DRAFT - IT Project Guidance

On Design Errors

Version:

0.1

## Purpose

This guidance paper documents foundational design errors commonly encountered in the development and operation of enterprise and government systems. It aims to clarify their historical context, explain why they emerged, and identify their long-term impacts on system integrity, maintainability, and analytical value. The document targets architects, developers, procurement teams, and policy decision-makers to ensure future designs avoid repeating patterns that lead to structural inefficiency, untraceable logic, or irreparable data loss.

## Synopsis

Many IT design practices originated under constraints or assumptions that no longer hold. Techniques like record deletion for storage efficiency, business logic embedded in stored procedures, or centralised rendering of client interfaces arose from technical limitations of past decades. Others, such as the adoption of firewalls as primary security measures or over-reliance on microservices, stem from vendor-led incentives rather than enduring architectural wisdom. This paper deconstructs these missteps, defines their impact, and reframes system design around principles of durability, auditability, separation of concerns, and platform neutrality.

## Contents

[Purpose 1](#_Toc195791704)

[Synopsis 1](#_Toc195791705)

[Contents 2](#_Toc195791706)

[Introduction 3](#_Toc195791707)

[Background of this work 3](#_Toc195791708)

[Context 3](#_Toc195791709)

[Background of issue 3](#_Toc195791710)

[Problem Statement 3](#_Toc195791711)

[H3 4](#_Toc195791712)

[H4 4](#_Toc195791713)

[Appendices 5](#_Toc195791714)

[Appendix A - Document Information 5](#_Toc195791715)

[Versions 5](#_Toc195791716)

[Images 5](#_Toc195791717)

[Tables 5](#_Toc195791718)

[References 5](#_Toc195791719)

[Review Distribution 5](#_Toc195791720)

[Audience 5](#_Toc195791721)

[Structure 5](#_Toc195791722)

[Diagrams 6](#_Toc195791723)

[Acronyms 6](#_Toc195791724)

[Terms 6](#_Toc195791725)

# Introduction

This paper does more than catalogue historical design missteps. It explores the structural, institutional, and cultural dynamics that led to these errors being widely adopted. Many originated as responses to then-valid constraints but persisted long after those constraints changed. Others emerged from vendor influence or popularisation through developer communities, rather than rigorous evaluation. This document aims to guide a return to principled design—resilient, comprehensible, and supportable.

We begin with a foundational challenge to one of the most unexamined assumptions in IT: the belief in "best practice."

# The Illusion of “Best Practice”

The history of software development is not a smooth evolution from poor to good—it is filled with cycles, reversals, and myths. Among the most persistent is the idea that there exists a single best practice for a given design problem. Best practice is typically a reflection of what is most adopted or most promoted, often shaped by the vendors that dominate the market. This makes it a lagging indicator of success—not a leading one.

Best practice is not the same as best design. Best practice tends to be anchored in tooling and adoption, while best design emerges from principles of separation, traceability, and structural soundness. The former is what developers are shown; the latter is what engineers must apply.

Systems built solely on best practice risk being optimized for ease of training or procurement rather than longevity, auditability, or change tolerance. As a result, many systems become unmaintainable not because their teams lacked skill—but because they followed advice optimised for vendors, not public outcomes.

The history of software design is not just a record of progress but also a catalogue of misunderstandings. What was once best practice often becomes tomorrow’s anti-pattern. This document provides a clear and comprehensive review of classic system design errors that have compromised data integrity, inflated operational cost, and undermined user trust. Each section includes origin, symptoms, systemic implications, and recommended correction or reinterpretation.

Beyond examining individual mistakes, this guidance offers a realignment: away from design-by-vendor and toward design-by-principle. It challenges the inherited wisdom of industry trends and calls for decisions grounded in structural logic, not commercial consensus.

# How This Document is Organised

The sections that follow are structured top-down—from decisions that distort whole-of-system strategy, to architectural drift within system components, down to repeated design errors in application-level practice. This staged structure helps reveal how misconceptions at one level propagate downstream.

The groupings are as follows:

1. Strategic Influence and Decision Distortion
   * CTO and EA Dependency on Gartner-Style Sources
   * The Engineering–Science Gap in IT
2. Departmental and System-Level Drift
   * The Myth of the Analytics End-State
   * Operational Hubs vs Data Warehousing
3. System Component Design Failures
   * Tier Drift and Anti-Pattern Accumulation
4. Application Layer Design Errors
   * Record Deletion
   * CRUD Misapplied
   * Logic in Stored Procedures
   * Data Schemas as Interfaces
   * DTOs and Unversioned APIs
   * Unqueryable APIs
   * Server-Side Rendering
   * Microservices Mania
   * Firewall Fetishism

Each issue is introduced, examined in historical and operational context, then closed with clear recommendations for course correction. The structure is cumulative—each level reinforces the one below it.

# Sector Capability and Structural Weakness

## Engineering Discipline in System Design

Information technology lacks the constraints imposed by licensing, structural safety, or physical physics found in traditional engineering. Anyone can build a system. And many do. As a result, the distinction between software craft and systems engineering has been blurred, often to the detriment of the latter.

Design in IT often defaults to what is popular, taught in tutorials, or supplied by default in vendor tooling. But good design in critical systems must meet the same bar as any engineered infrastructure: it must be traceable, testable, comprehensible, and capable of enduring change. Without enforced discipline, systems proliferate that are attractive on first launch but collapse under operational or evolutionary stress.

### Recommendation

Re-establish the distinction between artefact creation and systems engineering. Embed engineering principles—modularity, traceability, version control, fault isolation—into design reviews and solution architectures. Train analysts and architects to recognise when solutions are guided by fashion, not evidence. Advocate for structural peer review and post-implementation evaluations as a norm.

# Strategic Influence and Decision Distortion

## CTO and EA Dependency on Gartner-Style Sources

It is understandable that senior technology leaders turn to structured market analyses such as Gartner’s Magic Quadrant or Forrester Waves. These frameworks simplify decision-making by categorising tools and vendors in ways that appear rigorous and objective. However, they represent the state of the product market, not the discipline of system design. Their rankings are shaped by vendor marketing budgets, maturity of commercial packaging, and customer volume—not by architectural soundness, transparency, or long-term viability.

The absence of open source and custom development from these reports reflects not a technical inferiority, but a lack of monetisable packaging. There is little incentive for market analysts to highlight design patterns without products to sell. As a result, architecture discussions are prematurely narrowed to tools that fit pre-existing vendor categories. This distorts procurement, limits exploratory design, and reinforces legacy thinking at the highest strategic levels.

### Recommendation

Ensure strategic technology decisions are grounded in domain logic, system design fundamentals, and engineering evaluations—not in vendor rankings. Encourage CTOs and EAs to supplement analyst materials with peer-reviewed architectural case studies, post-mortems, and design critique. Introduce cross-functional review panels that can interrogate the logic of tool alignment with capability needs.

# Departmental and System-Level Drift

## The Myth of the Analytics End-State

Analytics is often mistaken as the destination in a digital transformation journey—the endpoint where value is finally realised. In this view, data is first collected, then cleaned, then centralised, and finally analysed by separate reporting functions. This is a misunderstanding.

Analytics is not the end. Action is. And ideally, action should be informed and automated wherever practical. Analytical insight is most powerful when embedded directly within operational systems—when decisions are made closer to the source of information, not at a distance.

The rise of data warehouses and analytics platforms shifted attention and investment away from the systems that generate data. Reporting became disconnected from operational workflows. Worse, analytics teams often worked on stale data, reinterpreting business meaning independently of the teams that created or acted on it.

### Recommendation

Reframe analytics as a means to action, not a destination. Embed lightweight analytical feedback loops into operational systems. Let systems support both live insight and historical analysis without duplicating responsibility across multiple tiers.

## TODO: The Myth of the Analytics End State

Analytics is often mistaken for the goal of system design—something you build towards and arrive at once operational data flows are in place. This is a misconception. Analytics is not the goal. Action is. Systems exist to serve, not just to observe. Analytics serves this purpose only insofar as it can inform or trigger decisions.

From an optimisation perspective, actions should be automated where feasible. This implies that the best location for analysis is not a retrospective data warehouse, but in proximity to live operations—at the moment of decision, not after the fact.

Data warehouses, by contrast, exist downstream. They collect what could not be analysed earlier, or what was generated by systems not designed for introspection. The insights they offer are often delayed, disconnected, or abstracted from real-time context. Reporting becomes a form of archaeology—studying what happened, rather than shaping what happens next.

### Recommendation

Design systems with embedded operational analytics that inform action directly, ideally through automation. Use data warehouses only for longitudinal studies, aggregated research, or retrospective audit—not as a substitute for live system understanding.

### Operational Hubs vs Data Warehousing

In many government and education settings, the accepted answer to reporting challenges has been the data warehouse: a central repository for cleaned, de-identified, standardised data extracted from live systems. While this provides performance and reporting isolation, it also disconnects insight from action and raises governance overhead.

Worse, the data warehouse becomes seen as the canonical source of truth, which is often incorrect. Warehouses contain transformed, sometimes incomplete, sometimes delayed representations of events. They reflect a version of the world curated for reporting, not for real-time decisions or operational accountability.

An alternative pattern is the use of operational data hubs: well-governed, queryable environments that expose real-time data through domain-specific APIs, with identity-aware controls and context-preserving access. These support regulated analytics without requiring wholesale data replication.

#### Recommendation

Do not elevate the data warehouse to canonical status. Preserve operational context. Use data hubs with policy-bound exposure and clear lineage tracking. Let analytics emerge from traceable operational states, not reverse-engineered reporting logic.

### TODO: Operational Hubs vs Data Warehousing

Many government and education systems have been encouraged to shift reporting out of their core applications and into central data warehouses. While this does insulate performance, it does so at a cost: timeliness, transparency, and accessibility. A better pattern is the use of operational hubs with governed access and anonymised analytics. These support on-demand insights without duplicating data across silos or replicating governance frameworks in parallel.

#### Recommendations

Do not treat the data warehouse as a canonical source. Preserve operational visibility. Embed analytical access controls inside live systems using identity-aware gateways and domain-level filtering. that, if left unexamined, can lead to significant structural or strategic risk. For each, we provide clarification, context, and a path toward better decision-making.

# Tier Design Failures

Traditional architecture was grounded in a clean separation of concerns: the storage tier handled persistence, the logic tier enforced rules and behaviour, and the interface tier presented outputs to users. On the client side, interaction was isolated, lightweight, and independent. This design wasn't just elegant—it enabled modular development, performance tuning, and long-term adaptability.

But in the late 1990s and early 2000s, a fundamental shift occurred: architectural patterns stopped coming from engineering disciplines or academic sources, and instead began to emerge from vendors with platforms to sell. No longer focused on supplying infrastructure or tools to support independently governed systems, vendors began offering integrated development platforms—complete with prescribed architectural models that aligned with their product stack.

These new models weren't optimised for clarity or structural integrity—they were optimised to easy learning and to enforce dependency. Tier boundaries, once clearly delineated for maintainability and flexibility, began to erode under commercial pressures. Database vendors encouraged logic to move down into the storage layer. Web vendors encouraged interface logic to move down into centralised server components. Each shift benefitted the vendor in the short term—locking customers into platform behaviours—while systematically undermining modularity, auditability, and long-term resilience.

This period was marked by two major collapses in layered architecture: the first, a movement of logic downward into the database tier; the second, a collapse of presentation upward into centralised server control.

## Logic Dropped to the Storage Tier (Stored Procedures)

In the 1990s and early 2000s, it became common—especially in banking and finance—to push validation and business rules into stored procedures. This was often justified by performance concerns, a desire to enforce integrity close to the data, and limitations in early application platforms.

But the strategic incentives were clearer: database vendors gained control. Oracle in particular made stored procedures central to system logic, creating a form of logic capture. Systems tied themselves to proprietary implementations in PL/SQL or T-SQL. Portability vanished. So did evolvability.

This was the first major fracture in tier architecture. Logic, once maintained in application code, was scattered across stored scripts with little version control or testing rigour. Over time, systems became opaque and brittle. Ownership blurred, and teams struggled to understand behaviour embedded in the data tier.

The result was a suspenders-and-belt scenario: both the application and database layers were expected to implement the same logic, often by different teams. Unsurprisingly, discrepancies crept in. Logic diverged, testing became laborious, and bug triage turned into blame shifting. Some attempted to eliminate the problem by removing the application logic entirely and consolidating everything in the database. This approach briefly gained momentum before its limitations became undeniable. Fortunately, the software engineering community began to reject this trajectory, and the design movement known as Domain-Driven Design reasserted the logic tier’s central role.

### Recommendation

Avoid logic in any other tier than the application tier. Let the database act as an relatively dumb, indexable, performant store of state—not as an execution environment. This restores modularity, traceability, and system portability.

## Presentation Dropped to the Server (Server-Side Rendering)

The origins of HTML were simple and powerful: it was a markup language for delivering static documents. Early websites were composed of text files with minimal styling, rendered by browsers without any server-side logic. This model required almost no server load—files were simply retrieved and displayed. But as user expectations grew, so too did the demand for dynamic behaviour.

To address this, Common Gateway Interface (CGI) was introduced, allowing for simple server-side logic. Interactions like form submission and lookup queries could be handled by executing scripts on the server. As expectations continued to grow—logins, validation, content updates—the server absorbed more and more responsibility.

Eventually, the entire rendering process shifted to the server. Applications began generating full HTML pages on each request. Frameworks like ASP.NET promoted this model, encouraging event-driven design where user interaction triggered a complete server-side rendering cycle. Layout, logic, authentication, and state management all lived on the server.

Microsoft had little reason to resist this shift—in fact, they encouraged it. IIS was given away to developers, accelerating adoption of server-heavy architecture. But this was strategic: more rendering on the server meant more compute usage, more servers, and more OS licenses. It wasn’t just architecture—it was a business model.

This centralisation came at a cost. Each interaction triggered full page loads, straining network capacity and CPU. Even as CPUs improved, network latency remained an unsolved constraint. Content Delivery Networks (CDNs) were introduced to push cached versions of assets closer to the user, but these didn’t solve the root problem. Complexity increased. Infrastructure footprints expanded. Vendors profited.

The shift might have persisted indefinitely—except for mobile. Cellular networks imposed harsh bandwidth and latency constraints. Users on phones couldn’t afford to reload full pages. Even aggressive CDN use couldn’t mask the inefficiency.

This pressure birthed the Single Page Application (SPA). SPAs changed the model. The interface shell was delivered once, then retained and reused. From then on, only data travelled. The browser handled rendering. This approach was responsive, efficient, and scalable.

And it rewrote the server’s role. Servers no longer rendered interfaces—they exposed data. The model flipped: rendering-first became API-first. A foundational transformation in web architecture.

### Recommendation

Decentralise rendering. Adopt edge-rendered or SPA-based designs wherever speed and autonomy are important. Let the server serve data, not HTML. Use server-side rendering only where necessary—such as for search engine indexing, content-heavy publishing, or heavily regulated environments.

## Tier Drift

Tier drift occurs when systems no longer respect the original separation of concerns between storage, logic, and presentation. These boundaries were not arbitrary—they enabled testability, scalability, and long-term substitution. But beginning with the collapse of logic into the database and presentation into the server, these separations were progressively weakened. Over time, expedient tools, full-stack frameworks, and time pressure led to increasingly blurred responsibilities across tiers.

Rendering logic has appeared in middleware. Authentication has shifted into frontend components. Workflow enforcement now occurs in databases or orchestration layers. Each shortcut made in the name of delivery speed or technical convenience increases accidental complexity. The more entangled the system becomes, the harder it is to change any one part without unexpected consequences.

This erosion also impacts onboarding. When new developers cannot trace where logic lives, or must learn exceptions to structural rules, teams slow down and the risk of failure increases. Poor modularity hampers optimisation, reuse declines, and maintenance becomes reactive rather than principled.

**Recommendation** Restore tier discipline. Clearly document the responsibilities and boundaries of each system layer. Resist convenience-driven consolidation of concerns. Use code review and deployment processes to enforce these boundaries. Design systems with evolvability and comprehension in mind—not just speed to launch.

## SaaS Platform Architecture (and Why It Still Matters)

The dominance of SaaS has changed how most people think about architecture. From the consumer’s perspective, the product is just a service—it works, or it doesn’t. But this apparent simplicity hides complexity. Underneath the interface lies a platform built using patterns and constraints chosen by the vendor. And those design choices still affect us.

When government or enterprise systems adopt SaaS platforms, they don’t get to shape the architecture—but they do inherit its consequences. If a vendor’s platform blends tiers, centralises logic, or hardwires workflows, those patterns become embedded in the system’s behaviour. If extensibility was not a design priority, it becomes nearly impossible to retrofit later.

This is why architectural discipline still matters—even when you’re not the one writing the code. Buying a system is effectively outsourcing your development team. Their choices, practices, and philosophies will shape your organisation’s capability for years to come.

### Recommendation

Treat SaaS procurement as an architectural decision. Demand architectural transparency. Ask vendors where logic resides, how state is managed, and how data is accessed. Require published reference architectures. Prefer platforms that preserve separation of concerns and expose modular APIs. SaaS can offer speed and efficiency, but only when its foundations are sound.

## Microservices –Logic Dropped to the Infrastructure Tier

The rise of microservices stemmed from a legitimate operational problem: the challenge of coordinating deployments across large, fast-moving teams. Amazon’s now-famous internal memo proposed a model of small, autonomous teams working on services that could be deployed independently. For organisations with multiple product lines, disjointed funding models, or independently managed release cadences, this separation of deployability made sense.

However, this architectural model was rapidly abstracted into a generic design principle. Vendors and consultants began promoting microservices not as a strategic deployment solution, but as a default system structure—regardless of whether the underlying organisational or operational need existed.

The result was widespread over-adoption. Systems that could have been simple, modular applications became tangled meshes of interconnected services—each requiring its own orchestration, monitoring, security, state handling, and operational support. This complexity multiplied points of failure, increased configuration drift, and made reasoning about the system's overall behaviour significantly harder.

Cloud vendors had every incentive to promote this model. Microservices consume more infrastructure: they generate more network traffic, require more logging and tracing, and involve more compute instances. What was sold as agility became a revenue model.

Microservices are not inherently bad—but they are not free. They impose real costs in terms of observability, coordination, latency, and resilience. They only make sense when the system’s operational context—particularly deployment independence—demands it.

### Recommendation

Use microservices only when operational deployment independence is essential—such as when multiple teams, departments, or jurisdictions work on the same programme but must release on different cadences. In most cases, prefer modular monoliths that enforce internal boundaries but retain coherence, testability, and deployability. Don’t mistake a deployment strategy for a system architecture. Design for simplicity first; fragment only when justified by real delivery constraints.

## TODO: Microservices Mania

Amazon’s famous memo—advocating small teams, independent services, and internal APIs—was a landmark moment. It birthed the modern microservices movement. But what worked for Amazon’s scale and incentives didn’t translate well to ordinary enterprises.

Cloud vendors encouraged the shift. Each microservice, run independently, consumed infrastructure: compute time, monitoring, logging, orchestration. This wasn’t free. It was profit.

For organisations without the discipline to manage inter-service contracts, deployment pipelines, and operational boundaries, microservices introduced sprawl. Integration multiplied. Testing slowed. Debugging became forensics.

Worse, systems became unportable. A business built on dozens of cloud-native functions found it couldn’t leave. Vendor lock-in was no longer just a risk; it was a fact.

The alternative—moduliths—gains strength. Modular monoliths enforce clean boundaries internally, but package logic coherently. They’re deployable, testable, comprehensible. They scale not by fragmenting, but by organising.

Recommendation

…

# Interface Layer Design Errors

## DTOs and Unversioned APIs

...blahs...

### Recommendation

….blah…

# Data storage Layer Design Errors

## Database (Schema) First

The industry has undergone a long evolution in the way it treats the data storage within system design. The earliest approaches were system centric, with storage a back system. When SQL came in decades later, it shifted to being database-centric—where schemas and table structures dictated application entities and by that behaviour. This gave rise to the notion of "Database First," in which the system was designed around data storage rather than use case.

Later, "Model First" attempted to restore some balance by allowing domain models to inform the data structure. But it wasn’t until the emergence of "Code First"—especially through modern Object-Relational Mapping (ORM) tools—that system design could once again be driven by domain logic. Domain-Driven Design (DDD) patterns reinforced this shift, encouraging the modelling of user needs and business meaning first, and delegating storage to serve that design.

Despite this, the idea still lingers that if the database is wrong, the system is wrong—and therefore that the database should be the first object of understanding, visibility, and access. But this logic is flawed. A database is not the system. It is only one of many components. You wouldn’t design a system around its external cache, file store, or mapping engine. The database should be no different.

The best design starts from user intention and domain purpose. It identifies the system’s domains, then defines schemas and entities to support those. Schemas serve persistence—not access. Data is not defined by how it is stored, but by what it means and what it enables.

Moreover, human-readable access to a database is not a virtue—it is a failure of interface design. When human users are forced to read raw tables, it's a signal that the system has not exposed proper APIs or queryable interfaces. Every time someone is told to "go check the database," it points to a design that abandoned abstraction.

### Recommendation

Treat the database as a storage component—not as the authoritative interface of your system. Design from domain logic outward. Expose application functionality through well-defined APIs. Use schemas to persist the system design—not to drive system design. A well-designed system should never require a human to navigate its database for operational insight. If users still do, the system is unfinished.

## Data Schemas as Interfaces

Relational schemas were never meant to be user-facing. They are internal representations, designed for machine efficiency, not users efficiency. Yet many legacy systems present them as the default interface: for reports, for integrations, even for direct user queries.

This collapses abstraction. Change becomes dangerous. Every index, join, or type adjustment risks breaking external dependencies. It becomes impossible to evolve.

APIs are the correct boundary. They present stable contracts, enforce validation, and shield internal change. The schema should be invisible. Only the interface matters.

### Recommendation

…

## Logic in Stored Procedures

Stored procedures gained traction in the 1990s, especially in banking, as a way to enforce data integrity close to the source. This made sense when network latency was high and applications were relatively thin. But embedding business logic in the database created duplication, hidden rules, and brittle systems.

Application developers found that logic had to be tested in both the application and the database. Synchronising this logic became expensive, and small inconsistencies led to large failures. Worse, it obscured ownership. Who maintained the logic—the application team, or the DBAs?

Eventually, database vendors proposed simplifying systems by consolidating logic inside the database. This appeared cost-efficient but violated separation of concerns. It also surrendered control: logic was now owned by the vendor, not the system owner. When systems were migrated, rewritten, or scaled out, they discovered the true cost of centralising logic in a proprietary tier.

### Recommendation

Best design restores the application layer as the place for rules, validation, and workflows. The database remains a dumb performant record-keeper—not an enforcer.

6. Data Schemas as Interfaces

Relational schemas were never meant to be user-facing. They are internal representations, designed for machine efficiency, not users efficiency. Yet many legacy systems present them as the default interface: for reports, for integrations, even for direct user queries.

This collapses abstraction. Change becomes dangerous. Every index, join, or type adjustment risks breaking external dependencies. It becomes impossible to evolve.

APIs are the correct boundary. They present stable contracts, enforce validation, and shield internal change. The schema should be invisible. Only the interface matters.

## CRUD

Worst.thing.ever. Catchy phrase, first popularised by xxx, but really worst.

Because Delete is only *one* State of Information.

Information as a science discusses other states:

Creation, Collaboration, Reviewing, Rejected, Accepted, Published, Updated, Mreged, Retired, Restored, Deleted. Only the last one is a physical State change.

So from a database perspective, which does physical change, Deletion is distinct operation, while all the State changes come under Update.

Except that developers are not necessarily either Db specialists or Information specialists therefore don’t know this, so they misapply a db tier pattern - CRUD – in the application/ logical tier.

### Recommendation

I humbly suggest that avoiding using the CRUD and instead first reach for CRUS (Create, Retrieve, Update, State).   
And I humbly recommend they know the UI acronym BREAD – Browse Read, Edit Add, Delete, being careful that the UI delete translates to a Logical change of state to Removed, that is not implemented in the database as a physical deletion, only a State update.

# Application Logic Layer Errors

Application-level mistakes often go unnoticed because they emerge from what seems practical in the short term: simplifying user interfaces, saving development effort, or accelerating time to market. But the design of the application layer—where business logic is shaped, user interaction is framed, and system states are most frequently manipulated—has deep implications.

This is the layer most accessible to developers and often the most customised. It’s where configuration becomes hard-coded, interfaces become opinionated, and decisions made for developer convenience quietly become operational constraints. Many of the most persistent design failures are born not of malice or incompetence, but of familiarity, outdated habits, or poorly understood patterns passed between development teams.

## Record Deletion

Record deletion began as a cost-saving mechanism. Storage media were expensive and slow, and data deemed old or inactive was moved to tape and later erased. This practice mirrored the physical world of shifting paper archives to cheaper, off-site storage facilities.

Over time, however, deletion lost its physical parallel and became conflated with ideas of backup, compliance, and even ethical responsibility. When GDPR regulations emerged, they were interpreted in panic. Entire industries sprang up around 'compliant deletion', promoting erasure as a virtue—even when it disabled operational analytics and crippled longitudinal studies. Deletion was rebranded as responsibility.

But here lies the contradiction: deleting records undermines the very audit trails that systems must preserve. It creates irreversible damage to both operational functionality and analytical value. Think of it as hiring skilled workers to build a load-bearing wall, while another team quietly removes bricks behind them. It may stand for a while, but collapse is inevitable.

Consider education data. If students over a certain again - e.g. 21+7 - are deleted in accordance with inaccurately developed data policies, longitudinal reports show teachers serving classrooms of zero students. The data lies—not by malice, but by absence. The alternative is to shift reporting to a secure data warehouse, but this introduces delay, bureaucracy, and opacity. The data is no longer visible to those who need it; it becomes hostage to process. (We return to the design implications of the data warehouse model in more detail later.)

Compliance must be more precisely therefore correctly applied. The GDPR refers to *personal data*, and to its *erasure*. Not all data is personal, and not all erasure is deletion. Unlinking personal identifiers from data—anonymisation—is sufficient. Disconnecting a structured record from its persona (or assigning it to an anonymous user) preserves its systemic value while honouring privacy law.

In designing for erasure, clarity of language matters. Deletion, Erasure, Removal—these are not synonyms. Nor is all information "personal." Regulation requires precision; so must our systems.

To reinforce clarity, these distinctions are vital not only for legal compliance but for system design integrity. We summarise them here for clarity and also include them formally in Appendix A – Definitions.

* Deletion refers to the permanent technical removal of data from a system such that it is no longer retrievable. It is a system-level act, often irreversible, and typically applied to records at the physical or database layer.
* Removal denotes the act of denying access to a record without necessarily deleting it. Data may still reside within a system but is rendered inaccessible through role-based access control, logical flagging, or detachment from any active user session or profile.
* Erasure is a broader legal term used most prominently in the GDPR. It involves the removal of personal data concerning a data subject, typically requiring deletion or irreversible anonymisation. Its focus is the disassociation of information from identity.
* Anonymisation is the transformation of personal data in such a way that it can no longer be attributed to a specific individual. When properly implemented, anonymised data is exempt from data protection obligations under GDPR.
* Pseudonymisation retains a link to the data subject but obscures identity through a reversible token or substitute identifier. It offers some protection but does not meet the threshold of anonymisation under GDPR.

### Recommendation

Anonymisation is sufficient to not require Erasure deletion. Removal is also enforced by the change to reduced permissions of public anonymous users.

This perspective is strongly reinforced by the Irish Data Protection Commission’s 2019 guidance on Anonymisation and Pseudonymisation. Their paper provides authoritative definitions, conceptual clarity, and legal framing that aligns with the interpretation of how systems should handle personal data without compromising analytical integrity. This source will be used throughout as a foundational reference to support arguments around compliant erasure versus destructive deletion.

## Diagnostics is not Auditing

A persistent misconception in system design is the conflation of diagnostics and auditing. Though both involve capturing information about system activity, their purposes, design requirements, and operational risks are fundamentally different.

Diagnostics are internal. They are designed for maintenance, troubleshooting, and system health monitoring. Diagnostic logs are volatile by nature—they grow quickly, contain granular technical detail, and must be truncated regularly to preserve system performance. For this reason, they should be externalised—stored outside the primary system data store and handled by rolling or time-bound retention strategies.

Auditing, by contrast, is external-facing. It exists to support accountability, traceability, and trust. It answers questions from customers, investigators, or governance bodies. Audit trails must be reliable, immutable, and readable within the system’s security and access framework. They cannot be erased, truncated, or offloaded without compromising integrity.

The confusion becomes dangerous when developers use diagnostics as a catch-all logging strategy. Sensitive information—such as credentials, tokens, or PII—can be accidentally captured in diagnostic logs, and those logs often escape the same security protections as application data. Worse still, some systems fail to keep a proper operational session log at all—undermining both internal security review and external inquiry.

Related to this is the misplacement of metrics. Like diagnostics, metrics are transient. They support automation, alerting, and visual dashboards—but are not intended to represent system state. Metrics should trigger action, not serve as a record of it.

### Recommendation

Use rolling, external storage for diagnostics and metrics—ensuring they are segregated from sensitive data, access-controlled, and purged regularly. Never log secrets. Within the system, maintain a durable, hash-protected session operations log that records user actions and system responses. This in-system audit record must be preserved, queryable, and protected against tampering. Systems without such audit logs are, functionally, black boxes.

# Presentation Layer Errors

## Modal Dialogue Confirmation Patterns

Modal dialogues—those pop-up prompts asking users to confirm deletion, submission, or some irreversible action—are often presented as safeguards. In reality, they are flow stoppers: intrusive by design, they interrupt a user's cognitive process to demand acknowledgment of risk. They exist not to improve design, but to compensate for poor system decisions—particularly the use of irreversible data deletion.

The modal dialogue emerged as a patch to an underlying problem: data operations that cannot be undone. But in modern information systems, this should rarely be the case. As discussed earlier in Record Deletion, systems should not physically delete records. They should retain full auditability and version history. A change in data should result in a new version—not the destruction of the old.

If data changes are versioned and reversible, the modal confirmation becomes largely redundant. The only true justification for modal confirmations lies in operations that send information outside the system—such as email or publishing to a public feed. Even then, as Gmail has demonstrated, there are better options. Gmail queues outgoing messages for a short grace period, allowing users to undo actions without disruptive prompts.

Modal-free design also significantly reduces training overhead. When users feel safe to experiment—knowing they can undo rather than fear irreversible damage—they become more confident, exploratory, and efficient in how they use the system. This boosts productivity, especially in organisational environments where user roles and comfort levels vary widely.

### Recommendation

Avoid modal dialogues for internal state changes. Design systems with versioning and reversibility so that destructive actions are not needed. Use pattern-based reversal instead of interruption—let users undo, rather than beg to confirm. Retain modals only where actions leave the system boundary and cannot be revoked. Fewer interruptions mean better flow, greater trust, and higher productivity.

## Server-Side Rendering

Early server-client systems were built on the model of the mainframe: a powerful server, a dumb terminal.

Early HTML built on this to let the server describe what the client should see, and browsers obediently rendered it. It seemed elegant, even revolutionary. But as systems scaled, this model strained.

Rendering is expensive. Every user action triggered server-side logic, rendering, and transmission of complete pages. To meet demand, servers bulked up. OS vendors benefited—more memory, more CPUs, more licenses.

Meanwhile, the idea of smart clients re-emerged. Single Page Applications (SPAs) reversed the flow: clients cached the logic and layout, and only raw data travelled between server and browser. CDNs distributed the application frame, keeping it close to the user. Latency dropped. Server load stabilised. Security improved.

The lesson is not that server-side rendering was wrong—it solved a real problem. But as a model, it outlived its usefulness. Systems must evolve. Rendering belongs at the edge; data at the centre. Separation matters.

### Recommendation

…

# Application Interface Layer Errors

## Confusion around API Purpose

A common design oversight in modern systems is the conflation of fundamentally distinct types of APIs: those intended for direct user-facing operations and those intended for system integration. This confusion undermines scalability, compromises security, and limits future interoperability.

Operational APIs—what might be better described as Application Operation Interfaces (AOIs)—serve human users interacting through the system’s front end. These APIs operate within the context of a specific session, user, and role. They are scoped to a tenancy or account, and their access is governed by the current user’s permissions.

In contrast, Integration APIs—Application Integration Interfaces (AIIs)—are designed for system-to-system communication. Their purpose is not to respond to single-user interactions but to facilitate bulk data exchange, replication, synchronisation, or coordinated workflows between platforms. These APIs should be designed with efficient querying, stateless interaction, and secure service authentication, not with the overhead and context requirements of interactive sessions.

Where these distinctions are not made, systems become brittle. For example, if a system hosting 3,000 accounts must provide integration to another system, but only exposes user-scoped APIs, the consuming system must emulate 3,000 user sessions—one per account. This is not only operationally infeasible, but also a security risk. Managing, storing, and rotating thousands of user credentials to accomplish what should be a system-level integration is a textbook example of bad architecture.

Moreover, this architectural failure reveals a deeper problem: a system that lacks integration APIs is not designed to interoperate. It resists data portability during operation and obstructs data extraction at end-of-life. It is, in effect, a closed system pretending to be open.

### Recommendation

Design or procure systems that clearly distinguish between user-oriented operation APIs and system-oriented integration APIs. Operational APIs should be session-bound, permission-aware, and tightly scoped to user actions. Integration APIs should support high-volume, multi-tenant data access through secure service credentials, not user impersonation. Evaluate API strategy as a core part of system architecture, not just an implementation detail.

REST-ful URLs

Some rubbish came up in the 2010s about making restful urls. What they were recommending was the url was intuitive

/education/school/123/student/456/  
  
The above has some merit.

Except that it perpetuates the notion that an entity has only one name or route. The above url makes total sense to an American. But no sense therefore is not intuitive to a French or Geman user.

It also embeds the notion that an entity has only one version.

Around the edges of the web came up the notion of permalink to compensate for link rot that eventuated from it. It was actually…the pattern reversed.

Any entity has mutlipe routes to it.   
The current url, in multiple languages,   
And the same for any older one versions of a resource.   
  
Recommendation  
REST-ful Urls is a too simplistic concept for a complex area of identifying resources.   
Design URLs to have versioning and redirection to counter url rot.

## Unqueryable APIs

The failure to implement queryable APIs is among the most expensive, least visible, and most deeply embedded design flaws in modern system interfaces. It is often the result of misunderstanding what REST offers and what it omits.

REST, despite its popularity, is not a standard. It originated as a set of architectural constraints in a doctoral dissertation—not as a practical blueprint for full-featured APIs. It intentionally avoided defining how to handle essential query operations like filtering, sorting, paging, joining, or projecting fields. The result: every REST-based API reinvents these operations differently. Each vendor, framework, or team builds custom endpoints for nearly every use case.

The cost is cumulative. Every analytical or operational use case must be specified, developed, tested, and deployed as a bespoke solution. Consumers of the API face inconsistent designs and unpredictable payloads. Integration becomes brittle. Performance tuning is difficult. And in most cases, access to data is shaped by the limits of a developer’s foresight, not the consumer’s need.

To address this, two main approaches have emerged:

* **OData**: A robust queryable API specification developed by Microsoft and now formalised as an OASIS and ISO/IEC standard. It supports deep filtering, sorting, joins, type introspection, and pagination in a standardised, discoverable way. Its strength is architectural completeness and enterprise readiness. Its weakness is complexity of implementation in smaller or immature development frameworks.
* **GraphQL**: A flexible, language-neutral query language that lets clients define their data needs dynamically. Originally developed by Facebook, it has gained traction in web development for its adaptability. It is widely supported, particularly in scripting and front-end ecosystems. However, GraphQL is not a standard, lacks consistent tooling maturity, and can introduce performance and complexity issues if misapplied.

Neither OData nor GraphQL is a silver bullet—but both represent significant advances over ad hoc REST patterns. Unfortunately, many systems implement neither. They rely on hardcoded, non-extensible REST endpoints that are tightly coupled to their front ends and unsuitable for reuse.

### Recommendation

Implement a queryable API standard as a first-class feature of system design. OData is preferred where standards, introspection, and rich query semantics are required. GraphQL may be used where flexibility or tooling support takes precedence. Avoid bespoke REST filtering conventions unless justified by significant technical constraints and accompanied by thorough documentation.

## Firewall Fetishism

Security by perimeter is a comforting myth. The idea of a moat—a firewall—protecting a vulnerable system is seductive. But real security is internal. A well-designed system should survive even if placed, metaphorically, in a public square.

Firewalls cannot fix:

* plaintext credentials in config files
* unencrypted storage
* over-permissive routes
* lack of session validation
* secret leakage through logs

Security requires embedded design. Secrets must be held in secure stores. Authentication must verify, not transmit, credentials. Logs must be short, session-scoped, and reviewable. Every function must check permission. Every session must expire.

The firewall is a checkpoint—not a cure.

# Conclusion

These errors are not just mistakes. They are habits—embedded in training, tooling, and thought. They persist not because they work, but because they are taught.

The phrase "best practice" has become a shield behind which poor design hides. But what is best for a vendor’s market may not be best for a public system’s lifespan. True design demands scrutiny, history, and humility. It learns not from trends, but from failures.

The science of IT design is not optional. It is overdue. This document is a call to reclaim it.

Appendices

Appendix A - Document Information

Authors & Collaborators

* Sky Sigal, Solution Architect

### Versions

* 1. Initial Draft

### Images

**No table of figures entries found.**

### Tables

**No table of figures entries found.**

### References

* Irish DPC: Anonymisation and Pseudonymisation Guidance, <https://www.dataprotection.ie/sites/default/files/uploads/2019-06/190614%20Anonymisation%20and%20Pseudonymisation.pdf>
* GDPR Article 17: Right to Erasure ("Right to be Forgotten")

**There are no sources in the current document.**

### Review Distribution

The document was distributed for review as below:

|  |  |
| --- | --- |
| Identity | Notes |
|  |  |
|  |  |
|  |  |

### Acronyms

API

: [Application Programming Interface](#Term_ApplicationProgrammingInterface).

DDD

: Domain Driven Design

GUI

: [Graphical User Interface](#Term_ApplicationProgrammingInterface). A form of [UI](#Acronym_UI).

ICT

: acronym for Information & Communication Technology, the domain of defining Information elements and using technology to automate their communication between entities. [IT](#Acronym_IT) is a subset of ICT.

IT

: acronym for Information, using Technology to automate and facilitate its management.

UI

: User Interface. Contrast with [API](#Acronym_API).

### Terms

Refer to the project’s Glossary.

Application Programming Interface

: an Interface provided for other systems to invoke (as opposed to User Interfaces).

Capability

: a capability is what an organisation or system must be able to achieve to meet its goals. Each capability belongs to a domain and is realised through one or more functions that, together, deliver the intended outcome within that area of concern.

Deletion

: Permanent technical removal of a record from the system.

Erasure

: Legal disassociation of data from identifiable persons, often involving deletion.

Domain

: a domain is a defined area of knowledge, responsibility, or activity within an organisation or system. It groups related capabilities, entities, and functions that collectively serve a common purpose. Each capability belongs to a domain, and each function operates within one.

Entity

: an entity is a core object of interest within a domain, usually representing a person, place, thing, or event that holds information and can change over time, such as a Student, School, or Enrolment.

Function

: a function is a specific task or operation performed by a system, process, or person. Functions work together to enable a capability to be carried out. Each function operates within a domain and supports the delivery of one or more capabilities.

Person

: a physical person, who has one or more Personas. Not necessarily a system User.

Persona

: a facet that a Person presents to a Group of some kind.

Personal Data

: Any information that can be used to identify a living person, directly or indirectly.

Removal

: Denial of access or extraction of data from operational view.

Quality

: a quality is a measurable or observable attribute of a system or outcome that indicates how well it meets expectations. Examples include reliability, usability, and performance. Refer to the ISO-25000 SQuaRE series of standards.

User

: a human user of a system via its UIs.

User Interface

: a system interface intended for use by system users. Most computer system UIs are Graphics User Interfaces ([GUI](#Acronym_GUI)) or Text/Console User Interfaces (TUI).

Appendix C – Related Concepts to be Covered

* CTO and EA Dependency on Gartner-Style Sources
* The Myth of the Analytics End-State
* Operational Hubs vs Data Warehousing
* Event Sourcing vs State Mutation
* The Myth of Statelessness
* Dangers of Silent Failures in Distributed Systems
* Role of Operational Audit in Capability Retention
* Identity and Role Separation in Multi-System Environments
* When Not to Use NoSQL
* Why High Availability ≠ High Integrity
* Tier Drift and Anti-Pattern Accumulation
* The Engineering-Science Gap in IT