Highly efficient and competitive organizations understand that an effective IT environment is a major pillar of their success.

Unfortunately, IT environments are often weighed down with legacy components that are costly, risky, and slow. The resulting imbalance between IT and business creates an execution gap that keeps IT from successfully supporting ongoing business requirements—requirements that are being driven by customers, competitors, and demands for compliance.

The solution for this execution gap? IT modernization.

IT modernization is the continuous evolution of an organization’s existing application and infrastructure software, with the goal of aligning IT with ever-shifting business strategies. IT modernization implies the acquisition and deployment of modern technologies, skill sets, and capabilities to replace legacy environments. These modern technologies must be based on open standards and must provide an open, complete, and integrated environment that is both economically efficient and able to support an organization’s strategic business goals. IT modernization can be done as quickly or as slowly as an organization requires: strategies and road maps can span multiple years, but must align with the organization’s business priorities and budget constraints.

This paper examines the various approaches to modernizing an IT environment. Typically, organizations will need a combination of approaches for a complete IT solution.

Planning an IT modernization effort includes defining a strategy and developing a plan that maps the current legacy environment to the desired state. The planning process also includes identifying target architecture, acquiring required software, and creating a multiphased execution plan.
Approaches to IT Modernization

IT modernization is the continuous evolution of an organization’s existing application and infrastructure software, with the goal of aligning IT with shifting business strategies. It implies the acquisition and deployment of modern technologies—along with their associated skill sets and capabilities—to replace legacy environments.

Oracle offers several approaches to IT modernization to address the complex challenges of modernizing legacy environments.

- Replacing legacy applications with packaged, commercial off-the-shelf (COTS) applications
- Enabling service-oriented architecture (SOA)
- Rearchitecting legacy applications
- Automating the migration of legacy applications based on 4GLs and other legacy languages
- Offloading a mainframe’s MIPS transactions to an open systems platform using data caching
- Rehosting application logic and data, intact, to a more-open, cost-effective, and agile platform

Organizations typically need a combination of these approaches for a complete modernization solution. Each of these approaches is examined in the following sections.

Replacing Legacy Applications with Packaged Applications

The approach that is most often considered in IT modernization is replacing legacy applications with one or more packaged COTS applications. This approach includes replacing horizontal, functional applications as well as vertical, industry-specific applications. Replacing legacy applications with SOA-based application packages can also be highly cost-effective.

Of course, legacy applications can be replaced with packaged applications only when appropriate application replacements exist. Both horizontal applications (such as payroll, accounting, billing, and customer relationship management) and many vertical applications (such as those designed for aerospace and defense, communications, industrial manufacturing, financial services, and utilities) are available today from Oracle and other software vendors.
To get maximum agility from replacement applications, organizations should replace legacy applications with applications made up of SOA components and applications that use SOA capabilities (such as SOA component orchestration). These SOA components can then be mixed with other modernized components resulting from rearchitecting, rehosting, and automatically migrating custom application components using an SOA platform and a services orchestration engine. This maximizes the agility of the complete application because the packaged applications are seen as sets of reusable components rather than as isolated application silos.

Even though packaged applications typically provide significantly more functionality than legacy implementations, there may be some unique facets of the legacy system that are not covered by the packaged application. In this case, the business logic from the legacy system can be retrained using one of the other modernization approaches described in this paper, such as rearchitecting or rehosting the application. The combination of Oracle Applications with Oracle Fusion Middleware using Oracle Application Integration Architecture allows for coexistence and integration of packaged and custom processes, giving customers greater flexibility.

Enabling Service-Oriented Architecture

This approach to IT modernization consists of wrapping legacy application services in place and presenting them as Web services to an enterprise service bus. The value of this approach is that it provides immediate integration of legacy systems with other systems, using middleware such as Oracle SOA Suite.

One of the key business benefits of enabling SOA integration is that organizations can buffer business processes from business applications so that changes to the application systems—or to the software they are built on—are not exposed to the process layer. The system is improved with limited impact—possibly no impact—on the business process.

SOA integration also allows organizations to move faster: They can run more projects concurrently because they know the projects will be tightly integrated at the end of the process. Registered SOA services can be reused and combined with applications that have been modernized through other, more-complex modernization techniques, such as rearchitecting.

By enabling SOA integration, an organization can begin to use SOA concepts—including the orchestration of SOA services into business processes—and leave legacy applications intact. Of course, appropriate interfaces to the legacy application must already exist, and the code behind these interfaces must perform useful functions that can be packaged as services.

SOA integration typically takes advantage of three legacy application interface points.

- **Presentation screens.** A legacy screen or group of screens is replaced with an SOA service that drives the underlying program the same way the original screen did. Presentation screens are often good candidates for SOA integration because many legacy applications use screens to drive a single transaction, such as adding a customer or approving a purchase order.

- **Functional calls.** Calls to a legacy procedure or program are replaced with an SOA service that issues the same call. Procedures or programs that can be called by the legacy application may have been written as reusable components and are candidates for reusable services.

- **Database calls.** A call to a legacy database or file system is replaced with an SOA service that issues the same native call and returns the requested data. Because the new environment uses relational databases for handling data, this call may be further modernized by allowing a structured query language (SQL) call to be issued, even though the data is not stored in a relational database environment.
SOA integration often involves communicating with the existing application on its current platform. However, it is also possible to use the SOA integration technique with an application after rehosting to another platform. (See “Rehosting Legacy Applications,” later in this white paper, for more information.) Using SOA integration in this way is useful because all the components—the rehosted application components that have been integrated with SOA, the new Java components, the packaged application components, and the orchestration engine that brings them all together—reside on the application grid leveraging common management, monitoring, and virtualization capabilities.

Because SOA integration does not typically require deep changes in the legacy application, legacy components can very quickly be used as part of an SOA infrastructure with little risk. Enabling SOA integration is definitely a first step toward a completely modern environment. However, because the code that implements the services remains unchanged, SOA integration does not solve the problems involved in maintaining a legacy environment, and provides only partial help in reducing the total cost of ownership (TCO), increasing application agility, reducing dependence on legacy skill sets, and achieving compliance.

Rearchitecting Legacy Applications

As an approach to IT modernization, rearchitecting means building new, replacement systems on the side; integrating these new systems with the old systems; and eventually, shutting down the old systems.

Legacy applications are a mix of business-relevant and technical code that implements legacy technical-support capabilities. Because the new IT environment provides much of this support functionality in other ways, the original technical code is no longer needed. Rearchitecting, then, focuses on recovering and reassembling business-relevant code from legacy applications while eliminating as much of the technology-specific code as possible.

Rearchitecting is typically used in modernization projects that involve changes in architecture, such as introducing object orientation and process-driven services. Rearchitecting recognizes that—in addition to the application code—application process interactions, data models, and workflows are also useful to the rearchitecting process.

Rearchitecting is typically done in four phases: recovery, redesign, refactoring, and regeneration.

Phase 1: Recovery

In the recovery phase, the original application is analyzed using an application portfolio analysis (APA) along with micro- and macro-analysis techniques to create a clear understanding of both the application and the models behind it. (These models are typically in the form of legacy modeling techniques, such as information engineering models.) This information is gathered top-down from walkthroughs of the existing system, existing workflows, and corporate process models. A bottom-up analysis is then done of existing data models, data dictionaries, and the legacy application code itself. The results of all these analyses are placed in a repository in preparation for the refactoring phase.

The recovery phase provides a platform-independent, present-case model of the current application design, as well as the analyzed code that implements the present-case model. The code may be represented either in an abstract syntax or in its original syntax. Even though the complete rearchitecting process requires human intervention, much of the recovery phase can be automated—especially when gathering information from existing computer sources.

1 An application portfolio analysis (APA) helps an organization better understand its current application environment.
Phase 2: Redesign

In the redesign phase, a future-case model—based on SOA principles and new modeling techniques—is developed to produce applications that are able to execute on SOA modernization architecture. New workflows are developed to cover the content of the present-case model, as well as the content obtained from use cases in the future-case model. New business process steps are developed using techniques that better align the new processes with the business processes and with the new, graphic system. Business-to-business interfaces are defined to cover the functionality of previous interfaces and to add new interfaces. This future-case model is stored in the same repository as the present-case model.

Redesign is often used in conjunction with the modernization approach of replacing legacy applications with packaged applications. If a packaged application exists that can replace all or part of the legacy application, then the packaged application components will form part of the redesigned architecture.

Components of the future-case model may correspond closely to the present-case model, but rearchitected changes or desired changes in business operations may mean that the two models differ substantially. For example, the redesign phase may examine processes to determine which are batch processes due to business requirements, and which are batch processes due to legacy technology constraints (such as CPU availability) that could be refactored. (See the next section for more information on refactoring.)

In some cases, design concepts that were coded in the legacy environment are replaced by native functionality in the new environment. For example, reports may be replaced by data warehouses and reporting tools, batch job control may be replaced with orchestration flow, data edits may be replaced with database stored procedures, and some business rules may be programmed into a business rules engine. This type of replacement mapping allows for a reduction in the number of lines of code in the new system, resulting in a more agile system that costs less to maintain.

With both the present-case model and the future-case model stored in the same repository, the stage is set for the refactoring phase.

Phase 3: Refactoring

In the refactoring phase, organizations examine the business logic of the present-case model to determine what can be mapped to the packaged application components that make up the new design architecture. In this way, rearchitecting provides a way to compare the mined legacy content to the new application functionality and ensures a more complete form of gap analysis than can be achieved by just talking to end users.

Functionality that does not map to packaged applications or to the native capabilities provided by the new technology environment is transformed and reassembled into new components that are based on the future-case model. For example, a set of individual procedural routines to add, update, approve, pay, or delete an invoice would become methods on an invoice object and would be used by a change-invoice process. The routines might be further refactored to eliminate and reconcile duplicate edits that were repeated in a number of places because of cut-and-paste development.

Because of the nature of refactoring, it can never be completely automated. However, it can be partially automated using tools that analyze legacy content to find candidates that map to the packaged application environment or candidates that can be reassembled into new components. For example, once a file is identified as a candidate for a new business object, code analysis can trace back from any file updates to determine the code that affects the field
values comprising the file records. The actions performed by the code then become candidates for packaged application capabilities or for methods on a new business object.

When complete, the refactoring phase yields a future-case application—defined at a platform-independent level and designed to maximize the use of packaged applications, SOA, and the capabilities of the new environment. Content that cannot be refactored to packaged applications can be refactored directly to a language, such as Java. Sometimes, a higher-level abstract representation is needed—for example, an application development framework that uses regeneration techniques.

Phase 4: Regeneration

The final phase of rearchitecting is regeneration. This phase maps nonpackaged applications, platform-independent refactored models, and code abstractions into a platform-dependent form. To carry out this mapping, the regeneration process takes advantage of application development frameworks and integrated development environments. These frameworks and tools make the creation of the final application simpler by providing both design templates and reusable implementation components that make maximum use of the new environment. At the same time, these frameworks and tools help hide any remaining platform specifics from the application code itself—increasing flexibility and lowering the cost of maintenance.

Although regeneration can be a manual process, in most cases application frameworks and design patterns make automation possible. Regeneration techniques can also be used to incorporate specific platform requirements, such as Java technical architectures.

Because rearchitecting takes maximum advantage of the knowledge contained in the existing application, it costs less and has fewer risks than developing an application from scratch. However, because rearchitecting does involve a high degree of change, there are still risks.

Other modernization techniques have been developed that do not offer all the benefits of rearchitecting. These other techniques can be used to break the process of modernization into a series of steps, which—although more costly—may have fewer risks.

Automating the Migration of Legacy Applications

In some cases, the process of migrating legacy applications to a new architecture can be automated. A migration is considered to be automated if at least 80 percent of the migration or transformation can be handled by migration tools rather than manually. Depending on the nature of the technologies involved (languages, databases, and so on) the degree of automation can range from 50 percent to the high-90s. To reach this degree of automation, the migration or transformation process must be algorithmic and should not require human intelligence during the transformation process (as is needed, for example, in the rearchitecting processes).

Automated migration does not typically change the design of an application, but it can provide specific enhancements if they are incorporated in the set of rules driving the tools (for example, field extensions). Automated migration takes existing code and runs it through a parser to create an abstract representation which is then fed to a utility that generates code in a new language (for example, migrating COBOL or Natural to Java). Automating migration is fast, but it only works if the gap between the legacy architecture and the new architecture is relatively small. Automated migrations are most successful when there is a well-defined mapping between the source and target architectures. Mapping that is not well-defined will cause problems with the migration, as described in the following sections.
Transforming Procedural Designs into Object-Oriented Designs

Although it is possible to transform procedural code (such as COBOL or PL/I) into an object-oriented language (such as Java), it is not possible to map the procedural design that surrounds COBOL programs into the object-oriented design that surrounds good Java programming. The fundamental design concepts of object-oriented programming—such as the class and its behavior—are architectural concepts that require human intelligence to design. The designer of a truly object-oriented application will use these techniques in new ways that cannot be recovered from an application developed using nonobject-oriented techniques. The challenges involved in maintaining applications that have been automatically migrated to a “procedural Java” implementation can be significant because there is no easy way to extend these applications using standard Java methods, class-based techniques, and so on.

Migrating legacy applications to object-oriented designs cannot be successfully automated using our definition of automated migration (where at least 80 percent of the migration is done with technology). This is equally true when automatically transforming legacy procedural flows into more-modern modeling approaches such as a Unified Modeling Language (UML). The transformation can be done, but the resulting UML will be a present-case UML—not the same UML that would be designed for a workflow-driven application.

Transforming Pseudo-Conversational Code into Conversational Code

One of the most interesting concepts in the history of computing was the introduction of pseudo-conversational programming by IBM in the 1960s. This concept, introduced at a time when machines were expensive and machine space was at a premium, forced programs to drop context information—including variable content and transactional content—from each screen input or output.

Although the loss of context with pseudo-conversational code is similar to the loss of context caused by thin client HTML applications, pseudo-conversational code is not the same as HTML when it comes to transaction handling—and it is very different from a conversational application where the application context is retained across any application display or event. Pseudo-conversational code requires human intelligence to map a set of pseudo-conversational screens to conversational code; transforming pseudo-conversational code to conversational code, then, cannot be successfully automated.

Pseudo-conversational code was developed and implemented to drive the efficiency of resource consumption, enabling tremendous scalability. While current-generation software employs different methods to achieve similar goals, if the resulting systems are to be as efficient and scalable as pseudo-conversational code, any transformation should be considered thoughtfully, with great care and attention to detail.

Transforming Legacy Procedures to Business Services

Like the SOA integration and rehosting approaches, automating migration does not change the core structure of the legacy application. To use migrated code as services, the legacy interfaces must first exist. Many mainframe applications developed on IBM CICS or IMS TM are structured in a way that enables reasonable mapping of business transactions to fine-grained services. Screen-based applications often offer basic services (for example, creating an invoice, updating an invoice, recording the payment of an invoice, and so on) which can be mapped to fine-grained services in a SOA. However, in some cases, the ideal services may not exist in legacy applications because the legacy applications simply were not designed that way.

Technical SOA interfaces can be created as part of an automated migration; however, business services that more closely follow the business process are much more difficult to create automatically unless the legacy components needed to support the services already exist.
Transforming to a Business-Process-Driven Environment

To get the maximum benefit from a new IT environment, the modernization process must transform legacy applications into SOA services. In this case, a business process automation tool is used to define both the human and computer processes that are driven by a business process management engine.

Unfortunately, most traditional applications have either no workflow concept outside of batch control scripts, or the workflow is embedded deep within the code. Transforming these applications so that the workflow is at the top of the architectural stack and can drive the rest of the application is not something that can be successfully automated. In fact, it may be that determining the current workflow will require interviewing current users—something that can’t be automated at all. In most cases, creating applications that use orchestrated SOA services to align with business processes cannot be done automatically.

Transforming from Batch to Online Processing

In many legacy applications, some aspects of online transactions are recorded and then processed later using batch processing. For example, stock trades cannot be valued until the stock market closes, so final processing cannot occur until the end of the day. Batch processing, such as end-of-day reconciliation or end-of-month reporting, is based on human clock cycles and does not disappear in the new IT environment. Some transactions will continue to be processed this way.

However, there are many traditional batch processes that can be wholly or partially modernized and transformed into online processes. For example, incoming transactions can be processed immediately, resulting in more-up-to-date information. Transforming these batch processes to the correct forms of online processing requires human thought and cannot be automated.

Making Automated Migration a Viable Alternative

The main concern of any automated migration process—in addition to architectural changes—is the quality of the source code. If the source code is poor, transforming it automatically to another language or environment will not improve its quality. However, if the source and target design are similar enough, automated migration can be a viable alternative, producing fast, consistent results. In many cases, the issues making automated migration difficult can be avoided or even eliminated. These issues are discussed in the following sections.

Migrating Applications That Use Legacy Databases and File Systems to Relational Databases

Legacy applications that are written in third-generation languages (3GL), such as COBOL and PL/1, and that use legacy database and file systems (such as VSAM, IMS DB, ADABAS, IDMS, and Datacom/DB) retrieve data by issuing calls embedded within the program code. Once a data model maps the legacy database or file formats to relational tables, you can use automated migration to remove the calls and replace them with SQL calls. The rest of the code is not affected. The same data model mapping can be used to migrate actual data from the legacy database or file system to a relational database.

Migrating Applications Written in Legacy Languages to Other 3GLs

While the vast majority of mainframe applications are written in COBOL, some mainframe systems still have programs written in IBM Assembler, PL/1, and other languages that cannot be well supported on open systems. Automated migration can be used to convert these components into more-acceptable 3GL alternatives, such as COBOL or C, for rehosting on open systems. For assembler migration, a multiphase approach is typically required to isolate business logic from technical use of assembler. Subsequently, the business logic can be
mapped to a COBOL or C representation with a good degree of accuracy. Customers with significant PL/I assets have also experienced success using automated migration that maps PL/I to C or to procedural Java.

Migrating Applications Written in Fourth-Generation Languages to Java or Rehosted COBOL

Fourth-generation language (4GL) environments (such as NATURAL, IDEAL, ADSO, and PowerBuilder) typically consist of both a language and a runtime environment that provide additional capabilities to support the executing programs. The 4GLs were created to move away from the complexity of 3GLs and environments by using a runtime environment to hide lower-level implementation details and provide a higher-level, more abstract interface for application developers.

As a result, 4GL environments are architecturally more modern than 3GL environments, and do not have as many of the architectural issues that make automated migration to Java virtually impossible for 3GL environments.

For example, an organization can automatically migrate a NATURAL/ADABAS application to Java/Oracle. The resulting application will not have a completely object-oriented design because the original NATURAL application was procedural. However, because NATURAL applications are coded at an abstract level that uses a conversational programming style, the procedural Java code created will be conversational and can use object-oriented libraries that supply the needed environmental functionality.

The result of this type of migration is a mixed-mode application that can take advantage of new capabilities and is reasonable to maintain. The code can be modernized even further to get all the benefits of the new environment.

Organizations can automatically migrate legacy mainframe 4GL languages to COBOL and then rehost them. Although this may seem like a step backward, the fact that 4GLs are generally procedural and use mainframe concepts (such as mainframe data typing) means that migrating them to COBOL is relatively straightforward. As part of the migration to COBOL, legacy databases associated with 4GLs can be eliminated, and the data can be rehosted on a relational database. This technique is particularly useful when a legacy 4GL application is intertwined with legacy COBOL.

Restructuring Program Code

To prepare code for other types of modernization, you can use automated migration techniques to "clean up" the code. Program and loop restructuring can also be used to eliminate dead code and GOTO statements.

Automating Migration: Advantages and Disadvantages

The clear advantages of automated migration are speed and consistency. Because a computer carries out the modernization process, it can be done quickly and consistently, and can even be repeated on a more recent copy of the source code to include ongoing changes. Modernization is done the same way every time, so even though automated migration is more invasive than either SOA integration or rehosting, it has a lower risk and requires less testing than a manual effort.

The disadvantage of automated migration is that only algorithmic transformations can be made. If the goal of a modernization effort is to make major changes to architecture and application design, automated migration will not work.
Offloading MIPS

Using a data grid to offload MIPS via data caching is an approach to legacy modernization that uses an open-system–based, in-memory data grid solution to alleviate excessive and unnecessary traffic between midtier servers and a mainframe. Applications cache data in the data grid and avoid expensive requests to back-end mainframe data sources. Reading from the cache is faster than querying the back-end mainframe and scales naturally with the application tier.

Offloading MIPS using a data grid to cache the mainframe data is most successful when extensive Web traffic hits the mainframe and relies on fairly static data, as with daily rate tables, account balances, and so on. Creating J2EE services around the data grid helps maintain the currency of the data, but if these services connect to the mainframe for updates as frequently as the original traffic, little time would be saved.

Oracle Coherence is the industry-leading in-memory data grid solution for legacy modernization. It alleviates latency problems and drives dramatic increases in application performance by moving data closer to SOA applications for efficient access. In-memory performance alleviates bottlenecks, reduces data contention, and improves the application’s overall responsiveness.

In some cases, customers have made the data grid the master repository of such data, leveraging its scalability and cost-effective virtualization technologies. When mainframe services require access to this data, they leverage mainframe integration technologies, such as Oracle Tuxedo Mainframe Adapters, to access that data in the grid and in trigger-related J2EE services.

A key business benefit of implementing a MIPS offload solution is the significant reduction of MIPS usage and the associated mainframe costs. In many cases, the MIPS offload approach serves as the first step in strategic, multiphased IT modernization projects. Money saved from the reduced MIPS consumption is then invested in subsequent modernization activities that offer more-substantial, longer-term business benefits.

Rehosting Legacy Applications

According to some industry estimates, more than 200 billion lines of COBOL code exists in enterprise applications—mostly in mainframe environments—and this number is growing by 3 to 4 percent a year. Rehosting is a modernization approach that is predominantly focused on migrating mainframe COBOL—intact and with no application logic changes—to open, lower-cost platforms. Of course, many mainframe applications rely on additional technologies, such as transaction monitors (such as IBM CICS, IMS TM, or Unisys MCS), pre-relational and relational databases (including IMS, IDMS, and DB2), and record-oriented file systems (such as VSAM, 3270, and terminal emulation). In addition, many applications have batch components that run in a job entry subsystem 2 or 3 (JES2/JES3) environment, orchestrated by job control language (JCL) and controlled by schedulers, such as CA-7 Workload Automation (CA-7), CA-11 Workload Automation Restart and Tracking (CA-11), BMC CONTROL-M for Distributed Systems, and so on). Rehosting has evolved as a complete methodology to address all of these mainframe technologies surrounding online and batch applications, leveraging a combination of emulation solutions (where possible) and automated migration (where required, for example in migrating a 4GL to COBOL and then rehosting it off the mainframe, or in migrating PL/I to C before rehosting it to open system platform).

As a proven and low-risk approach requiring minimal change, rehosting is best used when most of the application meets business requirements and quick migration can provide significant cost savings by eliminating, reducing, or containing MIPS consumption on the
mainframe. When migrating a small-to-medium mainframe environment, rehosting has been successful in significantly reducing the use and cost of the mainframe, and sometimes in eliminating the mainframe dependency altogether. For larger mainframe shops, rehosting for MIPS containment provides enough breathing room to avoid—or at least defer—an expensive mainframe upgrade.

The rehosting approach is based on

- Migrating the application off the mainframe to a compatible software environment on an open systems platform
- Preserving the language (if it is COBOL or C) and middleware services on which the application has been built

Rehosting protects legacy investments by relying on a mainframe-compatible software stack to minimize any changes in the core application and preserving the application’s business logic while running it on an open system operating system (OS) that has a more flexible and less expensive system infrastructure. Rehosting also keeps the customer’s options open for SOA enablement and rearchitecture by using a SOA-ready middleware stack to support Web services and enterprise service bus (ESB) interfaces for rehosted components. By using an extensible software platform that is open to integration with Java, J2EE, or .Net components; has BPM-based processes; and uses other key tools of the rearchitecture approach, customers can start rearchitecting selected components to support targeted functional changes at any time and still maintain strong integration with the rehosted services that are running the bulk of the business logic.

Reducing or eliminating the legacy mainframe costs and risks via rehosting can also help customers fund subsequent SOA enablement and rearchitecture aspects of legacy modernization and lay the groundwork for these approaches. Enabling SOA for a rehosted application is a much easier process on an open-systems–based, SOA-ready software platform. It is more efficient in its use of resources utilization, and therefore less expensive.

Rearchitecting selected components of a rehosted application is a lower-risk approach than rearchitecting the entire application. The risk can be further reduced by making sure that the target rehosting stack provides rugged, transparent integration between the rehosted services and the new components.

Three key success factors emerge from numerous mainframe rehosting projects:

- Leverage automation and proven migration practices to reduce project risk and its duration and to ensure a predictable schedule for cost reduction and demonstrable business value
- Preserve the application’s business content—both the application logic and the application data—to achieve functional equivalence sooner, minimize business impact, and avoid retraining
- Architect on an open system target environment to ensure mainframe-class reliability, availability, scalability, and other quality of service requirements and to ensure that migrated applications continue to meet performance and availability service level agreements

Meeting these needs requires a uniquely powerful database and application platform—one that natively supports key mainframe languages in a mainframe-compatible online and batch runtime; enables automated migration of application code and data; and delivers proven, mainframe-like quality of service on open systems. This database and application platform must also provide added flexibility so rehosted applications can be rapidly integrated in a SOA, and must be able to extend those applications with Java or .Net components as the needs of the business evolve.
Oracle’s rehosting solution combines Oracle Tuxedo (as the mainframe-compatible environment for hosting COBOL or C business logic) and Oracle Database. This mainframe-class platform natively supports COBOL and provides an execution environment compatible with the mainframe’s IBM CICS or IMS TM for online applications and provides a JES for batch. (See the Oracle white paper, *Mainframe Rehosting with Oracle Tuxedo: Accelerating Cost Reduction and Application Modernization*, for details.)

The robust foundation provided by Oracle Database and Oracle Tuxedo deployed in Oracle’s Maximum Availability Architecture on Oracle software technology ensures availability and performance that’s equal to or better than on a mainframe, and provides significant scalability advantages at a fraction of the cost.

During the migration process, rehosting carries a lower risk than attempting to migrate a large and complex application to a different language or architecture. Rehosting is an optimal choice when you need to preserve the core business logic and data of the mainframe application because the company continues to rely on the business processes supported by the mainframe. Rehosting eliminates the need to maintain antiquated skills for mainframe maintenance and can modernize the application platform and interfaces, making it easier to extend functions and integrate with other systems.

Rehosting can result in significant cost savings. While these savings can be achieved relatively quickly, the rehosting approach (as with the enabling SOA integration and the offloading MIPS approaches) retains some legacy architecture and procedural programming languages, which can present ongoing business risks because of increasingly rare legacy skill sets and potentially cumbersome implementation of services.

With a focus on deriving the maximum strategic business value while mitigating the risks presented by retaining legacy code, the rehosting approach is typically employed as the first step in strategic, multiphased IT modernization projects and can be followed by SOA enablement, partial rearchitecture, and other modernization steps. Budget savings from the rehosted applications can be invested in these subsequent modernization activities that offer additional business benefits. Rehosting can also be used to close the gap between the custom legacy application set and a packaged solution by providing a path forward for the existing functionality not supported by COTS applications.
# IT Modernization Approaches: Advantages and Disadvantages

<table>
<thead>
<tr>
<th>APPROACH</th>
<th>DESCRIPTION</th>
<th>ADVANTAGES AND DISADVANTAGES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Replacing legacy applications</td>
<td>This approach to IT modernization is considered most often. Replacing legacy</td>
<td>Advantages: Highly cost-effective.</td>
</tr>
<tr>
<td>with packaged applications</td>
<td>applications with Oracle Applications could include horizontal applications — such as Oracle E-Business Suite, Oracle's Siebel Customer Relationship Management (CRM), and Oracle's PeopleSoft Human Capital Management — along with industry-specific applications.</td>
<td>Disadvantages: Not possible with home-grown applications that have unique features and are critical to the business.</td>
</tr>
<tr>
<td>Enabling service-oriented</td>
<td>This approach consists of wrapping legacy application services in place and</td>
<td>Advantages: Noninvasive to the legacy application. Legacy components can be used quickly as part of the SOA infrastructure, with very little risk.</td>
</tr>
<tr>
<td>architecture</td>
<td>presenting them as Web services to an enterprise service bus. It provides</td>
<td>Disadvantages: The legacy code that implements SOA services remains unchanged, so the problem of maintaining a legacy environment remains.</td>
</tr>
<tr>
<td></td>
<td>immediate legacy integration to Oracle SOA Suite.</td>
<td></td>
</tr>
<tr>
<td>Rearchitecting legacy</td>
<td>This approach builds a new system on the side to replace the legacy system,</td>
<td>Advantages: Maximizes the benefits of SOA and new technology capabilities.</td>
</tr>
<tr>
<td>applications</td>
<td>integrates it with the old system, and then eventually shuts down the old system.</td>
<td>Disadvantages: Very costly.</td>
</tr>
<tr>
<td>Automating the migration of</td>
<td>This approach runs existing code through a utility to generate code in a new</td>
<td>Advantages: Speed and consistency.</td>
</tr>
<tr>
<td>legacy applications</td>
<td>language (for example, from COBOL to Java).</td>
<td>Disadvantages: Makes only algorithmic transformations. Not the right choice when the goal is to make major changes to architecture and application design.</td>
</tr>
<tr>
<td>Offloading MIPS</td>
<td>This approach offloads expensive mainframe MIPS to a mid-tier server through</td>
<td>Advantages: Reduced mainframe utilization and costs and increased application performance.</td>
</tr>
<tr>
<td></td>
<td>the use of an in-memory data grid (Oracle Coherence).</td>
<td>Disadvantages: Retains all of the legacy architecture and programming languages, forcing a continued reliance on legacy skill sets.</td>
</tr>
<tr>
<td>Rehosting</td>
<td>This approach to IT modernization migrates an application as-is to another</td>
<td>Advantages: Moves an application to another platform without changing the core application.</td>
</tr>
<tr>
<td></td>
<td>platform, while leaving the core application essentially untouched.</td>
<td>Disadvantages: Retains much of the legacy architecture and programming languages, forcing a continued reliance on legacy skill sets.</td>
</tr>
</tbody>
</table>
Conclusion

All organizations want to realize the highest business value possible from their existing investments in IT infrastructure and application software. However, maintaining the application and infrastructure software in a legacy environment consumes a disproportionate percentage of IT budget and human resources. The average company spends from 60 to 85 percent of its IT budget maintaining legacy applications that fail to meet the changing competitive needs of the business.

Oracle IT modernization approaches, as detailed in this white paper, support the complex variety of legacy modernization challenges and provide an IT environment that is both economically efficient and able to support an organization’s strategic business goals.

Using Oracle’s complete, open, and integrated software applications, organizations can implement business systems that align IT with business strategies and can grow the organization’s top and bottom lines.