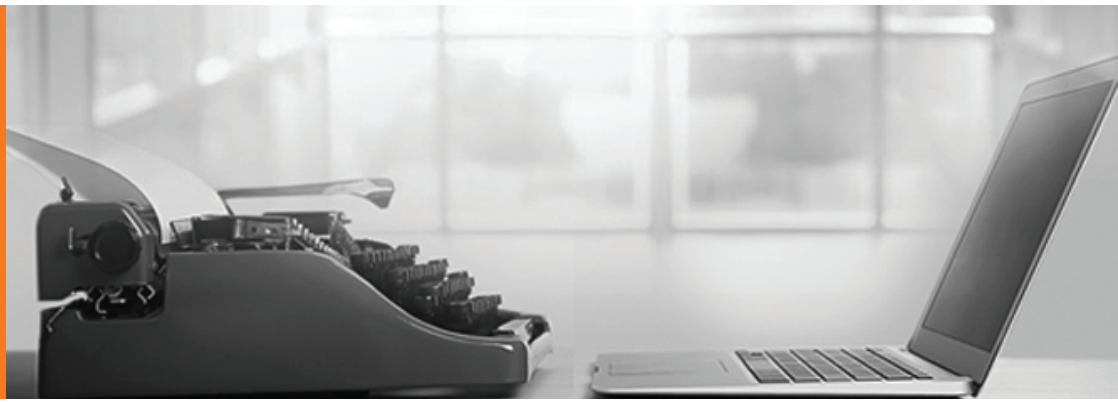
coMakeIT
continuous innovation

Demystifying
Application Modernization



A WHITE PAPER

Executive Summary

Software businesses that continue to offer and support *legacy applications* and enterprises that run mission-critical processes on such legacy software, realize that they are sitting on a ticking, time bomb. These software-driven businesses understand that unmitigated risk of legacy burden could potentially derail their core business, and also prevent them from innovation and success in a disruptive, digital world. Even though many CTOs and CIOs acknowledge that *legacy application modernization* is a *high-priority business imperative*, they are unsure of the right strategy and way forward. While choosing an application modernization strategy, one must be aware of the limitations of different conventional approaches, and must also be conscious of the pitfalls of quick fix approaches. This paper is written for technology practitioners and decision makers, and presents:

- a broad overview of legacy technologies
- nature of legacy constraints
- application modernization myths
- conventional approaches and limitations
- modern architectural paradigms
- coMakeIT's modernization strategies

CONTENTS

Legacy Landscape

Any ISV (*producer of software*) that entered the market between 1980-2005, or any enterprise (*consumer of software*) that is using software developed in this timeframe, almost certainly have a legacy application on their hands. Enterprise landscape is dotted with thousands of these legacy applications, built using a variety of programming languages, environments, platforms, and technologies as shown below:

Legacy Technologies

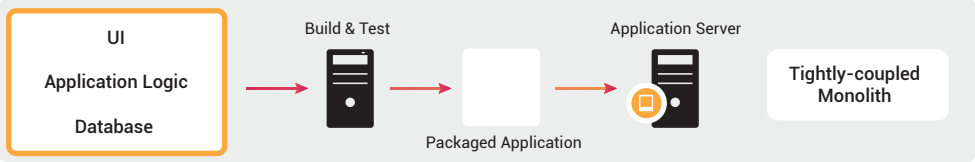
Programming Languages	COBOL, FoxPro, C etc.
Programming environments	PowerBuilder, Delphi, Developer/2000, VB etc.
ERP Tools	Native ERP development tools
Application Platform Suites	Progress, Forte
Mainstream technology stacks	Older versions of Microsoft, JAVA, LAMP

Most legacy applications have a monolithic (*tightly coupled*) code base and 2/3-tier architecture. In large enterprise-scale applications, explosion in number of modules with interdependent business and data logic, created spaghetti code with huge complexity.

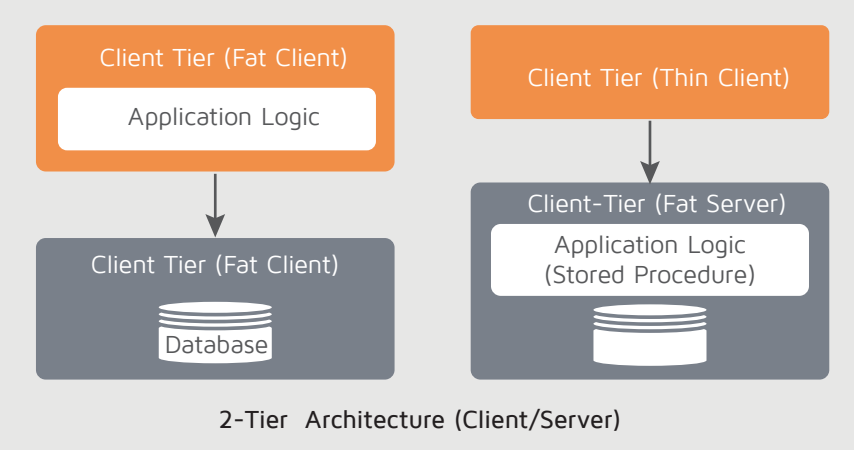
Evolution of 3-tier as the dominant architectural paradigm

> *2-tier architecture is inefficient, resource intensive, and due to the lack of separation of concerns, even incremental changes to the application will need significant effort*

Through the 80s and maybe even up to the early 90s, most software for business use was designed as 2-tier applications, with business logic either tightly coupled into the UI (what's shown to the user) or the backend database forming a single monolithic layer.



In a typical client-server setup, very often this separation was also physical in the sense that the front-end layer resided on the client system, and the database was hosted on a backend server.

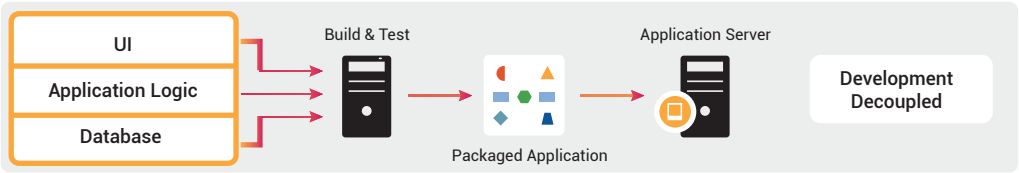


2-Tier Architecture (Client/Server)

As one can see from the above visualization, by design, 2-tier architecture is inefficient, resource intensive, and due to the lack of separation of concerns, even incremental changes to the application will need significant effort. The significant deployment & maintenance challenges as well as resource intensive nature of 2-tier applications led to the evolution of thin clients and 3-tier architectures.

3-tier architecture with clear separation of concerns into presentation, business logic, and persistence (or data) layers has evolved as the dominant architectural paradigm for enterprise applications

To overcome these constraints and also to build web/ browser based applications, a 3-tier, modular architecture with clear separation of concerns into presentation, business logic, and persistence (or data) layers has evolved. And over the past two decades, this has become the dominant and most widely used architectural paradigm, especially for enterprise software applications.



Limitations of 3-tier and constraints of legacy applications

> *Interdependent business and data logic between massive modules, resulted in convoluted, and tangled software, also known as spaghetti code.*

As enterprise software became large and complex, limitations and constraints of traditional 3-tier architectures and legacy applications became evident as described below:

Non-responsive UI

Conventional frameworks used to develop UI-layer are not responsive and cannot meet the demands of the new class of devices.

Software complexity & Spaghetti code

Thanks to interdependent business and data logic, software became twisted, convoluted, and tangled and came to be known as spaghetti code, a synonym for unmanageable legacy applications with massive modules, loads of code, and data complexity.

Monolithic code base

As applications turned complex with large monolithic code bases, they became unmanageable (in other words difficult to understand, maintain, and modify, especially for newer team members), resulting in a natural downward spiral in terms of code quality.

CRUD is crude

In the traditional *Create, Read, Update, and Delete* (CRUD) model of data persistence, typical operations such as reading data from the store, making modifications, and updating the current state of the data with new values, are done through transactions that lock the data. In a modern context with huge data volumes, this approach has severe limitations:

- In collaborative environments with lots of concurrent users, data conflicts are likely as update operations are performed on a single data item
- Due to proliferation of data and foreign key dependencies, even smallest changes to schema have become heavyweight
- Basic CRUD operations are well-suited to meet the needs of structured data, but cannot model and store connected data

> *Monolithic architecture and tightly coupled code makes scaling of applications possible only through running multiple instances of the application, which is a very resource-intensive, and inefficient way of handling transaction and data volumes.*

- In the absence of a data log, information on changes made i.e. history is lost
- Conventional RDBMS are built for stability, and not for complexity or business agility

Inflexibility, lack of modularity & integration challenges

Due to absence of modularity and interdependence between modules, even small changes call for a complete redeployment of the application. This made applications very inflexible, increased the risks associated with redeployment, discouraged frequent updates, and resulted in accumulated technology debt. Long-term technology lock-ins also makes it very difficult to adopt newer technologies, resulting in increasing obsolescence of these applications. Closed architectures made it very difficult to integrate legacy applications with other systems. This also led to the eventual emergence of SOA as a popular architectural paradigm for building modular and granular applications, that are easier to integrate and scale.

Inability to scale

Monolithic architecture and tightly coupled code makes scaling of applications possible only through running multiple instances of the application, which is a very inefficient way of handling transaction and data volumes. The tightly coupled architecture of legacy applications, doesn't allow for separation of concerns (resource requirements such as CPU, memory etc.), making it impossible to scale components independently.

Modernization Myths

"It is a myth to believe that quick fixes are genuine modernization strategies. At best, they will only result in partial modernization, and would neither remove legacy constraints, nor alter the application behavior in any meaningful manner."

Myth	Reality
If it ain't broke, don't fix it	**Modernization is not just for retrofitting, but it is also to:** • Make applications future-ready • Avoid being replaced by a nimble-footed competitor • Safeguard legacy investments
My customers are not complaining	**Legacy applications might serve current business needs and help retain current customers, but they:** • are incapable of meeting future business needs • will not be able to attract new customers or expand to adjacent markets • can't leverage emerging technologies
My monolith is rock solid	**Nothing could be farther from truth. This is a self-perpetuating myth and reality is far more complex:** • Hundreds of tables with thousands of Foreign Key relationships • Interdependent business and data logic and spaghetti code • Non-responsive UI • Monolithic architectures, no separation of concerns, and lack of modularity
Rehosting makes my application modern	**Merely Rehosting without architectural and technology modernization:** • only achieves partial infrastructure modernization. • doesn't remove any of the underlying constraints of the legacy application • will not enable the application to leverage full capabilities of modern infrastructure and cloud deployment
New UI is same as application modernization	**UI facelift is only a quick fix to web enable a legacy application and/or to make it look modern.** • A UI facelift doesn't alter behavior of the legacy application, and does nothing to make it scalable or extensible • A new UI doesn't remove code complexities or technology constraints of a legacy application • A truly modern application needs a responsive UI framework, which will work well, only when the underlying technologies and architecture are also modernized.

> *Modernization is a continuous process, and is certainly not a once and done thing.*

Adding a wrapper makes my product open	**It only helps in extending lifecycle of the legacy application, but doesn't remove the underlying constraints.** • At best, a wrapper exposes only a specific layer, and doesn't capture the holistic functionality of the legacy application • Basic behavior of the legacy application is neither modified, nor modernized
Code migration solves all my legacy problems	**This strategy will result in only partial modernization of the legacy code, and not the entire application.** • In the absence of architectural modernization constraints of legacy code will remain • A monolith will be recreated in a new technology with all the complexities of the legacy application • This often works better when migration is from an earlier version to a modern version of the same programming language or within the same technology stack • Difficult to anticipate how the migrated code will function, and will realize only when it fails
Application modernization is 'Once and Done'	**Modernization is a continuous process, and is certainly not a once and done thing.** • Emergence of disruptive technology paradigms are difficult to anticipate • In an inherently disruptive environment, enterprise business needs are rarely static, and are continuously evolving • One can be future-ready, but never future-proof.

Case for modernization

"In an era of unprecedented technology and business disruption, modernization is a business imperative"

Business drivers

- Shift from on-premise to cloud-based SaaS models, and opportunity to offer software-enabled services
- Demand for ability to integrate with other applications and to maximize value of legacy investments
- Enhanced customer experience with sophisticated systems of engagement to support consumerization and personalization
- Faster time-to-market and business agility with shorter dev, test, and deploy cycles
- Need modern stacks to remain competitive in a fast-changing market
- Increase operational efficiency by reducing technology debt, lowering TCO, and higher ROI
- Ability to support changing regulations and compliance burden
- Ability to support multiple business models in a platform environment

Technology enablers

Business drivers

- Changing enterprise needs and demand to leverage exponential technologies including SMAC, AI, ML, AR, VR, IoT etc.
- Proliferation and popularity of a new generation of robust programming languages, and architectural paradigms
- Demand for responsive UI to support a wide range of devices
- Demand for flexible, modular applications that can scale and also integrate easily with other applications and systems
- Ability to support state-of-the-art access and security in terms of people, organizations, and devices

Conventional modernization approaches & limitations

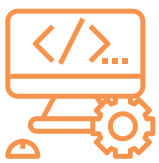
> *"Layered modernization might lower the operational risks, but it will only result in partial modernization. Most of the architectural and technology constraints will remain the same, and wouldn't yield a truly modern, flexible and responsive application"*

Legacy modernization can be classified into two broad categories – one is a *layered approach,* in which each layer such as the UI, database, and code are modernized; the 2nd approach is an *en masse* approach in which all layers of the legacy application and the manner in which it is deployed are modernized. Both approaches have their own limitations as described below:

Layered modernization

UI facelift

A UI facelift typically entails strategies such as web-enablement or HTML emulation. Web-enablement converts green screens into functional web pages, whereas HTML emulation creates a web or mobile interface to work with legacy application. In both cases, only partial UI modernization will be achieved, without altering the behavior of the legacy application or overcoming the constraints of the underlying architecture or technology. Technologies such as *Ajax* and *JQuery* intended to introduce dynamic behavior in the UI, shifted some of the client-side logic to the server side. Even though this achieved partially responsive UI, it also resulted in heavy, server-side frameworks (such as JSF), with complex and un-maintainable code and lower performance.



Code migration

This approach is akin to a technology upgrade and relies on using automated tools to either migrate the legacy code to a more recent version within the same stack (ex: from *VB to .Net*) or migrate to a more modern language (ex: from *Forte to Java*). While this strategy will result in partially modernizing the code, without architectural modernization, constraints of the legacy code will still

> *en masse modernization strategies are expensive, time-consuming, and high-risk endeavors ; they don't leverage legacy investments or accumulated domain expertise, and often demand significant investment with uncertain outcomes*

continue, and a monolith will be recreated in a new technology with the same complexities of the legacy application. A major limitation of code migration is also the fact that product developers will not understand the new technology, and product owners will not know until the migrated code fails

Database Migration

This approach is used to migrate data from legacy systems such as HDBMS (Hierarchical database management systems) or NDBMS (Network database management systems) to a more modern Relational Database Management Systems (RDBMS). This is typically accomplished using automated tools and data mapping. Database migration only helps in migrating data to a more modern and robust database, but doesn't alter the underlying UI, architecture or technology of the legacy application

en masse modernization

Rehost (also known as lift-and-shift)

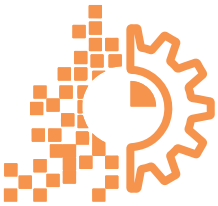
This strategy is based on redeploying legacy applications to a modern hardware and software infrastructure with minimal changes. This approach only achieves partial infrastructure modernization, and doesn't remove any of the underlying constraints of the legacy application in terms of technology and architecture. This is a myopic solution that only relieves the burden of maintaining the underlying infrastructure, but can never make the application truly SaaS and cloud-native.

Rebuild

This approach aims to modernize all tiers of the application along with deployment modernization, and calls for a complete rewrite of the application from scratch, using

> *It is very rare to find an off-the-shelf solution that fits the unique needs of a business, and they often demand significant customization*

modern technologies and newer architectural paradigms. While this strategy will yield a fully-modernized application, the fact remains that rebuilding or rewriting an application from scratch is often very expensive, time consuming, and a high-risk endeavor. Success of this approach will be dependent on the availability of people with skills in modern technologies and architectures, and ability to leverage domain or functional expertise of the existing team. This approach also doesn't safeguard legacy investments, while calling for significant new investments with uncertain outcomes.



Rip & Replace

This strategy calls for discarding the legacy application, and picking a commercial, ready-to-use modern application that meets the business needs of the enterprise. The reality is that it is very rare to find an off-the-shelf solution that fits the custom needs of a business, and another major disadvantage of this approach is the fact that significant legacy investments will have to be written off.

Architectural paradigms for modernization

> *Reactive Systems are flexible, scalable, and loosely coupled, which not only makes them easier to develop, but also makes them amenable to change (essential to be future-ready).*

Reactive systems

One of the key drivers for application modernization is a realization that the current and future business needs cannot be met with yesterday's software architectures. As the Reactive Manifesto describes, modern applications must be designed as Reactive systems that are Responsive, Resilient, Elastic and Message Driven.

Reactive Systems are flexible, scalable, and loosely coupled, which not only makes them easier to develop, but also makes them amenable to change (essential to be future-ready). These systems are also designed to be tolerant of failure, and are capable of handling failure very elegantly and preventing disasters from happening. Reactive systems are also highly responsive, ensuring a consistent performance, and depend upon asynchronous messaging for communication between components.

Event sourcing and CQRS

In the past few years, *Command Query Response Segregation* (CQRS) pattern, in combination with *Event Sourcing (*ES), has evolved as a powerful architectural paradigm to overcome the limitations of CRUD, and offers the following advantages:

- Event Sourcing records all changes to an application state as a sequence of events in an event store or event log
- Event log can be used to completely rebuild the application to any desired state
- As events are simple objects that can be saved through an append operation, they offer better performance and scalability, compared to complex relational stores.
- ES enables traceability of changes and audit logs
- CQRS pattern establishes clear segregation of responsibilities between Write and Read models, as opposed to CRUD where all operations are executed through RPC.
- Using CQRS and ES, write model captures all changes as events in a separate write-side store, which can be appended using simple Insert operations.

> *Microservices is an architectural paradigm, explicitly designed to overcome the complexity of interconnected, functional modules in large, complex applications.*

- Similarly, the read model builds the latest application state using the same events from a read-side store to
- support queries

Microservices

Microservices is an architectural paradigm, explicitly designed to overcome the complexity of interconnected, functional modules in large, complex applications. Compared to traditional 3-tier architecture, Microservices offers the following advantages:

- Loosely coupled services that are better aligned with business and domain functionality, and which can be independently deployed
- Each microservice will have its own database, avoiding the need for data sharing and tight coupling; when needed, data sharing between microservices is accomplished through asynchronous messaging and event sourcing, avoiding the need for foreign keys and cumbersome SQL JOIN's
- Decomposition into granular services makes it easier to upgrade, maintain, and modify services whenever there is a business need, and not the entire module or application

> *Modern applications demand responsive UI/UX frameworks to render on a wide range of devices, and also support user interactions from multiple sources including speech, vision, gestures, and touch*

NoSQL

NoSQL stands for "Not Only SQL", and is an alternative approach to database design to accommodate different data types and a variety of *schema-less data models* such as *Key/Value pairs, JSON documents, graph format* etc. NoSQL provides the flexibility to handle varying/changing data structures with less constraints. Over the past few years, popular NoSQL databases such as *MongoDB, Apache Cassandra,* and Redis have emerged as powerful alternatives, and offer the following benefits over relational databases:

- NoSQL databases allow you to stuff data into a database without defining a formal storage structure ahead of time
- NoSQL databases don't need rigid schema, making them lightweight
- NoSQL databases are elastic and scale better, as they are designed to run on distributed and clustered environments

Responsive UI/UX frameworks

Most legacy applications are designed for a specified, pre-defined input/output pattern, and very often the presentation layer is tightly integrated with business logic, making it very difficult to change. Traditional UI/UX is also mostly CRUD-driven with mass input and output. The current generation of applications demand goal-oriented UI/UX based on roles, tasks, state, and must be responsive to support a wide range of devices ranging from desktops to handhelds to mobile and wearables. At the same time, in addition to the standard keyboard-based input, they must also support user interactions

> *SecuRef is a unique modernization strategy that extracts business logic even from a monolithic application into a separate layer, without a code rewrite or extensive refactoring*

from multiple sources including speech, vision, gestures, and touch. Some of the popular UI/UX responsive frameworks used for creating modern, digital applications include *Bootstrap, Foundation, Skeleton, CreateJS* etc.

Model-driven Development

Model-driven Development (MDD) has emerged as one of the leading approaches for rapid, collaborative application development. MDD uses visual modeling techniques to define data relationships, process logic, build user interfaces, and construct complex applications. With MDD, business users can rapidly create and deploy applications. Model-driven development is significantly faster than traditional approaches to application development, and as it doesn't need extensive coding, it is also known as *Low Code Technology Platforms*. The right model-driven development platform provides significant productivity advantages over traditional development methods and enables project delivery by smaller teams.

Containerization

Containers are lightweight, stand-alone, executable packages of software bundled along with all dependencies needed to run the application: code, runtime environment, libraries, binaries, and configuration files. Containers virtualize the OS instead of the hardware, and multiple containers can run on the same machine and share the OS kernel, which makes them far more powerful and efficient than Virtual Machines (VM). With the advent of Docker in 2013, containerization has become a powerful tool to deliver and deploy applications/services; provisioning and scaling of resources as per need and in a dynamic manner became possible. Containers also reduced the gap between development and operations by enabling standardized packaging of software and its dependencies, and abstract away any differences in OS distributions and underlying infrastructure.

coMakeIT's Application Modernization Strategies

SecuRef

Notwithstanding technology and architectural constraints, legacy applications still have significant business value, along with functional and domain expertise accumulated over the years. Conventional modernization strategies fail to leverage these legacy investments. At coMakeIT, we have evolved a unique strategy, **SecuRef,** that not only leverages the accumulated functional and domain expertise, but also achieves modernization with a safety net. This is a strategy based on an inside-out approach to legacy modernization.

SecuRef is a modernization strategy that enables extraction of business logic into a separate layer, even from a monolithic application; in other words, transforming a 2-tier or a 3-tier into a reactive system. What makes this approach unique is the fact that it is achieved without a code rewrite or refactoring, and leverages technology and functional expertise of the existing team. This strategy enables application modernization from inside out, in a modular manner and also makes it possible to use open technologies and low-code platforms.

Rebuild and Reinvent

Depending upon the as-is state of the legacy application, and the desired end state, if none of the conventional approaches offer a meaningful modernization roadmap, then we recommend a strategy of rebuild and reinvent. This often calls for a complete rewrite of the application from scratch, and will yield a fully-modern, future-ready application. coMakeIT's rebuild and reinvent modernization strategy aims to create *reactive* applications using modern technologies and architectural paradigms such as:

- Event sourcing and orchestration using akka.io
- CQRS by taking advantage of NoSQL
- Microservices
- Case Management for complex entities using Cafienne (an implementation of CMMN - Case Management and Modeling Notation standard)
- Containerization

The end-state application will be a reactive system that is flexible, loosely coupled, scalable, resilient, and responsive.

Conclusion

There will always be competing pulls and multiple factors that influence the choice of modernization strategy, and one must be aware of the limitations of conventional approaches. A short-sighted modernization strategy may help deal with a few pressing challenges, and yield short-term results, but will not result in any meaningful modernization. The right modernization strategy must aim to achieve the following:

- A reactive system that is flexible, loosely coupled, scalable, resilient, and responsive.
- A future-ready application built with modern programming languages, using flexible and modular architecture, and capable of delivering consumer grade experience – at scale, usability and intelligence
- Process and Deployment modernization to support rapid development, continuous delivery and deployment
- A modern, reactive and open application that is easy to integrate and which can support multiple business models as part of different platforms

coMakeIT
continuous innovation

The Netherlands
coMakeIT B.V.
Stationsplein 62, 3743 KM Baarn
The Netherlands T: +31 35-303 5630

India
coMakeIT Software Pvt. Ltd.
Plot No.564/A 39, Road No.92, Jubilee Hills
Hyderabad 500 033, India T: +91 40 4035 1000

ISO 9001:2015
ISO / IEC 27001:2013 (ISMS)

www.comakeit.com