# Filer: Una Teoría de Imágenes de Memoria Unificadas

**Un documento fundamental sobre arquitectura, filosofía y la democratización de la creación de software**

---

## Introducción: La Tormenta Perfecta

¿Y si los datos, el esquema, el código y la interfaz de usuario vivieran en el mismo espacio de memoria, descritos por metadatos que se describen a sí mismos? ¿Y si cambiar un esquema adaptara automáticamente los datos existentes, sin necesidad de scripts de migración? ¿Y si las consultas en JavaScript reemplazaran a SQL, y los formularios se materializaran a partir de los metadatos en tiempo de ejecución?

Esta visión no es nueva. Los investigadores la propusieron en la década de 1980 (UNIFILE), y los profesionales construyeron variantes en la década de 2000 (Prevayler). Pero llegaron demasiado pronto. La tecnología no estaba lista. El ecosistema no estaba maduro. Y, de manera crucial, la revolución de la IA no había llegado.

**Tres fuerzas tuvieron que converger para hacer esta visión práctica:**

### 1. Maduración de JavaScript (2015: Proxies de ES6)

- Interceptación transparente sin manipulación de bytecode o hacking en tiempo de ejecución.
- El navegador como plataforma de despliegue universal (más de mil millones de dispositivos).
- Serialización nativa en JSON que refleja la estructura de los objetos.
- Funciones de primera clase que permiten el código como datos.

### 2. Evolución de la Plataforma del Navegador (década de 2010)

- IndexedDB para una persistencia sustancial del lado del cliente.
- Madurez del protocolo `file://` (aplicaciones sin servidores).
- Revolución del rendimiento (compiladores JIT, WebAssembly).
- Capacidades offline-first (Service Workers, API de Caché).
- Ubicuidad multiplataforma (Linux, Windows, macOS, iOS, Android).

### 3. Surgimiento de los LLM (2022+)

- **Construyendo Filer**: El desarrollo asistido por IA aceleró la implementación 6 veces (3 meses frente a 12-18 meses).
- **Usando Filer**: Lenguaje natural → modelos de dominio funcionales (minutos frente a semanas).
- **Reduciendo barreras**: Los expertos en dominios pueden crear aplicaciones sin programación tradicional.

**UNIFILE (1985)** tuvo la visión pero carecía de:

- Metaprogramación en tiempo de ejecución (C/Pascal no podían hacer proxies transparentes).

- Despliegue universal (pre-internet, sin un tiempo de ejecución ubicuo).
- Asistencia de IA (codificación puramente manual).
- Resultado: **Prototipo académico, nunca desplegado.**

**Prevayler (2002)** tenía event sourcing pero carecía de:

- Tiempo de ejecución ubicuo (Java requiere instalación de la JVM).
- Motor de síntesis (los desarrolladores todavía escribían clases manualmente).
- Plataforma de navegador (el despliegue seguía siendo pesado).
- Modelado asistido por LLM (ChatGPT estaba a 20 años de distancia).
- **Problema de tono**: Afirmaciones grandilocuentes ("¡9000 veces más rápido!") alienaron a los escépticos.
- Resultado: **Adopción de nicho, desapareció del mainstream.**

**Filer (2025)** llega en la confluencia:

- Proxies + Navegadores + LLMs = la creación de software se vuelve más accesible.
- No solo una mejor herramienta para desarrolladores, sino una plataforma para expertos en dominios.
- No solo un desarrollo más rápido, sino un enfoque diferente para construir sistemas.

Este documento explica la teoría detrás de Filer: por qué funciona, cómo funciona y qué hace posible. Está escrito tanto para sistemas de IA (para entender la arquitectura rápidamente) como para humanos (para comprender los fundamentos filosóficos y las implicaciones prácticas).

---

# Parte I: El Sufrimiento - Separaciones Artificiales

El desarrollo de software moderno sufre de separaciones forzadas que crean una complejidad accidental. Cada separación introduce capas de traducción, problemas de sincronización y cargas de mantenimiento dual. Estas separaciones parecen inevitables —"así es como funciona el software"— pero son accidentes históricos, no requisitos esenciales.

## Separación 1: Esquema vs. Datos

Las **bases de datos tradicionales** imponen una separación rígida:

- **Esquema** (DDL): `CREATE TABLE emp (empno INT, ename VARCHAR(50), sal DECIMAL)`
- **Datos** (DML): `INSERT INTO emp VALUES (7839, 'KING', 5000)`

**El dolor diario:**

1. **El Infierno de las Migraciones**:

```
-- Migración 001: Añadir columna hire_date
ALTER TABLE emp ADD COLUMN hire_date DATE;

-- Migración 002: Hacerla obligatoria (uy, ¿y las filas existentes?)
UPDATE emp SET hire_date = '2000-01-01' WHERE hire_date IS NULL;
ALTER TABLE emp MODIFY hire_date DATE NOT NULL;
```

Cada cambio requiere escribir scripts de migración, versionarlos, probarlos en diferentes entornos y coordinar el despliegue. Un paso en falso y se producen daños en los datos o fallos en la aplicación.

2. **Deriva de Versiones**:
    - La base de datos de producción está en la versión de esquema v12.
    - Staging en la v11.
    - El entorno local del desarrollador en la v13.
    - Una copia de seguridad antigua de la v8 no se puede restaurar sin ejecutar todas las migraciones intermedias.
3. **Dos Fuentes de Verdad**: El esquema se define en SQL y de nuevo en el código de la aplicación (por ejemplo, en un ORM). Deben mantenerse sincronizados manualmente.
4. **Fricción en la Exportación/Importación**: Los datos sin el esquema son inútiles, y viceversa. Deben versionarse juntos perfectamente.

**Por qué existe esto**: El almacenamiento en disco en la década de 1970 requería diseños fijos.

## Separación 2: Código vs. Datos (El Desajuste de Impedancia del ORM)

El **Mapeo Objeto-Relacional (ORM)** intenta unir modelos incompatibles: grafos de objetos y tablas relacionales.

**El dolor diario:**

1. **Problema de N+1 Consultas**: Un código de apariencia inocente puede desencadenar una consulta a la base de datos por cada objeto en un bucle, causando un rendimiento terrible.
2. **Abstracción con Fugas**: No se pueden tratar los objetos como objetos puros. Hay que pensar en las consultas de la base de datos mientras se escribe código orientado a objetos.
3. **Representación Dual**: La misma entidad se define dos veces (clase en el código + tabla en SQL).
4. **El Método `save()`**: Los cambios en los objetos no persisten hasta que se llama explícitamente a un método `save()`.

**Por qué existe esto**: Dos modelos incompatibles necesitan traducción.

## Separación 3: Presentación vs. Lógica

Los **frameworks modernos** separan la interfaz de usuario del dominio.

**El dolor diario:**

1. **Proliferación de Plantillas**: Cada entidad (Empleado, Departamento) necesita múltiples componentes de UI (formulario, vista de tabla, diálogo de edición).
2. **Sincronización Manual**: Añadir un campo al modelo de dominio requiere editar manualmente más de 4 archivos de UI.
3. **Duplicación de la Validación**: La misma regla de negocio (por ejemplo, el salario no puede ser negativo) se escribe dos veces: una en el cliente (JavaScript) y otra en el servidor (Python/Java).

**Por qué existe esto**: El patrón MVC y la separación de preocupaciones.

## Separación 4: Cliente vs. Servidor

Las **arquitecturas distribuidas** dividen la lógica a través de los límites de la red.

**El dolor diario:**

1. **Cascadas de Red**: Cargar datos relacionados a menudo requiere múltiples viajes de ida y vuelta a la red, lo que añade latencia.
2. **Sincronización de Estado**: Dos fuentes de verdad (cliente + servidor) que se desvían. Las actualizaciones concurrentes pueden perderse.
3. **Fallo sin Conexión**: Sin red, la aplicación es inutilizable.
4. **Versionado de API**: El cliente y el servidor deben desplegarse de forma coordinada para evitar fallos.

**Por qué existe esto**: Los navegadores de la década de 1990 eran "clientes delgados" que requerían que un servidor hiciera todo el trabajo.

## Separación 5: Desarrollo vs. Despliegue

La **infraestructura moderna** separa cómo se desarrolla de cómo se despliega.

**El dolor diario:**

1. **Carga de Infraestructura**: Para desplegar una simple aplicación de tareas, se necesita aprender sobre Docker, Kubernetes, AWS/GCP/Azure, etc.
2. **Deriva de Entornos**: "¡Funciona en mi máquina!" pero falla en staging o producción debido a sutiles diferencias de configuración.
3. **Infierno de Dependencias**: `npm audit` encuentra vulnerabilidades, y arreglarlas rompe la mitad de las dependencias.
4. **Ceremonia de Despliegue**: Un proceso largo y propenso a errores que implica CI/CD, pruebas, construcción y despliegue.

**Por qué existe esto**: La coordinación de múltiples servicios (frontend, backend, base de datos, caché) requiere orquestación.

## La Causa Raíz: Complejidad Accidental

Estas cinco separaciones son **accidentales**, no esenciales. Son artefactos de restricciones tecnológicas que ya no existen. Las hemos olvidado porque se enseñan como "la forma en que funciona el software" y porque industrias enteras se han construido a su alrededor.

**Filer elimina estas separaciones volviendo a un modelo más simple, pero ahora con tecnología que lo hace práctico.**

---

# Parte II: JavaScript - El Lenguaje en una Posición Única

Filer solo es posible en JavaScript. No porque sea el "mejor" lenguaje, sino por su posición única en el ecosistema de software en 2025.

## Características Clave del Lenguaje

### 1. Proxies de ES6 (2015) - Interceptación Transparente

Permiten interceptar el acceso y la mutación de propiedades sin cambiar el código del usuario. Esto es esencial para la persistencia transparente de Filer. El usuario escribe código natural de JavaScript, y el Proxy registra la mutación de forma invisible.

```
// El usuario escribe JavaScript natural
root.cuentas.janet.saldo += 100;

// El Proxy registra automáticamente:
{
  type: 'SET',
  path: ['root', 'cuentas', 'janet', 'saldo'],
  oldValue: 0,
  newValue: 100,
  timestamp: '2025-01-15T10:30:00Z'
}
// No se necesita un `save()` explícito.
```

### 2. Orientación a Objetos Basada en Prototipos - Todo son Objetos

En JavaScript, no hay una dicotomía clase/instancia en tiempo de ejecución; todo es un objeto. Esto permite que los metadatos (definiciones de tipo) y los datos (instancias) se traten de la misma manera, se serialicen con el mismo mecanismo y se almacenen en el mismo registro de eventos.

### 3. Funciones de Primera Clase - Código como Datos

Las funciones son valores que se pueden almacenar, pasar y serializar. Esto permite que los metadatos no solo describan la estructura, sino también el comportamiento (métodos, validadores).

### 4. JSON Nativo - La Estructura Coincide con la Serialización

Los objetos de JavaScript se serializan a JSON sin necesidad de una capa de traducción. La representación en memoria es estructuralmente idéntica a la representación serializada.

### 5. Tipado Dinámico - Metaprogramación en Tiempo de Ejecución

La ausencia de un paso de compilación significa que los tipos se pueden crear y modificar en tiempo de ejecución. Esto es lo que permite que los metadatos se conviertan en sistemas ejecutables inmediatamente.

## Ventajas de la Plataforma - Navegadores en Todas Partes

1. **Despliegue Universal**: Una aplicación Filer es un archivo HTML que se puede abrir en cualquier navegador, en cualquier sistema operativo. Funciona con el protocolo `file://`, sin necesidad de un servidor.
2. **Sin Fricción de Instalación**: Abrir un archivo. Eso es todo.
3. **Capacidad Offline-First**: Los Service Workers y IndexedDB permiten que la aplicación funcione sin conexión.
4. **Modelo de Seguridad Incorporado**: El sandboxing del navegador protege al usuario por defecto.

**En este momento de la historia, JavaScript es el medio en una posición única para esta visión.**

---

# Parte III: Linaje Histórico - Aprendiendo de los Intentos

La visión de Filer no es nueva. Entender por qué los intentos anteriores fracasaron ayuda a explicar por qué Filer podría tener éxito.

## UNIFILE (1986): Visión sin Plataforma

Un proyecto de investigación de Hitachi que proponía un "Sistema de Archivo Universal Personal" basado en un modelo de conceptos y relaciones. Tenían la visión de organizar el conocimiento a medida que se adquiere, pero carecían de la tecnología (metaprogramación en tiempo de ejecución, una plataforma ubicua, asistencia de IA).

## Prevayler (2002): Implementación sin Ubicuidad

Una biblioteca de Java de código abierto que implementaba el patrón de "Sistema Prevalente" (Imagen de Memoria). Utilizaba event sourcing con "Comandos" explícitos. Fracasó en ganar una adopción masiva debido a su dependencia de la JVM, la rigidez de los Comandos explícitos y un tono grandilocuente que alienó a los escépticos.

## Martin Fowler (2011): Documentación sin LLMs

Fowler documentó claramente el patrón como "Imagen de Memoria", pero la documentación por sí sola no cambia los ecosistemas. En 2011, no había una plataforma que lo hiciera accesible ni LLMs para la síntesis de metadatos.

**Lo que es diferente ahora:**

1. **Madurez de la plataforma**: JavaScript + Navegadores.
2. **Abundancia de RAM**.
3. **Revolución de los LLM**.
4. **Un cambio de tono**: Honesto, no grandilocuente.
5. **Centrado en los metadatos**: No solo event sourcing, sino sistemas autodescriptivos.

---

# Parte IV: El Cambio de los LLM - Dos Transformaciones

## Transformación 1: Construyendo Filer - Desarrollo Asistido por IA

La construcción de Filer, con su compleja interacción de Proxies, event sourcing y transacciones, fue acelerada aproximadamente 6 veces gracias a la colaboración con IA (Claude Code). La IA no solo generó código, sino que ayudó en la arquitectura, la generación exhaustiva de pruebas y la refactorización a gran escala, lo que resultó en un sistema de mayor calidad.

### Transformación 2: Usando Filer - Modelado de Dominio Asistido por LLM

Esta es la transformación más grande. El flujo de trabajo tradicional para crear una aplicación simple requiere meses de aprendizaje de múltiples tecnologías. Con Filer y un LLM, el flujo de trabajo se convierte en:

1. **Usuario**: "Quiero hacer un seguimiento de los libros que he leído, con calificaciones y notas".
2. **LLM**: Traduce esto a los metadatos formales `ObjectType` de Filer, restringido por el metamodelo de Filer.
3. **Filer**: **Promulga** instantáneamente estos metadatos en una aplicación en ejecución.
4. **Navigator**: **Sintetiza** una interfaz de usuario para que el usuario interactúe con su nuevo sistema.

El cronograma se reduce de meses a minutos. El aprendizaje se desplaza de la infraestructura al modelado conceptual.

---

## Parte V: Arquitectura - Los Tres Pilares

### Pilar 1: MemImg - Imagen de Memoria con Event Sourcing

**Estado**: ✅ Completo (913 pruebas, 94.74% de cobertura)

En lugar de persistir objetos en una base de datos, MemImg persiste la **secuencia de mutaciones** que crearon esos objetos.

- **Persistencia Transparente**: Las mutaciones de objetos JavaScript simples son interceptadas por Proxies de ES6 y se escriben en un registro de eventos. No se necesitan llamadas a `.save()`.
- **Mutations vs. Commands**: MemImg registra mutaciones de bajo nivel (`SET`, `DELETE`, `ARRAY_PUSH`), no Comandos de alto nivel. Esto hace que el sistema sea mucho más flexible y resistente a la evolución del esquema. Se puede refactorizar la lógica de negocio sin invalidar el registro de eventos.
- **Aislamiento de Transacciones**: Un sistema de "delta tracking" permite un estado de borrador. Los cambios se mantienen en una capa delta y se pueden confirmar (`save()`) o descartar (`discard()`).
- **Almacenamiento Conectable**: Funciona con IndexedDB en el navegador, archivos en Node.js, o cualquier backend personalizado.

### Pilar 2: Navigator - Interfaz Universal para la Exploración

**Estado**: ✅ Funcional (427 pruebas, 100% de aprobación)

Una interfaz de usuario universal para explorar y manipular imágenes de memoria.

- **Vista de Árbol**: Navega visualmente por el grafo de objetos.
- **Panel Inspector**: Examina las propiedades de cualquier objeto seleccionado.
- **REPL**: Ejecuta JavaScript directamente contra los datos en vivo.
- **Interfaz de Múltiples Pestañas**: Trabaja con múltiples imágenes de memoria independientes a la vez.

## Pilar 3: Metadata - La Piedra Angular Faltante

**Estado**: 🎯 Arquitectura clara, implementación pendiente

Esta es la parte transformadora. Los metadatos no solo describen el sistema, **SON el sistema**.

**El Metamodelo (Autodescriptivo)**
El metamodelo de Filer se describe a sí mismo usando sus propias construcciones (`ObjectType`, `PropertyType`, etc.). Esto permite tres manifestaciones de un único principio arquitectónico:

1. **Promulgación (Enaction)**: Los metadatos se convierten en código ejecutable. Un objeto `ObjectType` se convierte en una fábrica que produce instancias que son Proxies con seguimiento de mutaciones y validación.
2. **Síntesis de GUI**: Los mismos metadatos que ejecutan el sistema también describen cómo editarse a sí mismos. Navigator puede renderizar una interfaz de usuario para editar una instancia de `Account` usando la definición `AccountType`, y también puede renderizar una interfaz de usuario para editar la propia definición `AccountType` usando la definición `ObjectTypeMeta`.
3. **Esquema para LLM**: El metamodelo proporciona el esquema que restringe la generación del LLM, asegurando que siempre produzca metadatos válidos y ejecutables.

**El ciclo virtuoso:**

1. El usuario describe el dominio.
2. El LLM genera metadatos.
3. Los metadatos se convierten en un sistema ejecutable.
4. Navigator sintetiza una interfaz de usuario.
5. El usuario refina el dominio a través de la interfaz de usuario o la conversación.
6. El ciclo continúa...

---

# Parte VI: Por Qué Esto Importa - Implicaciones Más Allá de la Tecnología

## 1. La Computación Personal Realizada

La visión original de la computación personal era empoderar a los individuos para que crearan sus propias herramientas. Filer tiene el potencial de devolver esa capacidad, cambiando el enfoque del consumo de aplicaciones a la creación de herramientas personales. El usuario es dueño de los metadatos, los datos y el sistema.

## 2. Implicaciones Económicas

El costo de desplegar una aplicación CRUD simple tradicionalmente es de miles de dólares al año. Con Filer, el costo puede ser cero, ya que puede ejecutarse localmente desde un archivo sin necesidad de servidores, bases de datos o infraestructura de despliegue.

### 3. Un Nuevo Paradigma para la Creación

Filer propone un cambio de la "programación" a la "definición". En lugar de escribir código imperativo para manipular la interfaz de usuario o guardar en una base de datos, se define declarativamente el dominio, y el sistema se materializa a partir de esa definición.

---

## Cómo Empezar

### Pruébalo (Sin Instalación)

Puedes probarlo en línea (o descargar `dist/index.html` y abrirlo en tu navegador). Eso es todo.

### Desarrolla

```
git clone https://github.com/xrrocha/filer.git
cd filer
npm install
npm test          # 1,359 pruebas, 94% de cobertura
npm run dev       # Servidor de desarrollo con recarga en caliente
```

Consulta SETUP.md para una configuración detallada del entorno de desarrollo.