Delimitación del caso y diagnóstico del estado inicial. En primer lugar, se confirmó la idoneidad de Mealie como caso de software de código abierto y se acotó definitivamente el estudio al módulo de recetas. Sobre esa base, se inspeccionó el backend para localizar puntos críticos, dependencias concretas de infraestructura, mezcla de responsabilidades y dificultades de prueba aislada. De este bloque surgieron tanto el diagnóstico del problema como la necesidad de fijar un estado inicial reproducible.

Definición de una arquitectura objetivo parcial y de una estrategia incremental. A continuación, se formuló una arquitectura objetivo parcial, inspirada de forma pragmática en enfoques basados en límites explícitos y en la separación entre entrada, aplicación, dominio e infraestructura. Esa arquitectura no se concibió como un estado que debiera implantarse de una sola vez, sino como referencia para ordenar decisiones de diseño y para establecer una trayectoria de mejora controlada sobre el módulo.

Implementación por fases. La intervención se articuló a partir de cinco fases acumulativas, cada una de ellas asociada a un problema específico del estado inicial y a una mejora estructural reconocible.

- Fase 1: introducción de puertos y adaptadores para crear fronteras explícitas frente a infraestructura.
- Fase 2: extracción del flujo `CreateRecipeFromWeb` fuera del controlador HTTP y hacia una unidad de aplicación explícita.
- Fase 3: eliminación de semántica HTTP en la persistencia específica del módulo.
- Fase 4: extracción de la política de permisos a `RecipePermissionPolicy`.
- Fase 5: descomposición adicional de `RecipeService` mediante nuevos casos de uso y reducción parcial del patrón *service locator*.

Validación y trazabilidad del proceso. Por último, la validación no quedó reservada al final del trabajo, sino que condicionó el cierre de cada fase. Cada fase se consideró completada cuando existía evidencia coherente con su objetivo específico de que la modificación introducía una mejora estructural sin romper el comportamiento observable relevante. Según el objetivo de cada intervención, esa evidencia combinó análisis estructural, pruebas unitarias focalizadas, validación funcional mínima y documentación parcial acumulativa. Esta dinámica permitió enlazar de manera trazable planificación, implementación y evaluación, manteniendo la correspondencia entre cada cambio realizado, su justificación arquitectónica y la evidencia utilizada para valorarlo.

3.1.4. Justificación de la ejecución incremental por fases

La elección de una ejecución incremental por fases fue razonable por razones técnicas y metodológicas.

En primer lugar, el módulo intervenido ya estaba integrado en un sistema funcional real. Una reescritura amplia habría incrementado el riesgo de regresión, habría dificultado la atribución de efectos y habría desplazado el trabajo desde la mejora arquitectónica controlada hacia un rediseño de mayor alcance. La lógica de cambios pequeños y acumulativos descrita en el estado del arte [5,3,4] resultaba, por tanto, la estrategia más coherente con el tipo de sistema y con los recursos del TFM.

En segundo lugar, la organización del trabajo en fases permitió abordar problemas de distinta naturaleza sin superponer sus efectos. La incorporación de puertos y adaptadores, el aislamiento de un caso de uso, la eliminación de semántica HTTP en persistencia o el aligeramiento parcial de un servicio central responden a objetivos distintos. Separar estas actuaciones favoreció una atribución más clara de los cambios y evitó presentar como mejora global efectos que todavía no habían sido validados de forma específica.

Esta priorización progresiva es coherente con la literatura sobre decisión de refactorizaciones de deuda arquitectónica, que insiste en no agrupar intervenciones heterogéneas cuando ello dificulta atribuir su impacto estructural y su coste de validación [20]. Además, la estrategia incremental encaja con la interpretación de escalabilidad adoptada en este TFM: si la escalabilidad se entiende como capacidad de evolución estructural del desarrollo del software, la metodología debía mostrar que el módulo podía transformarse de manera controlada sin dejar de ser ejecutable, verificable y compatible con el sistema anfitrión.

3.1.5. Síntesis

La planificación del TFM partió de una definición estratégica del problema y fue adquiriendo precisión a medida que se concretaban el caso de estudio, la arquitectura objetivo parcial y la secuencia de intervención. El resultado no fue un cambio de dirección, sino una especialización progresiva de la misma línea de trabajo: mejora arquitectónica sobre software existente, aplicada de forma incremental, focalizada y verificable sobre el módulo de recetas. Esta planificación proporciona el marco metodológico que permite conectar el diagnóstico del estado inicial, el diseño de la solución, su implementación por fases y la evaluación consolidada del Capítulo 4.

3.2. Herramientas, tecnologías y metodología utilizada

Explicada la planificación real del trabajo, conviene precisar con qué medios técnicos y metodológicos se llevó a cabo la intervención. Esta sección describe el entorno técnico, las herramientas de desarrollo y validación, y la metodología aplicada sobre el módulo de recetas de Mealie. Estos elementos condicionaron tanto la forma de intervenir el código como la posibilidad de preservar su comportamiento observable y evaluar los cambios estructurales introducidos.

3.2.1. Stack tecnológico del sistema intervenido

El caso de estudio es Mealie, un sistema de código abierto cuyo backend está construido en Python y organizado alrededor de una API HTTP, una capa de servicios y un acceso a persistencia basado en SQLAlchemy. Aunque el proyecto incluye también frontend, despliegue y otros módulos funcionales, el alcance del TFM se restringe al backend del módulo de gestión de recetas.

Las principales tecnologías relevantes para este trabajo son las siguientes:

- Python 3.12 como lenguaje principal del backend y soporte de la mayor parte de la lógica intervenida.
- FastAPI como marco de entrada HTTP, fundamental para entender la organización de controladores y el problema de contaminación entre capas.
- SQLAlchemy como tecnología de acceso a datos y origen de varios acoplamientos detectados en repositorios y servicios.
- Pydantic como mecanismo de validación y serialización de esquemas de entrada y salida.
- Servicios auxiliares propios del proyecto, especialmente los relacionados con scraping de recetas, almacenamiento en filesystem, importación desde ZIP e integración con OpenAI.
- Event bus y componentes de publicación de eventos utilizados por algunos flujos del módulo.

Este stack resulta relevante porque el problema arquitectónico observado no surge en abstracto, sino de la interacción entre estos elementos. La combinación de controladores FastAPI, servicios de aplicación demasiado amplios, repositorios concretos, almacenamiento en filesystem e integraciones externas contribuye a los acoplamientos, la concentración de responsabilidades y la dificultad de prueba aislada que motivan el trabajo.

3.2.2. Herramientas de desarrollo, validación y medición

El desarrollo del trabajo se apoyó en herramientas ya presentes en el proyecto y en algunos artefactos específicos creados para este TFM.

Entorno de ejecución y trabajo sobre código real. El proyecto se ejecutó de forma local y controlada, lo que hizo posible verificar el comportamiento del módulo antes y después de aplicar los cambios arquitectónicos. Esta condición era fundamental, ya que el TFM no aborda un sistema artificial, sino una aplicación real cuya funcionalidad debía mantenerse estable a pesar de las modificaciones internas.

Herramientas de pruebas. La herramienta principal de validación fue `pytest`, empleada tanto para pruebas unitarias focalizadas como para pruebas de integración mínimas sobre los flujos afectados. Su papel fue doble: por un lado, permitió validar unidades extraídas, como casos de uso o políticas, de forma más aislada; por otro, permitió comprobar, cuando era necesario, que los endpoints o flujos de integración intervenidos conservaban el comportamiento observable esperado.

Herramientas de observación estructural. Además de las pruebas, se emplearon utilidades de análisis sobre el código fuente para observar tamaño, complejidad aproximada, número de métodos, puntos críticos y dependencias concretas. Esta observación no se apoyó únicamente en una impresión cualitativa, sino en conteos y criterios repetibles aplicados sobre el mismo módulo.

Instrumento específico de medición. El instrumento principal para la recolección de datos fue el script `support-files/recipe_module_metrics.py`. Esta herramienta permitió contrastar el estado inicial con los estados resultantes de la intervención mediante un grupo constante de métricas organizadas en cuatro dimensiones: acoplamiento, tamaño, complejidad y facilidad de prueba. Su uso aportó una forma reproducible de observar la evolución del módulo y evitó que la discusión del TFM dependiera únicamente de descripciones narrativas sobre el diseño.

Trazabilidad de los estados analizados. Para fijar el estado inicial del módulo y comparar posteriormente los resultados de cada fase, se trabajó sobre versiones claramente delimitadas del código. Esta delimitación permitió evitar comparaciones ambiguas entre estados del sistema y mantener la trazabilidad de la intervención.

3.2.3. Metodología de intervención incremental

La metodología general seguida en el trabajo fue la refactorización arquitectónica incremental sobre software existente. En línea con lo planteado en el Capítulo 2, la mejora no se buscó mediante la sustitución completa del módulo, sino mediante cambios acotados, acumulativos y compatibles con el comportamiento observable del sistema [5,3,4].

En el plano operativo, cada fase partió de un problema estructural identificado en el estado inicial, introdujo una modificación arquitectónica limitada y se cerró únicamente cuando existía evidencia suficiente de que el cambio no alteraba el comportamiento relevante dentro del alcance afectado. Este enfoque permitió trabajar sobre un sistema real manteniendo la relación entre diseño, implementación y validación.

Aunque la secuencia metodológica puede describirse de forma lineal, la ejecución real incluyó iteraciones locales. El diagnóstico refinó algunas decisiones de diseño, la integración obligó a introducir ajustes de compatibilidad fuera del punto crítico principal y la validación condicionó el cierre efectivo de cada fase. Por ello, la metodología debe entenderse como un proceso incremental con retroalimentación controlada, no como una cadena rígida de pasos cerrados.

Esta metodología no pretende demostrar la superioridad abstracta de un estilo arquitectónico sobre otro. Su propósito es más concreto: comprobar si una estrategia incremental, aplicada a un módulo real de Mealie, mejora atributos de calidad del desarrollo del software sin abrir una migración total del backend.

3.2.4. Estrategia de validación y medición

La validación del trabajo se apoyó en cuatro elementos metodológicos interconectados: estado inicial, fases de intervención, evidencias de validación y métricas estructurales.

Estado inicial. El estado inicial permitió fijar una línea base del módulo de recetas antes de introducir la nueva estructura arquitectónica. A partir de esta línea base se identificaron los principales puntos críticos, se localizaron acoplamientos relevantes y se establecieron los indicadores que después se compararían con los estados resultantes de la intervención.

Fases de intervención. Las fases se utilizaron como unidad de avance y de atribución. Cada una abordó un problema técnico distinto: creación de fronteras frente a infraestructura, extracción de lógica desde la entrada HTTP, corrección de dependencias entre persistencia y HTTP, aislamiento de políticas de permisos o reducción parcial de la concentración en **RecipeService**. Esta organización permitió separar efectos y evitar que las mejoras observadas se atribuyeran de forma imprecisa a la intervención en su conjunto.

Evidencias de validación. Para respaldar los resultados, la validación combinó tres tipos de evidencia: evidencia estructural reproducible sobre el código del módulo, evidencia unitaria sobre casos de uso o políticas extraídas y evidencia funcional mínima sobre los flujos HTTP o de integración directamente afectados. Esta combinación era necesaria porque cada tipo de evidencia cubría una dimensión distinta del problema: las métricas permitían observar cambios estructurales, las pruebas unitarias evidenciaban ganancias de testabilidad aislada y la validación funcional ayudaba a comprobar la preservación del comportamiento observable.

La combinación de evidencias se aplicó de forma proporcional al objetivo de cada fase. La Fase 1 tuvo un cierre principalmente estructural y preparatorio, mientras que las fases posteriores incorporaron, según su alcance, pruebas unitarias y validación funcional mínima sobre los flujos realmente intervenidos.

Métricas. El diseño de la evaluación recurrió a un conjunto de métricas específicas, expuestas con mayor profundidad en el Capítulo 4. En esta sección basta indicar su propósito: establecer un criterio común para observar complejidad, tamaño, testabilidad y acoplamiento, sin relegar el análisis a meras apreciaciones cualitativas. La estructura del trabajo se planificó para que las intervenciones generaran cambios observables mediante estas herramientas, aunque no todas las métricas mostraran la misma sensibilidad en cada fase.

3.2.5. Adecuación metodológica al problema

La metodología empleada resulta adecuada para el problema abordado por varias razones. En primer lugar, era compatible con la naturaleza heredada y funcional del módulo de recetas: al tratarse de una parte integrada en un sistema real, una intervención controlada y acumulativa resultaba más apropiada que una

sustitución amplia del diseño. En segundo lugar, reforzaba la trazabilidad del proceso, al vincular cada fase con un obstáculo específico identificado en el estado inicial. En tercer lugar, hacía posible una evaluación defendible, al combinar una línea base reproducible, cambios acotados y evidencias estructurales, unitarias y funcionales.

Además, esta metodología es coherente con la interpretación del título del TFM. La testabilidad se entiende aquí como la capacidad de validar comportamiento con mayor aislamiento. La mantenibilidad se interpreta como una mejora en la separación de responsabilidades, la reducción de acoplamientos concretos y la menor concentración funcional, en línea con el modelo de calidad ISO/IEC 25010 [6]. La escalabilidad se entiende como escalabilidad del desarrollo del software, es decir, como capacidad de evolución estructural del módulo sin crecimiento descontrolado del coste de cambio, en sintonía con la noción de arquitectura evolutiva de Ford, Parsons y Kua [4].

Esta sección delimita, por tanto, las opciones tecnológicas y metodológicas empleadas en el TFM. Los resultados numéricos, la comparación detallada entre estados y la comprobación consolidada de cada fase se desarrollan posteriormente en el Capítulo 4.

3.3. Análisis de los problemas abordados en el módulo de recetas

3.3.1. Propósito del diagnóstico inicial

Una vez definidas la planificación, las herramientas y la metodología, el siguiente paso consiste en fijar el estado de partida sobre el que se interviene. Antes de diseñar una solución arquitectónica resultaba necesario caracterizar con precisión qué problema se quería resolver en el módulo de recetas. Ese problema no era la falta de funcionalidad visible del sistema ni un fallo de rendimiento del producto final. El backend de Mealie ya resolvía los flujos principales del dominio asociado al módulo de recetas, pero lo hacía mediante una estructura interna que dificultaba la modificación segura, la prueba aislada y la evolución controlada del software.

El estado inicial descrito en este apartado no actúa únicamente como diagnóstico narrativo. Constituye también el punto de referencia reproducible que el Capítulo 4 reutiliza para comparar el efecto acumulado de las fases y para justificar por qué determinadas métricas y pruebas son relevantes para este módulo.

Por este motivo, el examen del problema se focalizó en los síntomas estructurales del sistema base. En particular, interesaba identificar aquellas características que encarecían la validación, modificación o ampliación del código: mezcla de responsabilidades, acoplamiento con infraestructura concreta, puntos críticos sobredimensionados, dependencias contextuales implícitas, uso de *service locator* y dificultades para ejecutar pruebas unitarias aisladas de recursos externos como base de datos, filesystem o integraciones.

Dentro del marco establecido en el Capítulo 2, este conjunto de síntomas resulta compatible con patrones descritos en la literatura sobre degradación arquitectónica, inconsistencia entre arquitectura deseada e implementada y deuda técnica arquitectónica [14,21,16,27].

3.3.2. Puntos críticos estructurales del estado inicial

El primer síntoma relevante del estado inicial era la existencia de varios puntos de concentración funcional. El análisis estructural previo a la Fase 1 puso de manifiesto, entre otros, los siguientes puntos críticos.

- `mealie/services/recipe/recipe_service.py`: 627 líneas de código.
- `mealie/routes/recipe/recipe_crud_routes.py`: 608 líneas de código.
- `mealie/services/scrapper/recipe_bulk_scraper.py`: 127 líneas de código.
- `mealie/routes/recipe/_base.py`: 56 líneas de código.

Al revisar la configuración interna de las clases fundamentales, este nivel de aglutinación resulta todavía más evidente.

- `RecipeService`: 27 métodos y un rango estructural de 502 líneas.
- `RecipeController`: 23 métodos y un rango estructural de 531 líneas.
- `RecipeBulkScraperService`: 5 métodos y un rango estructural de 107 líneas.

El problema no residía únicamente en el tamaño absoluto. Lo más relevante era el tipo de comportamiento que se concentraba en esos puntos. Las funciones más complejas del estado inicial estaban ligadas a tareas de orquestación e integración, no a reglas de dominio pequeñas y bien encapsuladas. En `RecipeService` destacaban, por ejemplo, `build_recipe_from_images`, `check_assets`, `_process_recipe_data`, `_resolve_ingredient_sub_recipes` y `create_from_zip`. En la capa HTTP destacaba `_create_recipe_from_web`, junto con otros métodos del controlador con una carga de coordinación elevada.

Esto conducía a una conclusión clara: la complejidad del módulo no estaba distribuida entre unidades pequeñas y cohesionadas, sino concentrada en unos pocos artefactos con responsabilidades heterogéneas.

Estos puntos críticos no se trataron como una apreciación meramente cualitativa, sino como parte del perímetro estable analizado mediante la herramienta de medición. De este modo, el diagnóstico inicial y la evaluación posterior se apoyan sobre el mismo conjunto de elementos comparables.

3.3.3. Mezcla de responsabilidades entre capas

El segundo problema relevante era la mezcla de responsabilidades entre entrada HTTP, aplicación, dominio e infraestructura.

En la capa de controladores, ciertas rutas no se limitaban a validar la entrada y adaptar la salida, sino que orquestaban directamente scraping, enriquecimiento

de datos, persistencia y publicación de eventos. El ejemplo más claro era el flujo de creación de recetas desde web, que permanecía alojado dentro del controlador autenticado del módulo.

Esta combinación de responsabilidades era aún más acusada en **RecipeService**, que reunía operaciones muy heterogéneas.

- Coordinación de flujos de aplicación.
- Decisiones de permisos sobre actualización, borrado y lock/unlock.
- Acceso a repositorios y ejecución de consultas.
- Manipulación de filesystem y assets.
- Importación de recetas desde ZIP.
- Construcción de recetas desde imágenes con apoyo de OpenAI.
- Renderizado de plantillas y otras operaciones auxiliares.

Esta concentración hacía de **RecipeService** un componente demasiado amplio para actuar como unidad coherente de aplicación. En lugar de expresar un único propósito arquitectónico reconocible, reunía varias responsabilidades cuyo cambio no necesariamente respondía a la misma razón. Ello elevaba el coste cognitivo del módulo y hacía más difícil razonar sobre el impacto de cualquier modificación.

3.3.4. Acoplamiento con infraestructura concreta

El tercer problema del estado inicial era el acoplamiento directo con implementaciones concretas de infraestructura. En este punto, el problema no consistía simplemente en “usar infraestructura”, algo inevitable en un backend real, sino en que una parte importante de la lógica de orquestación dependía de detalles concretos sin mediación de contratos explícitos.

El estado de partida del proyecto ofrecía diversas muestras ilustrativas de ello.

- **RecipeService** instanciaba directamente **RecipeDataService** en seis puntos distintos.
- El controlador de recetas y el servicio de scraping utilizaban llamadas directas a la función global `create_from_html`.
- Varias operaciones relevantes dependían de `UploadFile`, `ZipFile`, copia de árboles de directorios, borrado de temporales y otros detalles del filesystem.
- La construcción de recetas desde imágenes dependía de un servicio concreto de OpenAI integrado en el ecosistema del servicio principal.

Estas dependencias producían dos efectos problemáticos. Por un lado, reducían la capacidad de sustituir colaboradores por dobles de prueba o por implementaciones alternativas sin reescribir código de orquestación. Por otro, dificultaban la explicitud arquitectónica: el módulo usaba infraestructura de forma intensiva, pero muchas de esas dependencias permanecían ocultas dentro de clases grandes en lugar de aparecer expresadas mediante puertos, factorías o adaptadores bien identificables.

3.3.5. Limitaciones de testabilidad aislada

La consecuencia más visible de la organización anterior era la limitada testabilidad aislada del módulo. Con un criterio estricto, los flujos relevantes del estado inicial apenas podían probarse de forma unitaria sin base de datos, filesystem real o sustitución explícita de funciones globales.

Los casos más representativos de esta limitación eran los siguientes.

- `create_from_zip`, que dependía de `UploadFile`, lectura de ZIP, copia de archivos e imagen en almacenamiento.
- `create_from_images`, que dependía de `OpenAI`, temporales en disco y persistencia real.
- `update_recipe_image` y `delete_recipe_image`, que instanciaban directamente servicios de almacenamiento.
- `_create_recipe_from_web`, que dependía de scraping y de colaboradores concretos embebidos en el controlador.
- `RecipeBulkScraperService.scrape`, que reutilizaba también la dependencia directa de scraping global.

En este estado del módulo, la mayor parte de la validación tendía a apoyarse en contextos más pesados o más acoplados de lo deseable. Esto no significa que el sistema fuera imposible de probar, sino que la arquitectura no favorecía la prueba aislada. Desde la perspectiva del TFM, este rasgo resultaba central porque la mejora perseguida incluía precisamente aumentar la capacidad del módulo para validar comportamiento con menor dependencia de infraestructura real.

3.3.6. Contaminación HTTP en persistencia

Otro problema específico, aunque arquitectónicamente relevante, era la presencia de conceptos HTTP dentro de la persistencia del módulo. `RepositoryRecipes` importaba y lanzaba `HTTPException`, especialmente en operaciones de validación de consultas y sugerencias. Además, parte de la semántica de `query_filter` y `order_by` heredaba ya un significado explícitamente HTTP desde utilidades genéricas del repositorio.

Este patrón introducía varios problemas arquitectónicos.

En primer lugar, se invertía la responsabilidad esperable entre capas. La persistencia no sólo recuperaba o validaba datos, sino que condicionaba directamente la forma HTTP del error externo.

En segundo lugar, dificultaba la reutilización conceptual del repositorio fuera de FastAPI. Aunque esa reutilización no fuera un objetivo inmediato del TFM, la presencia de `HTTPException` dentro de la persistencia impedía considerar al repositorio como una pieza limpia respecto al transporte.

En tercer lugar, complicaba la prueba unitaria del nivel de persistencia. La validación de errores asociados a consultas quedaba ligada a excepciones y códigos propios de la capa web, en lugar de apoyarse en errores de persistencia o de aplicación independientes del protocolo HTTP.

3.3.7. Concentración de lógica en `RecipeService`

Si bien varios de los problemas anteriores ya apuntan hacia este componente, merece la pena subrayar de forma específica el papel de `RecipeService` en el estado inicial. Este servicio funcionaba como punto crítico principal del módulo por una combinación de tamaño, centralidad y heterogeneidad funcional.

El problema de `RecipeService` no era sólo cuantitativo. Que un archivo sea grande no implica por sí mismo un defecto arquitectónico. Lo problemático aquí era que el servicio actuaba al mismo tiempo como punto de entrada informal para múltiples casos de uso, concentrador de infraestructura, lugar de definición de permisos y soporte de operaciones muy distintas entre sí. Cuando un componente reúne tal variedad de motivos de cambio, cualquier evolución local corre el riesgo de producir efectos laterales difíciles de acotar.

A esto se sumaba que el papel central de `RecipeService` hacía especialmente delicado abordar su refactorización. Intervenirlo de forma global aumentaría el riesgo de introducir regresiones en flujos importantes del backend. Por otra parte, dejar este foco sin tratar habría supuesto renunciar a corregir uno de los problemas estructurales más arraigados del módulo. Este equilibrio justificó abordar la intervención de forma incremental, sin reescritura masiva pero sin posponer indefinidamente el problema.

3.3.8. Uso de *service locator* y dependencias contextuales implícitas

El estado inicial presentaba también varias invocaciones directas a `get_repositories()` dentro del módulo, con un total de seis usos relevantes en el perímetro observado. Este patrón introducía un problema diferente al del acoplamiento concreto con clases: no ocultaba necesariamente una implementación específica, pero sí ocultaba dependencias contextuales importantes detrás de una resolución global.

Desde el punto de vista arquitectónico, el *service locator* generaba diversos efectos negativos.

- Ocultaba parte de las dependencias del módulo, porque las colaboraciones necesarias no aparecían explícitamente en la interfaz de los componentes.
- Hacía más difícil aislar flujos en pruebas, ya que parte de la dependencia se resolvía dinámicamente a través del contexto.
- Mezclaba composición y comportamiento dentro de componentes que, idealmente, deberían limitarse a ejecutar un caso de uso o una responsabilidad concreta.

Tratar este problema no requería una eliminación total y simultánea de todos los usos de *service locator*. Resultaba más razonable acotar la intervención y reducir su presencia de forma selectiva en aquellos puntos donde dificultaba especialmente la testabilidad y la separación de responsabilidades.

3.3.9. Relación entre los problemas detectados

Los problemas observados no eran independientes entre sí. La mezcla de responsabilidades favorecía la aparición de puntos críticos. Los puntos críticos, a su vez, tendían a concentrar dependencias concretas de infraestructura. Dicho acoplamiento reducía la testabilidad aislada. La falta de fronteras explícitas facilitaba que conceptos HTTP se filtraran a persistencia y que la composición de dependencias se resolviera mediante *service locator*.

Por ello, el diagnóstico del estado inicial no debía interpretarse como una lista dispersa de defectos puntuales, sino como un patrón estructural coherente: el módulo funcionaba, pero su evolución estaba penalizada por una organización interna poco explícita, demasiado concentrada y con límites de capa insuficientemente definidos.

3.3.10. Justificación de la intervención arquitectónica

La intervención arquitectónica se justificaba porque los problemas detectados afectaban directamente a la mantenibilidad, la testabilidad y la capacidad de evolución del módulo. En este contexto, el TFM no necesitaba añadir nuevas funcionalidades visibles para aportar valor; lo prioritario era reorganizar parte del backend existente para que el módulo pudiera evolucionar con menor coste y mayor control estructural.

Una solución puramente local habría resultado insuficiente. Añadir pruebas alrededor de un servicio monolítico, por ejemplo, no corregía por sí mismo la mezcla de responsabilidades ni el acoplamiento con infraestructura. Del mismo modo, mover una dependencia aislada no resolvía el problema de concentración funcional en `RecipeService` ni el desbordamiento de responsabilidad en ciertos controladores.

Esto explica por qué la respuesta apropiada era una intervención arquitectónica incremental: era necesario introducir fronteras explícitas, redistribuir responsabilidades y crear unidades más pequeñas y testeables, pero sin romper el encaje del módulo dentro de Mealie ni abrir una migración total del backend.

Precisamente por tratarse de un problema con múltiples manifestaciones interrelacionadas, la solución descartó una reconstrucción global y única. El diagnóstico fijó el orden de trabajo: hacer explícitos los límites, separar el código HTTP de la lógica de aplicación, retirar las dependencias web filtradas hacia persistencia, aislar la política de permisos y, finalmente, descomponer parcialmente el servicio principal. Este orden articula los ejes de actuación que se desarrollan en los apartados siguientes.

3.3.11. Síntesis

El análisis del estado inicial revela un módulo funcionalmente útil, pero estructuralmente tensionado por varios rasgos interrelacionados: concentración funcional,

mezcla de responsabilidades, acoplamiento directo con infraestructura, limitada testabilidad aislada, contaminación entre capas y dependencias contextuales implícitas. Esta combinación no debe leerse como una suma casual de defectos locales, sino como un cuadro coherente con la literatura sobre erosión, *smells* arquitectónicos y deuda técnica arquitectónica [13,19,24]. Por ello, justificaba una intervención arquitectónica orientada a hacer más explícitas las dependencias, más cohesionados los flujos y más controlable la evolución futura del módulo.

3.4. Diseño de las soluciones adoptadas

3.4.1. Punto de partida del diseño

Una vez caracterizado el estado inicial, el siguiente paso consistía en decidir qué tipo de solución resultaba compatible con el problema detectado. La respuesta no podía ser una reescritura completa del módulo ni la aplicación rígida de un estilo arquitectónico idealizado. El trabajo se desarrollaba sobre un sistema real de código abierto, con comportamiento existente, dependencias ya integradas en Mealie y un alcance estrictamente limitado al módulo de gestión de recetas.

Por ello, el diseño de la solución partió de una premisa doble. Por un lado, se necesitaba una arquitectura objetivo suficientemente clara para orientar la transformación. Por otro, esa arquitectura debía utilizarse como referencia incremental, no como un estado final obligatorio que tuviera que alcanzarse en una sola intervención. Esta premisa es coherente tanto con la lógica de ocultación de decisiones de cambio probable formulada por Parnas [1] como con la lectura pragmática de refactorización sobre software existente defendida por Fowler y Feathers [5,3].

En consecuencia, la arquitectura propuesta no funcionó como un plan cerrado de migración total, sino como una referencia de diseño para priorizar fases, delimitar la intervención y comprobar si los cambios avanzaban hacia una estructura más mantenible, testeable y evolutiva.

3.4.2. Criterios de diseño

Las decisiones adoptadas en el diseño de la solución obedecieron a cinco criterios principales.

Compatibilidad con el sistema existente. Cada intervención tenía que preservar los contratos públicos y el comportamiento observable de los flujos afectados en la medida necesaria para el alcance del TFM. Este criterio excluía estrategias basadas en reescribir el módulo desde cero o en introducir cambios transversales difíciles de validar dentro de una intervención acotada.

Delimitación estricta del alcance. La solución se ciñó al módulo de recetas del backend. Aunque algunas carencias del diseño original estaban conectadas con patrones generales del sistema, la intervención no pretendía convertirse en

una reestructuración transversal de Mealie. Este condicionante explica que quedaran fuera decisiones de mayor alcance, como la reconfiguración completa del repositorio genérico compartido, cuando bastaba con intervenir la persistencia específica del módulo de recetas.

Mejora de atributos de desarrollo del software. La solución debía orientarse a mantenibilidad, testabilidad, desacoplamiento, separación de responsabilidades y escalabilidad del desarrollo entendida como capacidad de evolución estructural. El diseño, en consecuencia, no se evaluó por rendimiento, throughput ni despliegue, sino por la calidad de las fronteras introducidas, la reducción de acoplamientos concretos y la posibilidad de aislar flujos relevantes del módulo.

Introducción progresiva de fronteras explícitas. El diseño asumió que muchas dependencias del estado inicial no necesitaban desaparecer de inmediato, pero sí debían hacerse más explícitas y controlables. La estrategia adecuada no era eliminar toda infraestructura del módulo, sino expresarla detrás de puertos, adaptadores, políticas o casos de uso que permitieran separar mejor responsabilidades.

Atribución clara de resultados. Cada decisión de diseño debía vincularse con un problema del estado inicial y con una fase concreta de la intervención. Este criterio evitaba concentrar modificaciones heterogéneas en un único bloque y reforzaba la posibilidad de interpretar después qué efecto había producido cada cambio.

3.4.3. Arquitectura objetivo de referencia

La arquitectura objetivo adoptada como referencia se inspiró en una familia de enfoques basados en límites explícitos, dependencia hacia el interior y separación entre aplicación, dominio e infraestructura, próxima a puertos y adaptadores en sentido pragmático [1,7,21]. De acuerdo con el estado del arte, esta arquitectura no se formuló como una adhesión dogmática a un patrón, sino como un marco práctico para reorganizar el módulo.

En términos esquemáticos, la arquitectura objetivo distinguía cuatro niveles.

- Adaptadores de entrada: controladores HTTP y otros puntos de entrada responsables de validar, traducir y exponer contratos externos.
- Aplicación: casos de uso explícitos encargados de orquestar flujos concretos del módulo.
- Dominio: políticas y reglas de negocio que debían existir con identidad propia cuando su responsabilidad lo justificase.
- Adaptadores de salida e infraestructura: repositorios, almacenamiento, scraper, OpenAI, renderizado y publicación de eventos, consumidos a través de puertos cuando ello aportara valor estructural.

El propósito de esta solución era desplazar la orquestación fuera de los controladores de entrada y de servicios excesivamente concentrados como **RecipeService**, hacer explícitas las llamadas a servicios externos y encapsular dependencias técnicas detrás de interfaces adaptables.

Su función era, por tanto, servir como mapa de decisión y no como contrato cerrado de entrega. El valor de la arquitectura objetivo en este TFM no reside en declarar un estado ideal plenamente alcanzado, sino en hacer comprensible por qué determinadas refactorizaciones se priorizaron y otras quedaron deliberadamente fuera de alcance.

3.4.4. Elementos arquitectónicos introducidos

La solución diseñada se articuló en torno a cuatro familias de artefactos: puertos, adaptadores, casos de uso y políticas. Cada una respondía a una faceta concreta del diagnóstico inicial.

Puertos. El uso de puertos pretendía modelar las dependencias salientes del módulo evitando que la capa de aplicación quedara acoplada a implementaciones concretas. No se trataba sólo de añadir interfaces, sino de hacer explícitas y sustituibles aquellas colaboraciones que en el código original aparecían como llamadas directas integradas en el flujo principal. Responsabilidades como la persistencia, el almacenamiento de imagen, la importación por *scraping*, la publicación de eventos en el bus interno o el control de credenciales sobre servicios externos pasaron a expresarse como contratos inyectables y verificables de forma aislada. Esta decisión está alineada con la literatura que vincula la modularidad explícita y la reducción de dependencias estructurales con un menor deterioro arquitectónico a lo largo del tiempo [13,15].

Adaptadores. Los adaptadores permitieron conservar la infraestructura real del sistema sin que la capa de aplicación dependiera de ella directamente. SQLAlchemy, filesystem, scraper legacy, OpenAI, renderizado de plantillas y event bus continuaron existiendo, pero se integraron desde piezas explícitas y no únicamente desde servicios monolíticos o funciones globales.

Casos de uso. Los casos de uso se introdujeron para separar flujos completos de aplicación que en el estado inicial estaban embebidos en controladores o en **RecipeService**. El criterio para incorporarlos no fue convertir cualquier método en un nuevo objeto, sino identificar flujos con entidad propia, múltiples colaboradores y alto valor arquitectónico o de prueba aislada. En ese sentido, la solución adoptada se aproxima a la literatura que propone responder a *architecture smells* y a responsabilidades excesivamente concentradas mediante refactorizaciones estructurales explícitas [19,13].

Políticas. Las políticas se introdujeron para extraer decisiones de negocio que en el estado inicial aparecían mezcladas con el flujo de orquestación principal y penalizaban la cohesión del módulo. El caso más claro era la lógica de permisos: **RecipeService** concentraba al mismo tiempo la operación principal y las comprobaciones de autorización asociadas, lo que generaba un acoplamiento innecesario entre reglas de acceso y orquestación, además de dificultar su prueba aislada.

3.4.5. Justificación de la secuencia de fases

La división del trabajo en fases no fue una conveniencia meramente narrativa, sino una decisión de diseño. Se adoptó porque la arquitectura objetivo no podía alcanzarse de manera segura mediante una única intervención sin mezclar demasiados cambios heterogéneos.

Fase 1 como preparación de fronteras. No resultaba razonable extraer casos de uso o políticas sin haber creado antes puntos mínimos de desacoplamiento respecto a infraestructura. Por ello, la primera fase se diseñó como creación de puertos y adaptadores básicos. Su objetivo no era simplificar de inmediato los puntos críticos, sino abrir una base estructural que hiciera posibles fases posteriores.

Fase 2 como primer caso de uso completo. El flujo `CreateRecipeFromWeb` se eligió como primer caso de uso porque concentraba en una sola unidad operaciones típicas de un endpoint complejo: lógica de controlador, importación externa por *scraping*, enriquecimiento, persistencia y publicación de eventos. Resultaba, por tanto, un ejemplo representativo para demostrar cómo trasladar la orquestación a una unidad de aplicación manteniendo intacto el comportamiento observable desde el cliente.

Fase 3 como corrección de la dirección de dependencias. La presencia de `HttpException` en `RepositoryRecipes` constituía un problema específico, bien acotado y con una lógica arquitectónica clara. Diseñar una fase exclusiva para este punto permitía corregir la contaminación HTTP sin mezclarla con la reducción del punto crítico principal ni con la extracción de permisos.

Fase 4 como extracción de una política de negocio. La lógica de permisos merecía un tratamiento separado porque no era únicamente infraestructura ni entrada HTTP. Su naturaleza correspondía a una política de negocio dependiente de acceso a datos. Diseñar una fase propia evitaba que esa extracción quedara diluida dentro de una refactorización más amplia de `RecipeService`.

Fase 5 como reducción material del punto crítico residual. Sólo después de haber introducido puertos, extraído casos de uso representativos y separado políticas como la de permisos resultaba razonable abordar la reducción del punto crítico principal. Por eso la Fase 5 se reservó para descomponer la parte de `RecipeService` que seguía concentrando demasiadas responsabilidades, manteniendo un alcance acotado para que el coste de la transición siguiera siendo manejable.

3.4.6. Alcance ejecutado y decisiones de diseño acotado

Una distinción clave en este apartado es la diferencia entre la arquitectura objetivo como referencia y el alcance realmente ejecutado. El trabajo no afirma haber convertido el módulo completo en una arquitectura hexagonal homogénea. Lo que afirma es haber empleado esa arquitectura como guía para introducir mejoras concretas, acumulativas y verificables.

En el plano ideal, la arquitectura de referencia apuntaba a un módulo en el que los controladores actuaran sólo como adaptadores de entrada, los flujos relevantes estuvieran expresados como casos de uso explícitos, las políticas de negocio estuvieran aisladas de infraestructura, la persistencia no dependiera de semántica HTTP, la mayor parte de las colaboraciones externas estuvieran mediadas por puertos y la composición de dependencias fuera explícita.

En la intervención realmente ejecutada, ese objetivo se materializó de forma parcial y deliberada.

- Se introdujeron puertos y adaptadores en los puntos donde el acoplamiento era más relevante para el alcance del trabajo.
- Se extrajeron casos de uso para varios flujos significativos, pero no para toda la superficie funcional del módulo.
- Se externalizó la política de permisos, pero no toda la lógica de dominio asociada al módulo de recetas.
- Se eliminó la contaminación HTTP en la persistencia específica del módulo, sin abrir una refactorización completa del repositorio genérico compartido.
- Se redujo parcialmente el uso de *service locator*, pero no se eliminó de forma total.
- `RecipeController` siguió siendo un punto crítico relevante del lado de entrada HTTP y `OpenAIRecipeService` continuó existiendo como dependencia concreta dentro del sistema.

Esta limitación fue una decisión metodológica, no una carencia accidental. Llevar la arquitectura objetivo hasta su extremo habría convertido el TFM en una migración de alcance muy superior, con mayor riesgo de regresión y menor capacidad para atribuir resultados a intervenciones concretas. El diseño adoptado prefirió una mejora parcial, pero defendible, frente a una transformación total difícil de cerrar dentro del alcance del trabajo.

En consecuencia, varias decisiones quedaron deliberadamente fuera de la intervención: no se rediseñó todo el backend de Mealie, no se sustituyeron todos los servicios existentes por casos de uso independientes, no se reescribió la infraestructura de scraping, filesystem u OpenAI, no se eliminó completamente el punto crítico de entrada HTTP y no se convirtió la arquitectura objetivo en un compromiso de homogeneidad total del módulo. Esta delimitación permite interpretar los resultados con mayor precisión y evita medir la intervención contra una referencia teórica que no formaba parte del alcance real.

3.4.7. Relación con el diagnóstico y el marco teórico

El diseño de la solución responde de forma directa a los problemas analizados en el apartado 3.3 y se alinea con el marco del Capítulo 2. Los puertos y adaptadores responden al acoplamiento con infraestructura concreta; los casos de uso, a la orquestación alojada en controladores y servicios demasiado amplios; las políticas, a la mezcla de reglas de negocio con flujos operativos; y la descomposición parcial de `RecipeService`, a la concentración funcional detectada en el estado inicial.

Desde el punto de vista teórico, la refactorización incremental sobre software existente justifica que la transformación se ejecute por fases y con compatibilidad funcional [5,3]. La separación entre aplicación, dominio e infraestructura, entendida aquí en un sentido pragmático y parcial, justifica la introducción progresiva de puertos, adaptadores, casos de uso y políticas [1,7]. La atención a degradación estructural y *architecture smells* justifica que la respuesta se formule como redistribución explícita de responsabilidades y no como simple limpieza local [19,13]. Por último, la orientación a mantenibilidad, testabilidad y evolución estructural justifica que el éxito del diseño se mida por fronteras explícitas, reducción de puntos críticos y capacidad de prueba aislada, no por rendimiento del producto final.

3.4.8. Síntesis

La solución adoptada no consistió en un rediseño total del módulo de recetas, sino en una arquitectura objetivo parcial y pragmática, utilizada como guía para intervenir de forma incremental sobre los problemas más relevantes del estado inicial. Su núcleo consistió en introducir fronteras explícitas, desplazar orquestación a casos de uso, externalizar políticas de negocio relevantes y corregir dependencias impropias entre capas, manteniendo un alcance compatible con Mealie y con la evaluación posterior del TFM.

3.5. Implementación de la solución integración en Mealie

3.5.1. Lógica general de implementación

La implementación de la solución no se articuló como una secuencia de parches independientes, sino como una transformación progresiva del módulo de recetas. Cada fase se apoyó en la anterior y preparó las condiciones para la siguiente. De este modo, el módulo no pasó directamente de un estado inicial a una arquitectura final, sino por una serie de estados intermedios controlados en los que se fueron haciendo explícitas dependencias, extrayendo flujos y reduciendo responsabilidades concentradas.

Esta lógica acumulativa resultaba especialmente importante porque la intervención se realizó sobre un sistema real y funcional, no sobre un prototipo aislado. Por ello, cada cambio debía integrarse en el sistema real conservando compatibilidad funcional suficiente, sin romper la organización existente del proyecto y sin abrir una migración transversal no prevista. La secuencia, además, no fue completamente lineal: la integración sobre el sistema real obligó a introducir ajustes locales de compatibilidad y a cerrar algunas decisiones durante la propia validación.

3.5.2. Visión de conjunto de las fases

La implementación se organizó en cinco fases acumulativas, resumidas en la Tabla 1.

Tabla 1: Resumen de las cinco fases acumulativas de la implementación.

Fase	Problema principal abordado	Cambio central realizado	Aportación al conjunto
Fase 1	Dependencias concretas de infraestructura y ausencia de fronteras explícitas	Introducción de puertos y adaptadores básicos	Prepara el módulo para una migración incremental sin cambiar todavía la distribución principal de la lógica
Fase 2	Orquestación compleja alojada en el controlador HTTP	Extracción de <code>createRecipeFromWeb</code> a la capa de aplicación	Demuestra que un flujo completo puede salir de HTTP y convertirse en caso de uso aislable
Fase 3	Contaminación HTTP en persistencia	Sustitución de <code>HTTPException</code> por excepciones de aplicación y mapeo en entrada	Restituye la dirección de dependencias entre persistencia y adaptadores de entrada
Fase 4	Lógica de permisos embebida en <code>RecipeService</code>	Extracción de <code>RecipePermissionPolicy</code> y de su acceso a datos	Convierte la autorización en una política explícita y testeable
Fase 5	Punto crítico residual de <code>RecipeService</code> y uso disperso de <i>service locator</i>	Extracción de varios casos de uso y composición más explícita en controladores base	Reduce materialmente el punto crítico principal y consolida la estrategia iniciada en las fases anteriores

3.5.3. Fase 1: introducción de puertos y adaptadores

Problema al que respondía. La Fase 1 respondía al hecho de que el módulo dependía directamente de infraestructura concreta en varios puntos clave. `RecipeService` instanciaba servicios de almacenamiento y dependía de colaboradores muy ligados a SQLAlchemy, filesystem y OpenAI. A su vez, el controlador de

recetas y el servicio de scraping utilizaban funciones legacy concretas para la creación desde web. En este contexto, resultaba difícil extraer lógica a unidades nuevas sin abrir antes un mínimo de desacoplamiento.

Cambio realizado. El cambio principal de la fase se centró en la introducción de `mealie/services/recipe/ports.py` y `mealie/services/recipe/adapters.py` como piezas explícitas del módulo. En ellas se definieron contratos para persistencia, almacenamiento, scraping y construcción de recetas desde imágenes, junto con adaptadores compatibles con la infraestructura existente.

Los elementos principales introducidos en esta fase fueron:

- `RecipeRepositoryPort`
- `RecipeDataStoragePort`
- `RecipeDataStorageFactoryPort`
- `RecipeScraperPort`
- `RecipeOpenAIBuilderPort`
- `SqlAlchemyRecipeRepositoryAdapter`
- `FileSystemRecipeDataStorageFactory`
- `LegacyRecipeScraperAdapter`

Además, `RecipeServiceBase` pasó a aceptar dependencias abstractas para repositorio, almacenamiento y OpenAI, manteniendo valores por defecto compatibles con el sistema legacy. También se ajustaron el controlador base de recetas y el servicio de scraping para ensamblar y consumir estos puertos sin alterar contratos públicos.

Resultado aportado al conjunto. La Fase 1 no simplificó todavía los puntos críticos principales ni redujo la complejidad algorítmica del módulo. Su aportación fue crear fronteras mínimas y explícitas para que la refactorización posterior no dependiera de accesos directos a infraestructura concreta. En otras palabras, la fase abrió el espacio arquitectónico necesario para que fases posteriores pudieran extraer casos de uso y políticas sin necesidad de reescribir primero toda la base de código.

Conviene precisar dónde encaja esta primera intervención dentro del recorrido global del trabajo. La Fase 1 amplió el código del módulo en algunos puntos, pero ese aumento respondió a la introducción de puertos y adaptadores explícitos allí donde antes existían dependencias directas a infraestructura. La integración se mantuvo deliberadamente conservadora: no se introdujeron capas adicionales más allá de las imprescindibles para hacer explícitos los puntos de desacoplamiento.

3.5.4. Fase 2: extracción de `CreateRecipeFromWeb`

Problema al que respondía. Tras la Fase 1, seguía existiendo un problema claro en la capa de entrada: el flujo de creación de recetas desde web permanecía alojado en el controlador HTTP. Ese flujo combinaba scraping, enriquecimiento de tags y categorías, persistencia de la receta y publicación de eventos. Aunque

algunas dependencias ya estaban mediadas por puertos, la responsabilidad de orquestación seguía estando en el lugar equivocado.

Cambio realizado. La intervención principal se centró en introducir `CreateRecipeFromWeb` en `mealie/services/recipe/application/create_recipe_from_web.py` y trasladar ahí la secuencia completa del flujo. Para hacerlo posible, se añadieron nuevos puertos auxiliares y sus adaptadores correspondientes:

- `RecipeCreationPort`
- `RecipeScrapedDataEnricherPort`
- `RecipeCreatedEventPort`
- `CallableRecipeCreationAdapter`
- `ScrapedRecipeDataEnricherAdapter`
- `CallableRecipeCreatedEventAdapter`

Como consecuencia, el controlador HTTP correspondiente pasó a delegar el flujo en el nuevo caso de uso y a limitar su responsabilidad a la validación de la entrada, la invocación de `CreateRecipeFromWeb` y la traducción del resultado al contrato HTTP de salida. Lejos de ser una simple reubicación de código, este cambio reorganizó la responsabilidad de orquestación: el flujo quedó expresado como una unidad de aplicación con identidad propia, evaluable de forma aislada y trazable frente a sus puertos.

Durante la integración de la fase se detectó una recursión de inicialización en `RecipeServiceBase` al construir el colaborador por defecto para OpenAI. La corrección fue deliberadamente mínima y conservadora, reutilizando la propia instancia cuando correspondía y evitando así una construcción recursiva innecesaria. Este tipo de ajuste ilustra la necesidad de validar tempranamente sobre el sistema real, ya que ciertos problemas sólo aparecen al integrar la solución en su contexto completo.

Resultado aportado al conjunto. La Fase 2 aportó el primer flujo completo del módulo expresado como caso de uso explícito y aislable. Con ello, la transformación dejó de ser únicamente preparatoria y mostró un efecto arquitectónico visible: el controlador dejó de orquestar scraping, enriquecimiento, persistencia y publicación de eventos como responsabilidad propia.

La aportación de esta fase debe leerse de forma acotada. No transformó toda la capa de entrada del módulo ni aspiraba a hacerlo. Su valor reside en haber demostrado, sobre un flujo representativo y sin alterar el comportamiento externo, que era viable extraer una orquestación completa desde HTTP hacia aplicación manteniendo la integración con eventos, enriquecimiento de datos y persistencia existente.

3.5.5. Fase 3: eliminación de semántica HTTP en persistencia

Problema al que respondía. La Fase 3 abordó un problema distinto del de las fases anteriores: la persistencia específica del módulo seguía hablando en términos HTTP. `RepositoryRecipes` importaba y lanzaba `HTTPException`, especialmente en errores de consulta y ordenación. Este acoplamiento no afectaba

directamente al uso funcional del módulo, pero sí degradaba la dirección de dependencias entre capas.

Cambio realizado. El cambio consistió en sustituir las invocaciones a `HTTPException` dentro del repositorio por excepciones propias de aplicación y trasladar a la capa HTTP la traducción al código de error correspondiente. Para ello se introdujeron en el módulo dos nuevas excepciones, `QueryFilterValidationError` y `OrderByValidationError`, y se eliminó el uso directo de `HTTPException` desde `RepositoryRecipes`. La capa de entrada se encarga ahora de mapear esas excepciones al código HTTP 400 esperado por los endpoints afectados.

Un detalle relevante del diseño de implementación fue que no se tocó el repositorio genérico compartido del sistema. En su lugar, se introdujeron overrides locales en `RepositoryRecipes` para la lógica necesaria de paginación y ordenación. Esta decisión mantuvo el alcance centrado en el módulo de recetas y evitó convertir la fase en una refactorización transversal del subsistema de repositorios.

Resultado aportado al conjunto. La Fase 3 no redujo el punto crítico principal ni eliminó el *service locator*, pero sí corrigió un defecto arquitectónico de capa muy nítido: la persistencia dejó de depender de conceptos HTTP. Esto reforzó la coherencia del diseño objetivo, ya que el módulo no solo empezaba a expresar mejor sus colaboraciones, sino también a respetar mejor la dirección de dependencias entre persistencia y entrada.

3.5.6. Fase 4: extracción de `RecipePermissionPolicy`

Problema al que respondía. Tras las tres fases anteriores, `RecipeService` seguía concentrando la lógica de permisos del módulo: decidía directamente quién podía actualizar, eliminar o bloquear una receta, y entremezclaba esas comprobaciones con la operación principal de cada flujo. Además, parte de la evaluación se apoyaba en consultas SQL embebidas en el propio servicio, sin mediación de una capa de acceso a datos específica.

Cambio realizado. La Fase 4 introdujo `RecipePermissionPolicy` en `mealie/services/recipe/domain/permission_policy.py` como una política explícita del módulo. Junto con ella se añadieron dos contratos:

- `RecipePermissionAccessPort`
- `RecipePermissionPolicyPort`

Se incorporó además el adaptador `SqlAlchemyRecipePermissionAccessAdapter`, encargado de encapsular el acceso a datos necesario para evaluar ownership, bloqueo y restricciones de edición. A partir de ahí, `RecipeServiceBase` pasó a aceptar una política de permisos inyectable y el servicio dejó de contener directamente los métodos `can_update`, `can_delete` y `can_lock_unlock`.

En la práctica, esto permitió que las comprobaciones que antes residían en métodos como `_pre_update_check`, `update_recipe_image` o el flujo de borrado masivo `delete_many` delegasen ahora en la nueva política, evaluable de forma aislada y con un contrato explícito.

Resultado aportado al conjunto. La Fase 4 mejoró la cohesión del servicio principal sin requerir todavía una fragmentación completa del punto crítico. La autorización pasó a ser una unidad arquitectónica con identidad propia, más cercana al dominio y más fácil de validar de forma aislada. Esto era importante porque mostraba que la estrategia de la solución no sólo movía infraestructura detrás de puertos, sino que también permitía explicitar reglas de negocio que en el estado inicial estaban diluidas dentro de un servicio demasiado amplio.

3.5.7. Fase 5: descomposición adicional de `RecipeService` e integración más explícita

Problema al que respondía. Al cierre de la Fase 4, el principal problema residual del módulo continuaba siendo `RecipeService`. Aunque ya había perdido parte de su acoplamiento concreto y una responsabilidad de permisos relevante, seguía acumulando flujos heterogéneos de importación, duplicación y gestión de imagen. Al mismo tiempo, seguían existiendo múltiples invocaciones directas a `get_repositories()` y el número de flujos estrictamente aislables todavía era reducido para los objetivos del trabajo.

Criterio de selección del alcance real. El criterio adoptado en esta última fase descartó cualquier intento de descomposición total de `RecipeService` y optó por una intervención acotada y selectiva. La prioridad fue extraer sólo aquellos flujos que:

- residían todavía dentro del punto crítico principal;
- podían expresarse como casos de uso completos con puertos claros;
- disponían de cobertura funcional existente o construible para preservar comportamiento;
- y retiraban responsabilidad real del servicio, no meros helpers internos.

Con ese criterio se seleccionaron los flujos de creación desde ZIP, creación desde imágenes, duplicación, actualización de imagen y borrado de imagen. Además, se extrajo el renderizado de plantillas fuera de `RecipeService` y se centralizó la construcción de repositorios *group-scoped* en el controlador base de recetas.

Cambio realizado. La implementación incorporó cinco nuevos casos de uso en `mealie/services/recipe/application/`:

- `CreateRecipeFromZip`
- `CreateRecipeFromImages`
- `DuplicateRecipe`
- `UpdateRecipeImage`
- `DeleteRecipeImage`

Cada uno de estos casos de uso requirió introducir nuevos puertos sobre los colaboradores realmente necesarios: extracción y descompresión de archivos ZIP, sanitización de datos importados, transferencia y previsualización de imagen, persistencia de la receta y renderizado de plantillas. En paralelo, `mealie/routes/recipe/_base.py` concentró la composición de los `group_repositories` y

se convirtió en el punto donde se ensamblan los repositorios necesarios para los flujos extraídos.

Las rutas principales de `recipe_crud_routes.py` dejaron de delegar los flujos seleccionados en `RecipeService` y empezaron a invocar directamente esos nuevos casos de uso. Esto era una condición necesaria para que la reducción del punto crítico fuese material y no meramente cosmética. Si el servicio hubiera conservado métodos intermediarios internos, la responsabilidad real habría permanecido en el mismo punto aunque el código se hubiera desplazado ligeramente.

Como consecuencia, `RecipeService` dejó de contener `create_from_zip`, `create_from_images`, `duplicate_one`, `update_recipe_image`, `delete_recipe_image` y `render_template`. Además, dejó de resolver internamente el repositorio *group-scoped* mediante `get_repositories()` y pasó a aceptar un `RecipeRepositoryPort` inyectado desde el exterior.

Esta integración exigió algunos ajustes laterales en componentes que utilizaban directamente los flujos extraídos, como `bulk_actions.py`, `controller_group_recipe_actions.py` o `_migration_base.py`. Estos cambios fueron deliberadamente mínimos: no ampliaron el alcance del trabajo ni introdujeron nuevas responsabilidades en esos componentes, pero permitieron preservar el comportamiento funcional al apoyar los flujos modificados sobre las nuevas unidades de aplicación.

Resultado aportado al conjunto. La Fase 5 fue la fase que transformó de forma más visible la estructura del módulo. Su aportación no consistió sólo en añadir nuevas unidades de aplicación, sino en retirar de `RecipeService` varias responsabilidades de alto peso arquitectónico. Al mismo tiempo, la centralización del ensamblaje en el controlador base permitió reducir parte del *service locator* sin abrir una reescritura total del módulo.

Con ello, la fase consolidó el hilo de intervención completo: el módulo ya no dependía únicamente de puertos introducidos en fases tempranas, sino que empezaba a organizar parte sustancial de sus flujos alrededor de casos de uso pequeños, explícitos e integrados sobre Mealie.

3.5.8. Integración acumulativa de las fases

Leídas en conjunto, las cinco fases configuran una progresión coherente y acumulativa. La secuencia no pretendía aplicar de una sola vez la arquitectura objetivo, sino introducir cambios controlados que pudieran integrarse en Mealie sin abrir una transformación transversal del sistema. Cada fase abordó un problema concreto del estado inicial y, al mismo tiempo, dejó preparada una base más favorable para la siguiente.

La Fase 1 introdujo los puntos mínimos de desacoplamiento mediante puertos y adaptadores. Sobre esa base, la Fase 2 permitió comprobar tempranamente que un flujo completo podía salir del controlador HTTP y expresarse como caso de uso de aplicación, aun cuando todavía persistían otros problemas estructurales en el módulo. La Fase 3 corrigió después una dependencia impropia entre persis-

tencia y HTTP, y la Fase 4 aisló una política de negocio relevante. Finalmente, la Fase 5 abordó la reducción material del punto crítico residual concentrado en `RecipeService`.

El orden de integración fue, por tanto, deliberado. Extraer un primer caso de uso tras introducir puertos permitía validar pronto la viabilidad del enfoque sobre el sistema real, sin esperar a que todos los límites arquitectónicos estuvieran ya corregidos. Las fases posteriores refinaron esa estructura corrigiendo dependencias entre capas y separando políticas antes de intervenir con mayor intensidad sobre `RecipeService`. De este modo, la integración siguió una lógica incremental: abrir fronteras, validar el desplazamiento de un flujo representativo, corregir límites de capa, aislar reglas de negocio y reducir después el punto crítico principal sobre una base más estable.

3.5.9. Límites de la implementación

La implementación alcanzada no agota el alcance ideal descrito en el apartado 3.4. Una vez cerrada la intervención permanecen límites visibles, asumidos de antemano dentro del trabajo:

- `RecipeController` continuaba siendo un punto crítico importante del lado HTTP.
- Persistían usos residuales de `get_repositories()` dentro del módulo.
- `OpenAIRecipeService` seguía existiendo como dependencia concreta dentro del ecosistema del servicio.
- El módulo no se había convertido por completo en una arquitectura hexagonal homogénea.

Estos límites no contradicen la implementación realizada. Al contrario, delimitan con precisión el alcance del trabajo y evitan presentar como logrado lo que sólo formaba parte de la arquitectura objetivo de referencia.

3.5.10. Síntesis

La implementación en Mealie se desarrolló mediante una secuencia de cinco fases acumulativas que transformó progresivamente el módulo de recetas sin necesidad de una reescritura total. El resultado no fue una arquitectura completa o definitiva, sino una mejora estructural integrada sobre el sistema real: más dependencias explícitas, más flujos expresados como casos de uso, una política de permisos separada, persistencia sin semántica HTTP y una reducción material del punto crítico principal del módulo.

Esta descripción de la implementación no basta, por sí sola, para demostrar la mejora del módulo. Por ello, el Capítulo 4 retoma los estados generados por estas fases y los evalúa mediante métricas estructurales, pruebas focalizadas y validación funcional mínima, con el fin de valorar si la intervención produjo mejoras verificables en desacoplamiento, testabilidad y capacidad de evolución.

4. Medición y evaluación de la mejora conseguida

4.1. Diseño de la evaluación

4.1.1. Alcance y propósito de la evaluación

La evaluación se orienta a determinar si la intervención arquitectónica descrita en el Capítulo 3 produjo una mejora defendible en los términos planteados por el TFM. No se evalúa si el módulo de recetas de Mealie alcanza una arquitectura ideal, ni se miden rendimiento, latencia o comportamiento bajo carga. El análisis se centra en atributos de desarrollo del software: desacoplamiento, separación de responsabilidades, reducción de puntos críticos y testabilidad aislada, entendidos como indicadores de una mayor capacidad de evolución estructural.

Desde la literatura de evaluación arquitectónica, esta lógica resulta compatible, en sentido analógico y no como aplicación formal de un método completo, con el análisis basado en escenarios y con ATAM. Ambos enfoques sitúan la modificabilidad, los compromisos entre atributos de calidad y la discusión explícita de *trade-offs* como objetos legítimos de evaluación de arquitectura [9,10].

4.1.2. Estado inicial como línea base

La evaluación parte de un estado inicial reproducible del módulo de recetas previo a la introducción de puertos, adaptadores, casos de uso y políticas extraídas. Dicho estado inicial no funciona como una referencia abstracta, sino como el punto de comparación estable sobre el que se miden los efectos acumulados de la intervención y que después se reutiliza frente a los estados de cierre de las fases relevantes.

Su valor metodológico es doble. En primer lugar, permite describir con precisión el estado inicial del módulo antes de intervenirlo: puntos críticos principales, dependencias concretas, *service locator*, contaminación HTTP y escasa capacidad de prueba aislada. En segundo lugar, permite comparar estados sucesivos del mismo módulo sin atribuir a la mejora cambios que en realidad podrían deberse a otra evolución paralela del sistema.

La comparación con el punto de partida se realizó de forma sistemática mediante el script `support-files/recipe_module_metrics.py`, que mantuvo estable el perímetro de medición a lo largo de todas las fases. La descripción detallada de ese perímetro y de las métricas asociadas figura en el Anexo B.

4.1.3. Lógica de evaluación por fases

La evaluación no se diseñó como un juicio único emitido al final del trabajo, sino como una secuencia acumulativa de comprobaciones asociadas a las fases de intervención. Esta decisión resultaba necesaria porque las fases no perseguían exactamente el mismo tipo de mejora.

- La Fase 1 era preparatoria y buscaba abrir fronteras explícitas frente a infraestructura.
- La Fase 2 debía demostrar que un flujo completo podía salir del controlador HTTP y convertirse en unidad de aplicación aislable.
- La Fase 3 debía corregir la dirección de dependencias entre persistencia y entrada HTTP.
- La Fase 4 debía comprobar que la autorización podía existir como política explícita y testeable.
- La Fase 5 debía mostrar una reducción material del punto crítico residual de `RecipeService` y una mejora visible de la testabilidad del módulo.

Por ello, exigir a todas las fases el mismo patrón de mejora habría resultado metodológicamente incorrecto. Algunas fases estaban orientadas sobre todo a acoplamiento, otras a separación de responsabilidades y otras a reducción de puntos críticos o a prueba aislada. La evaluación debía respetar esa diferencia y, al mismo tiempo, conservar una lectura acumulativa del conjunto.

4.1.4. Selección de métricas estructurales

Como instrumento objetivo de comparación, se definió un conjunto fijo de métricas estructurales agrupadas en cuatro dimensiones: acoplamiento, complejidad, tamaño y testabilidad. Estas métricas no pretenden auditar el sistema en términos absolutos, sino ofrecer una base estable de comparación antes y después de la intervención. Su selección se apoya en los atributos de calidad recogidos en ISO/IEC 25010 [6] y en formulaciones clásicas como las de McCabe [8] o Chidamber y Kemerer [2], aplicadas con un criterio práctico orientado al contexto de la intervención. Esta lectura es coherente con la visión de evolución del software propuesta por Ford, Parsons y Kua [4], centrada en preservar atributos clave a lo largo del tiempo.

Métricas de acoplamiento

Tabla 2: Métricas estructurales de acoplamiento (C1–C4).

ID	Métrica	Sentido evaluativo
C1	Referencias concretas de infraestructura en la orquestación del módulo	Mide hasta qué punto servicios y controladores dependen directamente de implementaciones concretas en lugar de hacerlo mediante contratos explícitos

ID	Métrica	Sentido evaluativo
C2	Dependencias salientes inyectables explícitas	Mide el grado de explicitud arquitectónica de las colaboraciones externas
C3	Dependencias HTTP fuera de la capa de entrada	Mide contaminación de capas, especialmente en persistencia y servicios
C4	Invocaciones directas a <code>get_repositories()</code>	Mide la presencia del patrón <i>service locator</i> dentro del módulo observado

Métricas de complejidad y puntos críticos

Tabla 3: Métricas estructurales de complejidad y puntos críticos (X1–X3).

ID	Métrica	Sentido evaluativo
X1	Complejidad estructural por función	Permite observar si la complejidad se mantiene concentrada en unos pocos flujos o se redistribuye a unidades más pequeñas
X2	Número de métodos por clase	Permite detectar concentración funcional en puntos críticos de gran amplitud
X3	Extensión estructural de clase	Permite observar si clases críticas como <code>RecipeService</code> reducen realmente su peso estructural

Métricas de tamaño

Tabla 4: Métricas estructurales de tamaño (S1–S2).

ID	Métrica	Sentido evaluativo
S1	LOC por archivo crítico	Permite seguir la evolución de archivos concretos del módulo
S2	Tamaño total del perímetro instrumentado	Permite observar si la complejidad se concentra o se redistribuye dentro del conjunto observado, evitando interpretar el tamaño de forma simplista

Métricas de testabilidad

Tabla 5: Métricas estructurales de testabilidad (T1–T3).

ID	Métrica	Sentido evaluativo
T1	Dependencias externas sustituibles por contratos explícitos	Relee la explicitud estructural de C2 desde la capacidad potencial de aislamiento del módulo
T2	Flujos unitariamente evaluables sin base de datos o filesystem con criterio estricto	Mide testabilidad real en términos fuertes, no meramente teóricos
T3	Necesidad de reemplazos dinámicos de funciones globales	Mide dependencia residual de técnicas de aislamiento frágiles

Conviene precisar dos límites metodológicos desde el inicio. En primer lugar, no todas las métricas miden exactamente el mismo perímetro. En particular, S2 recoge el LOC total del conjunto de archivos críticos y componentes extraídos incluidos en el seguimiento del script, no el tamaño completo de Mealie.

En segundo lugar, C2 y T1 parten del mismo recuento de dependencias inyectables, pero se interpretan de forma distinta. C2 se utiliza como indicador de explicitud arquitectónica, mientras que T1 se interpreta como potencial de aislamiento. La evidencia más directa de testabilidad real se apoya en T2 y en las pruebas ejecutadas.

La selección de estas métricas está alineada con los objetivos formulados en el Capítulo 1: reducir acoplamiento directo, mejorar separación entre capas, hacer

visible la disminución de puntos críticos y aumentar la capacidad de prueba aislada del módulo.

4.1.5. Criterio de comparación antes y después

La comparación se mantuvo constante: medir un conjunto fijo de indicadores estructurales sobre el módulo intervenido, registrarlos al cierre de cada fase y analizar las variaciones únicamente respecto a los objetivos específicos de esa fase.

Este criterio conlleva dos implicaciones importantes. La primera es que la mejora no se deduce de una única métrica aislada. La segunda es que no toda variación debe interpretarse en el mismo sentido. Por ejemplo, un aumento de **S2** puede ser compatible con una mejora arquitectónica si ese crecimiento responde a la aparición de puertos, adaptadores, políticas o casos de uso pequeños que redistribuyen la complejidad fuera de un punto crítico monolítico.

Asimismo, una fase podía considerarse exitosa aunque algunas métricas globales permanecieran estables, siempre que las métricas y evidencias relevantes para su objetivo sí mostrasen mejora. Esta lógica fue especialmente importante en la Fase 3, orientada a **C3**, y en la Fase 4, centrada en cohesión y testabilidad aislada de la política de permisos.

4.1.6. Pruebas, validación funcional y medición reproducible

La evaluación de cada fase se sostuvo sobre tres tipos de evidencia complementaria.

Pruebas unitarias

Las pruebas unitarias permitieron comprobar si las nuevas unidades introducidas por la refactorización eran realmente aislables y no sólo teóricamente desacopladas. Su función principal consistió en respaldar sobre todo **T2** y verificar que el potencial de aislamiento sugerido por **C2** y releído en **T1** se materializaba en flujos ejecutables con dobles de prueba sobre puertos, sin necesidad de base de datos ni filesystem real.

Validación funcional mínima

La validación funcional cumplió un propósito distinto: comprobar que los flujos HTTP o de integración afectados por la refactorización conservaban el comportamiento observable relevante. Esta validación se diseñó siempre de forma mínima y focalizada, porque la intervención no pretendía ejecutar una regresión exhaustiva de todo Mealie, sino preservar el comportamiento de los flujos más sensibles respecto a cada fase. Esta exigencia de preservación de comportamiento durante la mejora del diseño se corresponde con la práctica recogida por Fowler [5] y Feathers [3] para la intervención controlada sobre código existente.

Script reproducible de métricas

El script `support-files/recipe_module_metrics.py` aportó la base cuantitativa de la comparación. Funcionó como instrumento de medida común, generando la fotografía inicial del módulo y los recuentos sucesivos asociados al cierre de cada fase. De este modo, la argumentación no descansa en apreciaciones cualitativas del autor, sino en datos producidos de forma reproducible sobre un perímetro instrumentado común del módulo, ampliado de forma declarada cuando la intervención incorporó nuevos componentes relevantes.

4.1.7. Síntesis

La evaluación se diseñó, en definitiva, como un marco de comparación reproducible entre el estado inicial y los estados sucesivos del módulo, apoyado en métricas estructurales y en dos formas de evidencia ejecutable: pruebas unitarias y validación funcional mínima. Esta combinación responde de forma directa al problema del TFM: no bastaba con rediseñar la arquitectura; era necesario demostrar que el módulo había mejorado en capacidad de evolución estructural sin perder el comportamiento observable de los flujos intervenidos.

4.2. Validación por fases de la intervención

4.2.1. Propósito de la validación por fases

La validación por fases no debe interpretarse como una colección de comprobaciones aisladas, sino como una lectura progresiva de la intervención. Cada fase se cerró con un tipo de evidencia acorde con el problema que abordaba. A medida que el módulo ganaba fronteras explícitas y unidades más pequeñas, también aumentaba el tipo de evidencia que podía exigirse al cierre de cada intervención.

4.2.2. Fase 1: validación estructural preparatoria

La Fase 1 tuvo una función distinta a la de las fases posteriores. Su objetivo era introducir puertos y adaptadores como frontera explícita frente a infraestructura concreta y preparar la capa de servicio para descomposiciones posteriores. Por ello, su validación se centró sobre todo en la comparación estructural respecto al estado inicial y no exigió todavía una batería específica de pruebas unitarias dedicadas.

Lo que se quiso validar en esta etapa era que el módulo dejara de depender directamente de varias implementaciones concretas y que comenzaran a existir puntos de sustitución explícita. La comparación mostró precisamente eso: las instanciaciones directas de `RecipeDataService` en `RecipeService` pasaron de seis a cero, las llamadas directas a `create_from_html` en la orquestación principal desaparecieron y el servicio principal ganó dependencias salientes expresadas mediante puertos.

El cierre metodológico de la Fase 1 fue, por tanto, preparatorio. La fase podía considerarse exitosa en desacoplamiento inicial, pero dejaba abiertas dos cuestiones que debían demostrarse en fases posteriores: que esas fronteras habilitaban flujos realmente aislables y que la mejora estructural inicial podía traducirse en una reducción visible de puntos críticos y responsabilidades.

4.2.3. Fase 2: validación del primer caso de uso explícito

A diferencia de la fase anterior, la Fase 2 exigía una validación más rigurosa. La evaluación debía comprobar tres aspectos: que el flujo `CreateRecipeFromWeb` podía sacarse del controlador HTTP, que la nueva unidad de aplicación podía probarse de forma aislada con criterio estricto y que el comportamiento externo observable de los endpoints afectados se mantenía intacto.

Para ello se ejecutaron tres tipos de evidencia:

- Una batería unitaria específica en `tests/unit_tests/services_tests/recipe_tests/test_create_recipe_from_web.py`, con cinco pruebas asíncronas sobre el caso de uso y dobles de prueba para scraping, enriquecimiento, creación y publicación de eventos.
- Una validación funcional mínima en `tests/integration_tests/user_recipe_tests/test_recipe_create_from_web_phase2.py`, con dos pruebas HTTP sobre `POST /api/recipes/create/url` y `POST /api/recipes/create/html-or-json`.
- La comparación cuantitativa reproducible mediante `support-files/recipe_module_metrics.py`.

La fase fue metodológicamente relevante porque, por primera vez, la intervención no sólo mostraba una mejora de estructura, sino también un flujo estrictamente aislable con criterio fuerte. La batería unitaria produjo 5 `passed` y la validación funcional mínima 2 `passed`. En las métricas, T2 pasó de 0 a 1, C2 subió de 3 a 7 respecto a la Fase 1 y T3 permaneció en 0, confirmando que el flujo podía aislarse sin reemplazar funciones globales durante la prueba.

En contrapartida, los problemas heredados que esta fase no abordaba se mantuvieron visibles. Ni C3 ni C4 mostraron mejora apreciable: la posibilidad de aislar un caso de uso concreto convivía todavía con dependencias HTTP en niveles inferiores y con invocaciones repetidas a `get_repositories()` desde el módulo.

4.2.4. Fase 3: validación de la separación entre persistencia y HTTP

La Fase 3 se diseñó para corregir un problema acotado: la dependencia de `RepositoryRecipes` respecto a `HTTPException`. Lo que se quiso validar fue que la persistencia dejara de usar semántica HTTP para expresar errores de consulta y que, al mismo tiempo, los endpoints afectados mantuvieran el contrato externo de error 400.

La evidencia ejecutada consistió en lo siguiente:

- tres pruebas focalizadas de repositorio en `tests/unit_tests/repository_tests/test_recipe_repository_phase3.py`, orientadas a `QueryFilterValidationError` y `OrderByValidationError`;
- dos pruebas HTTP en `tests/integration_tests/user_recipe_tests/test_recipe_phase3_http_mapping.py`, sobre `GET /api/recipes` y `GET /api/recipes/suggestions` con parámetros inválidos;
- comparación cuantitativa reproducible del perímetro instrumentado.

Los resultados respaldaron este cierre. La batería unitaria registró 3 **passed** y la validación funcional 2 **passed**. En la serie histórica, C3 pasó de 2 a 0, manteniéndose sin regresión los avances previos en C1, C2, T1, T2 y T3. El leve incremento observado en S2 se explica por el desplazamiento del manejo de errores HTTP desde la persistencia hacia la capa de entrada, donde ahora se realiza el mapeo explícito.

Tras esta fase seguían pendientes dos problemas principales: la lógica de permisos incrustada en `RecipeService` y la permanencia del *service locator*. El cierre de la Fase 3 resolvía, por tanto, la dirección de dependencias entre persistencia y entrada, pero no pretendía todavía simplificar el punto crítico principal del módulo.

4.2.5. Fase 4: validación de la política de permisos

La Fase 4 se orientó a la extracción de `RecipePermissionPolicy`. Lo que se quiso validar aquí era que la autorización dejase de estar incrustada como detalle interno de `RecipeService`, que la nueva política pudiera evaluarse unitariamente con criterio estricto y que los escenarios funcionales sensibles de permisos siguieran comportándose como antes.

La evidencia ejecutada combinó tres elementos:

- cinco pruebas focalizadas sobre `RecipePermissionPolicy` en `tests/unit_tests/services_tests/recipe_tests/test_recipe_permission_policy.py`;
- una selección de cuatro escenarios de integración ya existentes en `tests/integration_tests/user_recipe_tests/test_recipe_owner.py`;
- comparación cuantitativa reproducible del perímetro instrumentado del módulo.

El resultado fue consistente con el objetivo de la fase. La batería unitaria produjo 5 **passed**, la selección funcional 4 **passed** y la serie acumulada mostró que C2 pasaba de 7 a 9, T1 de 7 a 9 y T2 de 1 a 2. Además, `RecipeService` reducía su tamaño respecto a la Fase 3 y la política de permisos aparecía ya como unidad explícita del módulo.

Los límites de la fase también quedaron claros. C4 seguía sin moverse y el SQL de permisos seguía existiendo, aunque ahora encapsulado en un adaptador de infraestructura. Es decir, la fase mejoró cohesión y testabilidad, pero no podía presentarse como eliminación del *service locator* ni como descomposición completa del servicio principal.

4.2.6. Fase 5: validación de la reducción de `RecipeService`

La Fase 5 cerró la intervención con el cambio de mayor alcance estructural. Pretendía validar cuatro aspectos: que parte de la lógica concentrada en `RecipeService` podía extraerse como casos de uso explícitos, que esos casos de uso eran realmente aislables con criterio estricto, que el comportamiento externo observable de los flujos afectados se mantenía estable y que las métricas estructurales registraban descensos materiales en C4, X2 y X3, junto con un aumento relevante de T2.

La evidencia ejecutada comprendió:

- seis pruebas focalizadas en `tests/unit_tests/services_tests/recipe_tests/test_recipe_phase5_use_cases.py`;
- una selección funcional mínima sobre ZIP, OpenAI, actualización de imagen, borrado de imagen y duplicación entre hogares, reutilizando varios tests de integración existentes;
- la comparación cuantitativa del perímetro instrumentado del módulo frente al estado inicial.

Los resultados muestran el cambio estructural más visible de toda la serie. La batería unitaria produjo 6 `passed` y la funcional mínima 6 `passed`. En la comparación acumulada, C2 aumentó de 9 a 26, el recuento homólogo de T1 pasó también de 9 a 26, T2 de 2 a 7 y C4 bajó de 6 a 2. Al mismo tiempo, `RecipeService` pasó de 24 a 18 métodos, de 442 a 307 líneas estructurales y `recipe_service.py` de 598 a 452 líneas de código respecto a la Fase 4.

Este cierre permitió consolidar la validación de la intervención principal sin ocultar sus límites. Pese a la reducción material de invocaciones propias del patrón *service locator*, `RecipeController` conservó parte de su lógica de orquestación. La fase refuerza, por tanto, la lectura acumulativa de la mejora, aunque no equivale a una refundación arquitectónica completa del módulo.

4.2.7. Lectura acumulativa de la validación

Vista en su conjunto, la validación por fases muestra una progresión coherente. La Fase 1 preparó el terreno estructural. La Fase 2 aportó el primer flujo estrictamente aislable. La Fase 3 corrigió una frontera de capa específica. La Fase 4 convirtió la autorización en una política explícita y testeable. La Fase 5 tradujo esa base previa en una reducción material del punto crítico principal y en un aumento significativo de la testabilidad aislada.

Esta progresión resulta importante porque evita dos errores de interpretación. El primero sería exigir a las fases tempranas resultados de simplificación que aún no podían producir. El segundo sería leer la Fase 5 como una intervención aislada, cuando en realidad su capacidad de mejora dependía de las fronteras y unidades explícitas construidas en fases previas.

4.3. Evaluación final consolidada

4.3.1. Evolución global de las métricas

El balance global de la intervención se obtiene comparando el estado del módulo antes de la intervención con el alcanzado al cierre de la Fase 5, sobre un perímetro instrumentado común del módulo, ampliado de forma declarada para incorporar los componentes extraídos durante la intervención. Aunque la validación fase a fase permite interpretar el efecto específico de cada cambio, la comparación acumulada muestra la dirección global de la mejora.

Tabla 6: Evolución acumulada de las métricas estructurales del módulo intervenido.

Métrica	Estado inicial	Fase 1	Fase 2	Fase 3	Fase 4	Fase 5
C1	9	1	1	1	1	1
C2	0	3	7	7	9	26
C3	2	2	2	0	0	0
C4	6	6	6	6	6	2
S2	1418	1558	1684	1699	1819	2163
T1	0	3	7	7	9	26
T2	0	0	1	1	2	7
T3	2	0	0	0	0	0

La Tabla 6 sintetiza el efecto acumulado de la intervención sobre el perímetro observado, pero por sí sola no basta para afirmar una mejora arquitectónica. Su interpretación requiere relacionar cada métrica con los problemas del estado inicial, con las fases ejecutadas y con las evidencias funcionales y unitarias obtenidas durante la validación.

4.3.2. Interpretación de las métricas de acoplamiento: C1–C4

C1 pasa de 9 a 1. Este resultado refleja una reducción fuerte y temprana de referencias concretas de infraestructura en la orquestación del módulo. La mejora se produce esencialmente en la Fase 1 y se conserva sin regresión hasta el final. Esto respalda la afirmación de que el módulo deja de depender, en varios puntos clave, de clases o funciones concretas embebidas directamente en servicios y controladores.

C2 aumenta progresivamente desde 0 hasta 26. Este incremento no representa un deterioro, sino la materialización explícita de relaciones que antes permanecían implícitas dentro del servicio principal o de la lógica de entrada. El aumento más visible se produce en la Fase 5, cuando los nuevos casos de uso incorporan dependencias salientes declaradas de forma explícita mediante puertos.

C3 pasa de 2 a 0. La mejora corresponde de forma nítida a la Fase 3 y refleja la desaparición de semántica HTTP en la persistencia específica del módulo. Se

trata de una mejora completa dentro del alcance intervenido, sin necesidad de extender la refactorización al repositorio genérico compartido ni a otros módulos del sistema.

C4 pasa de 6 a 2. La reducción llega en la Fase 5 y es parcial, pero precisamente por ello resulta metodológicamente relevante. La intervención no necesitaba demostrar la eliminación total del patrón *service locator* para respaldar su hipótesis principal, pero sí mostrar que una parte material de ese patrón podía reducirse sin abrir una reescritura transversal de Mealie.

En conjunto, estas métricas respaldan una mejora verificable en el acoplamiento del módulo: se reducen dependencias directas con infraestructura concreta, aumentan las colaboraciones explícitas, se elimina la contaminación HTTP en persistencia y disminuye parcialmente el uso del *service locator*. La mejora queda respaldada dentro del alcance evaluado, aunque no equivale a una sustitución completa de todos los patrones heredados del estado inicial.

4.3.3. Interpretación de tamaño y testabilidad: S2, T1, T2 y T3

Las métricas de este bloque requieren una lectura especialmente cauta, dado que no todas operan sobre el mismo perímetro ni aportan evidencia independiente.

S2 pasa de 1418 a 2163 en el perímetro instrumentado seguido por el script. A primera vista, este crecimiento podría parecer contradictorio con la mejora buscada. Sin embargo, en el contexto de la intervención su lectura correcta es distinta. El aumento responde a la aparición de puertos, adaptadores, políticas y casos de uso explícitos que antes no existían como unidades separadas dentro del perímetro observado. Por tanto, S2 no puede interpretarse como deterioro automático ni como una medida del tamaño completo de todo el backend; debe leerse junto con la reducción material del punto crítico principal y con el incremento de testabilidad aislada.

T1 pasa de 0 a 26, pero no aporta evidencia independiente respecto a C2. Ambas métricas comparten la misma estructura de cómputo: C2 la interpreta como explicitud arquitectónica y T1 como potencial de aislamiento. Por ello, la verificación fuerte de testabilidad no descansa en T1, sino en T2 y en las pruebas unitarias efectivamente ejecutadas, que confirman que ese potencial se materializa en flujos realmente aislables.

T2 pasa de 0 a 7. Se trata de una de las evidencias más fuertes del trabajo. Al cierre de la intervención, el módulo dispone de siete flujos estrictamente evaluables sin base de datos ni filesystem real: `CreateRecipeFromWeb`, `CreateRecipeFromZip`, `CreateRecipeFromImages`, `DuplicateRecipe`, `UpdateRecipeImage`, `DeleteRecipeImage` y `RecipePermissionPolicy`. La mejora no es sólo teórica; viene respaldada por pruebas ejecutadas sobre esos componentes.

T3 pasa de 2 a 0. El módulo deja de depender de reemplazos dinámicos de funciones globales para aislar los flujos intervenidos. Este resultado es importante porque señala una mejora no sólo en el número de pruebas posibles, sino también en la calidad arquitectónica del aislamiento alcanzado.

4.3.4. Reducción de `RecipeService` y redistribución de complejidad

La lectura de puntos críticos y complejidad complementa las métricas globales con un análisis focalizado sobre el principal foco problemático del estado inicial: `RecipeService`.

Tabla 7: Reducción estructural de `RecipeService` y redistribución de complejidad.

Elemento	Estado inicial	Fase 4	Fase 5
<code>recipe_service.py</code>	627 LOC	598 LOC	452 LOC
Métodos de <code>RecipeService</code>	27	24	18
Extensión estructural de <code>RecipeService</code>	502	442	307

Estos datos evidencian una reducción material del punto crítico principal. La mejora no es cosmética, porque no se limita a introducir nuevos envoltorios alrededor del mismo servicio: desaparecen métodos concretos de alto peso arquitectónico y aparecen unidades de aplicación más pequeñas que absorben parte de esa responsabilidad.

No sucede lo mismo en la capa de entrada HTTP. `recipe_crud_routes.py` no sigue una trayectoria de reducción comparable y termina incluso por encima del estado inicial en términos de líneas de código. Esta asimetría es coherente con el alcance de la intervención: el trabajo se concentró sobre todo en extraer lógica de aplicación y responsabilidades del servicio principal, no en rediseñar toda la superficie de entrada del módulo.

La evolución muestra una redistribución progresiva de la complejidad fuera del punto crítico principal. Ya desde la Fase 2, métodos como `_create_recipe_from_web` ceden parte de sus responsabilidades a unidades de aplicación más pequeñas, y la Fase 5 consolida ese desplazamiento al extraer varios casos de uso completos. El resultado es un `RecipeService` con menos métodos, menor extensión estructural y un peso cognitivo más manejable, mientras que la testabilidad aislada del módulo crece de forma sostenida.

4.3.5. Trazabilidad entre diagnóstico, intervención y resultados

La relación entre diagnóstico, intervención y resultado puede sintetizarse del siguiente modo.

- El problema de dependencias concretas de infraestructura se abordó con puertos y adaptadores y se refleja en la caída de C1 y en el crecimiento de C2; leído desde testabilidad potencial, ese mismo recuento estructural se refleja también en T1.
- La contaminación HTTP en persistencia se abordó con excepciones de aplicación y mapeo explícito en entrada, y se refleja en la caída completa de C3.

- La concentración funcional en `RecipeService` se abordó mediante extracción progresiva de casos de uso y política de permisos, y se refleja en la reducción de métodos, extensión estructural y líneas del servicio principal.
- La baja testabilidad aislada se abordó haciendo explícitas colaboraciones externas y sacando flujos completos a unidades de aplicación o dominio, y se refleja en el crecimiento de T2 y la desaparición de T3.
- El uso de *service locator* no se eliminó completamente, pero se redujo de forma material en la Fase 5, lo que permite sostener una mejora parcial y honesta sobre ese eje.

Esta trazabilidad entre problema, cambio y evidencia constituye un elemento central de la evaluación. Permite defender que la mejora no se deduce de una impresión general sobre el código, sino de la relación entre defectos del estado inicial, intervenciones concretas y resultados observados sobre el mismo módulo.

4.3.6. Valoración de la hipótesis del trabajo

La hipótesis del trabajo planteaba que una refactorización progresiva, basada en la introducción de puertos, adaptadores, casos de uso y políticas explícitas, podía mejorar la capacidad de evolución del módulo de recetas sin necesidad de una reescritura total. La evidencia acumulada respalda esa hipótesis: las métricas estructurales muestran mejoras consistentes en los ejes evaluados, las pruebas unitarias y funcionales confirman la preservación del comportamiento observable, y el principal punto crítico del módulo reduce su peso de forma medible.

Estos resultados deben interpretarse dentro del alcance definido para el TFM. La intervención no buscaba una transformación completa del módulo hacia una arquitectura hexagonal homogénea, sino una mejora incremental y verificable sobre los problemas estructurales más relevantes del estado inicial. Desde esa delimitación, la reducción parcial del *service locator*, la persistencia del punto crítico HTTP de entrada y la continuidad de algunas dependencias concretas no contradicen la hipótesis, sino que acotan el grado de mejora alcanzado. La contribución principal reside en mostrar que la introducción progresiva de puertos, adaptadores, casos de uso y políticas produce mejoras estructurales verificables dentro de un sistema real de código abierto.

4.4. Amenazas a la validez y límites

4.4.1. Alcance de la intervención

El estudio queda restringido de forma estricta al módulo de gestión de recetas del backend de Mealie. No interviene frontend, autenticación, planificación de comidas ni otros módulos del sistema. Esta delimitación es metodológicamente coherente con el TFM, pero condiciona la generalización de los resultados.

Además, el alcance arquitectónico de la intervención fue deliberadamente parcial. El trabajo se concentró en extraer lógica de aplicación del servicio principal y en hacer explícitas algunas fronteras del módulo, pero no abarcó toda

su superficie. De ahí que sigan presentes focos como la lógica residual en `RecipeController` o las invocaciones supervivientes a `get_repositories()`, cuya transformación habría exigido una intervención de mayor envergadura.

4.4.2. Límites de las métricas

Las métricas empleadas son operacionales y reproducibles, pero no capturan por sí solas toda la mantenibilidad del sistema. Su finalidad consiste en comparar el mismo módulo antes y después de la intervención, no en ofrecer una medida absoluta de calidad del software.

Algunos resultados requieren, por ello, una lectura contextualizada. S2, por ejemplo, aumenta porque se calcula sobre un perímetro instrumentado que incorpora archivos críticos y componentes extraídos durante la intervención; su crecimiento refleja en parte la aparición de puertos, adaptadores, políticas y casos de uso explícitos, no necesariamente un deterioro estructural. De forma similar, T2 utiliza un criterio deliberadamente estricto: sólo contabiliza un flujo cuando puede probarse completamente sin base de datos ni filesystem real. Este criterio evita inflar resultados, aunque deja fuera mejoras parciales que no alcanzan todavía aislamiento completo.

También debe tenerse en cuenta que T1 no constituye una evidencia independiente de C2: ambas métricas comparten estructura de cálculo y deben leerse de forma conjunta. La lectura más directa de testabilidad real descansa en T2 y, sobre todo, en las pruebas unitarias efectivamente ejecutadas en cada fase.

Por último, el perímetro de medición incorporó los componentes nuevos creados por la intervención, como `RecipePermissionPolicy` o los casos de uso de la Fase 5, para poder observar la redistribución de responsabilidades dentro del módulo. Esta decisión fue necesaria para medir la arquitectura resultante, pero se declara explícitamente para mantener la transparencia metodológica de la evaluación.

4.4.3. Alcance de la validación funcional y unitaria

La validación funcional de todas las fases se mantuvo deliberadamente acotada. Se diseñó para comprobar la preservación de comportamiento en los flujos más sensibles de cada intervención, no para ejecutar una regresión exhaustiva de todo el módulo de recetas ni de todo Mealie.

Además, varias pruebas unitarias se apoyaron en dobles deterministas, como *stubs* y *fakes*, en lugar de invocar dependencias reales, en coherencia con el criterio estricto de T2. Tal como se documenta en la Fase 2, el scraper externo, las llamadas HTTP de salida y el acceso a persistencia se sustituyeron por implementaciones controladas durante la prueba. Esta decisión es coherente con el objetivo de aislamiento del módulo, aunque limita la posibilidad de extraer conclusiones sobre la integración real con esos componentes externos.

En la Fase 5 también se priorizó la estabilidad de la evidencia funcional. Para ello, se evitó depender de un test frágil por orden de ejecución y se utilizó un escenario funcional equivalente, autocontenido y más estable. Esta decisión no invalida la fase; al contrario, refuerza que la validez funcional del trabajo descansa sobre una selección razonada de casos representativos, no sobre una campaña de validación exhaustiva.

4.4.4. Generalización de los resultados

Los resultados del estudio son sólidos para el módulo intervenido, pero no admiten generalización automática a todo Mealie ni a cualquier backend. El módulo de recetas combina scraping, filesystem, OpenAI, reglas de permisos y varias formas de orquestación, lo que lo convierte en un caso especialmente interesante, pero también específico.

En suma, la demostración del trabajo es necesariamente local: los resultados se sostienen sobre el módulo intervenido y no permiten derivar automáticamente un método general aplicable a cualquier sistema. Lo que sí queda respaldado es que una intervención progresiva, focalizada y delimitada puede producir mejoras estructurales medibles dentro de su alcance, sin pretender ofrecer una solución arquitectónica general.

4.4.5. Síntesis de validez

Las amenazas y límites anteriores no invalidan la conclusión principal de la evaluación, pero sí delimitan con precisión su alcance. La memoria puede sostener con rigor que el módulo de recetas mejora estructuralmente y gana testabilidad aislada dentro del alcance intervenido. No puede sostener, en cambio, que el sistema haya sido completamente migrado a una arquitectura ideal ni que la mejora se extienda automáticamente a otros módulos o a propiedades operativas no medidas.

Esta delimitación permite que el Capítulo 5 formule conclusiones acotadas y líneas futuras coherentes con la evidencia obtenida.

5. Conclusiones y trabajos futuros

5.1. Conclusiones del trabajo

Las conclusiones del trabajo se apoyan en la evidencia consolidada en el Capítulo 4 y permiten responder a la pregunta de investigación y a la hipótesis formuladas en el Capítulo 1 dentro del alcance definido para la intervención.

El problema abordado en este TFM no consistía en una carencia de funcionalidad visible del producto ni en un problema de rendimiento operativo de Mealie. El problema era de desarrollo del software. El módulo de gestión de recetas funcionaba desde el punto de vista funcional, pero su estructura interna dificultaba

la evolución controlada del código por una combinación de acoplamiento con infraestructura, mezcla de responsabilidades entre capas, concentración funcional en puntos críticos, contaminación HTTP en persistencia y baja capacidad de prueba aislada. En sistemas reales y en evolución, este tipo de problema resulta relevante porque la sostenibilidad del desarrollo depende en gran medida de la calidad estructural con la que pueden introducirse cambios sin incrementar de forma desproporcionada el riesgo de regresión y el coste de validación, en línea con la preocupación por sostenibilidad arquitectónica discutida en el estado del arte [23].

Frente a esos problemas, el trabajo planteó una intervención acotada al backend del módulo de recetas, sin asumir una reescritura completa del sistema. La estrategia se apoyó en principios de separación de responsabilidades, en el uso pragmático de puertos y adaptadores y en una secuencia de fases incrementales orientadas a mejorar mantenibilidad, testabilidad, desacoplamiento y capacidad de evolución estructural.

La aportación principal del TFM consiste en haber documentado y evaluado un proceso trazable de diagnóstico, diseño, refactorización incremental y validación sobre un sistema real de código abierto. No se ha propuesto únicamente una arquitectura deseable en términos teóricos, sino que se ha aplicado una intervención concreta sobre código existente y se han contrastado sus efectos mediante métricas estructurales, pruebas unitarias focalizadas y validación funcional mínima.

La evidencia consolidada en el Capítulo 4 permite sintetizar los principales resultados de la intervención y vincularlos con los objetivos del trabajo:

1. **Reducción del acoplamiento concreto con infraestructura.** El módulo reduce de forma significativa su acoplamiento concreto con infraestructura: C1 desciende de 9 a 1, mientras que C2 aumenta de 0 a 26, reflejando que muchas dependencias antes implícitas pasan a expresarse mediante contratos, puertos y colaboradores identificables.
2. **Mejora de la testabilidad aislada.** La testabilidad aislada mejora de forma clara. T2 pasa de 0 a 7, lo que indica que al cierre de la intervención existen siete flujos completos evaluables con criterio estricto sin base de datos ni sistema de archivos real. Además, T3 desciende hasta 0, mostrando que los flujos intervenidos dejan de depender de reemplazos dinámicos de funciones globales para poder aislarse durante la prueba.
3. **Reducción del punto crítico principal.** `RecipeService` experimenta una reducción observable en tamaño, número de métodos y extensión estructural, y deja de concentrar varios flujos de importación, duplicación y gestión de imagen. Este resultado aborda uno de los problemas más claros del estado inicial: la acumulación de demasiadas razones de cambio en un mismo componente.
4. **Mejor definición de la frontera entre persistencia y entrada.** La intervención corrige una contaminación entre capas arquitectónicamente inadecuada. La persistencia específica del módulo deja de depender de semántica

HTTP, lo que permite sostener que la frontera entre persistencia y entrada queda mejor definida dentro del ámbito realmente intervenido.

5. **Reducción parcial del uso de *service locator*.** C4 desciende de 6 a 2. Este resultado no implica la desaparición completa del patrón, pero sí muestra una reducción efectiva dentro del alcance evaluado, sin abrir una reescritura transversal del backend.

Estos resultados permiten responder de forma afirmativa, dentro del alcance evaluado, a la pregunta de investigación formulada en el Capítulo 1. En el caso estudiado, una estrategia incremental de refactorización arquitectónica puede mejorar de forma defendible indicadores relevantes de mantenibilidad, testabilidad, desacoplamiento y capacidad de evolución estructural del desarrollo del software, sin evidencia de deterioro del comportamiento observable en los flujos representativos efectivamente validados.

En ese mismo sentido, la hipótesis de trabajo queda respaldada como hipótesis empírica local, no como validación universal de un patrón arquitectónico. La evidencia obtenida muestra que la explicitación selectiva de dependencias, la extracción de casos de uso, la separación de una política de negocio relevante y la reducción parcial de puntos críticos pueden mejorar el módulo en los indicadores realmente medidos y en la evidencia ejecutable obtenida, sin necesidad de un rediseño total del sistema.

El valor del trabajo reside, por tanto, en complementar la discusión teórica sobre arquitectura con la evidencia obtenida sobre código real. La intervención muestra que una refactorización incremental ejecutada en fases acotadas y verificadas puede mejorar de manera medible la estructura de un módulo perteneciente a un sistema real de código abierto, reforzando la viabilidad práctica de un enfoque iterativo, focalizado y reproducible.

Estos resultados deben leerse dentro del alcance definido: la intervención no elimina todos los puntos críticos, no equivale a una regresión funcional exhaustiva de todo Mealie ni alcanza por completo la arquitectura objetivo definida como referencia. Estos límites no contradicen la conclusión principal, sino que delimitan una mejora parcial, medible y defendible.

Al mismo tiempo, esta delimitación no reduce la relevancia del enfoque al caso concreto de Mealie. Aunque el trabajo no demuestra por sí solo una generalización directa, sí muestra que un rediseño arquitectónico incremental, focalizado y apoyado en evidencia puede servir como referencia para mejorar la infraestructura de desarrollo de productos software similares: sistemas abiertos, contributivos y en evolución, en los que la acumulación de cambios puede generar problemas estructurales comparables a los analizados en el Capítulo 2.

La conclusión más importante del trabajo puede expresarse, por tanto, del siguiente modo: en el módulo de recetas de Mealie, el diseño arquitectónico incremental no ha producido una transformación completa del módulo, pero sí una mejora estructural observable y respaldada por evidencia estructural y ejecutable. El módulo queda en mejor disposición para evolucionar como base de desarrollo: con dependencias más explícitas, menor concentración funcional en

su punto crítico principal, mejor separación entre capas y más comportamiento validable de forma aislada.

5.2. Líneas de trabajo futuro

Cualquier línea de continuación razonable debe partir de los límites detectados durante la evaluación del trabajo y no de una ampliación especulativa del alcance.

1. La línea futura prioritaria es la reducción de los usos residuales de `get_repositories()` dentro del módulo de recetas. Esta continuación es razonable porque la intervención ya ha mostrado que el *service locator* puede reducirse parcialmente sin reescribir el módulo completo. Llevar esa reducción a un cierre más completo permitiría consolidar la explicitud del ensamblaje y reforzar la coherencia arquitectónica alcanzada en la Fase 5.
2. La segunda línea prioritaria es intervenir arquitectónicamente sobre la capa de entrada HTTP, especialmente sobre `RecipeController` y `recipe_crud_routes.py`, que siguen actuando como puntos críticos relevantes. Esta continuación nace de un límite claramente identificado en la evaluación final: mientras `RecipeService` reduce de forma material su peso estructural, la superficie de entrada del módulo no experimenta una simplificación equivalente. Aplicar el mismo criterio incremental a esa capa permitiría completar el desplazamiento de orquestación iniciado en la Fase 2.
3. La tercera línea razonable es ampliar la validación funcional del módulo sobre los flujos ya intervenidos. El trabajo actual dispone de evidencia suficiente para sostener la hipótesis principal, pero esa validación es deliberadamente acotada y representativa. Una campaña más amplia de pruebas de caracterización y regresión sobre esos mismos flujos permitiría reforzar la confianza práctica en la línea de refactorización sin cambiar la tesis central ya demostrada.
4. La cuarta línea es extender el mismo protocolo de diagnóstico, intervención y evaluación a otro módulo del sistema. Esta continuación no se justifica por afán de ampliar artificialmente el trabajo, sino por una necesidad metodológica clara: reforzar la validez externa del enfoque. Si otro módulo de Mealie mostrara mejoras comparables bajo una estrategia incremental similar, la aportación del TFM dejaría de apoyarse sólo en el caso de recetas y ganaría mayor fuerza como propuesta de intervención arquitectónica sobre software de código abierto existente.
5. Como línea secundaria, pero razonable, puede considerarse la extensión controlada a otros puntos del backend de dos mejoras ya validadas en el módulo de recetas: separar la gestión de errores de persistencia de la semántica HTTP y hacer más explícito el ensamblaje de dependencias compartidas. Esta línea debe abordarse con prudencia, porque excede el alcance del presente TFM, pero resulta coherente con mejoras que ya han mostrado valor dentro del módulo de recetas.

6. Finalmente, una línea complementaria sería integrar parte del protocolo de medición y validación en un flujo de control continuo del proyecto. Incorporar métricas estructurales, pruebas focalizadas o comprobaciones de dependencias en un proceso recurrente permitiría detectar regresiones arquitectónicas futuras y convertir parte de la evaluación realizada en este TFM en un mecanismo preventivo de evolución controlada.

En su conjunto, estas líneas de trabajo futuro no plantean una expansión indiscriminada del estudio, sino una continuación directa de lo ya alcanzado. Primero conviene cerrar los límites residuales del propio módulo; después, si se desea ampliar la aportación del trabajo, resulta metodológicamente más sólido replicar el protocolo en otro contexto del sistema antes que abrir una reescritura transversal del backend.

Apéndices

A. Arquitectura objetivo del módulo de recetas

A.1. Propósito, alcance y relación con la memoria

Este anexo documenta la arquitectura objetivo utilizada como referencia durante el diseño de la intervención descrita en el Capítulo 3. No representa el estado final completo del módulo de recetas ni el conjunto exacto de artefactos implementados, sino el marco arquitectónico que permitió interpretar y priorizar las fases de refactorización incremental.

Su función es complementar la memoria principal con una visión estructurada de la arquitectura de referencia: límites entre capas, casos de uso candidatos, conceptos de dominio, puertos de salida y adaptadores de infraestructura. Esta arquitectura objetivo sirvió como mapa de decisión para determinar qué dependencias convenía hacer explícitas, qué flujos podían separarse de controladores o servicios demasiado amplios y qué responsabilidades debían adquirir identidad propia dentro del módulo.

El alcance de este diseño se limita al módulo de gestión de recetas del backend de Mealie. Su finalidad es orientar una mejora estructural centrada en reducir acoplamiento, aumentar testabilidad y separar progresivamente responsabilidades de entrada, aplicación, dominio e infraestructura, sin alterar el comportamiento funcional observable del sistema.

A.2. Condicionantes de diseño

La arquitectura objetivo se definió a partir de los problemas estructurales identificados en el estado inicial del módulo. De forma sintética, los principales condicionantes fueron los siguientes:

- Los controladores HTTP contenían lógica de orquestación de flujos e integraciones externas.
- `RecipeService` concentraba lógica de aplicación, permisos, acceso a datos, filesystem, importación ZIP y construcción de recetas desde imágenes.
- Existían dependencias directas desde la lógica de orquestación hacia FastAPI, SQLAlchemy, filesystem, scraper y servicios externos.
- Algunos flujos relevantes, como la creación de recetas desde web, dependían de funciones y servicios concretos, lo que dificultaba su aislamiento en pruebas.

Estos condicionantes no sustituyen el diagnóstico desarrollado en el cuerpo principal de la memoria, sino que resumen los factores que guiaron la definición de la arquitectura objetivo.

A.3. Arquitectura objetivo de referencia

La arquitectura objetivo se planteó como una separación progresiva entre dominio, aplicación, puertos de salida y adaptadores de infraestructura. Algunos de los elementos descritos en esta sección se materializaron durante la intervención, mientras que otros permanecieron como referencia de diseño para una evolución posterior del módulo.

A.3.1. Dominio

Los siguientes elementos representan conceptos de dominio relevantes para la arquitectura objetivo, con independencia de que todos ellos se hayan materializado como clases de dominio independientes dentro del alcance del TFM.

- **Recipe**: agregado principal del módulo.
- **RecipeSettings**: configuración asociada a visibilidad, comentarios, bloqueo y renderizado.
- **RecipeIngredient**, **RecipeStep**, **RecipeNote** y **RecipeAsset**: elementos internos asociados a la receta.
- **RecipeImportSource**: valor que representa el origen de una importación, como **manual**, **web**, **zip** o **image**.
- **RecipePermissionPolicy**: política de negocio para edición, borrado y bloqueo.
- **RecipeImage**: representación del estado de imagen y sus variantes asociadas.

En la arquitectura objetivo, el dominio concentra las reglas y conceptos que no deberían depender directamente del transporte HTTP ni de detalles concretos de infraestructura. Sus responsabilidades principales son:

- Garantizar invariantes relevantes del agregado receta.
- Validar relaciones internas, incluyendo referencias recursivas entre recetas cuando proceda.
- Definir políticas de permiso y reglas de mutación con identidad propia.

A.3.2. Aplicación

La capa de aplicación se concibe como el lugar donde se expresan los flujos completos del módulo. Los casos de uso actúan como unidades de orquestación entre la entrada HTTP, el dominio y los puertos de salida. La siguiente lista representa la descomposición deseable de la capa de aplicación, no el conjunto completo de casos de uso implementados durante el TFM.

- **GetRecipe**
- **CreateRecipe**
- **UpdateRecipe**
- **PatchRecipe**
- **DeleteRecipe**
- **DeleteRecipes**

- DuplicateRecipe
- UpdateRecipeImage
- DeleteRecipeImage
- UploadRecipeAsset
- CreateRecipeFromWeb
- CreateRecipeFromZip
- CreateRecipeFromImages
- ExportRecipe
- SuggestRecipes
- ListRecipes
- ListRecipeComments
- ManageRecipeTimelineEvent

Las responsabilidades de la capa de aplicación son:

- Orquestar el flujo entre dominio, políticas y puertos de salida.
- Recibir datos ya adaptados desde la capa de entrada y convertirlos en operaciones del módulo.
- Coordinar persistencia, almacenamiento, scraping, eventos o servicios externos sin depender de implementaciones concretas.
- Mantener los flujos principales en unidades explícitas, testeables y con un propósito reconocible.

A.3.3. Puertos de salida

En esta arquitectura objetivo, los casos de uso constituyen el punto de entrada principal a la capa de aplicación. Por ello, el anexo se centra especialmente en los puertos de salida, que son los más relevantes para desacoplar la lógica de aplicación de la infraestructura concreta.

Los principales puertos de salida previstos son los siguientes:

- RecipeRepositoryPort: persistencia del agregado receta.
- RecipeQueryPort: búsquedas, paginación y sugerencias.
- RecipeDataStoragePort: gestión de imágenes y assets asociados a recetas.
- RecipeDataStorageFactoryPort: creación de servicios de almacenamiento contextualizados cuando el flujo lo requiera.
- RecipeScraperPort: extracción estructurada de recetas desde URL, HTML o JSON.
- RecipeOpenAIBuilderPort: construcción de recetas desde imágenes mediante servicios externos.
- EventPublisherPort: publicación de eventos de dominio o aplicación.
- TemplateRendererPort: renderizado de plantillas y exportaciones.
- RecipePermissionAccessPort: acceso a datos necesario para evaluar permisos.
- RecipePermissionPolicyPort: contrato para consumir una política de permisos cuando sea necesario inyectarla desde la capa de aplicación o servicios existentes.

El objetivo de estos puertos es evitar que los casos de uso y políticas del módulo dependan directamente de SQLAlchemy, filesystem, servicios externos, bus de eventos o funciones globales heredadas. Al hacer explícitas estas colaboraciones, la arquitectura objetivo facilita la sustitución por dobles de prueba y reduce el acoplamiento entre la lógica de aplicación y la infraestructura.

A.3.4. Adaptadores de infraestructura previstos o reutilizados

Los adaptadores permiten conservar la infraestructura real de Mealie, pero integrándola mediante piezas explícitas que implementan los puertos definidos por el módulo. En la arquitectura objetivo se contemplan los siguientes adaptadores o familias de adaptadores:

- DB SQLAlchemy: adaptadores sobre `RepositoryRecipes`, `RepositoryUserRatings`, `RepositoryTimelineEvents` u otros repositorios necesarios para persistencia y consulta.
- Filesystem: adaptadores basados en `RecipeDataService` y posibles especializaciones para almacenamiento de imágenes, assets y transferencias temporales.
- Scraper: `LegacyRecipeScraperAdapter` sobre `create_from_html` y servicios heredados de scraping.
- OpenAI: `OpenAIRecipeService` como adaptador del puerto de construcción de recetas desde imágenes.
- Eventos: adaptadores sobre `EventBusService` o componentes equivalentes de publicación.
- Exportación y renderizado: adaptadores sobre `TemplateService` u otros servicios de plantillas.
- Permisos: adaptadores de acceso a datos, como `SqlAlchemyRecipePermissionAccessAdapter`, para separar la política de permisos de las consultas concretas.

No todos estos adaptadores se materializaron con el mismo grado de profundidad durante el TFM. Algunos formaron parte directa de la intervención, mientras que otros representan la dirección arquitectónica coherente con la arquitectura objetivo.

A.4. Diagrama conceptual de la arquitectura objetivo

El diagrama representa la arquitectura objetivo en términos conceptuales. No debe interpretarse como una descripción exacta de todos los paquetes o clases existentes al cierre de la intervención, sino como una vista de referencia sobre las relaciones deseables entre entrada, aplicación, dominio, puertos e infraestructura. En esta vista, los casos de uso actúan como punto principal de orquestación: consumen reglas de dominio, políticas y puertos de salida. Los puertos no se representan como dependencias genéricas del dominio, sino como contratos utilizados desde la capa de aplicación para acceder a infraestructura.

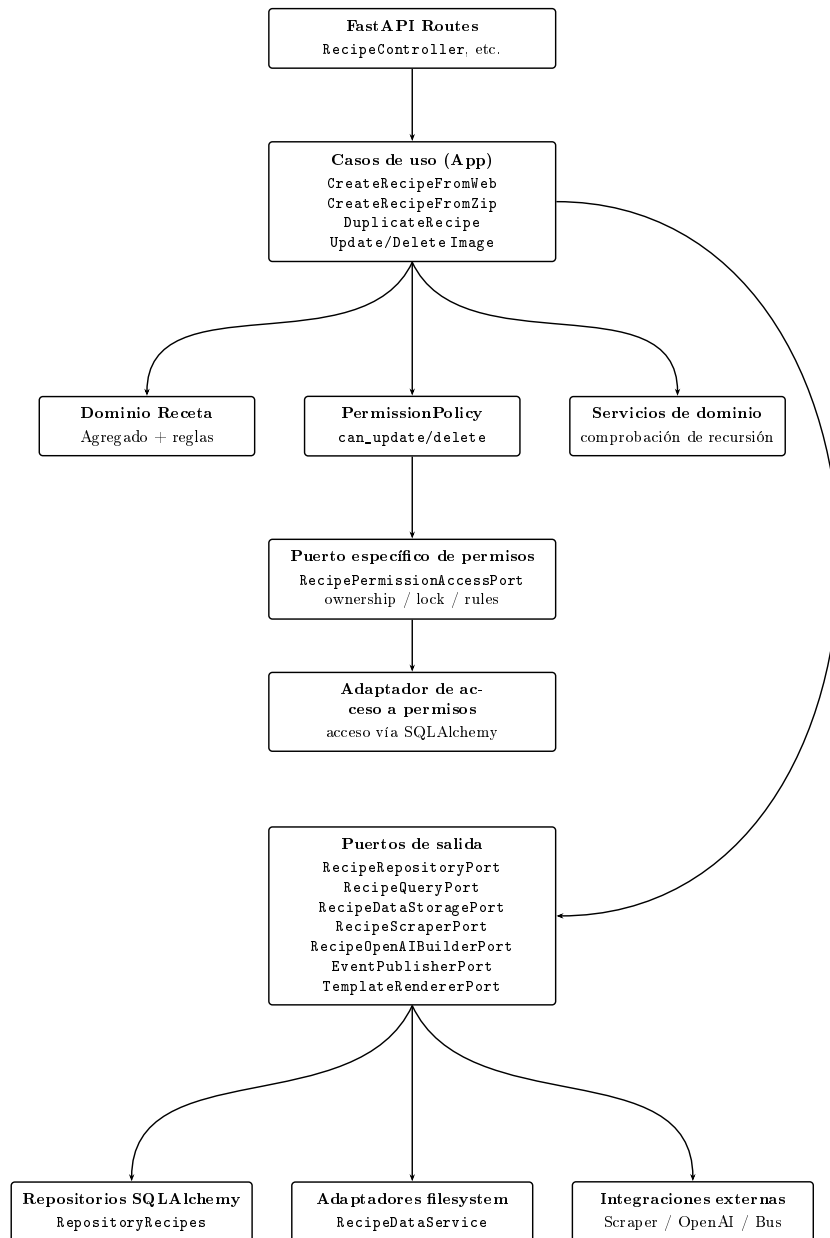


Figura 1. Arquitectura conceptual objetivo del módulo de recetas.

La política de permisos se muestra separada del resto del dominio para destacar que, aunque representa una regla de negocio, requiere información externa obtenida mediante un puerto específico. Por ello, `RecipePermissionAccessPort` aparece visualmente asociado a `RecipePermissionPolicy`, aunque conceptualmente forme parte del conjunto de puertos de salida. En cambio, `RecipePermissionPolicyPort` queda documentado en la lista textual de puertos, pero no se representa en la figura porque el diagrama prioriza las relaciones arquitectónicas principales y no todos los contratos de inyección.

A.5. Materialización parcial durante la intervención

La arquitectura descrita en este anexo no debe leerse como alcance completamente ejecutado. Durante la intervención se materializaron de forma parcial y acumulativa los elementos más relevantes para el problema estudiado:

- introducción de puertos y adaptadores para dependencias externas clave;
- extracción de casos de uso concretos desde la capa HTTP y desde `RecipeService`;
- explicitación de una política de permisos mediante `RecipePermissionPolicy`;
- separación entre persistencia y semántica HTTP en el módulo de recetas;
- reducción parcial del uso de `service locator`;
- mejora progresiva de la testabilidad aislada mediante unidades de aplicación y dominio más explícitas.

En cambio, algunos casos de uso, puertos y adaptadores listados en la arquitectura objetivo permanecen como referencia de diseño y no como implementación completa dentro del alcance del TFM. Esta distinción permite interpretar las fases del Capítulo 3 como aproximaciones sucesivas a una arquitectura objetivo coherente, sin confundir dicha referencia con el estado final completo del módulo.

B. Marco de medición y métricas reproducibles

B.1. Propósito y relación con la memoria

Este anexo documenta el marco operacional de métricas utilizado para respaldar la metodología del Capítulo 3 y la evaluación consolidada del Capítulo 4. Su función es hacer explícita la definición de cada métrica, el criterio de medición aplicado y el instrumento empleado para calcular los valores comparados a lo largo de la intervención.

De este modo, el Capítulo 4 puede centrarse en interpretar los resultados obtenidos, mientras que este anexo recoge el detalle metodológico necesario para comprender cómo se construyeron las mediciones. El anexo no pretende ofrecer una auditoría completa de la calidad de Mealie, sino documentar un marco reproducible de comparación aplicado al mismo módulo antes y después de las fases de refactorización.

B.2. Criterio general de medición

Las métricas se definieron para poder aplicarse al estado inicial del módulo de recetas y repetirse después al cierre de cada fase relevante. El objetivo no era capturar de forma exhaustiva toda la calidad del sistema, sino construir un conjunto estable de indicadores que permitiera comparar la evolución estructural del mismo módulo bajo un criterio homogéneo.

La comparación se apoyó en tres principios:

- mantener un perímetro de análisis estable sobre los archivos críticos y los componentes extraídos durante la intervención;
- aplicar los mismos criterios de conteo en el estado inicial y en los estados posteriores;
- interpretar cada métrica en relación con el objetivo concreto de la fase evaluada y no como indicador absoluto de calidad.

B.3. Perímetro instrumentado

El perímetro instrumentado se centró en el módulo de recetas del backend y en los componentes creados o modificados durante la intervención. Este perímetro permitió seguir la evolución de los principales puntos críticos y de las nuevas unidades introducidas por la refactorización.

De forma general, el seguimiento incluyó:

- archivos críticos del estado inicial, como `recipe_service.py`, `recipe_crud_routes.py`, `recipe_bulk_scraper.py` y componentes base del módulo de recetas;
- clases principales del estado inicial, especialmente `RecipeService`, `RecipeController` y `RecipeBulkScraperService`;
- nuevos componentes introducidos durante la intervención, como puertos, adaptadores, casos de uso y políticas;
- patrones arquitectónicos concretos observados en el diagnóstico, como referencias directas a infraestructura, dependencias HTTP fuera de la capa de entrada e invocaciones directas a `get_repositories()`.

Este perímetro no equivale al tamaño completo de Mealie ni al tamaño total de su backend. Su finalidad es permitir una comparación coherente del módulo intervenido y de los componentes directamente relacionados con la refactorización.

B.4. Métricas de acoplamiento

ID	Métrica	Definición operacional	Medición en el estado inicial	Medición tras cada fase
C1	Referencias concretas de infraestructura	Número de referencias directas a clases o funciones concretas de infraestructura dentro de servicios y controladores del módulo	Contar apariciones de <code>RecipeDataService()</code> , <code>create_from_html()</code> , <code>OpenAIRecipeService()</code> y otras dependencias concretas en archivos objetivo	Repetir el conteo y comparar la reducción o el traslado de dichas dependencias a adaptadores
C2	Dependencias inyectables explícitas	Número de dependencias externas expresadas como puertos en constructores, factorías o puntos explícitos de composición del módulo	Inspeccionar firmas de constructores y puntos de ensamblaje de servicios principales para contar puertos inyectables	Repetir el conteo por fase y analizar el crecimiento del desacoplamiento explícito
C3	Dependencias HTTP fuera de entrada	Número de usos de <code>HTTPException</code> o imports de <code>FastAPI</code> en repositorios o servicios del módulo	Buscar <code>HTTPException</code> y <code>from fastapi</code> en persistencia y servicios	Repetir tras cada fase para verificar la eliminación de semántica HTTP fuera de la capa de entrada
C4	Uso de <i>service locator</i>	Número de invocaciones directas a <code>get_repositories()</code> dentro del módulo de recetas	Buscar la expresión en servicios y controladores del módulo	Repetir el conteo para verificar su encapsulación o reducción progresiva

Estas métricas permiten evaluar si el módulo gana explicitud de dependencias y mejora la separación entre aplicación e infraestructura, dos condiciones relevantes para su evolución controlada.

B.5. Métricas de complejidad

ID	Métrica	Definición operacional	Medición en el estado inicial	Medición tras cada fase
X1	Complejidad por función	Aproximación estructural a la complejidad de función basada en nodos de control, como <code>if</code> , <code>for</code> , <code>try</code> , operadores booleanos y comprensiones	Ejecutar análisis AST sobre los archivos críticos y registrar las funciones con mayor puntuación	Repetir el análisis y comparar si la complejidad se desplaza hacia unidades más pequeñas y focalizadas
X2	Número de métodos por clase	Cantidad de métodos declarados en cada clase crítica	Analizar mediante AST clases como <code>RecipeService</code> , <code>RecipeController</code> y <code>RecipeBulkScraperService</code>	Repetir tras cada fase para observar redistribución o reducción de responsabilidades
X3	Extensión estructural de clase	Rango de líneas ocupado por cada clase crítica	Calcular <code>end_lineno - lineno + 1</code> mediante AST o inspección equivalente	Repetir tras cada fase para verificar la reducción de tamaño estructural en clases críticas

Estas métricas permiten observar si la intervención reduce la concentración funcional y redistribuye responsabilidades hacia unidades más pequeñas y cohesionadas. En particular, X2 y X3 resultan relevantes para interpretar la evolución de `RecipeService` como punto crítico principal del módulo.

B.6. Métricas de tamaño

ID	Métrica	Definición operacional	Medición en el estado inicial	Medición tras cada fase
S1	LOC por archivo crítico	Número de líneas por archivo relevante del módulo	Medir líneas de código en los archivos críticos del perímetro observado	Repetir tras cada fase para seguir la evolución de archivos concretos

ID	Métrica	Definición operacional	Medición en el estado inicial	Medición tras cada fase
S2	LOC total del perímetro instrumentado	Suma de líneas del conjunto de archivos críticos y componentes extraídos incluidos en el seguimiento del script	Calcular el total de líneas del perímetro instrumentado en el estado inicial	Repetir tras cada fase para observar si la complejidad se concentra o se redistribuye dentro del conjunto observado

El tamaño no se interpreta aquí de forma aislada. Un aumento de LOC puede ser compatible con una mejora arquitectónica si el crecimiento se produce por la aparición de componentes explícitos, pequeños y mejor aislados, como puertos, adaptadores, políticas o casos de uso.

B.7. Métricas de testabilidad

ID	Métrica	Definición operacional	Medición en el estado inicial	Medición tras cada fase
T1	Dependencias sustituibles mediante contratos explícitos	Número de colaboradores externos accesibles mediante puertos o inyección explícita; en el instrumento comparte el mismo recuento base que C2 y se interpreta desde el potencial de aislamiento	Inspeccionar constructores, factorías y puntos de composición del módulo para identificar dependencias inyectables	Repetir tras cada fase y cuantificar el crecimiento del potencial de aislamiento
T2	Flujos unitariamente evaluables sin base de datos ni filesystem real	Número de flujos relevantes del módulo que pueden probarse con dobles de prueba sin acceder a base de datos ni filesystem real	Aplicar un criterio estricto: el flujo sólo cuenta si sus dependencias externas son inyectables o están encapsuladas tras puertos	Repetir tras cada fase y documentar el número de flujos habilitados para prueba aislada

ID	Métrica	Definición operacional	Medición en el estado inicial	Medición tras cada fase
T3	Necesidad de reemplazos dinámicos de funciones globales	Número de situaciones en las que el aislamiento de un flujo exigiría reemplazar dinámicamente funciones globales o clases concretas	Identificar usos directos de funciones globales como <code>create</code> <code>_from_html</code> o dependencias equivalentes	Repetir tras cada fase y registrar su eliminación o reducción

La testabilidad constituye una dimensión central de la evaluación porque conecta directamente con el problema de desarrollo del software. La mejora no se interpreta sólo como posibilidad teórica de desacoplamiento, sino como capacidad efectiva de ejecutar pruebas unitarias sobre flujos relevantes con dobles de prueba y sin depender de infraestructura real.

B.8. Instrumentación y reproducibilidad

Para garantizar la repetibilidad del estudio, las métricas se recogieron con tres mecanismos complementarios:

- conteos automáticos de líneas y referencias concretas mediante utilidades de análisis sobre el código fuente;
- análisis estructural mediante scripts basados en AST para medir complejidad aproximada, número de métodos y tamaño de clases;
- inspección arquitectónica guiada para clasificar dependencias, flujos aislables y puntos de uso de *service locator*.

El artefacto principal de apoyo a esta medición es el script `support-files/receipe_module_metrics.py`. Este script permite comparar el estado inicial y los estados posteriores del módulo sobre una base reproducible. El instrumento admite distintas fuentes de análisis, como referencias Git, *working tree* y snapshots de directorio, de modo que el mismo criterio de cálculo puede reutilizarse frente al estado inicial y frente a los estados de cierre de las fases relevantes.

En el repositorio de trabajo, la reproducción histórica se apoya además en las etiquetas `tfm-baseline` y `tfm-phase-1` a `tfm-phase-5`. Esta organización evita depender del *working tree* actual para recomponer fases ya cerradas y permite aplicar de nuevo el mismo conjunto de métricas sobre estados concretos del código.

El proceso de reproducción puede resumirse en tres pasos:

1. seleccionar la referencia de código que se desea medir, por ejemplo el estado inicial o el cierre de una fase;
2. ejecutar el script de métricas sobre esa referencia o snapshot;

3. comparar los valores obtenidos con la serie histórica utilizada en el Capítulo 4.

De este modo, el marco de medición no depende únicamente de una observación manual del código, sino de un procedimiento repetible que permite reconstruir los valores empleados en la evaluación.

B.9. Alcance y límites del marco de métricas

Este marco no pretende capturar toda la mantenibilidad del sistema en sentido absoluto. Su finalidad es proporcionar un conjunto estable de indicadores para el mismo módulo y la misma intervención. Por ello, las métricas deben interpretarse junto con la evidencia unitaria, la validación funcional mínima y el análisis cualitativo de las decisiones de diseño.

Conviene explicitar dos límites metodológicos principales. El primero es que S2 no mide el tamaño completo de todo Mealie ni de todo el backend, sino el LOC total del perímetro instrumentado que el script sigue de forma estable. El segundo es que T1 no constituye un recuento independiente de C2: ambos reutilizan la misma base estructural de dependencias inyectables, aunque se interpreten desde perspectivas distintas.

En consecuencia, la evidencia más fuerte de testabilidad real debe apoyarse sobre todo en T2 y en las pruebas unitarias ejecutadas. Del mismo modo, la mejora arquitectónica no debe deducirse de una métrica aislada, sino de la convergencia entre indicadores estructurales, pruebas focalizadas y preservación funcional de los flujos intervenidos.

C. Intervención de Fase 1: puertos y adaptadores

C.1. Propósito y relación con la memoria

Este anexo amplía la descripción de la Fase 1 presentada en el Capítulo 3 y complementa la validación estructural recogida en el Capítulo 4. Su función es documentar con mayor detalle la primera intervención incremental sobre el módulo de recetas: la introducción de puertos y adaptadores como mecanismo inicial de desacoplamiento frente a infraestructura concreta.

A diferencia de las fases posteriores, esta intervención no buscaba todavía reducir de forma visible los puntos críticos principales ni extraer casos de uso completos. Su objetivo era habilitador: crear las primeras fronteras explícitas entre la lógica de orquestación y colaboradores como persistencia, almacenamiento, scraping o servicios externos. Por ello, el valor de esta fase debe interpretarse sobre todo en términos de preparación arquitectónica, trazabilidad de dependencias y mejora inicial del potencial de aislamiento.

El anexo no sustituye el diagnóstico general del estado inicial desarrollado en la memoria principal. Se centra únicamente en los elementos del estado de

partida que justificaban la Fase 1 y en los cambios concretos introducidos para abrir la trayectoria incremental seguida después.

C.2. Problema específico abordado por la Fase 1

La Fase 1 respondió a un problema concreto del estado inicial: varias partes del módulo dependían directamente de implementaciones de infraestructura o de funciones globales, lo que dificultaba sustituir colaboradores durante las pruebas y extraer posteriormente lógica hacia unidades de aplicación más pequeñas.

Los síntomas más relevantes para esta fase eran los siguientes:

- `RecipeService` instanciaba directamente `RecipeDataService` en varios puntos, lo que acoplaba la lógica de orquestación con detalles de almacenamiento.
- El controlador de recetas y `RecipeBulkScraperService` utilizaban llamadas directas a `create_from_html`, dificultando el aislamiento del flujo de creación desde web.
- `RecipeServiceBase` no exponía dependencias salientes mediante puertos o contratos explícitos.
- Varias operaciones seguían dependiendo de colaboradores concretos, como SQLAlchemy, filesystem, scraping u OpenAI, sin una frontera arquitectónica clara.

El propósito de la fase no era resolver todos los problemas estructurales del módulo. Otros aspectos, como la contaminación HTTP en persistencia, la lógica de permisos dentro de `RecipeService`, el uso de `get_repositories()` o la descomposición del servicio principal, quedaron deliberadamente para fases posteriores. La Fase 1 se limitó a crear una base mínima de desacoplamiento sobre la que dichas intervenciones pudieran apoyarse.

C.3. Intervención arquitectónica aplicada

La intervención consistió en introducir una capa mínima de puertos y adaptadores dentro del módulo de recetas, sin modificar los contratos públicos ni alterar el comportamiento observable del sistema. Esta capa permitió expresar dependencias que antes aparecían como instanciaciones o llamadas directas dentro de servicios y controladores.

C.3.1. Artefactos introducidos

La Fase 1 incorporó dos nuevos archivos principales en el módulo de recetas:

- `mealie/services/recipe/ports.py`: definición de contratos de salida del módulo.
- `mealie/services/recipe/adapters.py`: adaptadores compatibles con la infraestructura existente.

Los puertos añadidos fueron los siguientes:

- `RecipeRepositoryPort`
- `RecipeDataStoragePort`
- `RecipeDataStorageFactoryPort`
- `RecipeScraperPort`
- `RecipeOpenAIBuilderPort`

Los adaptadores añadidos fueron:

- `SqlAlchemyRecipeRepositoryAdapter`
- `FileSystemRecipeDataStorageFactory`
- `LegacyRecipeScraperAdapter`

La correspondencia entre los problemas detectados y los artefactos introducidos puede resumirse del siguiente modo:

- Las instanciaciones directas de `RecipeDataService` se abordaron mediante `RecipeDataStoragePort` y `RecipeDataStorageFactoryPort`.
- Las llamadas directas a `create_from_html` se sustituyeron por `RecipeScraperPort` y `LegacyRecipeScraperAdapter`.
- El acceso a persistencia se empezó a expresar mediante `RecipeRepositoryPort` y `SqlAlchemyRecipeRepositoryAdapter`.
- La construcción de recetas desde imágenes quedó preparada para depender de `RecipeOpenAIBuilderPort`, reduciendo el acoplamiento directo con el servicio concreto de OpenAI.

C.3.2. Archivos modificados y tipo de cambio

Además de los nuevos archivos de puertos y adaptadores, la Fase 1 modificó varios componentes existentes del módulo para consumir las nuevas abstracciones.

- `mealie/services/recipe/recipe_service.py`: se ajustó la base del servicio para aceptar dependencias abstractas relacionadas con repositorio, almacenamiento y construcción desde imágenes, manteniendo valores por defecto compatibles con la infraestructura existente.
- `mealie/routes/recipe/recipe_crud_routes.py`: se adaptó el uso de dependencias de scraping para que el controlador pudiera apoyarse en el puerto correspondiente en lugar de depender directamente de la función global heredada.
- `mealie/services/scrapper/recipe_bulk_scraper.py`: se sustituyó la dependencia directa con la función global de scraping por una colaboración expresada mediante el puerto correspondiente.
- `mealie/routes/recipe/_base.py`: se incorporaron ajustes de ensamblaje para permitir la creación y provisión de los adaptadores necesarios dentro del flujo existente del módulo.

Estos cambios se introdujeron de forma conservadora. La intención no era rediseñar por completo el módulo, sino abrir puntos explícitos de sustitución sin alterar el comportamiento observable de los flujos existentes.

C.4. Resultados específicos de la Fase 1

C.4.1. Tamaño

La Fase 1 incrementó el tamaño total del perímetro intervenido, principalmente por la incorporación de nuevos archivos de abstracción. Este crecimiento era esperable, ya que la fase añadía puertos y adaptadores sin extraer todavía casos de uso ni eliminar lógica heredada.

Elemento	Antes	Después	Variación
<code>recipe_service.py</code>	627 LOC	646 LOC	+19
<code>recipe_crud_routes.py</code>	608 LOC	607 LOC	-1
<code>recipe_bulk_scraper.py</code>	127 LOC	134 LOC	+7
<code>_base.py</code>	56 LOC	62 LOC	+6
Nuevos archivos de abstracción	0 LOC	109 LOC	+109

El aumento de LOC no debe interpretarse como deterioro arquitectónico automático. En esta fase, el crecimiento corresponde a la aparición de estructuras explícitas que antes no existían: contratos y adaptadores destinados a separar la lógica de aplicación de la infraestructura.

C.4.2. Acoplamiento

La mejora más clara de la Fase 1 se observa en el acoplamiento con infraestructura concreta. Varias dependencias que antes aparecían como instanciaciones o llamadas directas pasaron a expresarse mediante puertos o adaptadores.

Métrica	Antes	Después	Interpretación
Instanciaciones directas de <code>RecipeDataService</code> en <code>RecipeService</code>	6	0	Mejora clara: la dependencia queda encapsulada tras factoría y puerto
Llamadas directas a <code>create_from_html</code> en la orquestación de recetas	2	0	Mejora clara: el scraping pasa a consumirse mediante un puerto
Dependencias salientes inyectables en <code>RecipeServiceBase</code>	0	3	Mejora inicial: se habilita sustitución por dobles de prueba
Puertos explícitos del módulo de recetas	0	5	Mejora estructural: aparecen contratos de salida identificables

Métrica	Antes	Después	Interpretación
Adaptadores explícitos del módulo de recetas	0	3	Mejora estructural: la infraestructura se encapsula en piezas concretas

Estos resultados explican por qué la Fase 1 se considera una fase habilitadora. No reduce todavía el punto crítico principal, pero crea las primeras condiciones para extraer flujos y políticas en fases posteriores.

C.4.3. Complejidad

La complejidad estructural de las funciones críticas no experimentó una reducción significativa en esta fase. Las funciones más complejas siguieron situadas en los mismos puntos principales: `RecipeService`, `RecipeController` y `RecipeBulkScraperService`.

Este resultado es coherente con el objetivo de la Fase 1. La fase no estaba orientada a redistribuir todavía la lógica de aplicación ni a descomponer `RecipeService`, sino a introducir fronteras de desacoplamiento. Por ello, no cabía esperar una reducción inmediata en las métricas de complejidad o en el número de métodos de las clases críticas.

C.4.4. Testabilidad

La mejora en testabilidad durante la Fase 1 fue principalmente preparatoria. La introducción de puertos aumentó el potencial de aislamiento, pero todavía no generó flujos completos unitariamente evaluables con el criterio estricto utilizado en la evaluación.

Métrica	Antes	Después	Interpretación
Dependencias externas sustituibles por puertos en el servicio principal	0	3	Mejora inicial del potencial de aislamiento
Necesidad de llamadas directas a funciones globales de scraping para aislar flujos principales	2	0	Mejora clara: el scraping deja de depender directamente de funciones globales

Métrica	Antes	Después	Interpretación
Flujos relevantes unitariamente evaluables sin base de datos o filesystem, con criterio estricto	0	0	La mejora fuerte de T2 queda para fases posteriores

Aunque ya existían puntos de sustitución mediante puertos, muchas operaciones continuaban dependiendo de `repos`, `session.execute`, `get_repositories()`, `UploadFile`, `ZipFile` o servicios auxiliares aún no desacoplados. Por ello, con el criterio estricto adoptado en el TFM, T2 se mantuvo todavía en 0 tras esta fase.

C.5. Interpretación y límites de la fase

La Fase 1 mejoró el desacoplamiento inicial y preparó el terreno para pruebas más controlables, pero no pretendía resolver todavía los problemas principales de concentración funcional ni producir una mejora fuerte de testabilidad aislada. Su aportación específica al conjunto del TFM fue dejar establecidas las primeras fronteras explícitas entre orquestación e infraestructura.

Tras esta fase permanecieron abiertos varios problemas relevantes, que fueron abordados de forma progresiva en las fases posteriores:

- La lógica de creación desde web seguía residiendo en `_create_recipe_from_web` dentro del controlador, aspecto tratado posteriormente en la Fase 2.
- `RepositoryRecipes` seguía lanzando `HTTPException`, problema abordado en la Fase 3.
- La política de permisos continuaba integrada en `RecipeService`, responsabilidad extraída en la Fase 4.
- `RecipeService` seguía siendo un servicio de gran tamaño y alta concentración funcional, cuestión abordada de forma material en la Fase 5.
- Persistían dependencias directas o contextuales como `get_repositories()`, `session.execute`, `UploadFile`, `ZipFile`, `copytree` y `rmtree`, cuya reducción completa quedaba fuera del alcance de esta primera intervención.

En consecuencia, la Fase 1 no debe evaluarse por una reducción inmediata de complejidad o tamaño, sino por su capacidad para iniciar una dirección arquitectónica verificable: hacer explícitas dependencias externas clave y preparar el módulo para las extracciones incrementales que se desarrollan en las fases siguientes.

Por su carácter preparatorio, la Fase 1 no cuenta con un anexo de validación independiente. En esta etapa, la evidencia obtenida fue principalmente estructural: se redujeron dependencias directas, se introdujeron puertos y adaptadores, y se habilitaron los primeros puntos de sustitución. Las validaciones ejecutables específicas se incorporan a partir de la Fase 2, cuando empiezan a extraerse flujos completos acompañados de pruebas unitarias y funcionales.

D. Intervención de Fase 2: creación de recetas desde web

D.1. Propósito y relación con la memoria

Este anexo amplía la descripción de la Fase 2 presentada en el Capítulo 3 y complementa la validación recogida en el Capítulo 4. Su función es documentar con mayor detalle la primera extracción completa de un flujo de aplicación desde la capa HTTP hacia una unidad explícita de aplicación: `CreateRecipeFromWeb`.

A diferencia de la Fase 1, centrada en preparar fronteras mediante puertos y adaptadores, esta fase sí desplaza una responsabilidad completa fuera del controlador. Por ello, su valor principal reside en mostrar cómo un flujo real del módulo de recetas puede dejar de estar orquestado por la capa de entrada y pasar a expresarse como un caso de uso aislable, con dependencias externas representadas mediante contratos explícitos.

El anexo no sustituye la evaluación principal del Capítulo 4 ni el detalle probatorio de la validación asociada. Se centra en la trazabilidad técnica de la intervención: problema abordado, artefactos introducidos, cambios de responsabilidad, métricas específicas y riesgos asumidos.

D.2. Problema específico abordado por la Fase 2

Antes de esta fase, el flujo de creación de recetas desde web permanecía alojado en el controlador HTTP. El método `_create_recipe_from_web` reunía en un mismo punto la invocación del scraper, el enriquecimiento de tags y categorías, la persistencia de la receta y la publicación del evento de creación.

Aunque la Fase 1 ya había introducido puertos y adaptadores para reducir algunas dependencias concretas, la responsabilidad de orquestación seguía residiendo en la capa de entrada. Esto generaba dos problemas principales. En primer lugar, el controlador asumía decisiones propias de aplicación y no sólo tareas de adaptación HTTP. En segundo lugar, el flujo resultaba difícil de probar de forma aislada, porque su lógica estaba mezclada con detalles de transporte, colaboradores concretos y composición del sistema.

La Fase 2 abordó este problema trasladando la secuencia completa de creación desde web a un caso de uso explícito. El objetivo no era únicamente mover código, sino convertir el flujo en una unidad de aplicación con identidad propia, desacoplada de FastAPI y preparada para ser validada mediante dobles de prueba.

D.3. Intervención arquitectónica aplicada

La intervención consistió en introducir el caso de uso `CreateRecipeFromWeb` y reorganizar las dependencias necesarias para que el controlador dejara de orquestar el flujo directamente. A partir de esta fase, la capa HTTP quedó

limitada a construir el comando de entrada, invocar el caso de uso y adaptar la respuesta al contrato esperado por la API.

D.3.1. Caso de uso introducido

Se introdujo la clase `CreateRecipeFromWeb` en el archivo:

- `mealie/services/recipe/application/create_recipe_from_web.py`

Esta clase recibe cuatro dependencias abstractas:

- `RecipeScraperPort`
- `RecipeCreationPort`
- `RecipeScrapedDataEnricherPort`
- `RecipeCreatedEventPort`

La ejecución del caso de uso queda definida por una secuencia explícita:

1. invocar el scraper mediante puerto;
2. enriquecer la receta con tags y categorías cuando corresponda;
3. persistir la receta mediante un puerto de creación;
4. publicar el evento de creación mediante un puerto de publicación.

La clase no depende de `FastAPI`, `SQLAlchemy`, `RecipeDataService`, `get_repositories()`, `create_from_html` ni de utilidades propias del transporte HTTP. Esta separación permitió convertir el flujo en una unidad evaluable con dobles de prueba sobre sus colaboradores.

D.3.2. Puertos y adaptadores añadidos

Para soportar el nuevo caso de uso sin romper la compatibilidad con el sistema existente, se añadieron tres puertos auxiliares en `mealie/services/recipe/ports.py`:

- `RecipeCreationPort`: representa la operación de creación y persistencia de la receta.
- `RecipeScrapedDataEnricherPort`: encapsula el enriquecimiento de datos procedentes del scraping, especialmente tags y categorías.
- `RecipeCreatedEventPort`: abstrae la publicación del evento de creación de receta.

En `mealie/services/recipe/adapters.py` se incorporaron los adaptadores correspondientes:

- `CallableRecipeCreationAdapter`
- `ScrapedRecipeDataEnricherAdapter`
- `CallableRecipeCreatedEventAdapter`

El primero encapsula el uso del mecanismo existente de creación de recetas, el segundo concentra la resolución de tags y categorías a partir de los datos extraídos, y el tercero permite publicar eventos sin exponer la infraestructura concreta al caso de uso.

La trazabilidad entre problema y artefacto puede resumirse así:

- la orquestación alojada en `RecipeController` se trasladó a `CreateRecipeFromWeb`;
- la colaboración con scraping quedó expresada mediante `RecipeScraperPort`;
- el enriquecimiento de datos extraídos pasó a depender de `RecipeScrapedDataEnricherPort`;
- la persistencia de la receta se representó mediante `RecipeCreationPort`;
- la publicación de eventos se desacopló mediante `RecipeCreatedEventPort`.

D.3.3. Cambios en el ensamblaje y en el controlador HTTP

La integración del caso de uso exigió cambios en varios archivos del módulo:

- `mealie/services/recipe/application/__init__.py`: incorporación del nuevo paquete de aplicación.
- `mealie/services/recipe/application/create_recipe_from_web.py`: definición del nuevo caso de uso.
- `mealie/services/recipe/ports.py`: incorporación de los puertos auxiliares necesarios.
- `mealie/services/recipe/adapters.py`: incorporación de adaptadores compatibles con la infraestructura existente.
- `mealie/routes/recipe/_base.py`: ensamblaje del caso de uso y de sus colaboradores.
- `mealie/routes/recipe/recipe_crud_routes.py`: delegación del flujo desde las rutas HTTP hacia el nuevo caso de uso.
- `tests/unit_tests/services_tests/recipe_tests/test_create_recipe_from_web.py`: pruebas unitarias específicas del caso de uso.
- `tests/integration_tests/user_recipe_tests/test_recipe_create_from_web_phase2.py`: validación funcional mínima de los endpoints afectados.

En `BaseRecipeController` se añadió la propiedad `create_recipe_from_web_use_case`, encargada de construir el caso de uso con sus dependencias. También se incorporó el método `publish_recipe_created_event`, utilizado como adaptación conservadora hacia el mecanismo existente de publicación de eventos.

Como resultado, el controlador `RecipeController` dejó de orquestar directamente scraping, enriquecimiento, persistencia y publicación de eventos. Las rutas `/create/html-or-json` y `/create/url` pasaron a construir un `CreateRecipeFromWebCommand`, invocar el caso de uso y transformar el resultado a la respuesta esperada, manteniendo la validación de entrada, la captura de `ForceTimeoutException` y la adaptación de salida como responsabilidades propias de la capa HTTP.

Durante la integración de la fase se detectó una recursión de inicialización en `RecipeServiceBase` al construir el colaborador por defecto para OpenAI. La corrección fue local y conservadora, reutilizando la propia instancia cuando

correspondía y evitando una construcción recursiva innecesaria. Esta incidencia refuerza la importancia de validar tempranamente sobre el sistema real, ya que algunos problemas sólo aparecen al integrar la solución en su contexto completo.

D.4. Diagrama antes y después

La Figura 2 resume el cambio de responsabilidad introducido por la Fase 2. Antes de la intervención, el controlador concentraba la orquestación del flujo de creación desde web. Después, esa responsabilidad pasa al caso de uso `CreateRecipeFromWeb`, mientras que el controlador conserva tareas propias de adaptación HTTP: construir el comando, invocar el caso de uso y devolver la respuesta esperada.

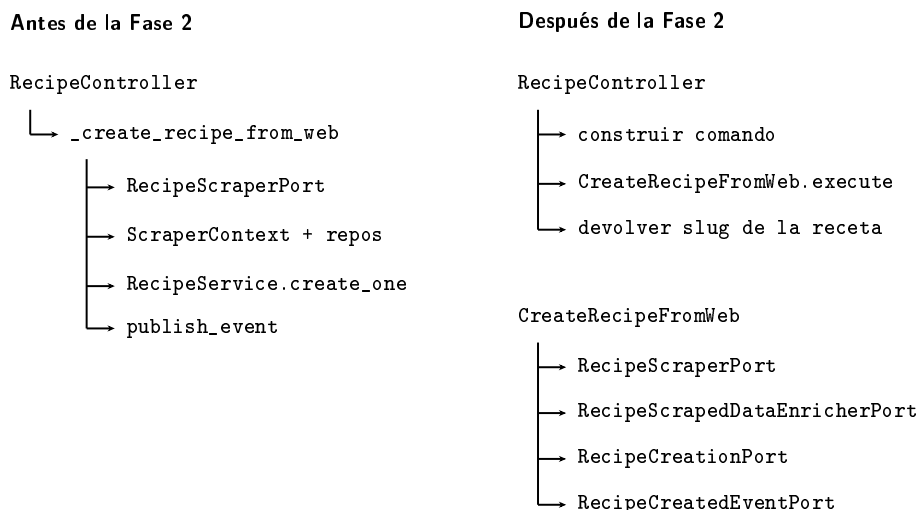


Figura 2. Cambio de responsabilidad del flujo `CreateRecipeFromWeb` en la Fase 2.

D.5. Resultados específicos de la Fase 2

Las métricas se presentan según el marco definido en el Anexo B. Los valores de estado inicial, Fase 1 y Fase 2 se reproducen sobre los snapshots etiquetados `tfm-baseline`, `tfm-phase-1` y `tfm-phase-2` del repositorio utilizado en la intervención.

D.5.1. Acoplamiento

ID	Métrica	Base	Fase 1	Fase 2	Observación
C1	Referencias concretas de infraestructura en la orquestación del módulo	9	1	1	La mejora principal ocurrió en la Fase 1; en la Fase 2 el flujo web queda desacoplado, aunque el módulo mantiene una instanciación concreta de <code>OpenAIRecipeService</code> en <code>RecipeServiceBase</code> .
C2	Dependencias salientes inyectables explícitas en los principales orquestadores	0	3	7	La Fase 2 añade cuatro dependencias abstractas al nuevo caso de uso, además de las tres ya introducidas en <code>RecipeServiceBase</code> .
C3	Referencias HTTP en la persistencia específica de recetas	2	2	2	No mejora todavía; esta métrica queda reservada para la Fase 3.
C4	Invocaciones directas a <code>get_repositories()</code> dentro del módulo de recetas	6	6	6	Sin mejora en esta fase; la extracción del caso de uso no modifica el uso de <i>service locator</i> en el resto del módulo.

D.5.2. Complejidad

ID	Métrica	Base	Fase 1	Fase 2	Observación
X1	Complejidad estructural del flujo web principal	8	8	2 / 1	En estado inicial y Fase 1, el flujo se concentraba en <code>_create_recipe_from_web</code> . En Fase 2 se reparte entre <code>_execute_create_recipe_from_web</code> y <code>CreateRecipeFromWeb.execute</code> .
X1	Complejidad máxima observada en <code>RecipeController</code>	8	8	8	No mejora a nivel global del controlador porque <code>get_all</code> sigue siendo el punto crítico dominante.

ID	Métrica	Base	Fase 1	Fase 2	Observación
X2	Número de métodos en RecipeController	23	23	23	Sin mejora cuantitativa; la simplificación del flujo no reduce el número total de endpoints.
X2	Número de métodos en la nueva clase de aplicación	N/A	N/A	2	La Fase 2 introduce un orquestador pequeño y cohesionado.
X3	Extensión estructural de RecipeController	531 líneas	531 líneas	512 líneas	Mejora moderada: el controlador reduce su rango estructural en 19 líneas respecto a la Fase 1.
X3	Extensión estructural del caso de uso CreateRecipeFromWeb	N/A	N/A	24 líneas	La nueva unidad de aplicación es compacta y de responsabilidad acotada.

En la fila relativa a la complejidad estructural del flujo web principal, el valor 2/1 indica la complejidad de `_execute_create_recipe_from_web` y de `CreateRecipeFromWeb.execute`, respectivamente.

D.5.3. Tamaño

ID	Elemento	Base	Fase 1	Fase 2	Variación Fase 1 - Fase 2
S1	<code>recipe_service.py</code>	627 LOC	646 LOC	649 LOC	+3
S1	<code>recipe_crud_routes.py</code>	608 LOC	607 LOC	588 LOC	-19
S1	<code>recipe_bulk_scraper.py</code>	127 LOC	134 LOC	134 LOC	0
S1	<code>_base.py</code>	56 LOC	62 LOC	92 LOC	+30
S1	<code>ports.py</code>	0 LOC	50 LOC	70 LOC	+20
S1	<code>adapters.py</code>	0 LOC	59 LOC	107 LOC	+48
S1	<code>application/create_recipe_from_web.py</code>	0 LOC	0 LOC	44 LOC	+44

ID	Elemento	Base	Fase 1	Fase 2	Variación Fase 1 - Fase 2
S2	Tamaño total del conjunto intervenido	1418 LOC	1558 LOC	1684 LOC	+126

El tamaño total aumenta en esta fase. Esta evolución es consistente con una estrategia incremental basada en extraer responsabilidades hacia unidades explícitas sin eliminar todavía todos los artefactos heredados. El incremento se explica principalmente por la aparición del caso de uso, los puertos auxiliares, los adaptadores asociados y los cambios de ensamblaje necesarios para integrarlos en Mealie.

D.5.4. Testabilidad

ID	Métrica	Base	Fase 1	Fase 2	Observación
T1	Dependencias externas sustituibles por contratos explícitos	0	3	7	La Fase 2 aporta cuatro colaboradores sustituibles por dobles de prueba al flujo <code>CreateRecipeFromWeb</code> ; sumados a los tres puertos introducidos en la Fase 1, el valor acumulado alcanza siete.
T2	Flujos unitariamente evaluables sin base de datos ni filesystem con criterio estricto	0	0	1	El flujo <code>CreateRecipeFromWeb</code> pasa a ser testeable de forma aislada mediante dobles de prueba.
T3	Necesidad de reemplazos dinámicos de funciones globales para aislar scraping	2	0	0	La mejora ya obtenida en la Fase 1 se mantiene en la Fase 2.

D.6. Interpretación arquitectónica de la fase

La Fase 2 produce una mejora localizada pero relevante. La complejidad del flujo web deja de estar concentrada en el controlador y pasa a distribuirse entre una adaptación HTTP ligera y un caso de uso de aplicación compacto. El método `_create_recipe_from_web`, que actuaba como orquestador principal, deja de concentrar la secuencia completa del flujo, y `CreateRecipeFromWeb` asume esa responsabilidad con dependencias explícitas.

Desde el punto de vista de separación de responsabilidades, el controlador se aproxima más al papel esperado de un adaptador de entrada: adapta la entrada HTTP a un comando, delega la ejecución en un caso de uso, transforma la salida en el `slug` de la receta creada y conserva el mapeo de una excepción técnica concreta a HTTP. La orquestación de scraping, enriquecimiento, persistencia y eventos queda desplazada a la capa de aplicación.

La mejora en testabilidad es especialmente relevante. Por primera vez en la intervención, existe un flujo del módulo de recetas que puede probarse de forma plenamente aislada con criterio estricto. Esta mejora no se limita a la posibilidad de escribir más pruebas, sino que reduce el coste de validación del flujo y refuerza la trazabilidad entre diseño arquitectónico y evidencia ejecutable.

No obstante, la mejora sigue siendo acotada. No se observan todavía cambios en C3 ni en C4: la persistencia continúa manteniendo referencias a `HTTPException` y el módulo sigue utilizando `get_repositories()` como mecanismo de *service locator* en varios puntos. Tampoco se reduce el número de métodos de `RecipeController`, ya que la fase no elimina endpoints ni rediseña toda la superficie de entrada del módulo. La mejora del controlador es principalmente estructural y de responsabilidad, no de reducción global del catálogo de métodos.

En consecuencia, la Fase 2 debe entenderse como una intervención orientada al desacoplamiento de un flujo representativo, no como una simplificación total del módulo. Su importancia reside en demostrar que la arquitectura objetivo podía materializarse en un flujo real de Mealie sin romper el comportamiento funcional y generando, por primera vez, una unidad de aplicación estrictamente aislable.

D.7. Riesgos y costes de la intervención

La Fase 2 introdujo varios riesgos controlados asociados a la estrategia incremental:

- incremento del número de artefactos arquitectónicos, lo que puede elevar inicialmente el coste cognitivo para nuevos contribuidores;
- posible dispersión temporal de la lógica entre adaptadores, caso de uso y controlador base mientras `RecipeService` siga concentrando otras responsabilidades;
- riesgo de duplicación futura si otros flujos similares se extraen sin consolidar patrones comunes de casos de uso.

Estos riesgos son razonables dentro de una estrategia incremental. La fase no buscaba cerrar toda la arquitectura del módulo, sino abrir una frontera clara entre entrada y aplicación sobre un flujo representativo, manteniendo la compatibilidad funcional con Mealie.

D.8. Evidencia asociada

La validación detallada de esta fase se conserva en el Anexo H. Ese anexo registra comandos, resultados, amenazas a la validez e incidencias descubiertas durante la ejecución.

En relación con el presente anexo, la evidencia asociada permite conectar la intervención técnica con su verificación: la extracción de `CreateRecipeFromWeb` no sólo reorganizó responsabilidades, sino que quedó respaldada por pruebas unitarias específicas, validación funcional mínima y comparación reproducible de métricas.

E. Intervención de Fase 3: separación entre HTTP y persistencia

E.1. Propósito y relación con la memoria

Este anexo amplía la descripción de la Fase 3 presentada en el Capítulo 3 y complementa la validación recogida en el Capítulo 4. Su función es documentar con mayor detalle la intervención orientada a eliminar la contaminación HTTP en la persistencia específica del módulo de recetas.

A diferencia de otras fases centradas en extraer casos de uso o reducir puntos críticos, esta intervención tuvo un alcance deliberadamente estrecho: corregir la dirección de dependencias entre persistencia y entrada HTTP. Por ello, el anexo se centra en las excepciones introducidas, en los cambios aplicados sobre `RepositoryRecipes`, en el mapeo de errores en la capa de entrada y en la evolución de la métrica C3.

El anexo no reabre la evaluación global de la intervención ni pretende documentar una refactorización completa del subsistema de repositorios. Su valor reside en mostrar cómo se corrigió un problema arquitectónico concreto sin ampliar el alcance hacia componentes compartidos del backend.

E.2. Problema específico abordado por la Fase 3

Antes de esta fase, la persistencia específica del módulo de recetas mantenía una dependencia impropia respecto a conceptos propios de la capa HTTP. En particular, `RepositoryRecipes` importaba y lanzaba `HttpException` en operaciones de consulta, especialmente en escenarios relacionados con filtros, ordenación y sugerencias.

Esta situación introducía una contaminación entre capas. La persistencia no sólo recuperaba o validaba datos, sino que condicionaba directamente la forma HTTP del error externo. Como consecuencia, el repositorio quedaba ligado a FastAPI y resultaba menos reutilizable conceptualmente fuera de un flujo HTTP.

El problema era acotado, pero arquitectónicamente relevante. No impedía que el módulo funcionara, pero sí degradaba la dirección de dependencias: una capa

interna conocía detalles de transporte que deberían resolverse en los adaptadores de entrada.

La Fase 3 abordó este problema sustituyendo las excepciones HTTP en persistencia por excepciones de aplicación y trasladando la conversión a respuesta HTTP 400 a los controladores correspondientes. El alcance se mantuvo limitado al módulo de recetas y no incluyó una refactorización transversal del repositorio genérico compartido.

E.3. Intervención arquitectónica aplicada

La intervención consistió en separar dos responsabilidades que en el estado inicial aparecían mezcladas:

- la persistencia debía expresar errores semánticos de consulta mediante excepciones propias de aplicación;
- la capa de entrada debía decidir cómo traducir esos errores al contrato HTTP externo.

De este modo, el comportamiento observable de los endpoints se preservó, pero la responsabilidad de construir respuestas HTTP dejó de residir en la persistencia.

E.3.1. Excepciones de aplicación introducidas

Se añadieron dos excepciones específicas en `mealie/core/exceptions.py`:

- `QueryFilterValidationError`
- `OrderByValidationError`

Ambas excepciones expresan errores de validación de consultas desde una perspectiva de aplicación, sin asumir transporte HTTP ni detalles de FastAPI. Su finalidad es permitir que el repositorio comunique errores semánticos sin decidir directamente el código de estado externo que debe recibir el cliente.

E.3.2. Desacoplamiento de `RepositoryRecipes`

`RepositoryRecipes` dejó de importar y lanzar `HTTPException`. Para ello se sustituyó la dependencia directa por las nuevas excepciones de aplicación y se introdujeron overrides locales de la lógica necesaria de paginación y ordenación dentro del repositorio específico de recetas.

Los métodos afectados fueron:

- `add_pagination_to_query`
- `add_order_attr_to_query`
- `add_order_by_to_query`

La decisión de implementar overrides locales, en lugar de modificar el repositorio genérico compartido, responde a un criterio metodológico de alcance

controlado. La Fase 3 debía eliminar la contaminación HTTP en la persistencia de recetas, no acometer todavía una refactorización transversal del subsistema de repositorios.

E.3.3. Mapeo HTTP en la capa de entrada

El controlador autenticado de recetas, ubicado en `recipe_crud_routes.py`, y el controlador público de exploración, ubicado en `controller_public_recipes.py`, pasaron a capturar `QueryFilterValidationError` y `OrderByValidationError`. Estas excepciones se transforman ahora explícitamente en respuestas HTTP 400.

La responsabilidad queda, por tanto, distribuida de forma más coherente:

- la persistencia informa de errores semánticos de consulta;
- la capa de entrada decide cómo traducir esos errores al contrato HTTP externo;
- el comportamiento observable para el cliente se mantiene estable.

E.4. Archivos modificados y tipo de cambio

La Fase 3 modificó un conjunto reducido de archivos de producción y añadió pruebas específicas para validar tanto el comportamiento interno como el contrato HTTP externo.

E.4.1. Archivos modificados en producción

- `mealie/core/exceptions.py`: incorporación de `QueryFilterValidationError` y `OrderByValidationError`.
- `mealie/repos/repository_recipes.py`: sustitución de `HTTPException` por excepciones de aplicación y overrides locales para la lógica de consulta necesaria.
- `mealie/routes/recipe/recipe_crud_routes.py`: captura de excepciones de aplicación y mapeo explícito a respuesta HTTP 400.
- `mealie/routes/explore/controller_public_recipes.py`: aplicación del mismo criterio de mapeo HTTP en el controlador público afectado.

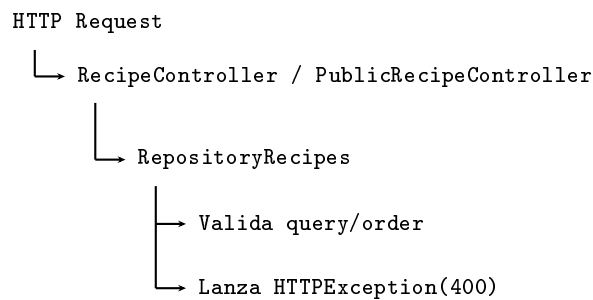
E.4.2. Archivos de validación añadidos

- `tests/unit_tests/repository_tests/test_recipe_repository_phase3.py`: pruebas focalizadas sobre las nuevas excepciones de aplicación en el nivel de repositorio.
- `tests/integration_tests/user_recipe_tests/test_recipe_phase3_http_mapping.py`: pruebas funcionales mínimas para comprobar que los endpoints afectados siguen devolviendo HTTP 400 ante parámetros inválidos.

E.5. Diagrama antes y después

La Figura 3 resume el desplazamiento de responsabilidad introducido por la Fase 3. Antes de la intervención, la persistencia decidía directamente el error HTTP. Después de la intervención, el repositorio emite una excepción de aplicación y la capa de entrada realiza el mapeo a HTTP.

Antes de la Fase 3



Después de la Fase 3

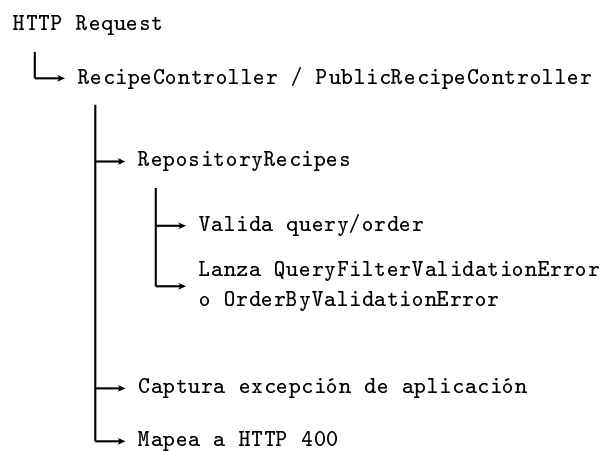


Figura 3. Cambio de responsabilidad en la gestión de errores de consulta durante la Fase 3.

E.6. Resultados específicos de la Fase 3

Las métricas se presentan según el marco definido en el Anexo B. Para mantener la trazabilidad de la intervención, se comparan estado inicial, Fase 1, Fase 2 y Fase 3 sobre los snapshots etiquetados `tfm-baseline`, `tfm-phase-1`, `tfm-phase-2` y `tfm-phase-3` del repositorio utilizado en el trabajo.

E.6.1. Acoplamiento

ID	Métrica	Base	Fase 1	Fase 2	Fase 3	Observación
C1	Referencias concretas de infraestructura en la orquestación	9	1	1	1	Sin cambio en esta fase; el objetivo no era seguir reduciendo dependencias concretas de orquestación.
C2	Dependencias inyectables explícitas	0	3	7	7	Se mantiene el nivel alcanzado en la Fase 2.
C3	Referencias HTTP en la persistencia específica de recetas	2	2	2	0	Mejora objetivo de la fase: la persistencia deja de depender de <code>HTTPEXception</code> .
C4	Invocaciones a <code>get_repositorios()</code> dentro del módulo	6	6	6	6	Sin cambio; esta métrica queda fuera del alcance específico de esta fase.

E.6.2. Complejidad y tamaño

La Fase 3 no buscaba reducir complejidad, sino corregir la dirección de dependencias entre capas. Por ello, las métricas de complejidad y tamaño deben interpretarse como costes locales derivados de reubicar la responsabilidad de mapeo HTTP en la capa de entrada.

ID	Elemento	Base	Fase 1	Fase 2	Fase 3	Observación
X1	Complejidad máxima observada en <code>RecipeController</code>	8	8	8	10	Ligero incremento local por el mapeo de errores; es un coste aceptable para situar la responsabilidad en la capa de entrada.
X3	Extensión estructural de <code>RecipeController</code>	531	531	512	527	Parte de la reducción de la Fase 2 se compensa por el nuevo mapeo explícito, sin revertir el desacoplamiento logrado.
S1	<code>recipe_service.py</code>	627 LOC	646 LOC	649 LOC	649 LOC	Sin variación respecto a la Fase 2.
S1	<code>recipe_crud_routes.py</code>	608 LOC	607 LOC	588 LOC	603 LOC	Incremento localizado en la entrada HTTP por el mapeo explícito de excepciones.
S1	<code>recipe_bulk_scrapaper.py</code>	127 LOC	134 LOC	134 LOC	134 LOC	Sin variación.
S1	<code>_base.py</code>	56 LOC	62 LOC	92 LOC	92 LOC	Sin variación.
S1	<code>ports.py</code>	0 LOC	50 LOC	70 LOC	70 LOC	Sin variación.
S1	<code>adapters.py</code>	0 LOC	59 LOC	107 LOC	107 LOC	Sin variación.
S1	<code>application/create_recipe_from_web.py</code>	0 LOC	0 LOC	44 LOC	44 LOC	Sin variación.
S2	Tamaño total del conjunto intervenido	1418 LOC	1558 LOC	1684 LOC	1699 LOC	Aumento de 15 LOC respecto a la Fase 2, concentrado en la capa de entrada.

El incremento de tamaño es mínimo y se concentra en la capa de entrada, donde ahora reside de forma explícita el mapeo HTTP que antes estaba incorrectamente oculto en persistencia.

E.6.3. Testabilidad

Las métricas agregadas de testabilidad no cambian en esta fase, porque la intervención no introduce nuevos casos de uso ni nuevos puertos de salida. Sin

embargo, la fase sí mejora la capacidad de probar la persistencia sin razonar en términos de HTTP.

ID	Métrica	Base	Fase 1	Fase 2	Fase 3	Observación
T1	Dependencias sustituibles por contratos explícitos	0	3	7	7	La fase no añade nuevos puertos, pero conserva la mejora ya obtenida.
T2	Flujos unitariamente evaluables sin base de datos o filesystem con criterio estricto	0	0	1	1	Se mantiene el caso de uso aislable introducido en la Fase 2.
T3	Necesidad de reemplazos dinámicos de funciones globales	2	0	0	0	La mejora previa se mantiene.

Además de estas métricas agregadas, la Fase 3 introduce una mejora cualitativa relevante: la capa de persistencia puede probarse respecto a validación de filtros y ordenación sin exigir conocimiento de códigos HTTP ni de excepciones propias de FastAPI.

E.7. Interpretación arquitectónica de la fase

La mejora central de la Fase 3 se produce en C3. La persistencia específica del módulo de recetas deja de contener referencias a `HTTPException` y deja de expresar errores de consulta con semántica HTTP. Esta reducción de 2 a 0 constituye una mejora arquitectónica directa, alineada exactamente con el objetivo de la fase.

Desde el punto de vista de dirección de dependencias, la modificación reubica la responsabilidad en la capa adecuada. Antes, la persistencia conocía FastAPI y decidía directamente la forma HTTP del error. Después, `RepositoryRecipes` emite excepciones de aplicación y los controladores afectados realizan el mapeo a HTTP 400. La dependencia impropia queda así eliminada dentro del alcance intervenido.

Este cambio también mejora la reutilización conceptual de la persistencia. Al eliminar `HTTPException` de `RepositoryRecipes`, el repositorio queda menos ligado a FastAPI y podría ser utilizado, al menos conceptualmente, desde otros contextos de entrada, como tareas internas, scripts o futuros adaptadores no HTTP. Aunque esa reutilización no se explota durante el TFM, la barrera arquitectónica queda eliminada en el punto intervenido.

El coste asumido es pequeño y localizado. La capa de entrada gana algo de lógica explícita de mapeo y `recipe_crud_routes.py` aumenta ligeramente de tamaño, pero ese crecimiento representa una reubicación correcta de responsabilidad: el código HTTP no desaparece, sino que pasa a residir en el lugar arquitectónicamente adecuado.

La fase no modifica otros problemas del módulo. No se observan mejoras en C4, el uso de `get_repositories()` permanece estable y la complejidad global de `RecipeService` no se ve afectada. Este resultado es coherente con el alcance de la Fase 3: corregir contaminación HTTP en persistencia no equivale a migrar permisos ni a descomponer el servicio principal.

E.8. Riesgos y costes de la intervención

La Fase 3 introduce riesgos controlados y acotados:

- Existe duplicación local de cierta lógica de paginación y ordenación en `RepositoryRecipes`, derivada de la decisión metodológica de no modificar todavía el repositorio genérico compartido.
- La capa de entrada gana lógica explícita de mapeo de errores, lo que aumenta ligeramente su complejidad local.
- Puede quedar trabajo técnico pendiente si en trabajos futuros se decide homogeneizar este patrón con otros repositorios o módulos del backend.

Estos riesgos son aceptables dentro del alcance de la intervención, porque preservan el foco del TFM y evitan abrir una refactorización transversal no prevista en la hipótesis de trabajo.

E.9. Evidencia asociada

La validación detallada de esta fase se conserva en el Anexo I. Ese anexo reúne comandos, resultados y amenazas a la validez vinculadas a la corrección de C3.

En relación con el presente anexo, la evidencia asociada permite conectar la intervención técnica con su verificación: la eliminación de semántica HTTP en persistencia quedó respaldada por pruebas focalizadas de repositorio, validación funcional mínima de los endpoints afectados y comparación reproducible de métricas.

F. Intervención de Fase 4: política de permisos

F.1. Propósito y relación con la memoria

Este anexo amplía la descripción de la Fase 4 presentada en el Capítulo 3 y complementa la validación recogida en el Capítulo 4. Su función es documentar con mayor detalle la extracción de una responsabilidad de negocio concreta desde `RecipeService`: la política de permisos del módulo de recetas.

A diferencia de las fases centradas en infraestructura o en la extracción de flujos completos desde la capa HTTP, esta intervención se orienta a mejorar la cohesión del servicio principal y la testabilidad aislada de una regla de negocio relevante. Por ello, el anexo se centra en la política `RecipePermissionPolicy`, en los puertos asociados, en el adaptador de acceso a datos y en el efecto local de la intervención sobre `RecipeService`.

El anexo no sustituye la evaluación consolidada del Capítulo 4 ni la validación detallada asociada a permisos. Su valor reside en documentar la trazabilidad técnica de la fase: qué responsabilidad se extrajo, qué artefactos se introdujeron, cómo se integraron en el servicio existente y qué límites permanecieron abiertos para fases posteriores.

F.2. Problema específico abordado por la Fase 4

Antes de esta fase, `RecipeService` seguía concentrando la lógica de permisos del módulo. El servicio decidía directamente quién podía actualizar, borrar, bloquear o desbloquear una receta, y mezclaba esas comprobaciones con operaciones de orquestación, acceso a datos y gestión de recursos asociados.

En el estado previo a la intervención, `RecipeService` combinaba tres tipos de responsabilidad dentro del mismo componente:

- orquestación de operaciones del módulo de recetas;
- acceso a persistencia y recursos asociados;
- decisiones de autorización sobre actualización, borrado y *lock/unlock*.

Además, parte de esas decisiones dependía de una consulta SQL incrustada directamente en el servicio. Esto dificultaba considerar la política de permisos como una unidad explícita del dominio y hacía más costosa su validación aislada.

La Fase 4 abordó este problema extrayendo una política explícita, `RecipePermissionPolicy`, y desplazando el acceso a datos necesario para evaluar permisos a un adaptador de infraestructura específico. El objetivo era doble: mejorar la cohesión de `RecipeService` y convertir la autorización en una unidad testeable mediante puertos.

F.3. Intervención arquitectónica aplicada

La intervención consistió en separar la regla de autorización de la lógica de orquestación del servicio principal. La política pasa a concentrar las decisiones sobre permisos, mientras que el acceso a datos necesario para evaluar esas decisiones queda encapsulado tras un puerto y un adaptador.

F.3.1. Política de permisos introducida

Se introdujo la clase `RecipePermissionPolicy` en el archivo:

- `mealie/services/recipe/domain/permission_policy.py`

Esta clase concentra las reglas de autorización del módulo para tres operaciones principales:

- actualización de recetas;
- borrado de recetas;
- bloqueo y desbloqueo de recetas.

La política ofrece métodos booleanos, como `can_update`, `can_delete` y `can_lock_unlock`, junto con métodos de aserción, como `assert_can_update`, `assert_can_delete` y `assert_can_lock_unlock`. Estos últimos lanzan `PermissionDenied` cuando la operación no está permitida.

Con esta decisión, la autorización deja de aparecer como una colección de métodos internos del servicio y pasa a formularse como una unidad con identidad arquitectónica propia. La política concentra decisiones de negocio, mientras que el acceso a datos necesario para evaluarlas se delega en un puerto específico.

F.3.2. Puertos y adaptador de permisos

En `mealie/services/recipe/ports.py` se añadieron dos contratos nuevos:

- `RecipePermissionAccessPort`
- `RecipePermissionPolicyPort`

`RecipePermissionAccessPort` expresa la capacidad de consultar la información necesaria para decidir permisos, como ownership, bloqueo de recetas o restricciones entre hogares. `RecipePermissionPolicyPort` permite que el servicio consuma una política de autorización sin quedar acoplado a una implementación concreta.

En `mealie/services/recipe/adapters.py` se añadió `SqlAlchemyRecipePermissionAccessAdapter`. Este adaptador encapsula la consulta SQL que anteriormente residía en `RecipeService.can_update` para comprobar:

- si el usuario actor es propietario de la receta;
- si la receta está bloqueada;
- si la política de edición entre hogares impide la modificación.

La consulta no desaparece, pero queda aislada como detalle de infraestructura detrás de un puerto. De este modo, la política de permisos puede evaluarse en pruebas mediante dobles de prueba, sin activar la consulta SQL real ni depender directamente de `SQLAlchemy`.

F.3.3. Simplificación de `RecipeService`

`RecipeServiceBase` pasó a aceptar `permission_policy` como dependencia opcional de tipo `RecipePermissionPolicyPort`. Cuando no se inyecta explícitamente, el servicio crea por defecto una instancia de `RecipePermissionPolicy` y respaldada por `SqlAlchemyRecipePermissionAccessAdapter`, manteniendo así la compatibilidad con la infraestructura existente.

Como consecuencia de esta integración:

- se eliminaron de `RecipeService` los métodos `can_update`, `can_delete` y `can_lock_unlock`;
- `_pre_update_check`, `update_recipe_image`, `delete_recipe_image` y `delete_many` delegan ahora en la política de permisos;
- la responsabilidad del servicio queda mejor delimitada como orquestación de operaciones, no como lugar de definición de la autorización.

F.3.4. Ajuste del instrumento de métricas

Para mantener la reproducibilidad de la evaluación, `support-files/recipe_module_metrics.py` se amplió de forma controlada con el fin de reconocer el nuevo artefacto introducido por la intervención.

El ajuste permitió:

- incluir `mealie/services/recipe/domain/permission_policy.py` en los archivos trazados;
- registrar `RecipePermissionPolicy` como unidad crítica relevante;
- contar esta política como flujo estrictamente testeable cuando sus dependencias se expresan mediante puertos.

Este ajuste no modifica retrospectivamente la línea base ni redefine las métricas empleadas en fases anteriores. Lo que hace es ampliar de forma declarada el perímetro instrumentado para incorporar un componente nuevo introducido por la intervención, manteniendo el mismo criterio operacional de medición sobre los distintos estados comparados. De este modo, la evaluación puede reflejar la arquitectura resultante sin alterar la comparabilidad metodológica de la serie.

F.4. Archivos modificados y tipo de cambio

La Fase 4 introdujo nuevos artefactos de dominio y modificó varios componentes existentes para integrar la política de permisos en el flujo del módulo.

F.4.1. Nuevos archivos

- `mealie/services/recipe/domain/__init__.py`: incorporación del nuevo paquete de dominio del módulo.
- `mealie/services/recipe/domain/permission_policy.py`: definición de `RecipePermissionPolicy` y de las reglas de autorización asociadas.
- `tests/unit_tests/services_tests/recipe_tests/test_recipe_permission_policy.py`: pruebas unitarias focalizadas sobre la política de permisos.

F.4.2. Archivos modificados

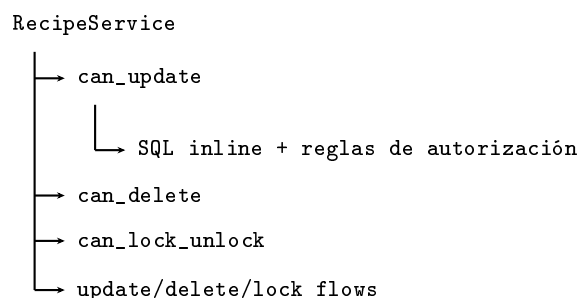
- `mealie/services/recipe/ports.py`: incorporación de `RecipePermissionAccessPort` y `RecipePermissionPolicyPort`.

- `mealie/services/recipe/adapters.py`: incorporación de `SqlAlchemyRecipePermissionAccessAdapter`.
- `mealie/services/recipe/recipe_service.py`: eliminación de métodos internos de permisos y delegación en la nueva política.
- `support-files/recipe_module_metrics.py`: ampliación del perímetro instrumentado para reconocer la nueva política como unidad relevante de medición.

F.5. Diagrama antes y después

La Figura 4 resume el desplazamiento de responsabilidad introducido por la Fase 4. Antes de la intervención, `RecipeService` contenía directamente la lógica de autorización. Después de la intervención, el servicio delega en una política explícita que obtiene la información necesaria mediante un puerto de acceso a permisos.

Antes de la Fase 4



Después de la Fase 4

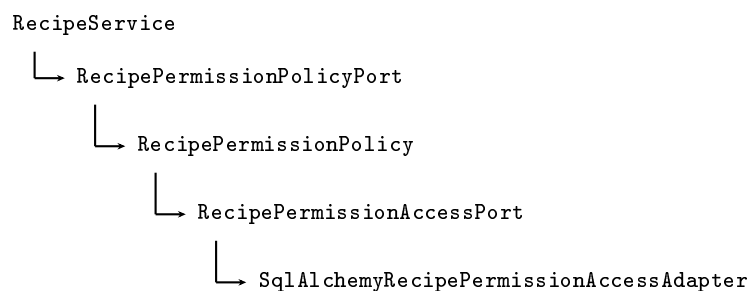


Figura 4. Cambio de responsabilidad en la gestión de permisos durante la Fase 4.

F.6. Resultados específicos de la Fase 4

Las métricas se presentan según el marco definido en el Anexo B. Para mantener la trazabilidad de la intervención, se comparan estado inicial, Fase 1, Fase 2, Fase 3 y Fase 4 sobre los snapshots etiquetados `tfm-baseline`, `tfm-phase-1`, `tfm-phase-2`, `tfm-phase-3` y `tfm-phase-4` del repositorio utilizado en el trabajo.

F.6.1. Acoplamiento

ID	Métrica	Base	Fase 1	Fase 2	Fase 3	Fase 4	Observación
C1	Referencias concretas de infraestructura en la orquestación del módulo	9	1	1	1	1	La mejora fuerte se mantiene; la Fase 4 no reintroduce dependencias concretas en la orquestación.
C2	Dependencias inyectables explícitas	0	3	7	7	9	Aumenta por la incorporación de <code>RecipePermissionPolicyPort</code> en <code>RecipeServiceBase</code> y <code>RecipePermissionAccessPort</code> en la nueva política.
C3	Referencias HTTP en la persistencia específica de recetas	2	2	2	0	0	Se conserva la mejora obtenida en la Fase 3.
C4	Invocaciones a <code>get_repositories()</code> en el módulo	6	6	6	6	6	Sin cambio; la reducción del <i>service locator</i> queda fuera del alcance de esta fase.

F.6.2. Complejidad

ID	Métrica	Estado previo relevante	Fase 4	Observación
X1	Complejidad de la responsabilidad extraída	La lógica de permisos estaba integrada en <code>RecipeService</code> junto con otras responsabilidades de orquestación	<code>RecipePermissionPolicy</code> presenta complejidades bajas, entre 1 y 3 por método	La fase no reduce toda la complejidad del módulo, pero extrae una responsabilidad concreta a una unidad pequeña y explícita.
X2	Métodos de <code>RecipeService</code>	El servicio acumulaba, además de la orquestación general, tres métodos de permisos embebidos	<code>RecipeService</code> queda en 24 métodos y la nueva <code>RecipePermissionPolicy</code> concentra 7 métodos en una clase separada	Se observa una fragmentación controlada de responsabilidades.
X3	Extensión estructural	<code>recipe_service.py</code> y mantenía un gran peso estructural en fases previas	<code>RecipeService</code> ocupa 442 líneas estructurales y <code>RecipePermissionPolicy</code> 57	Parte del volumen y de la responsabilidad se desplaza a una unidad pequeña, cohesionada y específica.

F.6.3. Tamaño

ID	Elemento	Base	Fase 1	Fase 2	Fase 3	Fase 4	Variación Fase 3–Fase 4
S1	<code>recipe_service.py</code>	627 LOC	646 LOC	649 LOC	649 LOC	598 LOC	-51
S1	<code>recipe_crud_routes.py</code>	608 LOC	607 LOC	588 LOC	603 LOC	603 LOC	0
S1	<code>recipe_bulk_scraper.py</code>	127 LOC	134 LOC	134 LOC	134 LOC	134 LOC	0
S1	<code>_base.py</code>	56 LOC	62 LOC	92 LOC	92 LOC	92 LOC	0
S1	<code>ports.py</code>	0 LOC	50 LOC	70 LOC	70 LOC	115 LOC	+45
S1	<code>adapters.py</code>	0 LOC	59 LOC	107 LOC	107 LOC	168 LOC	+61

ID	Elemento	Base	Fase 1	Fase 2	Fase 3	Fase 4	Variación Fase 3–Fase 4
S1	application/create_recipe_from_web.py	0 LOC	0 LOC	44 LOC	44 LOC	44 LOC	0
S1	domain/permissions_policy.py	0 LOC	0 LOC	0 LOC	0 LOC	65 LOC	+65
S2	Tamaño total del conjunto intervenido	1418 LOC	1558 LOC	1684 LOC	1699 LOC	1819 LOC	+120

La fase aumenta el tamaño total del conjunto observado, pero reduce de forma apreciable el volumen de `recipe_service.py`. El crecimiento neto responde a una redistribución explícita de responsabilidades: aparecen puertos, adaptadores y una política dedicada, mientras parte de la lógica deja de residir en el servicio principal.

F.6.4. Testabilidad

ID	Métrica	Base	Fase 1	Fase 2	Fase 3	Fase 4	Observación
T1	Dependencias sustituibles por contratos explícitos	0	3	7	7	9	La nueva política añade un punto más de sustitución y el servicio gana un colaborador abstracto adicional.
T2	Flujos unitariamente evaluables sin base de datos o filesystem con criterio estricto	0	0	1	1	2	<code>RecipePermissionPolicy</code> pasa a ser el segundo flujo del módulo validable de forma estricta sin infraestructura real.
T3	Necesidad de reemplazos dinámicos de funciones globales	2	0	0	0	0	La mejora previa se mantiene sin regresión.

F.7. Interpretación arquitectónica de la fase

La mejora principal de la Fase 4 se observa en cohesión y testabilidad. La política de permisos deja de estar diluida dentro de `RecipeService` y pasa a existir

como componente explícito del módulo. Esto eleva C2 de 7 a 9 y T2 de 1 a 2, lo que significa que el módulo gana una nueva unidad aislable y con dependencias abstraídas.

Desde el punto de vista de cohesión, **RecipeService** deja de ser responsable de decidir directamente reglas de autorización. El servicio conserva la coordinación de operaciones, pero delega la evaluación de permisos en un colaborador explícito. La autorización deja así de existir como detalle accidental del servicio y pasa a formularse como política con identidad propia.

La fase también mejora la distribución estructural del código. Aunque el tamaño total del conjunto crece, **recipe_service.py** reduce 51 líneas respecto a la Fase 3. Este resultado confirma que parte de la responsabilidad se ha extraído fuera del punto crítico principal, aunque el servicio siga manteniendo otras responsabilidades relevantes.

La mejora en validación aislada es igualmente relevante. **RecipePermissionPolicy** puede probarse con dobles de prueba sobre un único puerto de acceso a permisos. Esto reduce el coste de verificar reglas de ownership, borrado y *lock/unlock* sin activar base de datos ni recorrer endpoints HTTP.

La fase mantiene, no obstante, límites claros. No se observan mejoras en C4: el módulo sigue dependiendo de **get_repositories()** en los mismos puntos que en las fases anteriores. Tampoco cambia C1 ni C3, porque la intervención no persigue nuevas extracciones de infraestructura ni nuevas correcciones de semántica HTTP en persistencia. La complejidad global del módulo sigue dominada por **RecipeService** y **RecipeController**, por lo que la reducción material del punto crítico principal queda para una fase posterior.

El coste estructural asumido consiste en introducir nuevos contratos, un nuevo adaptador y una nueva política. Este incremento es metodológicamente aceptable porque hace visible una frontera arquitectónica que antes estaba implícita y era difícil de validar. En otras palabras, la fase intercambia tamaño global por mayor claridad estructural, cohesión y capacidad de prueba aislada.

F.8. Riesgos y costes de la intervención

La Fase 4 introduce riesgos controlados y acotados:

- El SQL de permisos continúa existiendo y sigue ligado a SQLAlchemy, aunque ahora queda encapsulado en un adaptador específico.
- El módulo gana nuevos artefactos arquitectónicos, lo que aumenta ligeramente el coste cognitivo inicial.
- La mejora de cohesión no elimina todavía el carácter amplio de **RecipeService**, que seguirá requiriendo una intervención posterior si se desea reducir el uso de *service locator* y simplificar más el servicio principal.

Estos riesgos son asumibles dentro de la estrategia incremental porque preservan el comportamiento y aíslan un problema real sin abrir todavía una refactorización transversal adicional del servicio principal.

F.9. Evidencia asociada

La validación detallada de esta fase se conserva en el Anexo J. Ese anexo reúne comandos, resultados e incidencias de validación vinculadas a permisos y testabilidad aislada.

En relación con el presente anexo, la evidencia asociada permite conectar la intervención técnica con su verificación: la extracción de **RecipePermission Policy** no sólo redistribuyó responsabilidades, sino que quedó respaldada por pruebas unitarias focalizadas, validación funcional mínima de escenarios sensibles y comparación reproducible de métricas.

G. Intervención de Fase 5: extracción de casos de uso

G.1. Propósito y relación con la memoria

Este anexo amplía la descripción de la Fase 5 presentada en el Capítulo 3 y complementa la evaluación recogida en el Capítulo 4. Su función es documentar con mayor detalle la intervención de mayor alcance estructural sobre **RecipeService**: la extracción de varios flujos completos hacia casos de uso explícitos de la capa de aplicación.

A diferencia de las fases anteriores, esta intervención no se centró en un único límite arquitectónico ni en una única responsabilidad aislada. Su objetivo fue reducir de forma material el punto crítico residual del módulo, aumentar el número de flujos unitariamente evaluables y disminuir parcialmente el uso de *service locator*. Por ello, este anexo se centra en los flujos extraídos, en los nuevos puertos y adaptadores, en el reensamblado del módulo y en la reducción local de **RecipeService**.

El anexo no sustituye la evaluación consolidada del Capítulo 4 ni la validación detallada asociada a la fase. Su valor reside en conservar la trazabilidad técnica de la intervención final: qué responsabilidades salieron realmente del servicio principal, cómo se reubicaron en casos de uso y qué límites permanecieron abiertos tras la descomposición.

G.2. Problema específico abordado por la Fase 5

La Fase 5 abordó el principal problema residual tras las fases anteriores: **RecipeService** seguía actuando como punto crítico del módulo de recetas. Aunque las fases previas habían introducido puertos, extraído un primer caso de uso, eliminado la semántica HTTP en persistencia y separado la política de permisos, el servicio principal continuaba acumulando operaciones heterogéneas.

Al cierre de la Fase 4 persistían cuatro síntomas relevantes:

- **RecipeService** continuaba acumulando operaciones de importación, duplicación y gestión de imagen.

- Varias de esas operaciones seguían mezclando coordinación, acceso a almacenamiento y decisiones de infraestructura dentro del mismo componente.
- El módulo conservaba invocaciones directas a `get_repositories()` en varios puntos.
- El número de flujos estrictamente testeables sin infraestructura real seguía siendo reducido, con T2=2.

Por tanto, la Fase 5 se planteó como una intervención de descomposición adicional, limitada y verificable, orientada a mejorar simultáneamente la cohesión de `RecipeService`, la explicitud arquitectónica de los flujos de receta, la testabilidad aislada del módulo y, cuando fuese posible sin ampliar el alcance, la reducción parcial del uso de *service locator*.

G.3. Criterio de selección del alcance

La Fase 5 no se ejecutó como una reescritura total del módulo. Se siguió un criterio de selección incremental basado en impacto arquitectónico y verificabilidad. Se priorizaron responsabilidades que cumplieran simultáneamente cuatro condiciones:

- residían todavía dentro de `RecipeService` y contribuían de forma directa a su carácter de punto crítico;
- podían expresarse como flujos completos de aplicación con puertos explícitos;
- disponían de cobertura funcional existente o fácilmente construible para validar la preservación de comportamiento;
- y permitían retirar comportamiento real del servicio, no sólo encapsularlo detrás de métodos delegadores internos.

Con este criterio se seleccionaron cinco flujos principales:

- creación de recetas desde ZIP;
- creación de recetas desde imágenes;
- duplicación de recetas;
- actualización de imagen de receta;
- borrado de imagen de receta.

También se extrajo el renderizado de plantillas de exportación fuera de `RecipeService` y se centralizó la construcción de los repositorios limitados al grupo doméstico del usuario, denominados repositorios *group-scoped*, en el controlador base de recetas, reduciendo así las llamadas internas a `get_repositories()`. Otros puntos del módulo, como timeline o rutas compartidas, se dejaron fuera del alcance de esta fase porque no contribuían directamente a reducir el punto crítico central ni a aumentar los flujos aislables.

G.4. Intervención arquitectónica aplicada

G.4.1. Casos de uso introducidos

Se introdujeron cinco casos de uso en `mealie/services/recipe/application` /:

- `CreateRecipeFromZip`
- `CreateRecipeFromImages`
- `DuplicateRecipe`
- `UpdateRecipeImage`
- `DeleteRecipeImage`

Cada uno de ellos encapsula una operación previamente alojada en `RecipeService` y expone un método `execute(...)` como punto de entrada principal. Esta decisión permite que cada flujo deje de estar implícito en un servicio generalista y pase a estar representado como una unidad arquitectónica reconocible, con colaboradores explícitos y validable de forma aislada.

G.4.2. Puertos y adaptadores añadidos

Para sostener estos casos de uso sin depender directamente de filesystem, normalización de datos, lectura de archivos ZIP o detalles de importación, se añadieron nuevos contratos en `ports.py`:

- `RecipeArchiveReaderPort`: lectura y extracción de recetas desde archivos ZIP.
- `RecipeDataNormalizerPort`: normalización de datos importados antes de persistirlos.
- `RecipeUploadedImageStagingPort`: preparación temporal de imágenes subidas.
- `RecipeDataCleaningPort`: limpieza de datos auxiliares durante procesos de importación.
- `RecipeAssetCopyPort`: copia de assets asociados a recetas duplicadas o importadas.
- `TemplateRendererPort`: renderizado de plantillas de exportación.

Junto con estos contratos se añadieron adaptadores concretos en `adapters.py` para la lectura de ZIP, staging de imágenes subidas, limpieza de datos, copia de assets y renderizado de plantillas.

La consecuencia arquitectónica es relevante: los nuevos flujos quedan desacoplados de detalles concretos de infraestructura sin alterar el comportamiento observable del sistema.

G.4.3. Reensamblado del controlador base

En `mealie/routes/recipe/_base.py` se centralizó la construcción de los repositorios *group-scoped* mediante la propiedad `group_repositories`. A partir de ella, el controlador base ensambla:

- `group_recipes`
- `group_cookbooks`
- El `RecipeService` principal con repositorio explícito;
- Los nuevos casos de uso de la Fase 5.

Esta decisión reduce llamadas dispersas a `get_repositories()` y desplaza la composición hacia un punto de ensamblaje más explícito. Aunque no elimina por completo el patrón *service locator*, sí reduce su presencia en el perímetro intervenido.

G.4.4. Reorganización de las rutas afectadas

Las rutas principales de `recipe_crud_routes.py` dejaron de delegar los flujos seleccionados en `RecipeService` y pasaron a invocar directamente los nuevos casos de uso.

Concretamente se reorganizaron los endpoints asociados a:

- Creación desde ZIP;
- Creación desde imágenes;
- Duplicación;
- Actualización de imagen;
- Borrado de imagen.

Esto era necesario para que la reducción sobre `RecipeService` fuese material y no meramente cosmética. Si el servicio hubiese conservado métodos delegadores internos, la responsabilidad real habría permanecido en el mismo punto crítico aunque parte del código se hubiera desplazado a nuevas clases.

G.4.5. Simplificación efectiva de `RecipeService`

Tras la intervención, `RecipeService` dejó de contener los métodos:

- `create_from_zip`
- `create_from_images`
- `duplicate_one`
- `update_recipe_image`
- `delete_recipe_image`
- `render_template`

Además, el servicio dejó de resolver internamente el repositorio *group-scoped* mediante `get_repositories()`. En su lugar, acepta explícitamente un `RecipeRepositoryPort` inyectado desde el exterior.

El resultado es una reducción material del volumen y de la amplitud funcional del punto crítico principal.

G.4.6. Ajustes de compatibilidad

Al dejar de construir internamente el repositorio *group-scoped* en `RecipeService`, algunos puntos externos que seguían creando el servicio necesitaron pasar a inyectar ese repositorio de forma explícita. Por ello se ajustaron:

- `mealie/routes/recipe/bulk_actions.py`
- `mealie/routes/households/controller_group_recipe_actions.py`

- `mealie/services/migrations/_migration_base.py`

Estos cambios no amplían el alcance de la intervención, pero preservan el comportamiento previo del sistema tras la simplificación del servicio principal.

G.5. Archivos modificados y tipo de cambio

La Fase 5 introdujo nuevos casos de uso, amplió los contratos del módulo, modificó el ensamblaje de dependencias y ajustó varios puntos de compatibilidad.

G.5.1. Nuevos archivos

- `mealie/services/recipe/application/create_recipe_from_zip.py`: caso de uso para creación de recetas desde ZIP.
- `mealie/services/recipe/application/create_recipe_from_images.py`: caso de uso para creación de recetas desde imágenes.
- `mealie/services/recipe/application/duplicate_recipe.py`: caso de uso para duplicación de recetas.
- `mealie/services/recipe/application/update_recipe_image.py`: caso de uso para actualización de imagen.
- `mealie/services/recipe/application/delete_recipe_image.py`: caso de uso para borrado de imagen.
- `tests/unit_tests/services_tests/recipe_tests/test_recipe_phase_5_use_cases.py`: pruebas unitarias focalizadas sobre los casos de uso extraídos.

G.5.2. Archivos modificados

- `mealie/services/recipe/ports.py`: incorporación de nuevos puertos para lectura de archivos, normalización, staging de imágenes, limpieza de datos, copia de assets y renderizado.
- `mealie/services/recipe/adapters.py`: incorporación de adaptadores para los nuevos puertos de salida.
- `mealie/services/recipe/application/__init__.py`: exposición de los nuevos casos de uso de aplicación.
- `mealie/services/recipe/recipe_service.py`: eliminación de métodos extraídos e inyección explícita de repositorio.
- `mealie/routes/recipe/_base.py`: centralización de la composición de repositorios *group-scoped* y ensamblaje de nuevos casos de uso.
- `mealie/routes/recipe/recipe_crud_routes.py`: delegación de flujos seleccionados hacia casos de uso explícitos.
- `mealie/routes/recipe/exports.py`: adaptación al nuevo puerto de renderizado.
- `mealie/routes/recipe/bulk_actions.py`: ajuste de compatibilidad para inyectar repositorio explícito.

- `mealie/routes/households/controller_group_recipe_actions.py`: ajuste de compatibilidad con la nueva composición de `RecipeService`.
- `mealie/services/migrations/_migration_base.py`: ajuste de compatibilidad para construcción explícita del servicio.
- `support-files/recipe_module_metrics.py`: ampliación controlada del perímetro instrumentado para reconocer los nuevos casos de uso.

G.6. Diagrama antes y después

La Figura 5 resume el cambio principal introducido por la Fase 5. Antes de la intervención, las operaciones seleccionadas estaban concentradas en `RecipeService`. Después, el controlador invoca casos de uso explícitos y el servicio conserva un conjunto más reducido de responsabilidades.

G.7. Resultados específicos de la Fase 5

Los valores de estado inicial a Fase 5 se apoyan en las comparaciones reproducibles conservadas mediante las etiquetas `tfm-baseline` y `tfm-phase-1` a `tfm-phase-5`. La medición de esta fase se realiza, por tanto, sobre el estado exacto con el que se cerró la intervención.

G.7.1. Acoplamiento

ID	Métrica	Base	Fase 1	Fase 2	Fase 3	Fase 4	Fase 5	Observación
C1	Referencias de infraestructura en la orquestación	9	1	1	1	1	1	Se consolida la mejora previa sin regresión.
C2	Dependencias inyectables explícitas	0	3	7	7	9	26	El aumento refleja la aparición de cinco casos de uso adicionales basados en puertos.
C3	Referencias HTTP en la persistencia de recetas	2	2	2	0	0	0	Se mantiene la limpieza conseguida en la Fase 3.

ID	Métrica	Base	Fase 1	Fase 2	Fase 3	Fase 4	Fase 5	Observación
C4	Invocaciones a <code>get_repositories()</code> en el módulo	6	6	6	6	6	2	La fase logra una reducción real, aunque no total, del <i>service locator</i> en el módulo observado.

G.7.2. Complejidad y punto crítico principal

ID	Elemento	Fase 4	Fase 5	Variación	Observación
X2	Métodos de <code>RecipeService</code>	24	18	-6	La reducción coincide con la extracción de cinco flujos y del renderizado de plantillas.
X3	Extensión estructural de <code>RecipeService</code>	442	307	-135	El punto crítico principal reduce de forma apreciable su amplitud.
T2	Flujos estrictamente aislables	2	7	+5	Aparecen nuevos casos de uso pequeños, especializados y testeables de forma aislada.

G.7.3. Tamaño

ID	Elemento	Base	Fase 1	Fase 2	Fase 3	Fase 4	Fase 5
S1	<code>recipe_service.py</code>	627 LOC	646 LOC	649 LOC	649 LOC	598 LOC	452 LOC
S1	<code>recipe_crud_routes.py</code>	608 LOC	607 LOC	588 LOC	603 LOC	603 LOC	639 LOC
S1	<code>recipe_bulk_scraper.py</code>	127 LOC	134 LOC	134 LOC	134 LOC	134 LOC	134 LOC

ID	Elemento	Base	Fase 1	Fase 2	Fase 3	Fase 4	Fase 5
S1	_base.py	56 LOC	62 LOC	92 LOC	92 LOC	92 LOC	162 LOC
S1	ports.py	0 LOC	50 LOC	70 LOC	70 LOC	115 LOC	153 LOC
S1	adapters.py	0 LOC	59 LOC	107 LOC	107 LOC	168 LOC	252 LOC
S1	application/create_recipe_from_web.py	0 LOC	0 LOC	44 LOC	44 LOC	44 LOC	44 LOC
S1	domain/permission_policy.py	0 LOC	0 LOC	0 LOC	0 LOC	65 LOC	65 LOC
S1	application/create_recipe_from_zip.py	0 LOC	0 LOC	0 LOC	0 LOC	0 LOC	46 LOC
S1	application/create_recipe_from_images.py	0 LOC	0 LOC	0 LOC	0 LOC	0 LOC	54 LOC
S1	application/duplicate_recipe.py	0 LOC	0 LOC	0 LOC	0 LOC	0 LOC	82 LOC
S1	application/update_recipe_image.py	0 LOC	0 LOC	0 LOC	0 LOC	0 LOC	41 LOC
S1	application/delete_recipe_image.py	0 LOC	0 LOC	0 LOC	0 LOC	0 LOC	39 LOC
S2	Tamaño total del conjunto intervenido	1418 LOC	1558 LOC	1684 LOC	1699 LOC	1819 LOC	2163 LOC

El tamaño total vuelve a crecer, pero el crecimiento se concentra en unidades explícitas y pequeñas. El dato más relevante para esta fase es que `recipe_service.py` cae 146 líneas respecto a la Fase 4 y 175 respecto al estado inicial.

G.7.4. Testabilidad

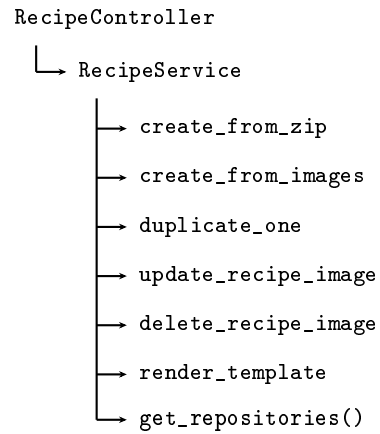
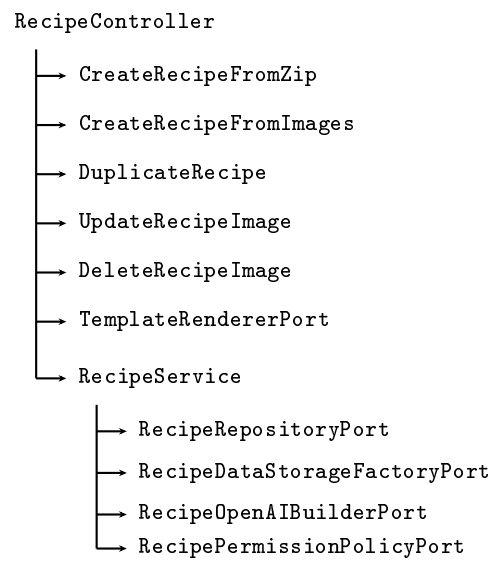
Antes de la Fase 5**Después de la Fase 5**

Figura 5. Cambio de responsabilidad sobre RecipeService durante la Fase 5.

ID	Métrica	Base	Fase 1	Fase 2	Fase 3	Fase 4	Fase 5	Observación
T1	Dependencias sustituibles por contratos explícitos	0	3	7	7	9	26	El módulo gana una red más amplia de colaboración por puertos.
T2	Flujos unitariamente evaluables sin base de datos ni filesystem con criterio estricto	0	0	1	1	2	7	La mejora más importante de la fase: aparecen cinco nuevos flujos estrictamente aislables.
T3	Necesidad de reemplazos dinámicos de funciones globales	2	0	0	0	0	0	No hay regresión respecto a la mejora obtenida en fases anteriores.

G.8. Interpretación arquitectónica de la fase

La Fase 5 produce la mejora estructural más visible de la intervención. **RecipeService** reduce métodos, reduce extensión estructural y pierde varias responsabilidades de alto peso arquitectónico. Esto confirma que la fase no añade únicamente una capa de abstracción, sino que retira comportamiento real del servicio principal.

La mejora en testabilidad también es significativa. T2 pasa de 2 a 7, lo que implica que ya no existen sólo una extracción puntual, como **CreateRecipeFromWeb**, y una política de permisos, sino un pequeño conjunto de casos de uso claramente aislables. Esta mejora refuerza la relación entre descomposición arquitectónica y capacidad de validación unitaria.

La reducción de C4 de 6 a 2 muestra que la fase también disminuye parte del uso de *service locator* en el perímetro observado. La mejora no es total, pero sí material: el ensamblaje de los repositorios *group-scoped* pasa a estar concentrado en el controlador base y deja de estar disperso en varios flujos internos del servicio.

El aumento de S2 es considerable, pero responde a una estrategia deliberada: sustituir concentración implícita por componentes explícitos, pequeños y testeables. En otras palabras, se asume más superficie arquitectónica para obtener menor ambigüedad funcional, menor concentración en el servicio principal y mejor validación aislada.

La fase mantiene límites claros. Persisten usos residuales de `get_repositories()`, **RecipeController** sigue siendo un punto de concentración relevante

del lado HTTP y `OpenAIRecipeService` continúa existiendo como componente concreto dentro del sistema. Por tanto, la Fase 5 no equivale a una conversión completa del módulo a una arquitectura hexagonal homogénea, sino a una reducción material y verificable del punto crítico principal.

G.9. Riesgos y límites que permanecen

La Fase 5 no elimina todos los riesgos del módulo. Tras la intervención permanecen abiertos varios aspectos relevantes:

- siguen existiendo dos invocaciones directas a `get_repositories()` dentro del módulo observado;
- `RecipeController` permanece como punto crítico relevante del lado de entrada HTTP;
- `OpenAIRecipeService` continúa siendo una dependencia concreta dentro del sistema;
- la mejora de cohesión no implica todavía una conversión completa del módulo a una arquitectura hexagonal homogénea.

Estos límites no invalidan la fase, pero acotan el alcance de lo conseguido. La intervención reduce de forma material el punto crítico principal y amplía la testabilidad aislada, aunque no cierra todas las oportunidades de mejora arquitectónica del módulo.

G.10. Evidencia asociada

La validación detallada de esta fase se conserva en el Anexo K. Ese anexo documenta la evidencia ejecutable, las incidencias de validación y el cierre metodológico focalizado en los flujos extraídos.

En relación con el presente anexo, la evidencia asociada permite conectar la intervención técnica con su verificación: la extracción de los casos de uso no sólo redistribuyó responsabilidades, sino que quedó respaldada por pruebas unitarias focalizadas, validación funcional mínima y comparación reproducible de métricas.

H. Validación detallada de la Fase 2

H.1. Propósito y relación con la memoria

Este anexo reúne la evidencia detallada que respalda la validación de la Fase 2 resumida en el Capítulo 4. Mientras que el Anexo D documenta la intervención técnica asociada a la extracción de `CreateRecipeFromWeb`, este anexo conserva la evidencia ejecutable utilizada para validar dicha intervención: comandos, resultados, métricas reproducibles, incidencias detectadas y amenazas específicas a la validez.

Su función es mantener fuera del cuerpo principal de la memoria el detalle operativo de la validación, sin perder trazabilidad. De este modo, el Capítulo 4 puede centrarse en la interpretación de los resultados, mientras que este anexo documenta cómo se comprobaron la testabilidad aislada del nuevo caso de uso, la preservación del comportamiento observable en los endpoints afectados y la evolución de las métricas estructurales correspondientes.

H.2. Objetivo de la validación

La validación de la Fase 2 persigue comprobar tres propiedades del flujo `CreateRecipeFromWeb` una vez extraído a la capa de aplicación:

- La intervención genera una mejora arquitectónica medible.
- El flujo resultante gana testabilidad real, más allá del desacoplamiento potencial.
- La funcionalidad observable del sistema se preserva en los endpoints HTTP afectados.

El alcance de la validación se limita al flujo intervenido por la Fase 2. No pretende cerrar todo el módulo de recetas ni sustituir una regresión funcional completa de Mealie.

H.3. Artefactos de evidencia

La evidencia de la Fase 2 se apoya en tres artefactos complementarios: pruebas unitarias, validación funcional mínima y comparación cuantitativa reproducible.

H.3.1. Evidencia unitaria

La evidencia unitaria se apoya en el archivo `tests/unit_tests/services_tests/recipe_tests/test_create_recipe_from_web.py`, que contiene cinco pruebas asíncronas sobre el caso de uso `CreateRecipeFromWeb` con dobles de prueba sobre los cuatro puertos del flujo: scraping, enriquecimiento, creación y publicación de evento. Su objetivo es respaldar el incremento de T2 con criterio estricto y mostrar que la explicitud estructural releída por T1 se traduce en un flujo realmente aislable, sin base de datos ni filesystem real.

H.3.2. Evidencia funcional mínima de integración

La evidencia funcional se apoya en el archivo `tests/integration_tests/user_recipe_tests/test_recipe_create_from_web_phase2.py`, que contiene dos pruebas HTTP de integración sobre `POST /api/recipes/create/url` y `POST /api/recipes/create/html-or-json`. Las pruebas utilizan una sustitución controlada del scraper y capturan el `dispatch` del event bus. Su objetivo es comprobar la preservación del comportamiento observable del flujo afectado por la refactorización.

H.3.3. Evidencia cuantitativa reproducible

La evidencia cuantitativa se apoya en el script `support-files/recipe_module_metrics.py`, que permite el cálculo reproducible de métricas C, X, S y T sobre etiquetas Git, *working tree* y snapshots de directorio. El script genera una comparación automática entre el estado inicial y el cierre de la fase, incluyendo detalle estructural de funciones, clases, LOC e inyección de dependencias. Su objetivo es sustentar con datos reproducibles la mejora arquitectónica atribuida a la Fase 2.

H.4. Ejecución reproducible

La validación se ejecutó desde la raíz del repositorio mediante los siguientes comandos:

```
python -m pytest tests/unit_tests/services_tests/recipe_tests/test_create_recipe_from_web.py
python -m pytest tests/integration_tests/user_recipe_tests/test_recipe_create_from_web_phase2.py
python support-files/recipe_module_metrics.py --source baseline=git:tfm-baseline --source current=git:tfm-phase-2 --pretty
```

El entorno de ejecución utilizado fue el entorno virtual local del workspace, `.venv`, con Python 3.12.10. La comparación de métricas se realizó contra las etiquetas `tfm-baseline` y `tfm-phase-2`, lo que permite reproducir el cierre de la fase sin depender del estado actual del *working tree*.

H.5. Resultados obtenidos

H.5.1. Resultado de la batería unitaria

La ejecución de `test_create_recipe_from_web.py` produjo 5 `passed`, junto con una advertencia no bloqueante de Alembic ajena a la intervención arquitectónica.

Este resultado confirma que:

- El caso de uso puede probarse de forma aislada.
- Las dependencias externas del flujo son sustituibles por dobles de prueba.
- Existe evidencia directa de mejora en testabilidad, no sólo evidencia estructural.

H.5.2. Resultado de la batería funcional mínima

La ejecución de `test_recipe_create_from_web_phase2.py` produjo 2 `passed`, junto con dos advertencias no bloqueantes de Alembic ajenas al cambio realizado. Cada una de las dos pruebas verificó simultáneamente:

- La creación exitosa de la receta a través del endpoint HTTP.

- La persistencia del enriquecimiento de tags y categorías derivado de `Scrape dExtras`.
- La publicación de un evento `recipe_created` con el `slug` de la receta y operación `create`.

Este resultado respalda que el comportamiento externo del flujo `CreateRecipeFromWeb` se preserve en los dos puntos de entrada afectados por la Fase 2.

H.5.3. Resultado de las métricas reproducibles

La comparación entre `tfm-baseline` y el snapshot etiquetado de Fase 2 produjo los siguientes valores principales:

Métrica	Estado inicial	Cierre Fase 2	Variación
C1	9	1	-8
C2	0	7	+7
C3	2	2	0
C4	6	6	0
S2	1418	1684	+266
T1	0	7	+7
T2	0	1	+1
T3	2	0	-2

Los resultados estructurales más destacados son:

- `_create_recipe_from_web` desaparece como punto crítico de controlador.
- `CreateRecipeFromWeb.execute` presenta complejidad estructural 1.
- `_execute_create_recipe_from_web` presenta complejidad estructural 2.
- `RecipeController` reduce su extensión estructural respecto al estado inicial.

La lectura de estos resultados es coherente con el objetivo de la fase: la mejora es fuerte en acoplamiento localizado y testabilidad, mientras que las métricas globales C3 y C4 permanecen fuera del alcance específico de esta validación. El aumento de tamaño es consistente con una estrategia incremental basada en extracción de responsabilidades, no en reescritura.

H.6. Incidencia descubierta durante la validación

Durante la ejecución de la batería funcional mínima se detectó una recursión de inicialización en `RecipeServiceBase`: el colaborador por defecto `openai_recipe_builder` intentaba instanciar `OpenAIRecipeService` incluso cuando la instancia actual ya pertenecía a esa clase.

La corrección aplicada fue mínima:

- Cuando el objeto actual es `OpenAIRecipeService`, el servicio reutiliza `self` como implementación del puerto.

- En el resto de casos mantiene la construcción por defecto previa.

Esta incidencia no modifica la hipótesis arquitectónica de la Fase 2, pero sí forma parte del cierre de la fase porque impedía validar el flujo HTTP real del sistema. Su detección muestra además la utilidad de validar tempranamente sobre el sistema integrado y no sólo sobre unidades aisladas.

H.7. Amenazas a la validez

La validez de la Fase 2 queda condicionada por los siguientes factores:

- La integración funcional se validó de forma intencionalmente mínima y no constituye una regresión exhaustiva del módulo.
- El scraper se sustituyó por un *stub* controlado, por lo que la validación se centra en la conservación del ensamblado del flujo y no en la robustez de todos los scrapers reales.
- Las métricas C, X, S y T son operacionales y reproducibles, pero no pretenden capturar toda la mantenibilidad del sistema en sentido absoluto.
- El estado de cierre incluye una corrección menor detectada durante la validación, por lo que el cierre final de la Fase 2 no coincide exactamente con el primer snapshot posterior a la extracción del caso de uso.

Estas amenazas son aceptables dentro del alcance de validación de la fase, siempre que se expliciten al interpretar los resultados.

I. Validación detallada de la Fase 3

I.1. Propósito y relación con la memoria

Este anexo reúne la evidencia detallada que respalda la validación de la Fase 3 resumida en el Capítulo 4. Mientras que el Anexo E documenta la intervención técnica asociada a la separación entre HTTP y persistencia, este anexo conserva la evidencia ejecutable utilizada para validar dicha intervención: pruebas focalizadas, comandos reproducibles, resultados obtenidos y amenazas específicas a la validez.

Su función es mantener fuera del cuerpo principal de la memoria el detalle operativo de la validación, sin perder trazabilidad. De este modo, el Capítulo 4 puede centrarse en la interpretación de los resultados, mientras que este anexo documenta cómo se comprobó la eliminación de dependencias HTTP en la persistencia específica del módulo y la preservación del contrato externo de los endpoints afectados.

I.2. Objetivo de la validación

La validación de la Fase 3 persigue comprobar dos propiedades concretas del módulo de recetas tras la intervención sobre persistencia:

- `RepositoryRecipes` deja de depender de conceptos HTTP para expresar errores de consulta.
- El comportamiento observable del sistema se preserva en los endpoints que consumen esa persistencia, manteniendo respuestas HTTP 400 cuando la consulta es inválida.

El alcance de la validación se limita a la intervención realizada en la Fase 3. No pretende demostrar todavía mejoras sobre permisos, *service locator* o descomposición completa de `RecipeService`.

I.3. Artefactos de evidencia

La evidencia de la Fase 3 se apoya en tres artefactos complementarios: pruebas unitarias de persistencia, validación funcional mínima de integración HTTP y comparación cuantitativa reproducible.

I.3.1. Evidencia unitaria de persistencia

La evidencia unitaria se apoya en el archivo `tests/unit_tests/repository_tests/test_recipe_repository_phase3.py`, que contiene tres pruebas focalizadas sobre `RepositoryRecipes`. Las pruebas verifican que se lanzan excepciones de aplicación ante `query_filter` inválido y `order_by` inválido, y comprueban explícitamente que dichas excepciones no son `HTTPException`. Su objetivo es confirmar que la persistencia expresa errores de consulta mediante excepciones de aplicación, no mediante semántica de transporte HTTP.

I.3.2. Evidencia funcional mínima de integración HTTP

La evidencia funcional se apoya en el archivo `tests/integration_tests/user_recipe_tests/test_recipe_phase3_http_mapping.py`, que contiene dos pruebas HTTP focalizadas sobre los endpoints `GET /api/recipes` y `GET /api/recipes/suggestions`. Las pruebas utilizan parámetros inválidos para comprobar que el contrato externo mantiene la respuesta HTTP 400. Su objetivo es verificar que el traslado del mapeo de errores a la capa de entrada no altera el contrato externo del API.

I.3.3. Evidencia cuantitativa reproducible

La evidencia cuantitativa se apoya en el script `support-files/recipe_module_metrics.py`, que permite una comparación reproducible entre el estado inicial y el cierre de la Fase 3 sobre el perímetro instrumentado del módulo. Su objetivo es verificar cuantitativamente que la métrica C3, definida como dependencias HTTP fuera de la capa de entrada, se reduce hasta cero en el módulo de recetas.

I.4. Ejecución reproducible

La validación se ejecutó desde la raíz del repositorio mediante los siguientes comandos:

```
python -m pytest tests/unit_tests/repository_tests/test_recipe_repository_phase3.py
python -m pytest tests/integration_tests/user_recipe_tests/test_recipe_phase3_http_mapping.py
python support-files/recipe_module_metrics.py --source baseline=git:tfm-baseline --source current=git:tfm-phase-3 --pretty
```

El entorno de ejecución utilizado fue el entorno virtual local del workspace, `.venv`, con Python 3.12.10. La comparación de métricas se realizó contra las etiquetas `tfm-baseline` y `tfm-phase-3`, lo que permite reproducir el cierre de la fase sin depender del estado actual del *working tree*.

I.5. Resultados obtenidos

I.5.1. Resultado de la batería unitaria de persistencia

La ejecución de `test_recipe_repository_phase3.py` produjo 3 `passed`, junto con dos advertencias no bloqueantes de Alembic ajenas a la intervención arquitectónica.

Las tres pruebas confirman respectivamente:

- `page_all` lanza `QueryFilterValidationError` cuando el `query_filter` es inválido.
- `page_all` lanza `OrderByValidationError` cuando `order_by` es inválido.
- `find_suggested_recipes` lanza `QueryFilterValidationError` ante filtros inválidos.

Este resultado permite sostener que:

- La capa de persistencia ya no depende de `HTTPException` para expresar estos fallos.
- El repositorio comunica errores de consulta mediante excepciones de aplicación.
- Existe evidencia directa de que el cambio afecta al comportamiento interno esperado del componente intervenido.

I.5.2. Resultado de la batería funcional mínima HTTP

La ejecución de `test_recipe_phase3_http_mapping.py` produjo 2 `passed`, junto con dos advertencias no bloqueantes de Alembic ajenas al cambio realizado.

Las pruebas verifican que:

- `GET /api/recipes` mantiene respuesta 400 cuando `queryFilter` es inválido.
- `GET /api/recipes/suggestions` mantiene respuesta 400 en el mismo escenario.

Este resultado respalda que el desplazamiento del mapeo de errores a la capa de entrada preserva el contrato externo del API y que la Fase 3 no introduce regresión funcional observable en los endpoints afectados.

I.5.3. Resultado de las métricas reproducibles

La comparación entre **tfm-baseline** y el snapshot etiquetado de Fase 3 produjo los siguientes valores principales:

Métrica	Estado inicial	Cierre Fase 3	Variación
C1	9	1	-8
C2	0	7	+7
C3	2	0	-2
C4	6	6	0
S2	1418	1699	+281
T1	0	7	+7
T2	0	1	+1
T3	2	0	-2

La lectura de estos resultados es coherente con el objetivo de la fase:

- La Fase 3 es la primera intervención que impacta directamente en C3.
- La reducción de C3 de 2 a 0 confirma la eliminación de referencias HTTP en la persistencia específica de recetas.
- La mejora no compromete las ganancias previas de desacoplamiento y testabilidad alcanzadas en las Fases 1 y 2.
- El incremento de tamaño es pequeño y corresponde al traslado explícito del mapeo HTTP a la capa correcta.

I.6. Amenazas a la validez

La validez de la Fase 3 queda condicionada por varios factores que conviene explicitar:

- La validación funcional es deliberadamente mínima y sólo cubre dos endpoints afectados por la intervención.
- La métrica C3 se evalúa sobre el módulo de recetas; otros módulos del sistema podrían seguir manteniendo contaminación HTTP en persistencia sin que ello afecte a este resultado.
- La solución aplicada introduce overrides locales en **RepositoryRecipes**, por lo que todavía no demuestra una estrategia global de saneamiento del repositorio genérico compartido.

Estas amenazas no invalidan la conclusión de la fase, pero delimitan con precisión el alcance de lo que puede afirmarse tras la validación: dentro del módulo de recetas, la persistencia específica deja de expresar los errores analizados

mediante semántica HTTP y el contrato externo de los endpoints afectados se preserva.

J. Validación detallada de la Fase 4

J.1. Propósito y relación con la memoria

Este anexo reúne la evidencia detallada que respalda la validación de la Fase 4 resumida en el Capítulo 4. Mientras que el Anexo F documenta la intervención técnica asociada a la extracción de `RecipePermissionPolicy`, este anexo conserva la evidencia ejecutable utilizada para validar dicha intervención: pruebas unitarias, selección funcional mínima, comandos reproducibles, resultados obtenidos, incidencia detectada y amenazas específicas a la validez.

Su función es mantener fuera del cuerpo principal de la memoria el detalle operativo de la validación, sin perder trazabilidad. De este modo, el Capítulo 4 puede centrarse en la interpretación de los resultados, mientras que este anexo documenta cómo se comprobó que la política de permisos extraída era testeable de forma aislada y que los escenarios funcionales más sensibles conservaban su comportamiento observable.

J.2. Objetivo de la validación

La validación de la Fase 4 persigue comprobar tres propiedades concretas tras la extracción de la política de permisos del módulo de recetas:

- La lógica de autorización deja de estar incrustada en `RecipeService` como detalle interno difícil de aislar.
- Las reglas de permisos pueden validarse unitariamente sin base de datos ni filesystem real.
- El comportamiento observable más sensible del sistema se preserva en escenarios representativos de ownership, borrado y bloqueo.

El alcance de la validación se limita a la intervención realizada en la Fase 4. No pretende demostrar todavía la eliminación del *service locator* ni la descomposición completa de `RecipeService`.

J.3. Artefactos de evidencia

La evidencia de la Fase 4 se apoya en tres artefactos complementarios: pruebas unitarias de la política de permisos, validación funcional mínima de integración y comparación cuantitativa reproducible.

J.3.1. Evidencia unitaria de la política de permisos

La evidencia unitaria se apoya en el archivo `tests/unit_tests/services_tests/recipe_tests/test_recipe_permission_policy.py`, que contiene cinco

pruebas focalizadas sobre `RecipePermissionPolicy`. Las pruebas verifican la delegación correcta al puerto de acceso a permisos, la denegación de actualización, la regla de ownership para *lock/unlock*, el bypass de borrado para administradores y la preservación del mensaje plural de denegación en borrado múltiple. Su objetivo es demostrar que la política de permisos es una unidad testeable con criterio estricto y sin infraestructura real.

J.3.2. Evidencia funcional mínima de integración

La evidencia funcional reutiliza el archivo `tests/integration_tests/user_recipe_tests/test_recipe_owner.py`, del que se seleccionaron cuatro escenarios representativos: `test_user_locked_recipe`, `test_other_user_cant_lock_recipe`, `test_other_user_cant_delete_recipe` y `test_admin_can_delete_locked_recipe_owned_by_another_user`. Su objetivo es comprobar que la extracción arquitectónica preserva el comportamiento observable en escenarios de permisos ya existentes en el sistema.

J.3.3. Evidencia cuantitativa reproducible

La evidencia cuantitativa se apoya en el script `support-files/recipe_module_metrics.py`, que permite una comparación reproducible entre el estado inicial y el cierre de la Fase 4 sobre el perímetro instrumentado del módulo. Su objetivo es comprobar cuantitativamente que la extracción de `RecipePermissionPolicy` y queda reflejada en las métricas estructurales, especialmente en C2, T1 y T2.

J.4. Ejecución reproducible

La validación se ejecutó desde la raíz del repositorio mediante los siguientes comandos:

```
python -m pytest tests/unit_tests/services_tests/recipe_tests/test_recipe_permission_policy.py
python -m pytest tests/integration_tests/user_recipe_tests/test_recipe_owner.py -k "locked_recipe or cant_lock_recipe or cant_delete_recipe or admin_can_delete_locked_recipe"
python support-files/recipe_module_metrics.py --source baseline=git:tfm-baseline --source current=git:tfm-phase-4 --pretty
```

El entorno de ejecución utilizado fue el entorno virtual local del workspace, `.venv`, con Python 3.12.10. La comparación de métricas se realizó contra las etiquetas `tfm-baseline` y `tfm-phase-4`, lo que permite reproducir el cierre de la fase sin depender del estado actual del *working tree*.

J.5. Resultados obtenidos

J.5.1. Resultado de la batería unitaria de permisos

La ejecución de `test_recipe_permission_policy.py` produjo 5 `passed`, junto con dos advertencias no bloqueantes ajenas a la intervención arquitectónica.

Las pruebas confirman respectivamente:

- `RecipePermissionPolicy` delega correctamente el cálculo de actualización en el puerto de acceso.
- `assert_can_update` lanza `PermissionDenied` cuando la actualización no está permitida.
- El `lock/unlock` exige ownership de la receta.
- Un administrador conserva capacidad de borrado sin pasar por la consulta de permiso estándar.
- El mensaje de denegación en borrado múltiple mantiene la semántica previa.

Este resultado permite sostener que:

- La política extraída es evaluable de forma aislada.
- La lógica de permisos puede verificarse mediante dobles de prueba sobre un puerto de acceso.
- La mejora en T2 no es sólo estructural, sino que queda respaldada por evidencia ejecutable.

J.5.2. Resultado de la batería funcional mínima de integración

La ejecución focalizada sobre `test_recipe_owner.py` produjo 4 `passed` y 8 `deselected`, junto con una advertencia no bloqueante ajena al cambio realizado.

Los escenarios validados confirman que se preservan comportamientos relevantes del módulo:

- Una receta bloqueada sigue restringiendo la edición según las reglas existentes.
- Un usuario no propietario no puede bloquear una receta ajena.
- Un usuario sin permiso no puede borrar una receta protegida.
- Un administrador mantiene capacidad para borrar una receta bloqueada.

Este resultado respalda que la extracción de la política no altera el contrato observable de permisos en los casos críticos seleccionados y que la refactorización mantiene compatibilidad funcional en el eje de autorización intervenido.

J.5.3. Resultado de las métricas reproducibles

La comparación entre `tfm-baseline` y el snapshot etiquetado de Fase 4 produjo los siguientes valores principales:

Métrica	Estado inicial	Cierre Fase 4	Variación
C1	9	1	-8
C2	0	9	+9
C3	2	0	-2
C4	6	6	0
S2	1418	1819	+401
T1	0	9	+9
T2	0	2	+2
T3	2	0	-2

Resultados estructurales destacados del estado de cierre:

- `RecipePermissionPolicy` aparece como nueva unidad explícita con siete métodos y 57 líneas estructurales.
- `RecipeService` queda en 24 métodos y 442 líneas estructurales.
- `recipe_service.py` reduce su tamaño total a 598 LOC frente a 649 LOC en la Fase 3.
- `T2_details` reconoce dos flujos estrictamente testeables: `CreateRecipeFromWeb` y `RecipePermissionPolicy`.

La lectura de estos resultados es coherente con el objetivo de la fase:

- La Fase 4 mejora de forma directa C2 y, por la misma base estructural, T1.
- La evidencia fuerte adicional de testabilidad aparece en T2, que pasa de 1 a 2 respecto a la Fase 3.
- La mejora de testabilidad es acumulativa y ahora alcanza dos flujos aislables con criterio fuerte.
- No hay avance en C4, por lo que la fase no puede presentarse como solución al *service locator* ni como descomposición completa del servicio principal.

J.6. Incidencia descubierta durante la validación

Durante la primera ejecución de la validación funcional apareció una incidencia de compatibilidad no relacionada con el comportamiento de permisos: un test existente importaba `RecipeDataService` desde `mealie.services.recipe.recipe_service`, y esa importación había desaparecido durante la limpieza inicial del archivo.

La corrección aplicada fue mínima:

- Se restauró la importación de `RecipeDataService` en `recipe_service.py` para preservar compatibilidad con el ecosistema actual de pruebas.

Esta incidencia no invalida la hipótesis de la Fase 4 ni altera el diseño de la política de permisos. No obstante, forma parte del cierre de la fase porque afectaba a la capacidad de ejecutar la evidencia funcional existente.

J.7. Amenazas a la validez

La validez de la Fase 4 queda condicionada por los siguientes factores:

- La validación funcional es deliberadamente mínima y se centra en cuatro escenarios representativos, no en una regresión exhaustiva de todos los permisos del módulo.
- La consulta SQL de permisos sigue existiendo, aunque ahora encapsulada en un adaptador; por tanto, la fase no elimina todavía toda la complejidad de persistencia asociada a la autorización.
- La métrica T2 mejora porque el instrumento reconoce la nueva política explícita como flujo aislable. Este ajuste del instrumento está justificado metodológicamente, pero debe declararse para mantener transparencia metodológica.
- El uso de `get_repositories()` permanece intacto, de modo que la mejora de cohesión no implica todavía una reducción del *service locator*.

Estas amenazas no invalidan la conclusión de la fase, pero delimitan con precisión qué puede afirmarse tras la validación: la política de permisos queda separada como unidad explícita y testeable, mientras que la eliminación del *service locator* y la descomposición material del servicio principal permanecen fuera del alcance de esta fase.

K. Validación detallada de la Fase 5

K.1. Propósito y relación con la memoria

Este anexo reúne la evidencia detallada que respalda la validación de la Fase 5 resumida en el Capítulo 4. Mientras que el Anexo G documenta la intervención técnica asociada a la extracción de varios casos de uso desde *RecipeService*, este anexo conserva la evidencia ejecutable utilizada para validar dicha intervención: pruebas unitarias, selección funcional mínima, comandos reproducibles, resultados obtenidos, incidencias de validación y amenazas específicas a la validez.

Su función es mantener fuera del cuerpo principal de la memoria el detalle operativo de la validación, sin perder trazabilidad. De este modo, el Capítulo 4 puede centrarse en la interpretación de los resultados, mientras que este anexo documenta cómo se comprobó que los flujos extraídos eran testeables de forma aislada, que el comportamiento observable se preservaba en escenarios representativos y que la mejora sobre C4, X2, X3 y T2 quedaba respaldada por una ejecución reproducible del instrumento de métricas.

K.2. Objetivo de la validación

La validación de la Fase 5 persigue comprobar cuatro propiedades concretas tras la descomposición adicional de *RecipeService*:

- Los flujos seleccionados salen realmente del punto crítico principal y pasan a existir como casos de uso explícitos.
- Los nuevos flujos pueden validarse unitariamente mediante puertos, sin base de datos ni filesystem real.
- El comportamiento observable se preserva en operaciones representativas de importación, duplicación y gestión de imagen.
- La mejora sobre C4, X2, X3 y T2 queda respaldada por una ejecución reproducible del instrumento de métricas.

El alcance de la validación se limita a los flujos intervenidos por la Fase 5. No pretende ejecutar una regresión funcional exhaustiva de todo el módulo de recetas ni demostrar una eliminación completa de todos los puntos críticos restantes.

K.3. Artefactos de evidencia

La evidencia de la Fase 5 se apoya en tres artefactos complementarios: pruebas unitarias de los nuevos casos de uso, validación funcional mínima de integración y comparación cuantitativa reproducible.

K.3.1. Evidencia unitaria de los nuevos casos de uso

La evidencia unitaria se apoya en el archivo `tests/unit_tests/services_tests/recipe_tests/test_recipe_phase5_use_cases.py`, que contiene seis pruebas focalizadas. Estas cubren los flujos principales de `CreateRecipeFromZip` y `CreateRecipeFromImages`, incluyen la validación de error cuando no se consigue staging de imágenes y verifican los casos de uso `DuplicateRecipe`, `UpdateRecipeImage` y `DeleteRecipeImage`. Su objetivo es demostrar que los flujos introducidos en la Fase 5 son unidades aislables y ejecutables mediante dobles de prueba sobre puertos.

K.3.2. Evidencia funcional mínima de integración

La evidencia funcional reutiliza varios archivos de pruebas de integración existentes. Se seleccionaron escenarios representativos de creación, duplicación y gestión de imagen a partir de `test_recipe_crud.py`, `test_recipe_create_from_image.py`, `test_recipe_owner.py` y `test_recipe_cross_household.py`, lo que da lugar a seis ejecuciones efectivas.

Su objetivo es comprobar que la refactorización arquitectónica conserva el comportamiento observable en los flujos más sensibles extraídos de `RecipeService`.

K.3.3. Evidencia cuantitativa reproducible

La evidencia cuantitativa se apoya en el script `support-files/recipe_module_metrics.py`, que permite comparar de forma reproducible el estado inicial y

el cierre de la Fase 5. Esta comparación evalúa la reducción parcial de C4, la evolución de `RecipeService` mediante X2 y X3, y el incremento de T2 asociado a los nuevos flujos aislables.

K.4. Ejecución reproducible

La validación se ejecutó desde la raíz del repositorio mediante los siguientes comandos:

```
python -m pytest tests/unit_tests/services_tests/recipe_tests/test_recipe_phase5_use_cases.py
python -m pytest tests/integration_tests/user_recipe_tests/test_recipe_crud.py::test_create_recipe_from_zip tests/integration_tests/user_recipe_tests/test_recipe_create_from_image.py::test_openai_create_recipe_from_image tests/integration_tests/user_recipe_tests/test_recipe_owner.py::test_user_can_update_recipe_image tests/integration_tests/user_recipe_tests/test_recipe_owner.py::test_user_can_delete_recipe_image tests/integration_tests/user_recipe_tests/test_recipe_cross_household.py::test_duplicate_recipe_changes_household
python support-files/recipe_module_metrics.py --source baseline=git:tfm-baseline --source current=git:tfm-phase-5 --pretty
```

El entorno de ejecución utilizado fue el entorno virtual local del workspace, `.venv`, con Python 3.12.10. La comparación de métricas se realizó contra las etiquetas `tfm-baseline` y `tfm-phase-5`, lo que permite reproducir el cierre de la fase sin depender del estado actual del *working tree*.

K.5. Resultados obtenidos

K.5.1. Resultado de la batería unitaria de Fase 5

La ejecución de `test_recipe_phase5_use_cases.py` produjo 6 `passed`, junto con dos advertencias no bloqueantes ajenas a la intervención arquitectónica.

Las pruebas confirman que:

- La importación desde ZIP queda orquestada mediante puertos.
- La importación desde imágenes aísla correctamente staging, OpenAI, limpieza y persistencia.
- El caso de error sin imágenes en staging falla de forma explícita.
- La duplicación crea nuevas entidades sin depender de infraestructura real.
- La actualización de imagen valida permisos y coordina almacenamiento.
- El borrado de imagen mantiene la semántica esperada.

Este resultado muestra que los nuevos casos de uso son ejecutables de forma aislada y que su aumento en T2 está respaldado por evidencia ejecutable.

K.5.2. Resultado de la batería funcional mínima de integración

La ejecución de la batería funcional produjo 6 `passed`, junto con dos advertencias no bloqueantes.

Los escenarios validados confirman:

- La creación desde ZIP sigue funcionando correctamente.
- La creación desde imágenes mantiene su comportamiento asociado.
- La actualización y borrado de imagen funcionan para usuarios autorizados.
- La duplicación sigue produciendo una nueva receta en el hogar destino.

Estos resultados muestran que la extracción de los flujos no introduce regresiones observables en los escenarios representativos seleccionados.

K.5.3. Resultado de las métricas reproducibles

Métrica	Estado inicial	Cierre Fase 5	Variación
C1	9	1	-8
C2	0	26	+26
C3	2	0	-2
C4	6	2	-4
S2	1418	2163	+745
T1	0	26	+26
T2	0	7	+7
T3	2	0	-2

Los resultados estructurales muestran una reducción material de `RecipeService` y un incremento significativo de flujos aislables. En concreto, el servicio reduce métodos y extensión estructural respecto a la Fase 4, mientras que T2 reconoce siete flujos estrictamente testeables.

En conjunto, estos datos confirman que la Fase 5 reduce el punto crítico principal y aumenta la testabilidad, aunque el tamaño total del sistema crece como consecuencia de la redistribución explícita de responsabilidades.

K.6. Incidencias y decisiones de validación

Durante la validación se detectó que `test_duplicate` dependía del estado generado por otras pruebas y fallaba antes de ejecutar el flujo relevante. En lugar de considerarlo una regresión, se decidió sustituirlo por un caso funcional equivalente y autocontenido: `test_duplicate_recipe_changes_household`.

Esta decisión mejora la calidad de la evidencia, ya que evita dependencias del orden de ejecución de tests y mantiene el objetivo de validar correctamente el flujo de duplicación.

K.7. Amenazas a la validez

La validez de la Fase 5 queda condicionada por los siguientes factores:

- La validación funcional es intencionadamente limitada y no cubre todo el módulo.
- La reducción de C4 es parcial y no elimina completamente el *service locator*.
- `RecipeController` sigue siendo un punto de concentración en la capa HTTP.
- El aumento de métricas estructurales depende de la incorporación explícita de nuevas unidades al instrumento.

Estas limitaciones no invalidan la fase, pero delimitan su alcance: la Fase 5 mejora de forma material la descomposición del módulo y la testabilidad, sin cerrar completamente todos los problemas arquitectónicos.

Referencias

1. Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12), 1053-1058.
2. Chidamber, S. R., & Kemerer, C. F. (1994). A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6), 476-493.
3. Feathers, M. C. (2004). *Working effectively with legacy code*. Prentice Hall Press.
4. Ford, N., Parsons, R., & Kua, P. (2017). *Building evolutionary architectures: Support constant change*. O'Reilly Media, Inc.
5. Fowler, M. (2018). *Refactoring: Improving the design of existing code*. Addison-Wesley Professional.
6. International Organization for Standardization. (2023). *Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Product quality model* (ISO/IEC 25010:2023).
7. Martin, R. C. (2017). *Clean architecture: A craftsman's guide to software structure and design*. Prentice Hall Press.
8. McCabe, T. J. (1976). A complexity measure. *IEEE Transactions on Software Engineering*, (4), 308-320.
9. Kazman, R., Abowd, G., Bass, L., & Clements, P. (2002). Scenario-based analysis of software architecture. *IEEE Software*, 13(6), 47-55.
10. Kazman, R., Klein, M., & Clements, P. (2000). *ATAM: Method for architecture evaluation*.
11. Koziolok, H. (2011, June). Sustainability evaluation of software architectures: a systematic review. In *Proceedings of the Joint ACM SIGSOFT Conference-QoSA and ACM SIGSOFT Symposium-ISARCS on Quality of Software Architectures-QoSA and Architecting Critical Systems-ISARCS* (pp. 3-12).
12. Breivold, H. P., Crnkovic, I., & Larsson, M. (2012). A systematic review of software architecture evolution research. *Information and Software Technology*, 54(1), 16-40.
13. Terra, R., Valente, M. T., Czarnecki, K., & Bigonha, R. S. (2012, March). Recommending refactorings to reverse software architecture erosion. In *2012 16th European Conference on Software Maintenance and Reengineering* (pp. 335-340). IEEE.
14. Macia, I., Arcoverde, R., Garcia, A., Chavez, C., & von Staa, A. (2012, March). On the relevance of code anomalies for identifying architecture degradation symptoms. In *2012 16th European Conference on Software Maintenance and Reengineering* (pp. 277-286). IEEE.
15. Bavota, G., Gethers, M., Oliveto, R., Poshyanyk, D., & De Lucia, A. (2014). Improving software modularization via automated analysis of latent topics and dependencies. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23(1), 1-33.
16. Kazman, R., Cai, Y., Mo, R., Feng, Q., Xiao, L., Haziye, S., ... & Shapochka, A. (2015, May). A case study in locating the architectural roots of technical debt. In *2015 IEEE/ACM 37th IEEE international conference on software engineering* (Vol. 2, pp. 179-188). IEEE.
17. Martini, A., Bosch, J., & Chaudron, M. (2015). Investigating architectural technical debt accumulation and refactoring over time: A multiple-case study. *Information and Software Technology*, 67, 237-253.
18. Martini, A., & Bosch, J. (2015, May). The danger of architectural technical debt: Contagious debt and vicious circles. In *2015 12th Working IEEE/IFIP Conference on Software Architecture* (pp. 1-10). IEEE.

19. Samarthayam, G., Suryanarayana, G., & Sharma, T. (2016, September). Refactoring for software architecture smells. In *Proceedings of the 1st International Workshop on Software Refactoring* (pp. 1-4).
20. Martini, A., & Bosch, J. (2016, May). An empirically developed method to aid decisions on architectural technical debt refactoring: AnaConDebt. In *Proceedings of the 38th International Conference on Software Engineering Companion* (pp. 31-40).
21. Ali, N., Baker, S., O’Crowley, R., Herold, S., & Buckley, J. (2018). Architecture consistency: State of the practice, challenges and requirements. *Empirical Software Engineering*, 23(1), 224-258.
22. Martini, A., & Bosch, J. (2017). On the interest of architectural technical debt: Uncovering the contagious debt phenomenon. *Journal of Software: Evolution and Process*, 29(10), e1877.
23. Venters, C. C., Capilla, R., Betz, S., Penzenstadler, B., Crick, T., Crouch, S., ... & Carrillo, C. (2018). Software sustainability: Research and practice from a software architecture viewpoint. *Journal of Systems and Software*, 138, 174-188.
24. Besker, T., Martini, A., & Bosch, J. (2018). Managing architectural technical debt: A unified model and systematic literature review. *Journal of Systems and Software*, 135, 1-16.
25. Verdecchia, R., Malavolta, I., & Lago, P. (2018, May). Architectural technical debt identification: The research landscape. In *Proceedings of the 2018 International Conference on Technical Debt* (pp. 11-20).
26. Perez, B., Correal, D., & Astudillo, H. (2019, May). A proposed model-driven approach to manage architectural technical debt life cycle. In *2019 IEEE/ACM International Conference on Technical Debt (TechDebt)* (pp. 73-77). IEEE.
27. Baabad, A., Zulzalil, H. B., Hassan, S. A., & Baharom, S. B. (2020). Software architecture degradation in open source software: A systematic literature review. *IEEE Access*, 8, 173681-173709.
28. Sharma, T., Singh, P., & Spinellis, D. (2020). An empirical investigation on the relationship between design and architecture smells. *Empirical Software Engineering*, 25(5), 4020-4068.
29. Verdecchia, R., Lago, P., Malavolta, I., & Ozkaya, I. (2020, May). ATDx: Building an architectural technical debt index. In *15th International Conference on Evaluation of Novel Approaches to Software Engineering, ENASE 2020* (pp. 531-539). SciTePress.
30. Bogner, J., Fritzsche, J., Wagner, S., & Zimmermann, A. (2021). Industry practices and challenges for the evolvability assurance of microservices: An interview study and systematic grey literature review. *Empirical Software Engineering*, 26(5), 104.
31. Waseem, M., Liang, P., Shahin, M., Di Salle, A., & Marquez, G. (2021). Design, monitoring, and testing of microservices systems: The practitioners’ perspective. *Journal of Systems and Software*, 182, 111061.
32. Mumtaz, H., Singh, P., & Blincoe, K. (2021). A systematic mapping study on architectural smells detection. *Journal of Systems and Software*, 173, 110885.
33. Cortellessa, V., & Di Pompeo, D. (2021). Analyzing the sensitivity of multi-objective software architecture refactoring to configuration characteristics. *Information and Software Technology*, 135, 106568.
34. Verdecchia, R., Kruchten, P., Lago, P., & Malavolta, I. (2021). Building and evaluating a theory of architectural technical debt in software-intensive systems. *Journal of Systems and Software*, 176, 110925.

35. de Toledo, S. S., Martini, A., & Sjøberg, D. I. (2021). Identifying architectural technical debt, principal, and interest in microservices: A multiple-case study. *Journal of Systems and Software*, 177, 110968.
36. Abi-Antoun, M., & Coelho, W. (2005, November). A case study in incremental architecture-based re-engineering of a legacy application. In *5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05)* (pp. 159-168). IEEE.
37. BenIdris, M., Ammar, H., & Dzielski, D. (2021). The technical debt density over multiple releases and the refactoring story. *International Journal of Software Engineering and Knowledge Engineering*, 31(1), 99-116.
38. Laser, M. S., Le, D. M., Garcia, J., & Medvidovic, N. (2021, May). Architectural archipelagos: Technical debt in long-lived software research platforms. In *2021 IEEE/ACM International Conference on Technical Debt (TechDebt)* (pp. 94-98). IEEE.
39. Heitlager, I., Kuipers, T., & Visser, J. (2007, September). A practical model for measuring maintainability. In *6th International Conference on the Quality of Information and Communications Technology (QUATIC 2007)* (pp. 30-39). IEEE.
40. Muccini, H., Dias, M. S., & Richardson, D. J. (2005). Towards software architecture-based regression testing. *ACM SIGSOFT Software Engineering Notes*, 30(4), 1-7.
41. Li, R., Liang, P., Soliman, M., & Avgeriou, P. (2022). Understanding software architecture erosion: A systematic mapping study. *Journal of Software: Evolution and Process*, 34(3), e2423.
42. Seaman, C., Guo, Y., Zazworka, N., Shull, F., Izurieta, C., Cai, Y., & Vetro, A. (2012, June). Using technical debt data in decision making: Potential decision approaches. In *2012 Third International Workshop on Managing Technical Debt (MTD)* (pp. 45-48). IEEE.
43. Mealie Recipes. (2026). *Mealie* [Computer software]. GitHub. <https://github.com/mealie-recipes/mealie>
44. Mealie documentation project. (2026). *Mealie documentation*. <https://docs.mealie.io/>