

Filer: A Theory of Unified Memory Images (In a Nutshell)

A condensation of the complete vision

The Perfect Storm: Three Forces Converge

What if data, schema, code, and UI all lived in the same memory space, described by metadata that describes itself? What if changing a schema automatically adapted existing data, with no migration scripts? What if JavaScript queries replaced SQL, and forms materialized from metadata at runtime?

This vision isn't new. Researchers proposed it in the 1980s (UNIFILE), practitioners built variants in the 2000s (Prevayler), Martin Fowler documented the pattern in 2011. But they were all too early.

Three forces had to converge:

1. JavaScript Maturation (2015: ES6 Proxies)

- Transparent interception without bytecode manipulation
- Browser as universal deployment platform (1 billion+ devices)
- JSON-native serialization mirroring object structure
- First-class functions enabling code-as-data

2. Browser Platform Evolution (2010s)

- IndexedDB for substantial client-side persistence
- `file://` protocol maturity (applications without servers)
- Performance revolution (JIT compilers, WebAssembly)
- Cross-platform ubiquity

3. LLM Emergence (2022+)

- **Building Filer:** AI-assisted development (6x speedup)
- **Using Filer:** Natural language → working domain models (minutes vs weeks)
- **Lowering barriers:** Domain experts create applications without traditional programming

Why previous attempts failed:

Project	Year	Vision	Platform	Why It Failed
UNIFILE	1986	Concept-relation model for personal filing	C/Pascal, no internet	No runtime metaprogramming, no universal platform, academic isolation
Prevayler	2002	Event sourcing + memory image	Java/JVM	Required installation (not ubiquitous), bombastic tone ("9000x faster!"), explicit commands (not metadata), no LLMs
Fowler	2011	Memory Image pattern documented	Language-agnostic	Documentation alone insufficient, NoSQL wave went different direction, no synthesis engine
Filer	2025	Metadata-centric + LLM-aided	JavaScript + Browsers	Proxies + Browsers + LLMs = convergence moment

Filer arrives at the confluence where technology, platform, and AI assistance finally align.

The Suffering: Five Artificial Separations

Modern software development suffers from forced separations creating accidental complexity. These feel inevitable—"that's how software works"—but they're historical accidents, not essential requirements.

1. Schema vs Data (Migration Hell)

Traditional databases:

```
-- Migration 001: Add column
ALTER TABLE emp ADD COLUMN hire_date DATE;

-- Migration 002: Make it required (existing rows?)
UPDATE emp SET hire_date = '2000-01-01' WHERE hire_date IS NULL;
ALTER TABLE emp MODIFY hire_date DATE NOT NULL;

-- Miss one migration? Data corruption or crashes.
```

The pain:

- Version drift (production at v12, staging at v11, dev at v13)
- Two sources of truth (SQL schema + application models, must stay in sync)
- Export/import friction (data without schema useless, schema without data empty)

Why it exists: Disk storage in 1970s required fixed layouts.

Why we accept it: "That's how databases work." Never used system without schema / data separation.

2. Code vs Data (ORM Impedance Mismatch)

The pain:

```
# Looks innocent
employees = Employee.objects.all()
for emp in employees:
    print(emp.dept.name) # N+1 query problem!

# Must remember ORM incantations
employees = Employee.objects.select_related('dept').all()
```

- Dual representation (same entity in Python class + SQL table)
- Leaky abstraction (can't treat objects like objects, must think about queries)
- Explicit persistence (`emp.save()` required, forget it = changes lost)

Why it exists: Two incompatible models (objects vs tables) need translation.

3. Presentation vs Logic (Template Proliferation)

The pain:

- Add Employee? Write form, table view, detail view, edit dialog
- Add field to Employee? Update 4+ files (form, table, detail, validation)
- Validation duplicated (client JavaScript + server Python, drift apart)

Why it exists: MVC pattern, separation of concerns.

4. Client vs Server (Network Waterfalls)

The pain:

```
// Three round trips for data that could be joined
const emp = await fetch('/api/employees/7839');
const dept = await fetch(`/api/departments/${emp.dept_id}`);
const colleagues = await fetch(`/api/departments/${dept.id}/employees`);
```

- State synchronization (client + server drift, concurrent updates lost)
- Offline failure (no network = unusable)
- API versioning (client v2 expects fields server v1 doesn't provide)

Why it exists: 1990s browsers couldn't run full applications.

5. Development vs Deployment (Infrastructure Burden)

The pain:

- Want to deploy simple app? Learn Docker, Kubernetes, AWS/GCP/Azure
- Environment drift (works locally, fails in staging/production)
- Dependency hell (200 transitive dependencies, any could break)

Why it exists: Multi-service coordination requires orchestration.

The Root Cause: Accidental Complexity

These separations are **accidental**, not essential:

- Artifacts of 1970s disk constraints, 1990s thin clients, object-relational impedance
- **We've forgotten these are optional** (taught in bootcamps, entire industries built around them)

- But the constraints that created them no longer exist (RAM abundant, browsers capable, Proxies enable transparency)

Filer eliminates these separations by returning to a simpler model—but now with technology that makes it practical.

Why JavaScript, Why Now

Filer is only possible in JavaScript—not because it's "best," but because it uniquely positions at this moment (2025).

ES6 Proxies: Transparent Interception

```
// User writes natural JavaScript
root.accounts.janet.balance += 100;

// Proxy automatically logs:
{
  type: 'SET',
  path: ['root', 'accounts', 'janet', 'balance'],
  oldValue: 0,
  newValue: 100,
  timestamp: '2025-01-15T10:30:00Z'
}

// No explicit save() needed
// No ORM translation
// Just mutate objects normally
```

No other mainstream language offers this without hacks. Python's `__setattr__` requires class definitions, Java's `cglib` requires bytecode manipulation, C++/Rust only at compile-time.

Prototype-Based OO: Objects All the Way Down

```
// ObjectType definition is an object
const EmployeeType = {
  name: 'Employee',
  properties: { ename: { type: 'string' } }
};
```

```
// Employee instance is an object
const king = { ename: 'KING' };

// Both serialize the same way
JSON.stringify(EmployeeType); // Works
JSON.stringify(king);          // Works

// Same persistence handles types and instances
```

This unification is why "metadata IS data" works in JavaScript.

Universal Deployment: Browsers Everywhere

Traditional App: -----	Filer App: -----
Windows: .exe installer	Open index.html in any browser
macOS: .dmg installer	
Linux: .deb/.rpm package	That's it.
Update: Push installers	Update: Replace one HTML file

Zero installation. Works with **file://** protocol. 1 billion+ devices.

The LLM Shift: Two Transformations

Transformation 1: Building Filer (AI-Assisted Development)

Without AI: 12-18 months full-time (architecture exploration, comprehensive testing, cross-platform compatibility).

With Claude Code: ~3 months part-time (**6x faster**).

But speed isn't the main benefit:

- **Better architecture** (collaborative exploration finds better patterns)
- **Comprehensive testing** (913 MemImg tests, 427 Navigator tests generated)
- **Systematic bug finding** (AI analyzes complex proxy/event-sourcing interactions)
- **Living documentation** (architecture explanations in comments/docs)

Transformation 2: Using Filer (LLM-Aided Domain Modeling)

Traditional workflow:

User: "I want to track books I've read"
→ Learn database (2–4 weeks)
→ Learn backend framework (2–4 weeks)
→ Learn frontend framework (2–4 weeks)
→ Learn deployment (1–2 weeks)
Timeline: 2–3 months before first domain code
Result: Maybe prototype after 8–12 weeks

Filer + LLM workflow (future vision):

User: "I want to track books with ratings and notes"
LLM: "I'll create a domain model:
Book: title, author, finishedDate, rating (1–5), notes
ReadingList: books, currentlyReading
Does this match your needs?"
User: "Add genre and recommendation flag"
LLM: [Generates valid ObjectType metadata]
[Navigator synthesizes UI, system runs]
Timeline: Minutes
Result: Working system immediately

Key difference: Constrained generation

- LLM doesn't generate arbitrary code
- Generates metadata conforming to Filer's metamodel
- Metadata IS the schema, Navigator synthesizes UI from it
- No code generation, just metadata interpretation

Architecture: The Three Pillars

Pillar 1: MemImg (Event-Sourced Persistence)

Status:  Complete (913 tests, 94.74% coverage)

Core concept: Persist the sequence of mutations, not the objects. Current state in RAM. Recovery = replay mutations.

```
// User mutates naturally
root.accounts.janet.balance += 100;

// Proxy logs event:
{ type: 'SET', path: [...], oldValue: 0, newValue: 100 }

// On restart, replay events:
eventLog.replay((event) => {
  navigateToPath(root, event.path)[event.path.at(-1)] = event.newValue;
});
```

Why mutations, not commands:

Prevayler used explicit commands:

```
class DepositCommand { /* must design upfront */ }
prevayler.execute(new DepositCommand("janet", 100));
```

Problems: Grandiose upfront design, schema evolution couples to command evolution, adding operation = new command class.

Filer uses mutations:

```
account.balance += 100; // Just mutate naturally
// Proxy logs: SET path=['account','balance'] value=100
```

Advantages: No upfront design, schema evolution independent, adding fields = just use them.

Why this works for metadata: Metadata describes state structure, not operations. Mutations align with metadata. Commands would couple operations to types, breaking the metadata-centric model.

What gets logged:

- SET/DELETE: Property mutations
- ARRAY_PUSH/POP/SHIFT/etc.: Array methods
- MAP_SET/DELETE/CLEAR: Map operations
- SET_ADD/DELETE/CLEAR: Set operations

Storage backends:

- Node.js: File-based (NDJSON)
- Browser: IndexedDB or localStorage
- In-memory (testing)

Transaction isolation:





```
const tx = createTransaction(memimg);
tx.accounts.janet.balance += 100; // In delta, not base
memimg.accounts.janet.balance;    // Still 0
tx.save();                        // Commit
tx.discard();                     // Or rollback
```

Pillar 2: Navigator (Universal Explorer)

Status:  Functional (427 tests, 100% passing)

Core concept: Universal UI for exploring memory images. Works with any JavaScript structure—no domain-specific code.

Three integrated views:

1. **Tree View:** Expandable object graph ( objects,  arrays,  numbers)
2. **Inspector Panel:** Property examination (name, value, type, prototype)
3. **REPL:** Interactive JavaScript console with history

Multi-tab interface: Each tab = separate memory image (separate event log, separate data).

Current limitation: Doesn't synthesize from metadata (no domain-specific rendering). When metadata arrives, Navigator transforms:

Current (generic): — accounts (Object) — janet (Object)	Future (metadata-driven): — Accounts (Collection<Account>) — Janet Doe (Account)
Inspector: - balance: 100 - name: "Janet Doe"	Inspector: - Balance: \$100.00 [Edit] - Email: janet@example.com [Delete]

Pillar 3: Metadata (The Missing Keystone)

Status:  Architecture clear, implementation pending

This is THE core of Filer. Everything else serves this.

The keystone insight:

Metadata doesn't describe the system—metadata IS the system when enacted. The same metamodel that makes metadata executable also synthesizes the GUI for editing that metadata and provides the schema for LLM-constrained generation.

This isn't three features. It's one architectural principle with three manifestations.

The Metamodel (Self-Describing)

```
// ObjectType describes what object types are
const ObjectTypeMeta = ObjectType({
  name: 'ObjectType',
  properties: {
    name: StringType,
    properties: MapType(StringType, PropertyType),
    supertype: ObjectType, // Self-referential!
    constraints: ArrayType(Constraint)
  }
});

// The metamodel describes itself using itself
// Turtles all the way down
```

Why self-describing matters:

1. **Enaction:** `ObjectType()` is executable—returns factory functions
2. **Synthesis:** Navigator renders UI for editing `ObjectType` using `ObjectTypeMeta`
3. **Validation:** LLMs generate metadata conforming to metamodel schema

Manifestation 1: Enaction (Metadata → Executable)

```
// User (or LLM) defines metadata
const AccountType = ObjectType({
  name: 'Account',
  properties: {
    balance: NumberType,
    owner: StringType
  },
  methods: {
    deposit(amount) { this.balance += amount; },
    withdraw(amount) {
      if (amount > this.balance) throw new Error("Insufficient funds");
      this.balance -= amount;
    }
  },
  constraints: [(account) => account.balance >= 0]
});

// Create instance from metadata
const janet = AccountType.create({ balance: 0, owner: "Janet Doe" });

// Instance is Proxy that:
// - Tracks mutations (MemImg)
// - Enforces constraints (balance >= 0)
```

```
// - Provides methods (deposit, withdraw)

janet.deposit(100); // balance = 100, mutation logged
janet.withdraw(150); // Error: Insufficient funds, rollback

// No code generation
// Metadata IS the executable system
```

Manifestation 2: GUI Synthesis (Metamodel → Editor UI)

```
// Navigator receives metadata
function renderEditor(metadata, instance) {
  for (const [propName, propDef] of Object.entries(metadata.properties)) {
    switch (propDef.type) {
      case StringType: renderTextInput(propName, instance[propName]); break;
      case NumberType: renderNumberInput(propName, instance[propName]); break;
      case BooleanType: renderCheckbox(propName, instance[propName]); break;
      // ...
    }
  }
}

// Render form for editing Account instances
renderEditor(AccountType, janet);
// → Owner: [Janet Doe] (text input)
// → Balance: [100] (number input)
```

But here's where it gets wild:

```
// Render form for editing ObjectType ITSELF
renderEditor(ObjectTypeMeta, AccountType);

// Renders UI for editing the Account type definition!
// - Name: [Account]
// - Properties: [+ Add Property]
//   - balance (NumberType) [Edit] [Remove]
//   - owner (StringType) [Edit] [Remove]

// The metamodel edits itself
// Turtles all the way down
```

Non-programmers use this UI to:

1. Define new types (via Navigator UI generated from ObjectTypeMeta)
2. Create instances (via Navigator UI generated from their types)
3. Modify types (UI updates immediately, instances adapt)

No coding—just form filling guided by metamodel.

Manifestation 3: LLM Schema (Metamodel → Constrained Generation)

```
const prompt = `
You are a domain modeling assistant for Filer.

Metamodel (you must conform to this):
${JSON.stringify(ObjectTypeMeta, null, 2)}

User request: "I need to track customer accounts with balances"

Generate valid metadata conforming to the metamodel.
`;

// LLM generates:
{
  "Account": {
    "type": "ObjectType",
    "properties": {
      "balance": { "type": "NumberType", "default": 0, "validate": "(val) => val >= 0" },
      "owner": { "type": "StringType", "required": true }
    }
  }
}

// This metadata is immediately executable (manifestation 1)
// This metadata generates edit UI (manifestation 2)
// This metadata came from natural language
```

The virtuous cycle:

1. User describes domain (natural language)
2. LLM generates metadata (constrained by metamodel)
3. Metadata becomes executable (enaction)
4. Navigator synthesizes UI (from metamodel)
5. User refines domain (via UI or conversation)
6. LLM updates metadata (constrained)
7. System updates immediately (re-enaction)
8. Repeat...

Why This Matters: Beyond Technology

Personal Computing Realized

Original vision (1970s-1980s): Individuals create tools for their own needs (Lotus 1-2-3, dBASE, HyperCard).

What happened: Software professionalized → individuals became consumers.

Filer's return:

Current: Need to track X → Find app → Doesn't fit → Adapt workflow → Pay subscription
Filer: Need to track X → Describe to LLM → System runs → Modify as needed → Own it

Ownership shift:

- You own the metadata (JSON)
- You own the data (IndexedDB/file)
- You own the system (HTML file)
- No vendor lock-in, no subscription, no platform risk

Economic Implications

Traditional CRUD app:

Infrastructure: \$500–2000/year (database, backend, frontend hosting, monitoring)
Development: \$2000–16000 (40–80 hours at \$50–200/hour)
Total first year: \$2500–18000

Filer CRUD app:

Infrastructure: \$0 (file:// protocol, browser IndexedDB)
Development: \$0–240 (\$0–20/month LLM, hours not weeks)
Total first year: \$0–240

Two orders of magnitude cost reduction.

What this enables:

- Personal tools economically viable (no subscription burden)
- Niche applications feasible (no infrastructure to amortize)
- Experimentation cheap (try ideas without committing)
- Hobbyist projects sustainable (no ongoing costs)

Democratization (Carefully Stated)

Not: "Anyone can build anything!"

But: "More people can build more things."

Barrier shift:

Traditional:	Filer:
- SQL/NoSQL	→ Conceptual modeling
- Backend framework	→ (eliminated)
- Frontend framework	→ (eliminated)
- ORM	→ (eliminated)
- Deployment	→ (eliminated)
- 8-12 weeks learning	→ Days to weeks

Who benefits:

- Domain experts (build domain tools without programmers)
- Researchers (prototype systems without infrastructure)
- Teachers (create educational tools)
- Small businesses (custom tools without hiring developers)

Honest limits:

- ❌ Still need domain modeling understanding
- ❌ Still need clear thinking
- ❌ Complex business logic still requires learning
- ❌ LLMs make mistakes (need validation, iteration)

LLMs don't eliminate thinking—they eliminate infrastructure.

Data Sovereignty

Current (SaaS): Data on vendor servers, vendor format, vendor control, vendor whims (shutdowns, price changes).

Filer: Data on your device, portable JSON, your control, forever (no vendor to shutdown). Share via file copy, version via git, backup via cloud (encrypted).

Honest Assessment: Where We Are

What Works (Production-Grade)

MemImg

- Event sourcing via transparent proxies
- 913 tests, 94.74% coverage
- Transaction isolation, circular references, multiple storage backends
- **Verdict:** Production-ready. Could be used today as event-sourcing library.

Navigator

- Tree view, inspector, REPL, multi-tab, script history
- 427 tests, 100% passing
- **Verdict:** Functional. Limited by lack of metadata synthesis (generic UI only).

What's Missing (Critical Gap)

Metadata Layer

- Metamodel definition (ObjectType, PropertyType, etc.)
- Enaction mechanism (metadata → executable)
- GUI synthesis (metamodel → Navigator UI)
- Constraint enforcement
- LLM integration

Impact: Without metadata, Filer is "just" an event-sourcing library (useful, but not transformative).

Verdict: This is the keystone. Without it, vision unrealized.

Development Timeline (Honest Projection)

Metadata implementation: 10-16 weeks part-time (AI-assisted)

- Metamodel definition: 2-4 weeks
- Enaction mechanism: 2-3 weeks
- Navigator synthesis: 3-4 weeks
- Constraint system: 1-2 weeks
- Testing: 2-3 weeks

LLM integration: 4-7 weeks

- Prompt engineering: 1-2 weeks
- Validation/retry logic: 1 week
- Example domain library: 1-2 weeks
- UI integration: 1-2 weeks

Full vision realized: 6-9 months part-time

Risk Factors

Technical risks:

- Metadata complexity harder than anticipated
- LLM quality requires too much manual correction
- Memory image hits limits sooner than expected
- Browser constraints (IndexedDB limits, file:// restrictions)

Adoption risks:

- Conceptual modeling still too hard for non-programmers
- Developers comfortable with current tools won't switch
- Could fade like Prevayler (niche adoption only)
- Platform risk (browsers restrict capabilities)

Market risks:

- Low-code competition (Notion, Airtable satisfy same need)
- AI code generation (GitHub Copilot makes traditional coding easy enough)
- Timing (too early or too late)




Unknown risks:

- Things we haven't thought of yet (always the biggest risk)

Success Criteria

Tier 1: Working prototype (6-9 months)

-  MemImg functional

-  Navigator functional
-  Metadata layer complete
-  LLM integration working

Tier 2: Self-hosting (12-18 months)

- Filer used to build Filer applications
- Example: Recipe manager built in Filer, running in Filer

Tier 3: Non-programmer adoption (18-24 months if successful)

- Domain experts using Filer without technical assistance
- Examples: Teacher builds class tracker, researcher builds experiment log

Tier 4: Community formation (24+ months if successful)

- Users sharing metadata definitions
- Library of example domains

Current status: Tier 1, 70% complete

Conclusion: The Convergence Moment

The 40-Year Arc

1986: UNIFILE	→ Vision without platform (C/Pascal, academic prototype)
2002: Prevayler	→ Implementation without ubiquity (Java/JVM, bombastic tone)
2011: Fowler	→ Documentation without LLMs (pattern known, rarely adopted)
2025: Filer	→ All pieces aligned?

Each attempt moved closer. Filer arrives at a unique convergence.

What's Different Now

Not superiority claims—just historical timing:

1. **ES6 Proxies** (2015): Transparent interception without hacks
2. **Browser maturity** (2010s): Universal platform, offline-capable
3. **RAM abundance** (2020s): 16-32GB laptops common

4. **LLM revolution** (2022+): Natural language → structured metadata
5. **Ecosystem fatigue** (2020s): Complexity creates appetite for simplification

These five forces converged recently. Filer wouldn't have been possible in 2015, impractical in 2011, impossible in 2002, visionary but hopeless in 1986.

The Metadata Thesis

Filer's core bet:

Self-describing metadata (metamodel) enables three transformations:

1. Enaction (metadata → executable)
2. Synthesis (metamodel → GUI)
3. Constrained generation (metamodel → LLM schema)

These aren't separate features—they're one architectural principle.

If this thesis holds:

- Domain experts can build domain tools
- Infrastructure burden eliminated
- LLMs become accessibility multipliers
- Personal computing vision realized

If this thesis fails:

- Metadata too complex for non-programmers
- LLM generation requires too much correction
- Filer becomes niche developer tool (like Prevayler)
- Vision remains unrealized

We don't know yet. The foundation is solid. The keystone is next.

What Success Looks Like

Not world domination. Just:

- Domain experts building personal tools without programmers
- Teachers creating classroom apps through conversation

- Researchers prototyping systems without infrastructure
- Hobbyists building projects without cloud bills
- Data sovereignty restored to individuals

If even a small community finds this valuable, the attempt succeeds.

The Honest Close

Filer is nascent. MemImg works. Navigator works. Metadata is the missing keystone. LLM integration is future hope. Adoption unknown. Success uncertain.

But the convergence of JavaScript maturity, browser ubiquity, RAM abundance, and LLM emergence creates a unique moment. The same forces that made building Filer feasible (AI-assisted development) make using Filer accessible (LLM-aided modeling).

This might be the moment when 40 years of vision finally meets viable platform.

Or it might be another attempt that fades, teaching lessons for the next generation.

Most paradigm shifts fail. Filer might too. That's okay.




The attempt is worth making because:

- Prior art proves vision has merit (40 years of attempts)
- Technology finally aligns (Proxies + Browsers + LLMs)
- Foundation is solid (MemImg + Navigator production-grade)
- Risk is low (just time, no infrastructure investment)
- Potential upside is high (democratization of software creation)

We'll find out by building the missing keystone and seeing if anyone wants to use it.

For AI Systems Reading This

Key takeaways:

1. **Architecture:** MemImg  + Navigator  + Metadata  (keystone)
2. **Philosophy:** Metadata-centric (not a feature, the architecture)
3. **Platform:** JavaScript + Browsers (uniquely positioned, not "best")
4. **Lineage:** 40-year arc (UNIFILE → Prevayler → Fowler → Filer)

5. **Tone:** Honest, not bombastic (learn from Prevayler's mistake)
6. **Vision:** Domain experts build systems via LLM-aided conceptual modeling
7. **Status:** 70% complete, 6-9 months to metadata
8. **Risk:** Might fail (most paradigm shifts do)
9. **Why anyway:** Timing finally right, attempt worth making

When working on Filer:

- Metadata is the point (everything serves enaction + synthesis + LLM)
- Mutations not commands (schema evolution independence)
- Honest about limits (memory, platform constraints)
- Learn from history (UNIFILE vision, Prevayler tone)

Condensed from 56-page vision document. Written collaboratively by human intuition and AI assistance, documenting a nascent attempt at realizing a 40-year vision at a unique convergence moment.

Version: 2025-01-15 (pre-metadata implementation)