IT Project

Guidance: Development – Rationale for use of .NET Core for custom enterprise grade services

Version: 0.1

Purpose

To justify the selection of .NET Core as the preferred platform for custom system development in public and enterprise contexts. The paper evaluates its performance, scalability, supportability, and economic impact in comparison to no-code platforms and scripting-based stacks.

Synopsis

This paper outlines the architectural, operational, and economic reasons for selecting .NET Core over alternative technologies for developing large-scale, maintainable, and adaptable systems. It addresses misconceptions around no-code platforms and scripting languages, and affirms that with disciplined practices, .NET Core supports robust, cost-effective solutions that align with public sector delivery expectations.

Contents

Purpose 1
Synopsis 1
Contents 2
Introduction 4
Advantages and Disadvantages 5
Advantages 5
Maturity and Supportability 5
Portability 5
Performance and Memory Efficiency 6
Infrastructure and Cost Efficiency 6
Ecosystem and Development Pipeline 7
Certification and Operational Readiness 7
Disadvantages and Constraints 8
Opinion 8
Startup performance 8
Size 8
Speed of development 8
Comparisons 9
Comparison to No/Low Code 9
Hidden Costs and Dependency Risks of No-Code Platforms 9
Advantages of Custom Development with .NET Core 9
Comparison to Scripted Languages 9
Scope of Front-End Development 10
Depth of Knowledge in Full-Stack Development 10
Performance Considerations 10
Security Implications 11
The Critical Role of Analysis and Leadership 11
Conclusion 12
Appendices 14
Appendix A - Document Information 14
Versions 14
Images 14
Tables 14
References 14
Review Distribution 14
Audience 14
Structure 14
Diagrams 15
Acronyms 15
API 15
GUI 15

ICT 15
IT 15
UI 15
Terms 15
Application Programming Interface 15
User 15
User Interface 15

Introduction

The .NET Core runtime is among the fastest general-purpose application platforms currently available. Independent benchmarks from TechEmpower and others show that .NET Core consistently handles approximately 80,000 HTTP requests per second under load. By comparison, Node.js reaches around 40,000, Python with FastAPI approximately 30,000, PHP frameworks such as Laravel 15,000, and Java Spring Boot about 70,000. GoLang can exceed 100,000 but demands more manual work to match the operational standards achieved by .NET Core out of the box.

For public sector systems, this performance makes a measurable difference. Fewer resources are needed to serve a larger user base, meaning reduced infrastructure cost and improved scalability. This document outlines why .NET Core is not merely a viable platform for government and enterprise-grade systems, but one of the most cost-effective, evolvable, and maintainable frameworks available—while also identifying its disadvantages in order to present a complete and supported position.

Advantages and Disadvantages

As a runtime and development platform, .NET Core offers a set of characteristics well-suited to enterprise and public sector systems. Its strengths lie in performance, supportability, deployment flexibility, and integration with modern engineering practices. Like any platform, it also presents limitations—typically where extreme optimisation, platform-specific dependencies, or language-neutral tooling are required. The following sections summarise these key advantages and disadvantages in the context of government-grade system development.

Advantages

.NET Core provides a modern, high-performance environment that balances delivery speed with operational stability. It is designed to meet the demands of contemporary development—supporting cross-platform deployment, cloud readiness, containerisation, and service-oriented design—while maintaining the rigour and structure required for large-scale, long-lived systems. Its strong tooling, comprehensive documentation, and widespread industry adoption mean that teams can move quickly while maintaining strong governance and supportability. .NET Core encourages clean architecture, testability, secure-by-default patterns, and strong separation of concerns, all of which contribute to predictable system behaviour and long-term adaptability.

Maturity and Supportability

.NET Core is part of the .NET family, which has existed for over 20 years. It was rebuilt as open source and cross-platform in 2016. It now supports Windows, Linux, and macOS, and its development model is fully transparent under the .NET Foundation. Microsoft continues to invest in the platform, with new LTS releases every two years and strong backwards compatibility. Long Term Support guarantees patching and security coverage, while optional interim releases allow for innovation.

This maturity brings benefits such as predictable upgrade paths, documentation suited to enterprise environments, and compatibility with formal development and accreditation practices. It supports existing team skills from the .NET Framework era while providing modernised tooling, compiler performance, and platform flexibility.

Portability

In practical terms, code portability has shifted from being a major system objective to a quality of integration points. While ISO-25010 once emphasised portability, ISO-25012 now acknowledges that most systems are cloud-hosted and tightly integrated with third-party services such as storage, identity, AI, and search. Once integrated with a specific cloud provider’s suite, service portability is limited regardless of language.

Nonetheless, .NET Core allows services to be written for any target platform and compiled for Linux or Windows hosts. Containerisation and cross-platform CLI tools mean services can be moved between hosts or environments with minimal friction. Open source transparency ensures no lock-in at the runtime or library level.

Performance and Memory Efficiency

.NET Core offers near-optimal performance among PCode platforms while preserving a high degree of abstraction. Its garbage collector has significantly improved over successive versions, especially with the introduction of server-optimised GC and Span memory access patterns.

Where early .NET implementations suffered from memory bloat and inefficient GC pauses, recent benchmarks show far better memory retention and throughput under load. Compared to interpreted or scripted runtimes, .NET Core requires fewer threads, consumes less memory per request, and delivers consistent latencies. Startup time remains slower than scripted platforms (e.g. Node.js, Python), especially for cold-start scenarios in serverless environments. However, technologies like ReadyToRun, tiered compilation, and profile-guided optimization reduce this impact. For high-throughput APIs and services, the trade-off strongly favours .NET Core.

Infrastructure and Cost Efficiency

A key advantage of .NET Core is its ability to deliver high performance using fewer compute resources. Benchmark scenarios simulating 1 million daily users show .NET Core requiring 30–50% fewer execution environments compared to alternatives like Java, Node.js, or PHP—assuming clean, asynchronous, well-architected code. In hosted environments like Azure, this translates directly into cost savings through smaller VM instances, fewer required replicas, and more efficient use of threads and memory.

Concrete modelling for 1,000,000-user loads over a 10-year span shows a clear cost difference based purely on runtime selection. PHP-based services under load can require over NZD $11,900/month in compute costs, whereas .NET Core or Go can sustain the same throughput at closer to NZD $7,500/month. This equates to approximately NZD $528,000 saved over a decade—before factoring in staffing, support, or user growth. This figure is based solely on compute resource consumption under typical load; additional savings or losses would depend on design discipline and operational maturity.

However, those savings depend on architectural discipline. Real-world systems—especially those built under pressure or by transient contractors—often suffer from blocking logic, overuse of synchronous I/O, poor caching, and inefficient business logic in the application layer. In such cases, even .NET Core may require twice as many servers, eroding savings. Conversely, well-built PHP can outperform poorly built Node.js or Python stacks. This reinforces a critical point: **framework choice matters, but system design discipline matters more**.

Even with a 30–50% real-world overhead added to baseline figures, .NET Core consistently delivers between NZD $500,000 and $750,000 in infrastructure savings over

10 years for high-volume systems—making it not only performant but economically defensible. These figures do not include staff availability, maintainability, or integration cost—which further favour mainstream, mature frameworks like .NET Core.

While additional savings are often hypothesised by shifting to Functions-as-a-Service (FaaS) models such as Azure Functions, the reality is more complex. Any theoretical reduction in compute cost is typically offset—and often exceeded—by the orchestration overhead, increased complexity of reasoning about system flow, elevated skill requirements, and the fragmentation of development responsibilities. These platforms demand high design clarity and careful lifecycle management to avoid runaway sprawl. In most public-sector contexts, they introduce more confusion than benefit and erode transparency around ownership, debugging, and delivery.

.NET Core allows for scalable hosting on traditional infrastructure or containers, with clarity of execution and simplified operations—avoiding unnecessary complexity while still offering optional access to advanced platform features.

Ecosystem and Development Pipeline

.NET Core's ecosystem is suited for regulated environments. Libraries used in production are vetted and versioned under Microsoft's governance. Supply chain integrity is actively monitored, and the .NET CLI warns on insecure packages. Unlike free-for-all code library ecosystems (e.g. Node.JS & NPM), the .NET environment parts prioritises security, compatibility, and observability.

Microsoft's DevOps toolchain (Azure DevOps, GitHub Actions, Application Insights) integrates directly with .NET. Identity, caching, ORM, and telemetry libraries are supported natively. EF Core allows for schema-controlled data access with fine-grained permissions, making it ideal for public sector data governance.

Certification and Operational Readiness

.NET Core enables system certification under government requirements by supporting standardised identity management, auditable data access, and platform hardening. It integrates with enterprise logging, auditing, and secrets management. Key public sector priorities such as audit trails, multi-tenant isolation, accessibility support, and data export requirements can be addressed within the .NET ecosystem without needing third-party scaffolding.

Using a Domain-Driven Design approach, .NET Core supports decoupled development with layered architecture. This enhances testability, aligns with capability-based procurement models, and simplifies long-term maintenance. Versioning, migrations, and system upgrades can be planned and executed in line with architectural governance.

Disadvantages and Constraints

Despite its many strengths, .NET Core is not without trade-offs. As a strongly typed, structured environment, it demands more up-front design discipline and familiarity with object-oriented patterns. Compared to dynamic languages or no-code environments, the learning curve can be steeper for teams unfamiliar with architectural concepts such as dependency injection, asynchronous control flow, and layered design. The breadth of available tooling and libraries, while a strength, can also present decision fatigue and risk of misconfiguration if not carefully governed. And while cross-platform, .NET Core's most mature hosting and diagnostic tools remain oriented toward the Azure ecosystem, which may limit perceived flexibility in some infrastructure strategies. These considerations do not outweigh the benefits, but they must be acknowledged and planned for.

Opinion

One persistent issue with .NET is historical bias. Many still associate it with proprietary Microsoft tooling, Windows-only runtimes, or expensive licensing. In reality, .NET Core is fully open source, licensed under MIT, and runs seamlessly on Linux.

Startup performance

Another disadvantage is startup performance. In serverless and container cold-start situations, .NET Core lags behind lightweight interpreters. Developers must be aware of AoT strategies, dependency trimming, and startup code minimisation to mitigate this.

Size

Framework size and package bloat are valid concerns. Careless use of NuGet packages can produce large deployment units. However, these can be mitigated through linker configuration and package analysis.

Speed of development

Finally, rapid evolution in competing ecosystems can lead to perceived slowness in .NET Core innovation. For example, certain experimental web APIs or generative AI libraries may appear first in Python or JavaScript. This lag is intentional. Microsoft prioritises security, compatibility, and lifecycle over adopting every emerging trend. In regulated – or simply *responsible* - environments, this results in better long-term value.

Comparisons

Comparison to No/Low Code

Enterprise architects, especially those who have witnessed the pitfalls of poorly executed custom development projects, may gravitate toward no-code platforms as a seemingly safer option. However, it's crucial to assess the long-term implications of this choice, particularly concerning costs, flexibility, and organizational autonomy.

Hidden Costs and Dependency Risks of No-Code Platforms

No-code platforms often operate on per-user, per-month pricing models. As organizations scale and user counts increase, these costs can escalate significantly. For instance, some platforms charge substantial monthly fees plus additional costs per user, which can become financially burdensome for enterprises with large user bases.

Furthermore, the specialized skills required to maximize the potential of no-code platforms are frequently scarce and command premium rates. This scarcity often leads organizations to rely heavily on external contractors. Due to project demands, these contractors may focus more on rapid development than on comprehensive knowledge transfer, leaving the organization dependent on external expertise for ongoing maintenance and future enhancements.

Additionally, many no-code platforms are proprietary, leading to potential vendor lock-in scenarios. Transitioning away from such platforms can be challenging and costly, as customizations and data may not easily transfer to other systems .

Advantages of Custom Development with .NET Core

In contrast, .NET Core, part of the .NET ecosystem established over two decades ago, offers a mature, open-source framework supported by a vast community of developers. This extensive support network ensures a readily available talent pool, facilitating easier recruitment and reducing dependency on specialized, high-cost contractors.

Custom development with .NET Core allows organizations to tailor solutions precisely to their needs, ensuring that systems evolve in alignment with organizational changes rather than forcing the organization to adapt to the limitations of a third-party platform. By adhering to established frameworks and best practices, organizations can achieve long-term sustainability and flexibility.

Comparison to Scripted Languages

In the realm of full-stack development, a prevalent belief is that scripting languages like JavaScript, particularly when used with platforms such as Node.js, offer superior speed and efficiency. While these technologies have their merits, it's essential to recognize the broader context of system development and the depth of knowledge required to build robust, secure, and scalable applications.

Scope of Front-End Development

Front-end developers primarily focus on crafting user interfaces and enhancing user experiences. Their expertise lies in creating responsive designs, implementing interactive elements, and ensuring cross-browser compatibility. While these skills are vital, they represent just one layer of the application stack. The development of a complete system encompasses much more, including server-side logic, database management, authentication, authorization, and infrastructure considerations.

Depth of Knowledge in Full-Stack Development

Building a comprehensive application requires a profound understanding of various components:

- **Protocols and Standards**: Knowledge of HTTP/HTTPS, RESTful principles, WebSockets, and other communication protocols is crucial for ensuring efficient and secure data exchange.
- **Security**: Implementing measures to protect against threats such as SQL injection, cross-site scripting (XSS), and cross-site request forgery (CSRF) is essential. This involves understanding both client-side and server-side vulnerabilities and their mitigations. moldstud.com
- **Infrastructure and Deployment**: Familiarity with server configurations, load balancing, containerization (e.g., Docker), orchestration tools (e.g., Kubernetes), and continuous integration/continuous deployment (CI/CD) pipelines is necessary for deploying and maintaining applications at scale.
- **Data Management**: Designing efficient database schemas, understanding data normalization, and ensuring data integrity and consistency are foundational to backend development.

Front-end developers may not typically engage deeply with these areas, leading to a potential gap in understanding the complexities involved in full-system development.

Performance Considerations

While Node.js, leveraging JavaScript, is renowned for its non-blocking, event-driven architecture that excels in handling asynchronous operations, it may not be the optimal choice for CPU-intensive tasks. In contrast, frameworks like .NET Core are optimized for high-performance scenarios, effectively managing multithreading and delivering superior performance in compute-bound operations.

Security Implications

Security is a paramount concern in application development. Front-end developers often focus on client-side security, but server-side security is equally critical. Backend developers play a vital role in implementing robust security protocols, managing authentication and authorization, and safeguarding sensitive data. A lack of comprehensive security knowledge can lead to vulnerabilities that compromise the entire system.

The Critical Role of Analysis and Leadership

Regardless of the development approach—custom, no-code, or scripting-based—the success of any system depends on rigorous analysis and strong leadership. The foundation of any project must be a shared and accurate understanding of the problem domain, expressed in terms of achievable, bounded contexts. Without this, implementation becomes reactive and piecemeal.

Effective architecture starts with identifying the capabilities a solution must provide, understanding which of those should be automated, and sequencing delivery accordingly. Priorities should follow a deliberate order: securing the system first, enabling stable delivery second, and only then accelerating development. Development leadership plays a key role in keeping the work aligned to the core framework—resisting the temptation to chase unproven technologies, speculative patterns, or reinvented infrastructure.

A disciplined development approach—grounded in formal analysis, clear scope definition, and architectural coherence—offers the best safeguard against both technical drift and delivery failure. It enables teams to move quickly without compromising security, maintainability, or clarity. When applied consistently, this discipline is what turns custom development into a strength rather than a liability.

Conclusion

Choosing a system development approach is a strategic decision. It must be based not only on perceived speed or cost, but on long-term viability, security, maintainability, and the ability to evolve with organisational needs. In this regard, .NET Core stands out as a mature, complete, and high-performance framework that supports disciplined, sustainable development. It enables domain-aligned modelling, secure application patterns, and integration across all major infrastructure and service concerns—without requiring organisations to invent architectures or navigate fragmented ecosystems. It is not just capable—it is dependable.

By contrast, no-code and low-code platforms, while attractive in constrained scenarios, introduce hidden costs and long-term risks. Their licensing models often scale poorly, locking organisations into recurring fees that can rise sharply with usage or policy changes. Their technical constraints can force operational processes to conform to the tool, rather than allowing the tool to support process evolution. More critically, the skills required to customise or operate these platforms are often scarce, expensive, and external—leaving little room for knowledge transfer. As a result, organisations can become hostages to the platform and its specialists.

Likewise, while scripting languages such as JavaScript (Node.js) or Python may enable rapid development of lightweight services or user interfaces, they are often championed by developers who operate on the edges of the system—typically at the presentation layer. These developers bring valuable front-end skillsets but may not engage deeply with the full stack. Protocols, security, orchestration, identity, deployment, and infrastructure concerns often remain out of scope. The belief that scripting environments are inherently faster or better is true only within a narrow slice of system activity. Full-scale systems require predictable behaviour, governed schemas, deep operational understanding, and the ability to enforce consistent standards—areas where general-purpose scripting stacks frequently fall short.

What emerges clearly from experience—and from the failures of past projects—is that the most sustainable, adaptable systems are not those that rely on the simplicity of platforms or the apparent speed of scripts, but those that are designed deliberately, developed with discipline, and anchored in mature frameworks. Custom development within .NET Core, following well-established design and analysis practices, remains one of the most cost-effective and robust approaches available. It supports long-term maintainability without sacrificing adaptability. It empowers teams to evolve systems with confidence, rather than constrain them with caution.

For these reasons, this paper affirms the value of custom development—not as an abstract ideal, but as a disciplined practice undertaken within the guardrails of a platform like .NET Core. When anchored in the deliberate application of sound architectural and development patterns, and supported by mature tooling, widely available talent, and clear strategic alignment, custom development offers a more adaptable, maintainable, and cost-effective foundation. Compared with no-code platforms and scripting-based stacks, it provides greater long-term value, control, and resilience.

Appendices

Appendix A - Document Information

Authors & Collaborators

- Sky Sigal, Solution Architect

Versions

0.1 Initial Draft

Images

No table of figures entries found.

Tables

No table of figures entries found.

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	
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).

Appendix B – Regarding concerns as to Microsoft, the organisation

> *Note:*
> *The following is a summary extracted from the document titled “IT Project Guidance – Rationale for using Microsoft.*

Concerns about Microsoft’s historical dominance are valid but increasingly anachronistic. Over the past 15 years, Microsoft has undergone a structural, cultural, and strategic transformation, shifting from a proprietary-first software vendor to one of the largest contributors to open-source and multi-platform tooling. This appendix aims to clarify this shift and address residual perceptions, especially when comparing Microsoft to other technology providers such as Amazon Web Services (AWS), IBM, and Salesforce/MuleSoft.

1. Change of Leadership, Change of Direction

Microsoft's transformation traces back to the 2014 appointment of Satya Nadella as CEO. Under Nadella, Microsoft pivoted sharply towards openness, cloud-first strategies, and cross-platform interoperability. This is not superficial branding; the company has embraced competitors' platforms and invested deeply in open standards.

Examples include:

- .NET Core (now .NET 8) being open-sourced under the MIT licence
- TypeScript, VS Code, and C# language specifications all available in public repositories
- Windows Subsystem for Linux (WSL), a direct support of Linux-first workflows

2. Open Source and Free Software

Microsoft is now among the top corporate contributors to GitHub. Major development tools are free and open source:

- **.NET Core / .NET 8:** Fully open-source runtime and libraries, with extensive community input
- **Visual Studio Code:** The editor is built on an open-source core (vscode), though the official Microsoft distribution includes proprietary extensions (e.g. telemetry, debugging)
- **PowerShell Core:** Cross-platform, open-source CLI shell
- **DAPR (Distributed Application Runtime):** Open-source microservices infrastructure

Critics sometimes cite proprietary extensions in Visual Studio Code. This concern is technically true: the distributed binaries include Microsoft-maintained plugins for better telemetry, debugging, and language support. However, **fully open distributions** such as VSCodium are available, and the entire plugin ecosystem supports third-party development.

3. Products, Services, and the Subscription Economy

Like Apple, AWS, IBM, Google, and Adobe, Microsoft has shifted from perpetual licences to a services model. This is a global industry trend. Microsoft still offers:

- Free IDEs and frameworks
- Free tiers of Azure (including always-free services for learning and startups)
- Student/educational access to Office and development tools
- Freemium pricing across its stack

Products like Microsoft 365 (formerly Office 365) are not open-source and never were. That is not a monopolistic behaviour—it is a commercial strategy, and users have alternatives like LibreOffice or Google Workspace.

4. Not a Monopoly by Any Metric

The notion that Microsoft maintains a monopoly today is not supported by modern usage metrics:

- **Browser Share (2024):** Microsoft Edge has ~11%, vs Chrome's ~65% (StatCounter)
- **Server OS:** Linux dominates in public cloud workloads (>70% of AWS, Google Cloud)
- **Cloud Market Share (2024):** AWS ~31%, Azure ~25%, Google Cloud ~11% (Synergy Research)
- **IDE/Dev Tools:** JetBrains IDEs, Eclipse, and VS Code split the developer market, with no monopoly

5. Azure vs AWS: A Comparable Landscape

AWS, often perceived as the “open” alternative, employs similar proprietary strategies. Examples:

- **Aurora:** A fork of MySQL and PostgreSQL, with proprietary enhancements only available on AWS
- **Redshift:** A closed-source, Postgres-inspired data warehouse incompatible with standard engines
- **Proprietary Services:** Lambda, Kinesis, DynamoDB, and Step Functions all tie workloads into AWS

Just as Azure can be cost-effective or expensive depending on usage patterns, AWS pricing can spiral if not tightly managed. Both require architectural maturity to optimise cost and avoid lock-in.

6. IBM’s Historical Position

IBM once dominated the enterprise software and services market. It pioneered lock-in via middleware, proprietary APIs (WebSphere, DB2, Rational), and large consulting agreements. While it now supports open standards (e.g. Red Hat, OpenShift), IBM’s enterprise sales culture continues to focus on high-value, high-margin engagements with less visibility in modern developer ecosystems. Unlike Microsoft or AWS, IBM’s open-source contributions are narrower in scope and often centred on Linux and enterprise integration layers.

7. Salesforce and MuleSoft: True Lock-In at Premium Cost

Salesforce is a closed ecosystem. Its language (Apex), runtime, and data model are proprietary. Customers cannot export and run applications outside of Salesforce. Similarly, MuleSoft—acquired by Salesforce—remains a heavyweight integration suite with high licensing fees and steep learning curves. Unlike .NET Core or Azure Functions, it is neither open nor cheap.

Modern application architectures often avoid Salesforce due to:

- Lack of portability
- Cost barriers
- Poor developer experience compared to modern toolchains

8. Microsoft's Pragmatic Model

Microsoft spans multiple market positions:

- **Open frameworks:** .NET, PowerShell Core, VS Code, DAPR, Orleans, MAUI
- **Free services:** Dev tunnels, GitHub Codespaces (limited), Azure free tiers
- **Affordable personal products:** Microsoft 365 Home
- **Enterprise services:** Azure, Dynamics 365, Microsoft Entra, Defender, Fabric

In this spectrum, some offerings are proprietary and paid, while others are open and free. This is neither unusual nor monopolistic. The company behaves like any modern platform vendor, balancing openness, control, and return on investment.

9. The Maturity of the Consumer

Ultimately, platform costs—on Microsoft, AWS, or Google—depend on how well the systems are designed. Azure can be very cost-effective at scale with proper governance. Microsoft offers cost calculators, Azure Advisor, and hybrid licensing to help reduce cost for mature clients. So do AWS and Google. It is not a function of openness or monopoly—it is architectural competency.

Conclusion

Microsoft is not a monopoly. It is a mature, diversified corporation whose frameworks are among the most open and accessible in the industry. While some commercial services remain closed or subscription-based, its developer tooling and infrastructure are both transparent and free to use. Competitors like AWS and Salesforce engage in equally aggressive ecosystem strategies, often with less openness. Caution is always warranted in platform decisions—but Microsoft's openness is now a strength, not a liability.

Appendix C – Supporting References

The following are some sources to pursue for facts supporting this documents conclusions.

1. **.NET Core Performance Benchmarks**
   Independent benchmarks, such as those from TechEmpower, demonstrate that .NET Core consistently handles a high number of HTTP requests per second under load.
   TechEmpower Framework Benchmarks
2. **Node.js Performance Metrics**
   Benchmarking studies indicate that Node.js can handle approximately 1,000 requests per second on a single-core EC2 instance.
   Under Pressure: Benchmarking Node.js on a Single-Core EC2
3. **Python FastAPI Performance**
   FastAPI, a modern Python web framework, is recognized for its high performance, often outperforming other Python frameworks.
   Benchmarks - FastAPI
4. **PHP Laravel Performance**
   Performance tests show that Laravel handles around 521 requests per second with sessions enabled.
   Benchmarking Laravel, Symfony, & Zend
5. **Java Spring Boot Performance**
   In performance benchmarks, Java Spring Boot maintains a throughput of approximately 301 requests per second.
   Go Fiber vs. Java SpringBoot vs. Express.js
6. **GoLang Performance**
   GoLang has been shown to handle a high number of requests per second, with some benchmarks indicating over 20,000 requests per second.
   Go vs Python more request per second
7. **Azure Functions Cold Start Performance**
   Discussions around Azure Functions have highlighted cold start latencies, with efforts ongoing to improve performance.
   Azure Functions cold start improvement
8. **No-Code Platform Costs**
   No-code platforms often operate on per-user, per-month pricing models, which can become costly as user numbers increase.
   The 8 best no-code app builders in 2025
9. **Vendor Lock-In Risks with No-Code Platforms**
   There are concerns about vendor lock-in with no-code platforms, potentially

leading to challenges in transitioning away from such platforms.
Vendor Lock-In Risks: Why Low-Code Platforms Must Prioritize

10. **.NET Core vs. Node.js Performance**
Comparative analyses suggest that .NET Core offers superior performance metrics compared to Node.js in certain scenarios.
Node.js (Express) vs .Net: Hello World Performance
11. **Security Vulnerabilities in Scripting Languages**
Studies have identified common security vulnerabilities, such as cross-site scripting (XSS) and SQL injection, in scripting languages like PHP.
Security Vulnerability Classes in Popular Programming Languages
12. **Importance of Leadership in Software Development**
Effective leadership is crucial in guiding software development teams to success and adapting to unexpected obstacles.
Effective Leadership in Software Development Teams