# Refactoring a Monolithic Enterprise .NET project, with the main focus on reducing Downtime

Thilakshanee Iresha Jayabahu
thilakshanee@gmail.com

## Introduction

This paper introduces the purpose of refactoring a monolithic system into a containerized architecture with an emphasis on minimizing downtime to prevent outages. It explains why reducing downtime matters and outlines patterns and methodologies for performing software changes and upgrades.

What are the main principals for good software engineering. The table below shows some main software goals and features, *('Fundamental Principles of Good Software Design', Teteeda)*

**Table 1**
*Main software goals and features*

| Goal/Feature | Effective Principles/Methods |
|---|---|
| **Correct Integration** | Correctness, Abstraction, Architecture |
| **Efficiency** | Optimal Resource Consumption, Refinement |
| **Scalability** | Modularity and Scalability, Patterns |
| **Adaptability to Change** | Maintainability, Refactoring |
| **High-quality Design** | Completeness, Data Protection, SOLID Principles |
| **Security** | Data Protection, Architecture, SOLID Principles (particularly Dependency Inversion Principle for modular security design) |
| **Reusable and Maintainable Code** | DRY (Don't Repeat Yourself), Modularity and Scalability, SOLID Principles (Single Responsibility Principle for focused modularity, Open/Closed Principle for extensibility) |

However, the challenge of achieving these essential goals where a software system needs upgrading, new functionality is required or the code base needs improving, in a critical software system that is accessible in a production environment. The question is how to perform such changes in a production environment where any downtime of the system is not acceptable.

We investigate different strategies for achieving large scale system refactoring, with zero downtime drawing on some real-world examples. These examples will highlight some proven architectural patterns, deployment and operational strategies that have a proven track record to update systems safely while maintaining continuity of service.

Modern systems are designed to operate continuously without interruption and the support the growing needs of end users that required access and services twenty-four hours a day. The reliability of a system is critical to the business, and any outages could lead to financial loss and reputation. If a system needs to be upgraded and refactored any downtime would be seen has loss of revenue to a company. There is it always important to improve maintainability, scalability and performance when refactoring a system.

Taking a monolith architectural type system, which was the traditional approach to building applications, where all components like user interface, business logic, and data access are combined into one single system. Some of the main factors that make it difficult to implement a clean upgrade cycle. In the diagram there are a few examples of the challenges faced with upgrading older monolithic system.

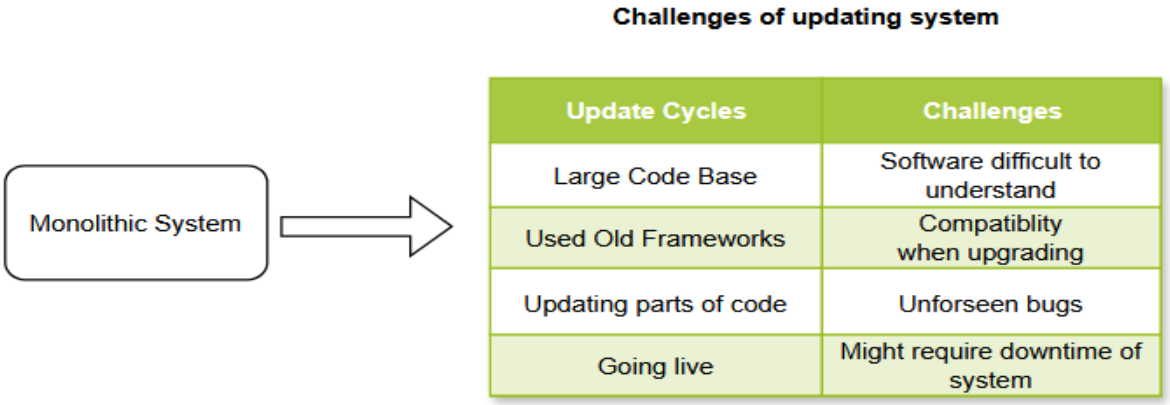


**Figure 1**
*Monolithic Architecture and System Update Challenges*

*Note.* Image highlighting some main update cycles required for a monolith system and the challenges those updates might have. Since monolith systems can be complex and built on layers of changes the risk and the need that the system will require downtime when deploying increases.

**Best practices for Refactoring a Monolithic system into more modernized system**

1. Critical points of refactoring older systems

Large scale systems have a wide range of complexities that have to be taken into consideration. Over time, as requirements change and new technologies emerge, maintaining and extending these monolithic applications becomes increasingly challenging (‘*Refactoring Old Monolith Architecture: A Comprehensive Guide’,Keşkekoğlu, Alperen* )

- Scalability, traffic volumes and availability – Systems require to be accessible all the time, therefore no room for service interruption.
- Legacy Systems and Complexity – Some systems contain old code that is embedded into a system, changing one part of a code, that seems unrelated could create bugs and errors in some other part of the system.
- Documentation and Knowledge – Older systems may have poor documentation and lack teams that have knowledge of the system.
- Dependency Management – Systems may rely on databases, therefore updates have to consider any changes which might affect the data integrity.
- Test coverage – Modern systems have improved on automated testing, and this should be standard practice for any software projects. Though older systems could have limited test coverage which would need to improve.
- Regression Testing – Testing of a complete system after a change can be challenging and time consuming.

2. Core values for Zero downtime Refactoring

Refactoring improves existing code, enhance code readability and maintainability (*'Refactoring Strategy: Zero Downtime',Idoko, Nicholas).* A few guidelines that teams can follow to improve a more successful refactor systems without any downtime.

- Small incremental updates – More smaller frequent updates means that updates are more manageable. More frequent update cycles allow team members to practice and refine skills, which gives deeper understanding of the process.
- Logging and tracking – Using more modernized logging and tracing, therefore teams can implement better ways to observe changes in systems.
- Monitoring – Analysing logging and react to problems. Tracking a system performance and monitoring its health, means that a response can implemented faster.
- Compatibility. Insure older legacy systems remain compatible with, the ever-evolving modern computer architecture.
- Secure Deployment – Used controlled release to minimize risk and protected the integrity of code. Modernize deployment strategies to include continuous integration and continuous deployment to automate the process.

3. List of examples that can help with Zero downtime Refactoring

Common deployment and release strategies used to introduce new functionality or replace legacy components with minimal risk and downtime. For each strategy shows, a core concept and a primary benefits and operational considerations.

**Table 2**
*Common deployment and release strategies*

| Pattern | Description | Benefits |
|---|---|---|
| **Strangler Fig Pattern** | Replace legacy components gradually with new modules. Introduce new components alongside legacy system and any traffic can be redirected. | Continuous delivery of components.<br>Smaller changes avoid the risk of major overhauls.<br>Gradual changes are more manageable. |
| **Feature Toggles** | Deployment of new functionality, but code is not active. New features are only active when the feature is switched on. | Test new changes in production when feature is switched on.<br>Instantly switch flag off if rollback is required. |
| **Blue-Green Deployment** | Two environments that run identical systems.<br>One active the other idle.<br>Changes applied to idle and switched when required. | Near zero downtime.<br>Faster rollbacks by switching back environment. |
| **Canary Release** | Changes are released to a subset of the users on the system. | More accurate testing of changes.<br>Any problems only affect the subset of users. |
| **Refactor** | More complexities involved due to state of data | Following the three phases |

| | | |
|---|---|---|
| **Databases** | must be maintained.<br>Changes can be<br>• Expand: Modify existing schema elements without deleting old ones.<br>• Migrate: Update the application logic, and any transition data.<br>• Contract: Deprecated any elements. | means compatibility is maintained throughout the process. |

4. Tooling systems currently available

For modern .NET systems, the following provide excellent support for zero downtime.

- ASP.NET – Enables cross platform framework development from Microsoft, designed with scalability and modular design practices at its core.
- CI/CD – Continuous Integration and Continuous deployment design pattern means pipelines support the automated process of software deployment. Features and changes are pushed into branches and merged into main deployment branches. Each branch can be tested before being merged.
- Containerization – A system where an individual software application, is package with all it required dependencies. So that the created image can be spun up within minutes, quickly and reliable across different environments. Each container is isolated from the host system.
- Monitoring – Logging tools allow real time monitoring, which is essential for looking into system behaviour and fault finding.
- Message Queues – Implementation decouples services and improves exchange of information asynchronously.
- Automated Testing – Using testing tools like Selenium or Playwright so systems can be quickly regression tested.

5. Scenario

A company with a large enterprise system, deals with financial large transactions volumes over an old legacy system. The transactions are based on processing payments therefore the demand peaks at certain periods of the month.

The system is based on a monolithic architecture, where whole ecosystem is encompassed in one application but is deployed across two cloud based virtual machines behind a load balancer. When the system was created the system had enough capacity to deal with the small user base and flow of transactions per day. Even on peak demands the system was robust enough to cope. However, over time with the increased volume the application has now hit performance issues, especially during peak times of the month. These issues means that company brand is being affected by negative customer experience and since the under arching architectural monolithic system, changing it will involve time, and costs but the result will be beneficial to the company and its brand.

Because the company was dealing with large transactions volumes, any downtime would be costly so achieving minimum to zero downtime was the main goal. The approach was to set up multiple teams that would be task to implement specific areas of the system. Taking the approach of implementing sections of the system which could run independently of the monolithic system. This approach meant that team gained experience of the core features.

The legacy system was analysed, and a few cores areas were identified, these were be broken down into key components that could be deployed and updated as independent services. The load balancer was used to divert traffic and the new features where tested, plus the system implemented feature toggles so that the underline older legacy code was still available. The new features where deployed on both nodes and the feature toggles were used to control the rollout of the updates. At each stage the company's internal testers could stress test the parts of the new system without affecting the current implementation. After the internal testing was complete the company roll out some changes to a small subset of its user base, to validate the updates under real world conditions.

The company used the Strangler fig patterns has shown here.

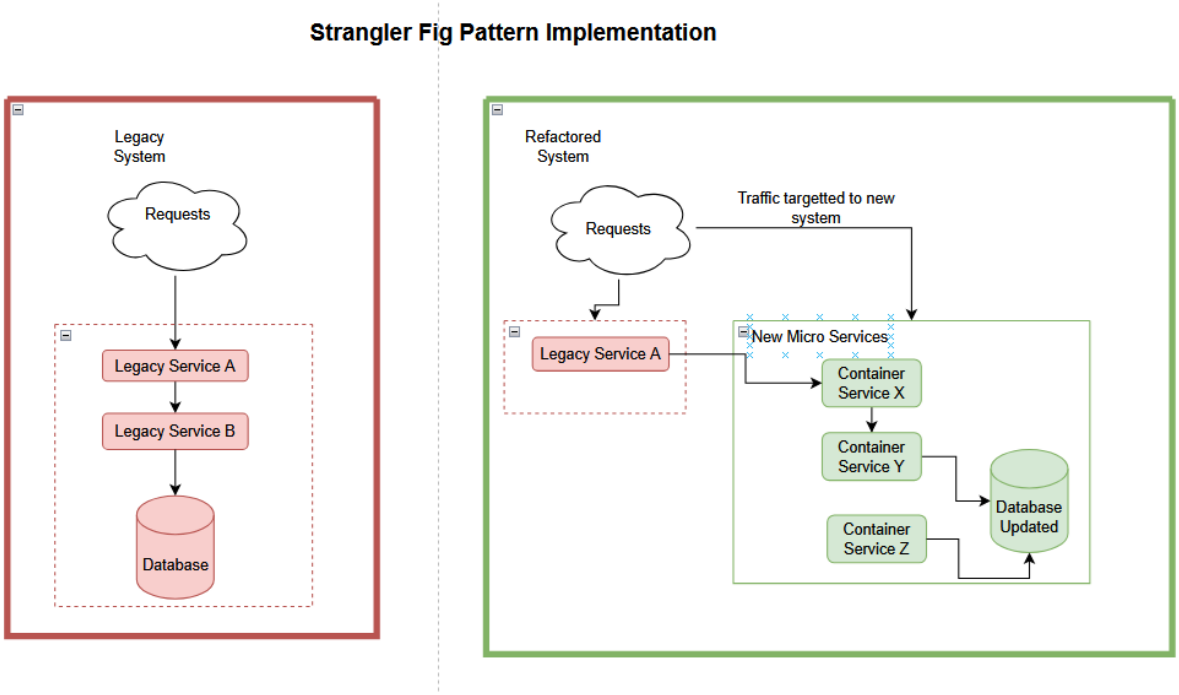


**Figure 2**
*Implementation of the Strangler Figure Pattern*

*Note.* The legacy showing all services and bound into one service.
Refactored system showing some of the legacy services have been placed into containers while some of the original service is still running in the old legacy service.

Any database changes used the design pattern of expand and contract, which allowed the old system to run alongside the new system during migration.

This approach meant the system was modernized to a more container-based system, improving flexibility, scalability and performance of the system, without any service downtime.

6. List of some guidelines and mistakes

The table below contrasts some actionable best practices for safe, maintainable software change with frequent pitfalls that undermine releases.

**Table 3**
*Common deployment and release strategies*

| Good practical guidelines | Avoid common mistakes |
|---|---|
| Small and continual iterations. | Attempting to change large monolithic system at once. |
| Focused on stability rather than performance. | Ignoring the complexities and compatibility of database changes. |
| Importance of monitoring and reporting of logs. | No rollback strategies implement. |
| Refactoring is a continuous process. | Insufficient communication with stakeholders |

## Conclusion

It is essential for computer applications to be refactored without causing any downtime, this can be challenging but by adopting best practices strategies and proven architectural patterns systems can be updated in a robust fashion while maintaining continuous services. This discipline should be the focus always on improving the ongoing refactoring of .Net enterprise systems.

## Author Bio for JSEP

Thilakshanee Iresha Jayabahu is a Senior Software Engineer with more than 15 years of experience in enterprise software development across aviation, fintech, and technology sectors. She specializes in .NET architecture, which provides scalable system design, and building maintainable enterprise platforms. Her work focuses on designing resilient cloud-ready applications, using modern structured design patterns to support long-term system sustainability. Thilakshanee is particularly interested in designing robust and modular architectures, distributed systems, and applying engineering best practices to help organizations build adaptable software systems, with a strong focus on refactoring monolithic enterprise .NET applications.

You can reach her at thilakshanee@gmail.com or connect on LinkedIn: linkedin.com/in/thilakshanee-jayabahu.

## References

Keşkekoğlu, Alperen. "Refactoring Old Monolith Architecture: A Comprehensive Guide." Insider One Engineering, Medium, 10 June 2024, https://medium.com/insiderengineering/refactoring-old-monolith-architecture-a-comprehensive-guide-7c192d7612e8.

Idoko, Nicholas. "Refactoring Strategy: Zero Downtime." Nicholas Idoko, n.d., https://nicholasidoko.com/blog/refactoring-strategy-zero-downtime/.

“Tateeda.” “Fundamental Principles of Good Software Design.” Tateeda, https://tateeda.com/blog/fundamental-principles-of-good-software-design. Accessed 23 May 2026.