# **Dig|lit Living Technical Doctrine**

## **The Adaptive Brain: Technology-Agnostic Architecture Principles**

"Tools change every decade. Principles endure for centuries."  
 This document transcends React, Python, and cloud providers—it defines HOW we think about technology.

## **COMMANDMENT I: ARCHITECTURAL FIRST PRINCIPLES**

### **1.1 Own Your Data Layer (Forever)**

**Principle:** Never build on a platform where data export requires permission or proprietary formats.

**Enforcement:**

* Every database must support standard SQL or NoSQL export (PostgreSQL, MongoDB, etc.)
* Binary blobs stored with open codecs (AVIF/WebP, not proprietary)
* APIs must provide bulk export endpoints (CSV, JSON, Parquet)

**Examples:**

* ✅ **Good:** Supabase (PostgreSQL), self-hosted Minio (S3-compatible)
* ❌ **Bad:** Salesforce proprietary database, Notion API (no bulk export)

**Why It Lasts 100 Years:** In 2075, PostgreSQL may be obsolete, but the *principle* of ownable data survives. Whatever replaces SQL will still need export portability.

### **1.2 Abstraction Layers Are Mandatory**

**Principle:** Never couple business logic directly to infrastructure. Always have a translation layer.

**Implementation:**

┌─────────────────────┐

│ BUSINESS LOGIC │ ← Pure functions, no vendor references

└──────────┬──────────┘

│

┌──────────▼──────────┐

│ ABSTRACTION LAYER │ ← Interfaces/protocols

└──────────┬──────────┘

│

┌─────┴─────┐

▼ ▼

┌─────────┐ ┌─────────┐

│ AWS S3 │ │ Vercel │ ← Swappable providers

└─────────┘ └─────────┘

**Code Example (Storage Abstraction):**

// GOOD: Provider-agnostic interface

interface StorageProvider {

upload(file: File, path: string): Promise<URL>;

download(path: string): Promise<Blob>;

delete(path: string): Promise<void>;

}

// Implementations can swap without touching business logic

class S3Storage implements StorageProvider { ... }

class CloudflareR2Storage implements StorageProvider { ... }

class LocalFileStorage implements StorageProvider { ... }

// Business logic never knows which provider is active

const storage: StorageProvider = getConfiguredStorage();

await storage.upload(userAvatar, '/avatars/user-123.jpg');

**Why It Lasts 100 Years:** Interfaces outlive implementations. When AWS is replaced by quantum storage in 2070, we swap the implementation—business logic untouched.

### **1.3 Security by Default, Not by Addition**

**Principle:** Security cannot be "added later." It's foundational, like a building's structure.

**Non-Negotiable Layers:**

1. **Authentication:** Multi-factor by default (TOTP, WebAuthn)
2. **Authorization:** Role-based access control (RBAC) + attribute-based (ABAC)
3. **Encryption:**
   * At rest: AES-256-GCM minimum
   * In transit: TLS 1.3+ only
   * In use: Homomorphic encryption for sensitive compute (future-ready)
4. **Zero Trust:** Every request authenticated, even internal services
5. **Quantum Resistance:** Prepare for post-quantum cryptography (NIST standards)

**Threat Model Hierarchy:**

Level 1: Script kiddies → Rate limiting, CAPTCHA

Level 2: Professional hackers → WAF, intrusion detection

Level 3: Nation-state actors → Air-gapped secrets, hardware security modules

Level 4: Insider threats → Audit logs, least-privilege access

Level 5: AI-powered attacks → Adversarial ML defenses (future)

**Why It Lasts 100 Years:** Attackers get smarter, but defense-in-depth is eternal. Quantum computers may break RSA, but the *principle* of layered security survives.

### **1.4 Performance is a Feature, Not an Optimization**

**Principle:** Slow software is broken software. Performance budgets are requirements, not goals.

**Immutable Performance Targets:**

| **Metric** | **Target** | **Rationale** |
| --- | --- | --- |
| Time to First Byte (TTFB) | <200ms | Human perception threshold |
| Largest Contentful Paint (LCP) | <2.5s | Google Core Web Vitals |
| API Response (p99) | <500ms | Keep users in flow state |
| Database Query (p99) | <100ms | Prevent N+1 query hell |
| Build Time | <5 min | Fast feedback loops |

**Enforcement Mechanisms:**

* **Budget Guards:** CI/CD fails if bundle size exceeds thresholds
* **Real User Monitoring:** Track p50/p95/p99 latencies in production
* **Chaos Engineering:** Intentionally degrade performance to test resilience
* **Annual Performance Audits:** Third-party review by web.dev or similar

**Why It Lasts 100 Years:** Human patience hasn't increased in 10,000 years. Even with 10Gbps internet in 2075, users will still rage-quit slow apps.

### **1.5 Composition Over Inheritance**

**Principle:** Build systems from small, reusable pieces. Avoid deep class hierarchies.

**Design Pattern:**

// BAD: Deep inheritance tree (fragile)

class Animal { ... }

class Mammal extends Animal { ... }

class Dog extends Mammal { ... }

class Labrador extends Dog { ... } // Now locked into 4-level hierarchy

// GOOD: Composition (flexible)

interface Movable { move(): void; }

interface Audible { makeSound(): void; }

interface Trainable { learn(trick: string): void; }

class Labrador implements Movable, Audible, Trainable {

// Compose behaviors as needed

private locomotion = new QuadrupedMovement();

private vocalization = new BarkSound();

private cognition = new ObedienceTraining();

move() { this.locomotion.walk(); }

makeSound() { this.vocalization.emit('woof'); }

learn(trick: string) { this.cognition.train(trick); }

}

**Why It Lasts 100 Years:** Nature uses composition (DNA is modular). Rigid hierarchies go extinct when environments change.

## **COMMANDMENT II: TECHNOLOGY EVALUATION FRAMEWORK**

### **2.1 The Lindy Effect Filter**

**Rule:** Prefer technologies that have survived >10 years. New tech must prove 10x better to replace old.

**Technology Tiers:**

TIER S (Immortal): SQL, HTTP, HTML, CSS, Git

↓ Use liberally, these will outlive you

TIER A (Stable): PostgreSQL, Linux, Python, JavaScript

↓ Safe bets, incremental improvements

TIER B (Mature): React, Docker, Kubernetes

↓ Dominant but watch for successors

TIER C (Emerging): Rust, WebAssembly, Svelte

↓ Use for new projects, avoid critical path

TIER D (Experimental): Deno, Bun, AI frameworks

↓ R&D only, expect rewrites

**Evaluation Scorecard (0-10 scale):**

| **Criterion** | **Weight** | **Example: Supabase** |
| --- | --- | --- |
| Open Source | 2x | 10 (PostgreSQL core) |
| Community Size | 1.5x | 8 (50K+ GitHub stars) |
| Data Portability | 2x | 10 (standard SQL export) |
| Security Track Record | 1.5x | 7 (young but no breaches) |
| Performance | 1x | 9 (sub-100ms queries) |
| Documentation | 1x | 10 (excellent) |
| Vendor Lock-In Risk | 2x | 8 (can self-host) |
| **Total Score** | - | **94/110** → Adopt |

**Thresholds:**

* 90+: Adopt immediately
* 70-89: Adopt with abstraction layer
* 50-69: Use for non-critical features
* <50: Avoid or sandbox only

### **2.2 The Boring Technology Club**

**Manifesto:** You get 3 "innovation tokens" per project. Spend them wisely.

**Example Budget:**

Project: AI-Powered ERP System

Innovation Token 1: Custom AI agent architecture (core differentiation)

Innovation Token 2: WebAssembly for client-side data processing (10x perf)

Innovation Token 3: [SAVED FOR FUTURE]

Boring Choices:

- PostgreSQL for database (not MongoDB or exotic DB)

- React for frontend (not bleeding-edge framework)

- Docker for deployment (not Kubernetes initially)

- Stripe for payments (not custom crypto contracts)

**Why 3 Tokens?**

* 0 tokens: Commodity product, no competitive advantage
* 1-3 tokens: Differentiated but debuggable
* 4+ tokens: Hero project that collapses when creator leaves

**Why It Lasts 100 Years:** The principle of "conserve innovation budget" is timeless. Even in 2075, chasing shiny objects will still cause project failures.

### **2.3 Open Source Contribution Mandate**

**Policy:** For every 10 hours spent using open source, contribute 1 hour back.

**Contribution Ladder:**

1. **Novice:** Report bugs with reproduction steps
2. **Contributor:** Fix typos in documentation
3. **Developer:** Submit bug fixes or small features
4. **Maintainer:** Own a package or subsystem
5. **Creator:** Launch a new open-source project (1000+ stars)

**Strategic Projects to Sponsor (2025-2045):**

* **Database:** Fund PostgreSQL extensions for AI workloads
* **Frontend:** Contribute to React or successor framework
* **AI:** Open-source model training pipelines
* **Security:** Quantum-resistant crypto libraries
* **DevOps:** Self-healing infrastructure tools

**Why It Lasts 100 Years:** Open source is a commons. Tragedies are avoided through reciprocity. Companies that only take eventually face forks or abandonment.

## **COMMANDMENT III: ARCHITECTURE PATTERNS FOR LONGEVITY**

### **3.1 The Hexagonal Architecture (Ports & Adapters)**

**Structure:**

┌───────────────────────┐

│ BUSINESS LOGIC │

│ (Core Domain) │

└───────────┬───────────┘

│

┌────────────┼────────────┐

│ │ │

┌────▼────┐ ┌───▼────┐ ┌───▼────┐

│ Port │ │ Port │ │ Port │ (Interfaces)

└────┬────┘ └───┬────┘ └───┬────┘

│ │ │

┌────▼────┐ ┌──▼─────┐ ┌──▼──────┐

│Adapter │ │Adapter │ │ Adapter │ (Implementations)

│(REST) │ │(GraphQL)│ │(gRPC) │

└─────────┘ └────────┘ └─────────┘

**Benefits:**

* Swap APIs (REST → GraphQL) without touching core logic
* Test business logic without databases (mock ports)
* Add new interfaces (CLI, WebSockets) easily

**Example:**

// CORE: Business logic (no infrastructure)

class OrderService {

constructor(

private paymentGateway: PaymentPort,

private inventory: InventoryPort,

private notifications: NotificationPort

) {}

async placeOrder(order: Order): Promise<OrderResult> {

// Pure business logic

const payment = await this.paymentGateway.charge(order.amount);

await this.inventory.reserve(order.items);

await this.notifications.send(order.customerEmail, 'Order confirmed');

return { orderId: payment.id, status: 'confirmed' };

}

}

// PORTS: Interfaces (what core needs)

interface PaymentPort {

charge(amount: Money): Promise<PaymentResult>;

}

// ADAPTERS: Implementations (how it's done)

class StripeAdapter implements PaymentPort {

async charge(amount: Money) {

return await stripe.charges.create({ amount: amount.cents, currency: amount.currency });

}

}

class CryptoAdapter implements PaymentPort {

async charge(amount: Money) {

return await tronWeb.contract().transfer(WALLET, amount.toUSDT());

}

}

// ASSEMBLY: Dependency injection

const service = new OrderService(

new StripeAdapter(), // Can swap to CryptoAdapter without changing OrderService

new SupabaseInventory(),

new SendgridNotifications()

);

### **3.2 Event-Driven Architecture for Scale**

**Principle:** Systems communicate through events, not direct calls. Enables async, decoupling, and replay.

**Event Flow:**

USER ACTION → COMMAND → EVENT → HANDLERS → SIDE EFFECTS

Example:

User clicks "Buy"

→ PlaceOrderCommand

→ OrderPlacedEvent

→ [ReserveInventory, ChargePayment, SendEmail]

→ Success/Failure Events

**Event Store Schema:**

interface DomainEvent {

id: string; // UUID

type: string; // 'OrderPlaced', 'PaymentCharged'

aggregateId: string; // Order ID, User ID, etc.

data: any; // Event payload

metadata: {

timestamp: Date;

userId: string;

causationId: string; // What triggered this event

correlationId: string; // Trace full transaction

};

}

// Store ALL events forever (immutable log)

CREATE TABLE events (

id UUID PRIMARY KEY,

type TEXT NOT NULL,

aggregate\_id TEXT NOT NULL,

data JSONB NOT NULL,

metadata JSONB NOT NULL,

created\_at TIMESTAMPTZ DEFAULT NOW()

);

CREATE INDEX idx\_events\_aggregate ON events(aggregate\_id, created\_at);

CREATE INDEX idx\_events\_type ON events(type, created\_at);

**Why It Lasts 100 Years:** Event sourcing is how the universe works (physics = events through time). Append-only logs are the most durable data structure.

### **3.3 Microservices? No. Modular Monolith? Yes.**

**Philosophy:** Start with a well-structured monolith. Extract microservices only when proven necessary.

**Monolith Structure:**

app/

├── modules/

│ ├── orders/

│ │ ├── domain/ # Business logic

│ │ ├── api/ # REST endpoints

│ │ ├── events/ # Event handlers

│ │ └── db/ # Data access

│ ├── payments/

│ ├── inventory/

│ └── notifications/

├── shared/

│ ├── auth/

│ ├── database/

│ └── config/

└── main.ts # Application entry

**Microservice Extraction Criteria:**

* ✅ Module has clear bounded context (payments, search)
* ✅ Different scaling needs (search needs 10x compute)
* ✅ Team size >20 people (Conway's Law: architecture mirrors org)
* ✅ Regulatory isolation required (HIPAA-compliant service separate)

**When NOT to Extract:**

* ❌ Premature optimization ("we might need to scale")
* ❌ Resume-driven development ("I want to learn Kubernetes")
* ❌ Cargo culting ("Netflix does microservices")

### **3.4 Database-Per-Service (Eventually)**

**Evolution Path:**

PHASE 1 (Year 0-2): Single PostgreSQL database, multiple schemas

orders\_schema.orders

payments\_schema.transactions

PHASE 2 (Year 2-5): Logical separation, still one database

Strict foreign key rules: No cross-schema FKs

PHASE 3 (Year 5+): Physical separation when needed

orders\_db.orders

payments\_db.transactions

Communication via events, never direct DB queries

**Why Gradual?**

* Distributed transactions are HARD (2PC, Sagas, eventual consistency)
* Network calls are 100,000x slower than memory
* Operational complexity explodes (monitoring, backups, migrations)

**Why Eventually?**

* Scale: 1M+ transactions/sec requires sharding
* Compliance: GDPR right-to-erasure easier with user-specific DB
* Failure isolation: Payment DB crash doesn't take down search

## **COMMANDMENT IV: TECHNICAL DEBT LEDGER**

### **4.1 Debt is Not Evil, Untracked Debt Is**

**Metaphor:** Technical debt is like financial debt. A mortgage for a house is smart debt. Credit card debt for luxury goods is dumb debt.

**Debt Classification:**

TYPE 1: Strategic Debt (Good)

- Ship MVP fast, refactor after market validation

- Hardcode config initially, build admin UI later

- Manual deployments → Automate after 100th deploy

TYPE 2: Tactical Debt (Acceptable)

- Copy-paste code to meet deadline (refactor next sprint)

- Skip tests for prototype (add before production)

- Use suboptimal algorithm (optimize when it's proven bottleneck)

TYPE 3: Accidental Debt (Needs Repayment)

- Misunderstood requirements → wrong abstraction

- Technology choice didn't pan out (framework deprecated)

- Team knowledge gaps (junior wrote spaghetti code)

TYPE 4: Reckless Debt (FORBIDDEN)

- No error handling ("it works on my machine")

- Hardcoded credentials in code

- SQL injection vulnerabilities

- No documentation for critical systems