IT Project Guidance

Design: Rationale for the Use of ORMs

Version: 0.1

Purpose

This document explains why Object-Relational Mapping (ORM) remains a preferred pattern for enterprise and public sector systems. It provides a rationale for adopting ORMs within structured, maintainable, and high-performance environments, while addressing common objections. It supports informed architectural decisions by grounding ORM usage in real-world development, deployment, and governance contexts—specifically within services developed within .NET Core-based for which rationales are also provided.

Synopsis

Object-Relational Mapping (ORM) tools have become central to modern system design, enabling consistent, testable, and maintainable access to relational data. Yet misconceptions persist—especially about performance, visibility, and schema control. This paper clarifies the benefits of ORMs, debunks common criticisms, and shows how mature frameworks like Entity Framework Core (EF Core) enable safe, scalable, and domain-aligned data access. Emphasis is placed on how ORMs support public sector standards for auditability, security, and system evolution—without compromising performance. EF Core is used as a reference example, reflecting its deep integration with the .NET Core platform, which remains one of the highest-performing and most widely adopted environments for government applications.

Contents

Introduction

…

Background

Systems store data for two main reasons: to handle more information than memory alone can hold, and to preserve that data across restarts or power loss. As physical storage has advanced, so too have our methods of structuring it. Relational databases introduced a durable, general-purpose way to represent known entities and relationships. Over time, alternative models emerged—key-value, document, graph, blob, time-series, vector—each serving more specialised or dynamic needs. Yet relational thinking remains foundational, especially where clarity, governance, and interoperability are required.

While relational databases still underpin most enterprise and public systems, both schema definition and data access methods have evolved.

Schema Development Changes

Schema design began with handwritten SQL DDL, which was error-prone and laborious. This gave way to database-first workflows, where schemas were constructed directly in the database and then documented. Although this improved visibility, it failed to solve version control, testing, or automation challenges. It served maintenance stakeholders, but not agile service teams.

Model-first tools attempted to improve this by generating schemas and code from visual diagrams. These soon proved incompatible with modern development practices. They lacked effective support for versioning, were hard to integrate into source control and CI/CD pipelines, and often imposed constraints unrelated to business behaviour. Tooling-first models quickly fell behind DevOps.

Code-first approaches addressed these issues by defining schemas through application code. This brought schema definition, logic, and service behaviour into a single versioned and testable unit, deployable as part of the application lifecycle. It marked a shift in ownership—from centralised database teams to domain-aligned service teams—and enabled systems to be driven by service outcomes, not schema constraints. This also introduced new responsibilities: teams had to manage indexing, constraints, and data lifecycle within their own services.

Data Querying Changes

Querying data also changed. Early systems relied on handwritten SQL—powerful but repetitive and fragile. Later frameworks aimed to standardise access but often missed the mark. By the early 2000s, ORMs had matured into a practical solution for abstraction, consistency, and maintainability. Object-Relational Mapping (ORM) tools allow structured data to be accessed via familiar object models. They reduce boilerplate, enforce

parameterisation, and align well with Domain-Driven Design principles, the most mature and value-based approach to development of resilient, maintainable and improvable enterprise grade services.

Security played a key role in ORM adoption. By enforcing parameterised queries, ORMs significantly reduced SQL injection risk—crucial in systems handling personal, financial, or legally protected data. But convenience and safety were not the only drivers. ORMs also supported how modern teams work: service orientation, testability, continuous integration, and automation.

As ORMs matured, they expanded to cover schema generation and migration management. This convergence of abstraction, structure, and deployment transformed them into the integration layer between application code and data persistence. The shift from database-first to service-first became not just viable but preferable. Services could now be designed around business needs—not just table layouts.

Common Objections to ORMs

Some resistance to ORMs persists, typically from roles rooted in legacy practice—such as database administrators or architects who favour centralised control. Their objections often centre on maintainability, visibility, and performance.

A common concern is visibility: that ORMs obscure the underlying schema or produce SQL that is less readable than hand-developed code. However, this is not relevant: requirement to use SQL often signals a deeper problem—a lack of clear service boundaries. Systems should expose services and APIs, not invite ad hoc database access. If raw SQL is the only interface, the system design is incomplete.

Others argue that schema design should favour maintainers, not developers. But designing for the application code and potentially API layer actually improves maintainability: it enables testability, encapsulation, and version control. The discipline of maintaining schema is not lost—it is reallocated. Developers define structure as part of the domain model, while data specialists provide targeted expertise on indexes, constraints, and performance tuning.

The belief that direct SQL is inherently faster also persists, but is rarely correct. Performance improvements come from proper indexing, reduced round-trips, and understanding execution plans—not bypassing abstraction. ORMs, when used well, expose exactly the structures that allow this tuning. Dropping to raw SQL reintroduces risk without guaranteeing performance.

Finally, some object that schema evolution via code creates risk. Yet unmanaged databases are riskier still. Code-first migrations are testable, versioned, and automatable. This is precisely what makes them safer than manual changes or undocumented scripts.

Current ORM Capabilities

Modern Object-Relational Mapping (ORM) frameworks offer far more than simplified query syntax. A mature ORM provides a complete bridge between code and database — enabling developers to define, evolve, and interact with data structures using domain models, without forfeiting the rigour of database engineering.

At its core, a mature ORM allows entity manipulation through objects rather than raw data access. Queries are constructed as expressions over entities — using code constructs that mirror the domain model rather than relying on strings or manual joins. These queries can express filtering, ordering, projection, joining, aggregation, and grouping — with the same depth and nuance available in hand-written SQL, but with greater readability and less room for error. This improves developer productivity and enforces query safety by design — eliminating SQL injection risk through parameterisation and automatic escaping.

A mature ORM also supports full schema definition. Developers can configure key strategies (including GUIDs, composite keys, and database-generated sequences), relationships (one-to-many, many-to-many with join entities), indexing, and field constraints such as length, precision, or nullability. All of this is done through code — using patterns that are testable, repeatable, and portable across environments.

ORMs today are typically database-agnostic. The same entity model can target different back-end engines (e.g. SQL Server, PostgreSQL, SQLite) with minimal change. This portability is valuable not only for deployment flexibility, but for testing, containerisation, and future-proofing system infrastructure.

At the same time, mature ORMs do not conceal the underlying database engine. They provide hooks for advanced scenarios — such as invoking stored procedures[1], leveraging database-specific features (like temporal tables or filtered indexes), or integrating with custom scripts where needed. This allows teams to benefit from abstraction without being limited by it.

Perhaps most significantly, modern ORMs manage schema evolution. They support versioned migrations — tracking structural changes over time, applying them incrementally, and generating SQL scripts for review or automation. Where changes require data movement or temporary scaffolding (e.g. splitting columns, introducing new constraints), a migration pipeline allows both schema and data to be modified safely, predictably, and traceably. This enables teams to maintain live systems through controlled change, not through manual database surgery.

[1] However, use this sparingly as the requirement of SP's often indicates larger design issues.

These capabilities combine to make a mature ORM not just a tool for convenience, but a critical layer for aligning domain logic, data structure, and deployment safety — at the scale and complexity required by modern systems.

Why mention Entity Framework

It is difficult to discuss Object-Relational Mapping (ORM) practices entirely in the abstract. While the principles of schema modelling, relationship design, and migration management are broadly applicable, the implementation details are always shaped by the capabilities and constraints of a specific tool. For this reason, where needed, this guidance refers to Entity Framework Core (EF Core) as the working example.

EF Core is referred to because it has the capabilities, and therefore meets a specific set of needs relevant to public sector systems, including performance at scale, tooling maturity, developer familiarity, and alignment with secure, maintainable deployment practices. It enables developers to model domains in code, define schema structure, manage migrations, and access data in a single, consistent framework that integrates tightly with the .NET ecosystem already common across many enterprises and government systems.

.NET Core

> *While this document is focused on the principles and value of Object-Relational Mapping, the examples provided draw from the .NET Core environment, using Entity Framework Core. The reasons for this are covered in detail in the companion guidance paper Evaluating the Use of .NET Core for Enterprise-Grade Systems. That paper discusses performance, maturity, and supportability in full. For this document, the relevant .NET Core sections are included only to clarify why EF Core is used as the reference implementation—readers do not need to refer to the other paper unless seeking platform-level detail.*

The .NET Core runtime is consistently among the fastest general-purpose platforms available. Independent benchmarks show that .NET Core handles approximately 80,000 HTTP requests per second under load—roughly twice as many as Node.js (~40,000), nearly three times more than Python with FastAPI (~30,000), over five times that of PHP frameworks like Laravel or Symfony (~15,000), and slightly ahead of Java Spring Boot (~70,000). Go achieves the highest raw throughput (~100,000), but doing so typically demands greater manual effort to meet the same standards of security, testability, and operational maintainability.

In practical terms, .NET Core offers an optimal balance: near-top-end performance with significantly lower delivery overhead. This makes it particularly suitable for government-grade systems, where operational consistency, cost-effective scaling, and long-term maintainability are as important as throughput. It supports asynchronous programming

natively, reducing resource contention at scale, and integrates cleanly with structured development practices, identity models, and deployment pipelines.

EF Core, as a native component of the .NET ecosystem, inherits these performance characteristics while enabling schema evolution, data modelling, and secure access control within the same streamlined platform.

EF Core

EF Core builds on over a decade of ORM evolution. It supports the full set of patterns required in modern development: code-first schema generation, fluent configuration, dependency injection, asynchronous execution, and query composition using LINQ. Unlike simpler ORMs, EF Core includes mature migration tooling, allowing schema changes to be tracked and applied with the same rigour and automation used for application code. This ensures that domain logic, data structure, and infrastructure can all be versioned, tested, and deployed through unified DevOps pipelines.

Its integration with the broader .NET ecosystem is another reason for its selection. EF Core fits cleanly into common development workflows, test frameworks, hosting environments, and identity management models. It eliminates the need for third-party glue code or brittle custom components. It also provides strong defaults for common concerns such as tenancy separation, multitenant indexing, and logging. These benefits make EF Core a natural choice for enterprise-grade systems where traceability, resilience, and maintainability are required.

Other ORM tools remain viable and may offer specific advantages in certain scenarios. Lightweight options like Dapper can deliver faster raw query execution by removing abstraction overhead. Others may appeal through simplicity or fit better into non-.NET environments. However, these trade-offs often come at a cost: reduced integration, weaker support for migrations, more custom code to manage infrastructure concerns, and reintroduced risks—such as SQL injection—if query composition isn’t handled correctly.

In most cases, these marginal gains do not justify the operational risk or long-term complexity. Optimisation, when required, is better achieved through proper indexing, efficient data modelling, and understanding of query execution plans—not by abandoning abstraction. Introducing bespoke persistence layers or custom scripting strategies adds more surface area for bugs, regressions, and security flaws. EF Core, when used properly, already delivers strong performance while supporting the practices that actually lead to scalable and resilient systems.

All modelling principles in this guidance are tool-agnostic. Everything presented here can be implemented using any capable ORM. EF Core is used as the reference not because it is uniquely capable, but because it is complete, stable, and well-suited to the realities of government-grade systems. Other tools are possible, but the burden of proof—and operational cost—rests with those who choose to diverge from a tested and supported standard.

Selecting an ORM: Considerations and Trade-Offs

While the value of using an Object-Relational Mapper in general is now well-established, the choice of a specific ORM should be made carefully. Different tools offer different strengths, trade-offs, and failure modes. The key is not choosing the “best” ORM in the abstract, but selecting the one that aligns with system architecture, team skillsets, governance needs, and lifecycle expectations.

Entity Framework Core (EF Core) is used throughout this guidance as the reference ORM because it is stable, complete, well-supported, and fully integrated with .NET Core. However, other ORMs—such as Dapper, NHibernate, or even hand-written micro-mappers—may be better suited in highly constrained environments or for teams optimising for low-level control.

Lightweight ORMs and Micro-Mappers

Tools like Dapper favour performance by avoiding abstraction. They allow explicit SQL and offer minimal overhead, which can be ideal in read-heavy systems with stable schemas. However, they reintroduce the need for manual mapping, increase the chance of query mismatches, and leave data lifecycle and schema evolution to be managed separately. This often results in tight coupling to the database and brittle migration paths over time. As stated earlier, rarely is the optimisation gained from this aspect – greater gains and less risk and maintenance issues will comes from better analysis of indexing requirements.

Complex ORMs (e.g., NHibernate)

ORMs like NHibernate offer sophisticated features such as caching layers, custom query languages (e.g., HQL), and advanced mapping scenarios. While powerful, they introduce a steeper learning curve, slower runtime performance, and a more complex debugging experience. They could be questioned as trying to do too much in one platform to succeed at any part as effectively as they could otherwise. These tools are generally suited to legacy systems or very specific enterprise requirements, not to modern DevOps-aligned service teams.

Hybrid Approaches

Some teams mix ORMs with raw SQL for edge cases. While this is viable, it must be governed with the upmost caution. The danger is not the presence of raw SQL but its silent spread across layers—undermining abstraction, security, and maintainability. Where raw queries are necessary, they should be clearly encapsulated, tested, and documented—ideally invoked via extension points or raw SQL execution APIs already provided by the ORM.

For Further Discussion: Entity Isolation and DTO Mapping

It is critical to understand that ORM entities are internal constructs, not API contracts. While modern frameworks make it technically easy to expose entities directly from services—especially during early development or proof-of-concept work—this practice is wholly inappropriate for production-grade, enterprise systems.

In any serious application architecture, entities must be mapped to Data Transfer Objects (DTOs) before being exposed through service APIs. This mapping serves multiple essential purposes:

- **Security**: Internal entities often expose far more structure and metadata than is intended or safe to disclose. Exposing them risks leaking internal logic, relationships, or sensitive fields that were never meant for external consumption.
- **Usability**: Consumers of a service often require shaped, simplified, or enriched data to match their domain needs. DTOs enable tailored responses without overloading consumers with irrelevant or technical structure.
- **Maintainability**: Mapping to DTOs creates a protective boundary between internal data models and public interfaces. This allows internal entities to evolve over time—reflecting changes in business logic, optimisation, or system structure—without breaking downstream consumers who rely on the contract remaining stable.

This separation becomes even more important in public sector and enterprise systems, where services have long operational lifespans and serve multiple external consumers. Unlike internal teams, external consumers cannot be expected to synchronise their upgrades with internal development cycles. Their budgets, schedules, and strategic goals differ—and often conflict—with those of the host agency. A minor schema change, if exposed directly, can cascade into downstream system failures that could have been avoided with disciplined interface design.

Frameworks like AutoMapper can be used to streamline the transformation between entity models and DTOs, especially when working with EF Core. However, the practice itself is non-negotiable. API contracts must be explicitly designed, independently versioned, and documented. Entities are for the system. DTOs are for the consumer.

Additionally, DTOs should not be treated as static or inert. Services should expose queryable endpoints—either through formal standards like OData or structured query languages like GraphQL. Both allow clients to shape the data they retrieve without requiring bespoke endpoints for each new use case. This removes the need for ongoing cycles of use-case definition, development, testing, and release just to support variant read patterns. Queryable endpoints shift the system from static delivery to dynamic consumption, aligning better with the real-world needs of reporting, analytics, and integration scenarios.

These patterns are often misunderstood. For example, both OData and GraphQL have been unfairly characterised as security risks—particularly in relation to potential denial-of-service through complex or long-running queries. In practice, this is trivial to mitigate. In the case of OData, a single line of configuration can cap the depth and complexity of queries. The actual risk lies not in the tools or protocols - but in failing to apply the most basic constraints.

To our knowledge, only .NET Core currently supports true standards-aligned queryable DTOs in a secure, governable, and production-ready way. It remains the only major platform that has successfully integrated dynamic querying into its mainstream service development model without requiring third-party extensions or architectural compromise.

This capability, when properly understood and applied, offers one of the most powerful tools for scaling service delivery—without scaling the budget.

Conclusion

The main risks in ORM selection arise not from tool limitations but from misalignment with system architecture and team discipline. The best ORM is the one that supports versioned schema evolution, query abstraction, data safety, and integration with the platform’s lifecycle. EF Core is used here because it delivers these qualities within .NET Core. Other tools may be valid choices—but the operational burden of proving and maintaining those alternatives rests with the team that selects them.

Appendices

Appendix A - Document Information

Authors & Collaborators

- Sky Sigal, Solution Architect

Versions

0.1 Initial Draft

Images

Tables

References

There are no sources in the current document.

Review Distribution

The document was distributed for review as below:

Identity	Notes
Russell Campbell, Project Manager	
Duncan Watson, Enterprise Architect	
Carl Klitscher, Solution Architect	
Gareth Philpott, Solution Architect	
Vincent Wierdsma, Lead Developer	

Chanaka Jayarathne,
Senior Developer

Audience

The document is technical in nature, but parts are expected to be read and/or validated by a non-technical audience.

Structure

Where possible, the document structure is guided by either ISO-* standards or best practice.

Diagrams

Diagrams are developed for a wide audience. Unless specifically for a technical audience, where the use of industry standard diagram types (ArchiMate, UML, C4), is appropriate, diagrams are developed as simple “box & line” monochrome diagrams.

Acronyms

API : Application Programming Interface.

GUI: Graphical User Interface. A form of UI.

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

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

UI : User Interface. Contrast with API.

Terms

Refer to the project’s Glossary.

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

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) or Text/Console User Interfaces (TUI).