
The System Architect's Essential Role in Open Systems Implementations

Cary V. Millsap
Oracle Corporation

May 19, 1998

The enormous variety of technology options available in today's open systems environment makes the role of *system architect* more critical than ever on both large and small projects. More component options have led to greater flexibility and lower component prices, but also to greater complexity and, in turn, to greater risk of implementation project failure or delay due to technical miscalculations or inappropriate assumptions. *System architecture* is the discipline that addresses the difficult technical issues inherent in a complex software system implementation.

In spite of the importance of system architecture design, many information systems practitioners lack adequate understanding of this subject area, resulting in many costly mistakes throughout information technology organizations worldwide. This paper explains the types of technical challenges the system architect must address. It illustrates the role of the system architect, the type of person required to fill that role, and the types of authority and information access that this person needs to perform effectively in the role. Finally, it describes procedures and an approach that a good system architect uses to reduce the technology risk in a software application implementation project.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the Oracle Corporation copyright notice and the title of the publication and its data appear, and notice is given that copying is by permission of Oracle Corporation. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1997, 1998 Oracle Corporation. No Oracle part number has been assigned.

Executive Summary

Open systems implementers bear more technology risk than their predecessors. The system architect's job is to mitigate this risk. To do this, the architect must make decisions in every phase of the implementation project to ensure that the resulting integration of hardware, software, configuration options, procedures, and people will deliver the desired functional service and system performance to the business. By allocating at least ten percent of an average implementation budget for system architectural services—twenty percent or more for innovative architectures—an implementation project team reduces its technology risk and dramatically improves the resulting system's probability of meeting its functional, performance, cost, and flexibility requirements.

Contents

1. INTRODUCTION
2. WHAT IS ARCHITECTURE?
3. THE ARCHITECT'S ROLE
4. MANAGING TECHNOLOGY RISK
5. WORKLOAD AND CAPACITY
6. ARCHITECTURE TASKS
7. APPROACH
8. CONCLUSION

1. Introduction

The traditional view of computer system architectures is the result of a decades-old culture of buying ready-made hardware and software suites off the shelf from an industry giant. IBM and other mainframe vendors sold preset system configurations with applications whose performance was generally consistent from one implementation to the next. If something didn't work for you, then your one-stop hardware, software, and service provider would work hard to *make* it work for you. In this environment, *architecture* was something that somebody else worried about for you.

Today, open systems vendors fulfill their promise of universal data management, intuitive graphical interfaces, rapid software customizations, ad hoc inquiry, and applications that are capable of doing business the way *you* do business. However, in the flexible open systems marketplace, configurations are not preset, and system performance has become unique to every implementation. The flexibility of open systems burdens the customer with a new set of risks in the areas of architecture design and validation. In this paper, we shall discuss how to mitigate those risks.

2. What is Architecture?

Computing professionals have adopted for themselves the word *architecture*, which for centuries has been the art and science of designing and erecting buildings.¹ Although the end result of any architect's work is an element in the physical world, the architect's *art* is an abstract one that begins with determining the client's requirements.

Architecture is a tricky business, because contrary to the first law of retail, the architect's customer is *not* always right: the architect must seek a better understanding of his client's true requirements than the client has himself. For example, a client may dream of a 90-foot tall glass foyer, but is his dream important enough to him to outweigh the dramatic difference in heating and cooling costs relative to a more conventional foyer design? An application system client may want a distributed database application on multiple NT clusters, but does he want it badly enough to budget the enormous incremental IT spend required to run it relative to a more conventional UNIX centralized server design?

The architect that builds exactly what you ask is not a good architect if the resulting system—whether it's a building or a software application—doesn't meet the requirements that you'll face in reality. The architect must navigate the boundary between stated requirements and the constraints imposed by natural law, continually anticipating the hidden costs and forecasting the benefits of each decision he makes along the way.

Being a successful architect is all about endowing an end result with the right attributes. With buildings, many of the required attributes are easy to "see," like

¹ From *The American Heritage Dictionary of the English Language, 3ed.* Houghton Mifflin, Boston MA.

floor space, plumbing, appliances, hardware, and the building's overall style. Other important attributes require greater depth of study to appreciate, such as structural integrity, operational cost efficiency, and subtle things that only a few elite commercial architects understand, such as the way adjacent tall buildings can channel a light breeze into a howling, irritating wind at their bases.

Like buildings, software application systems also have some attributes that people can "see," such as functional behavior and user interfaces, but most attributes of a successful application are abstract and more difficult to understand. An application system achieving the highest standard of quality must possess every one of the following four key attributes:

1. *Functional integrity*—production of complete, functionally correct output. Measures include bugs, defects, completeness, and ease of use.
2. *Performance, including availability*²—production of sufficient output volume within the system users' response time tolerances. Measures include throughput, response time, uptime, and downtime.
3. *Manageability*—execution within the system owner's cost boundaries. Measures include procurement costs, construction costs, operational costs, and upgrade costs.
4. *Flexibility*—a system's ability to adapt to changing requirements without violating functional correctness, performance, availability, or cost requirements. Measures include upgradeability and scalability.

Negotiating the tradeoffs among the constraints imposed by these four attributes is a difficult job at which many implementation project teams fail. At least two factors contribute to these failures.

First, the multidisciplinary technology expertise required to design an application is rare. System architecture demands subject matter expertise not commonly found in the training or experience of traditional functional analysts and software developers. Good software system architects must have knowledge in hardware, networks, operating sys-

tems, software design, database internals, system performance modeling, application performance optimization, business forecasting, technology risk management, and overall program management.

Second, technology expertise in the specialized field of system performance is rare. The performance, availability, or operational cost efficiency impact of a functional design decision is not intuitive to most analysts until the impact manifests itself as a real problem. An implementation project executed without competently designed interim performance and availability milestones looks successful to the untrained eye, right up to the time it fails on Day One of its production status. Yet, experience has proven repeatedly that the operational phase of an implementation is the most expensive time to detect and correct a system inadequacy [Boehm 1981, p40]. Sometimes in the late phases of a project, there's just no tolerably painless way to make a bad implementation better.

3. The Architect's Role

Many unsuccessful open system implementations can trace their failures back to an unsolved system performance problem that began as an unidentified or unmitigated risk and ended as an unexpected career change. Performance problems are particularly important to avoid because performance impacts the work life of *everybody* who uses a system. The architect's job is to focus on the attributes of performance, availability, manageability, and flexibility from project start to project finish—from the early requirements analysis phases of your project all the way into its production operation.

By focusing a talented architect on these system attributes, of course, you maximize your probability of attaining them. Without a system architect, many of the issues like those shown in the box on page 5 will simply go unanswered until a project failure risk becomes so obvious that circumstances force someone to attend to it. To successfully negotiate these types of issues, the architect must be an experienced leader who possesses these key attributes:

- Multidisciplinary technology expertise that enables the architect to optimize among the domains of business processes, application functions, database services, system software and operations, hardware, and networks.
- The vision to formulate a coherent business solution to meet *all* of the owner's requirements,

² *Performance* and *availability* appear in a single category because performance and availability metrics are interdependent. For example, any measure of throughput expressed in events per unit of time (e.g., transactions per second) necessarily depends upon application outages (consider the effect of an hour's downtime upon a day's throughput) [Gray and Reuter 1993; Gunther 1998].

from the universe of possible component combinations.

- The leadership skills to guide programs and projects in the right technology direction and execute those programs or projects according to plan.
- The problem-solving skills to focus the team's attention swiftly and accurately onto the correct areas of action to solve system technology oriented project problems.

The architect's team must be an appropriately chosen collection of people who can execute tasks including the following:

- Record service level requirements: interview users, gauge the technical feasibility of business desires, record the requirements in a measurable format, and so on.
- Review application designs for technical attributes: performance, availability, manageability, and flexibility.
- Test performance of individual application modules: create test plans, generate test data, create transactions to use in testing, execute the tests, interpret their results, force the issue of performance repair in areas jeopardizing the project, and so on.
- Repair inefficient SQL: trace interactive and batch sessions, identify the bottlenecks, implement SQL changes to improve performance, recommend schema changes, and so on.
- Manage system workload and capacity: monitor system performance, limit system workload, assess hardware capacity, forecast future workload, and so on.
- Manage the development, test, and production systems: run the operating system and the database, create operational procedures, manage space consumption, manage access, backup and recover the system, produce operational statistics, and so on.

The proportion of the project budget that the architect's team consumes will vary with the complexity of the implementation. Most implementation projects with values less than US\$10 million require allocation of about ten percent of their project budgets to architecture services (roughly one employee dedicated to system architecture and system management tasks out of every ten full-time software application

implementation team members). For projects in which the implementation will require either unproven system components or proven components arranged in an innovative way, system architecture tasks may consume twenty percent or more of a project's budget:

- *n*-tier implementations exploiting middleware or other technologies that require custom development or customization of basic services;
- implementations based upon new or unproven technology in any layer of the stack (including networks, hardware, operating systems, and database or application software);
- application architectural customizations, such as those required when implementing integrated database applications upon databases that employ distributed data or distributed processing;
- implementation programs (collections of implementation projects) that span multiple sites, which require data or processing interchange, especially in environments requiring multilingual support;
- applications whose workload will possibly exceed the capacity of the largest available system configuration.

The abstract attributes of performance, availability, manageability, and flexibility are critical to the success of an implementation project. Attending insufficiently to system architecture tasks can open your project to significant failure risks at the worst possible time: when your business must rely upon your new system. The architect's job is to protect you from those risks. The next section of this paper is dedicated to helping you understand the value of the architect in your implementation project.

Sample Issues for the System Architect

Project management

- When to define the production configuration?
- How to phase the hardware construction?
- How to phase the software rollout?
- When and how to test performance?
- How to mitigate go-live date risks?
- What special support for go-live?

Hardware and application topology

- Centralized or distributed processing?
- Centralized or distributed data?
- MPP, SMP, or single-CPU configuration?
- One-tier? two tier? three tier? n -tier?
- What network protocol?
- What type of middleware?

Hardware capacity

- How much CPU capacity for batch database calls?
- How much CPU capacity for online database calls?
- How much disk storage capacity?
- How much I/O throughput capacity?
- How much shared and unshared memory capacity?
- How much network throughput capacity?

Hardware configuration

- How much virtual memory?
- Which components to duplex?
- Which RAID levels for which disk arrays?
- File systems or raw devices?
- What striping units and array sizes?
- What O/S, LVM, and database block sizes?
- What offline storage devices?
- What packet sizes and timeout intervals?

Application design and configuration

- Is the business process flow optimized?
- Is the data model optimized?
- Is the SQL optimized?
- What data archival and purging procedures?
- Are the processes in each tier optimized?
- What chart of accounts configuration?
- What summary templates?
- What approval processes?

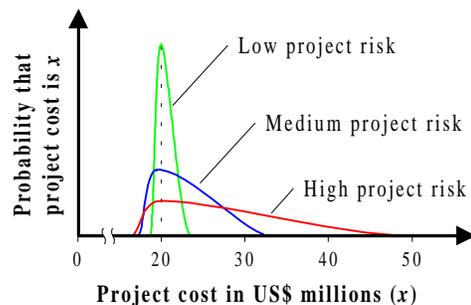
Operational procedures

- What operational readiness preparation?
- What system management training?
- What backup and restore procedures?
- What change control procedures?
- What workload management procedures?
- What job queue management procedures?
- What performance monitoring procedures?
- What space management procedures?
- What capacity planning procedures?

4. Managing Technology Risk

Risk is product of impact and probability. It is the impact of a given problem multiplied by the probability that that problem will occur. Because risk includes the notion of a probability, risk exists because of incomplete information. When complete information about a problem is known, its probability of occurring becomes either zero or one, and then the problem ceases to be a risk—it becomes either an irrelevance or a certainty. Risk is a quantitative measure of information incompleteness.

Imagine a large system implementation project that you believe would cost about twenty million US dollars. Imagine asking thirty different project teams at thirty different companies around the world to execute this project. Not all of the thirty project teams would come in at exactly \$20 million. Some of the thirty projects would be over budget, and some would probably come in under budget. If you were to plot a probability histogram of the costs of these thirty projects, you would produce a curve resembling one of the three curves shown below.



The tallest curve represents a small variance in project cost across the thirty implementations; the narrowness of the curve indicates that all of the thirty project costs came in close to the \$20 million budget. The risk of the project represented by this curve is low; if you executed the project a 31st time, you would expect the cost to be very close to \$20 million. Perhaps the project implemented a stable, high-quality, pre-packaged application; and perhaps the workload requirements upon that application were relatively undemanding.

In the middle curve, many more of the thirty projects ran over the \$20 million estimate; the risk inherent in the project described by this curve is greater than that of the narrowest curve. In this type of project, perhaps some project teams consumed more resources than they had originally expected because they had to redesign or re-implement major pieces of

their projects. Perhaps they had to replace their hardware shortly before going live because they had underestimated the amount of computing power the application would require. Or perhaps they spent a lot of time trying to diagnose and repair a database latch contention problem that eventually relented to a SQL optimization effort.

In the type of project described by the shortest, widest curve, the risk is treacherous—most of the thirty projects ran over budget, and many of the project teams went over by 100% or more. Perhaps the project owners discovered after losing valuable business that their application's performance was intolerably poor. Perhaps they sank huge amounts of resources into trying to patch up the implementation before they determined that the application and its architecture would require significant redesign before it could accommodate the workload generated by their business.

You may never be afforded the luxury of observing the behavior of thirty or more historical projects to determine your exact project risks, but this thought experiment can help you understand the behavior of risk in a project. Software application implementations can bear minimal risk, treacherous risk, or any amount in between. Common technology risks include:

- Will the system be fast enough to support the transaction rates and user response times that your business plan requires?
- Will the system be reliable enough to support the high availability schedule that your business plan requires?
- Will the system be cost-effective enough to serve your business within your cost constraints?
- Will the system be scalable enough to accommodate the growth rates your business expects in user population, data volume, network communication, and new application implementations?

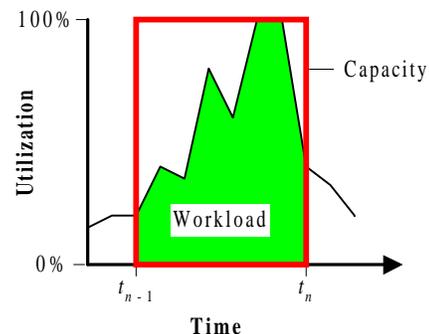
Risk is a measure of the unknown perils that threaten your project. The easiest way to avoid risk is to execute easy projects, like projects that have been done before elsewhere, or projects whose goals are not particularly ambitious. However, avoiding risk isn't your job. Your pursuit of market advantage can easily lead you into an aggressive technology risk portfolio. Your job is not to *avoid* risk; your job is to *manage* risk.

Obtaining more complete information is the best way to mitigate your technology risks. Smart people who can forecast the behavior of your application early in the analysis and specification phases of your project are your most efficient keys to mitigating technology risk. People who can manage system performance throughout the entirety of your project, from analysis through production, through system operation, are key to building a robust application system that will support your business requirements in the shortest time, at the lowest cost.

The next section is dedicated to helping you understand the focal points of the architect's attention. You will learn why the job of providing system performance requires an architect's attention beginning in the very first conceptual stages of an implementation project.

5. Workload and Capacity

System performance is the result of ensuring that the desired workload fits within the available capacity. *Capacity* is the amount of a service that a configuration can provide to an application, and *workload* is the amount of that capacity that the application consumes. All of the system architect's work toward providing system performance emanates from understanding of capacity and workload. Capacity and workload are easy to identify in a standard utilization graph like the following:



The system capacity from time t_{n-1} to time t_n is the area of the rectangular box that represents 100% utilization for that time interval. The workload for the same interval is the area beneath the utilization curve for the interval. For a system to provide satisfactory service, its desired workload must not exceed its capacity. If desired workload exceeds capacity for a given time interval, the system will process as much workload as it can, and the unprocessed workload will “spill over” into a later time interval.

In the graph above, note that just to the left of time t_n , the workload reached 100% of the available system capacity. It is highly likely that at this time, the system's users noticed a degradation in system performance until the system "caught up" with the workload.

Software application architects regard system capacity in four dimensions: CPU, memory, disk, and network. The system architect's job is to construct a system on which workload is optimized and capacity exceeds the largest workload that the system must accommodate. To do this, the architect *quantifies* both the anticipated workload and the expected capacity of the system being implemented.

Quantifying workload and capacity presents difficult challenges, because workload and capacity are normally expressed in different units. For example, a business that sells automotive components might express part of its workload in terms of sales orders processed per day, but the capacity of that business' disk drives is expressed in terms of read and write calls per second (called *I/Os per second*, *IO/s*, or *I/Ops*). The task of converting orders per day into IO/s requires time and a thorough understanding of the application, the database software, and the specific I/O subsystem being considered. A similar type of analysis is required for memory and network design.

Analyzing CPU workload and capacity is more difficult, because there is no universally accepted, useful unit of CPU capacity like the IO/s unit published by disk manufacturers. Most CPU manufacturers quote CPU performance in MHz (megahertz, or millions of cycles per second) or some other measure of the number of instructions they can process in a second. The problem with this metric is that there is no universal definition of an "instruction"—an instruction on an apparently fast CPU may represent a fraction of the capacity represented by an instruction on an apparently slow CPU. Even expressing CPU power in units that resemble your application (like one of the TPC benchmarks) is not a consistently useful way to express CPU capacity, because you can be practically assured that there's not a benchmark that does exactly what your application is going to do.

Oracle architects have generally converged upon a unit called the *logical I/O (LIO)* as their unit of choice for analyzing CPU workload and capacity. The LIO is a unit representing a single call to the database server, and a LIO executed by one process generates approximately equivalent workload as any LIO executed by any other process. However, con-

verting a business metric like "orders per day" into LIOs per second (LIO/s) and then LIO/s into the hardware vendor's preferred capacity units is a challenging task often requiring small-scale testing to produce a reliable estimate.

The next challenge is to quantify the *entire* workload. We may have a workload forecast expressed in "orders per day," but we also know that order processing makes up only part of the system's total workload. Quantifying the remainder is generally more difficult than quantifying the focal workload. Other sources of workload include system overhead, other applications like accounting or reporting, and system management overhead. And of course, the accuracy of the workload forecast will degrade over time if the forecast is not periodically re-analyzed, because there are always frequent workload-altering events that were unanticipated in the original analysis. Deciding to run that monthly GL trial balance report every day can change everything.

One strategy to handle these difficulties is to design a system that will handle the anticipated peak workload with some spare capacity to accommodate additional workload. The system architect can institute a structured *workload discipline* consisting of procedures that restrict the amount of unplanned workload that people will be permitted to execute in a given time interval.

A final hard challenge that an architect can help you overcome is that workload is a highly personal measure of the specific use of an application. It is generally *unreliable* to forecast one company's workload based upon the workload of another company, even if they use identical applications and identical hardware. Workload depends upon several factors whose influences are difficult to quantify, including:

- *Business volume*—Companies with different business volume requirements will generate different workloads even if they're using identical applications with identical hardware and software configurations.
- *Business policy*—Different business policies drive huge workload differences among different companies.³ For example, two similar implementations will generate dramatically different workloads if one company's CFO requires a

³ Changing business policies can even drive huge workload differences at the same company from one month to the next.

nightly detailed trial balance report, and the other company's CFO requires only one such report a month.

- *Software configuration*—There is no such thing as a “standard workload” for flexible, configurable applications. For example, Oracle's financial applications provide options that allow users to configure their accounting structure any way they like. Two similar implementations that use slightly different configuration options will use different SQL statements for a given operation. Different SQL statements, of course, produce different workloads.
- *Hardware configuration*—Different hardware implementations generate different workloads. For example, a RAID level 5 disk implementation will generate two times more disk workload for a random write call than the same application implemented with mirrored disks. Two similar implementations that use slightly different hardware configurations will generate different workloads.
- *Data distribution*—Though relational database query optimization technology is advancing rapidly, the physical distribution of values within the database still affects the workload required to retrieve those rows. For example, imagine two manufacturing firms with identical business volumes, one with one central warehouse and the other with one thousand distributed warehouses. The workload generated by SQL joins on the warehouse table in these two implementations will differ dramatically.

Because these factors are so difficult to quantify, workload forecasting almost always requires tasks for custom analysis to be allocated in the implementation project plan. Workload forecasting forms the basis for the architecture tasks that are described in the next section.

6. Architecture Tasks

The system architect must engage in several tasks that amount to *configuration planning*. Configuration planning includes tasks that cross both the functional service and system performance domains for the duration of an implementation project, like determining...

- how to partition data and processing among tiers or nodes to optimize throughput, response time, and availability at the best economic value;

- how to match the network topology, the disk configuration, and other hardware configuration elements to the demands of the application;
- how to configure an application's software parameters (e.g., configuring an accounting application's chart of accounts) to optimize system performance without sacrificing business autonomy and accessibility requirements;
- how to optimize the economic decisions among options to buy more powerful hardware, invest into application software optimization, change workload priorities, or modify business requirements;
- how to implement and maintain a structured but non-intrusive change control discipline that includes performance testing and a feedback loop between the testing process and the system configuration design process;
- or how to organize the system's operational staff so that sufficient attention is focused upon the tasks of system management, from ensuring that system parameters are continually adjusted to meet changing workload demands, to execution of a batch queueing discipline to meet business needs while optimizing system throughput.

The task of determining how much capacity a configuration will require is called *capacity planning*. A system's initial capacity planning process is sometimes called *hardware sizing*, because a prominent output of the initial capacity plan is the purchase order for system hardware. A valid capacity plan depends upon the valid output of two interrelated processes:

- *service level requirement specification*, in which the system designers specify the functional service and performance requirements of the system;
- and *workload forecasting*, in which the capacity planner must forecast a system's workload using information about the system's functional service requirements.

Because workload forecasting is so difficult, many implementation teams make the mistake of executing these two processes independently, even if, to their credit, they attempt to execute them at all. The service level requirement specification and workload forecasting processes should be inextricably linked, because executing them independently often leads to

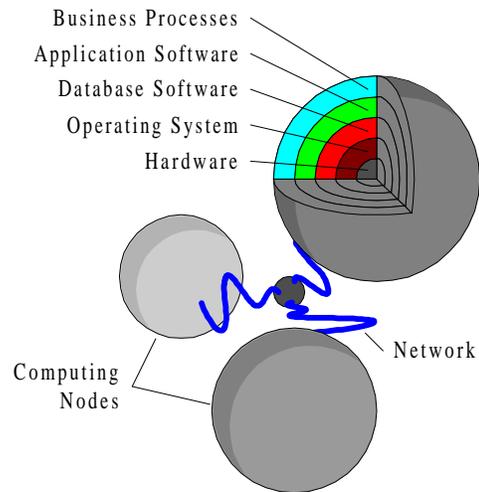
requirements that contradict each other or possibly even violate the laws of physics.⁴

During the entire configuration and capacity planning process, the architect must continually attend to *workload optimization*. Workload optimization is the process of adjusting system components to consume the least possible amount of capacity to meet a business need. Remember, system performance comes from ensuring that capacity exceeds workload. There are two ways to win that game: either increase the capacity, or—the important task we're highlighting here—eliminate inefficiencies in the workload.

The less experienced the architect, the greater his tendency to focus strictly upon the purely technological layers of an application system, such as the applications SQL or the database configuration. More experienced architects exploit opportunities in every layer. For example, an experienced architect is able to...

- provide cost/benefit justification for determining whether specific desired functional services are economically feasible to deliver;
- provide a means of replacing detail reports with more efficient summary reports to conserve wasted capacity;
- help front-end software designers to reduce network bandwidth consumption requirements;
- introduce a sensible batch workload discipline to balance workload across idle periods to improve peak period throughput;
- recommend application schema or SQL optimizations to reduce database call volumes;
- propose I/O subsystem configuration optimizations that will preserve system capacity for each application I/O call.

⁴ I am reminded of an ill-planned service level agreement in which a data center signed up to providing sub-second response times for a client-server transaction executing three round-trips to a server that was 10,000 miles away. By the time you deduct the 0.3226 seconds burned by electrons traveling the world encumbered only by physical restrictions on the speed of light, there's not much time left in the sub-second requirement for database calls, I/O processing, and the inevitable network loads that force WAN latencies into performance that is worse than theoretical perfection.



The degree of holism in the approach to workload optimization is what distinguishes workload optimization from mere *performance tuning*, which is the process of optimizing the purely technical components of a system. Systemic workload optimization—a process in which users, business policy, and system management procedures are recognized as vital system components—is a core skill of the professional system architect.

Both workload forecasting and workload optimization are components of a more pervasive policy of *workload management*. An optimal workload management policy requires that you either optimize or discard workload that jeopardizes system performance, or that you increase capacity suitably before allowing such workload to enter the system. Workload management tasks include...

- for *existing workload*, continual measurement to detect changes that could impact system performance,
- and for *new workload*, the creation of a performance testing discipline so that workload added to a system will not violate system requirements.

A discipline of workload management is the key to maintaining system performance in an environment that includes changing table sizes or data distributions (from insert, update, and delete activity), changing schema definitions (such as from software upgrades or index optimizations), or changing SQL (from additions or changes to reports, transactions, or applications). Workload management is a critical full-time job during an open systems implementation

project, and it remains an important responsibility of the database manager throughout any implementation's operational lifespan.

Another major system architecture component is *hardware subsystem design*. In this task, the architect designs a hardware subsystem that will meet the functional service and performance requirements defined in the service level requirement specification and workload forecasting processes. For example, the I/O subsystem design process results in a blueprint that tells the hardware installer how many disks to install, how large each disk should be, what the disk array sizes should be, what RAID organization to use for each array, what striping unit to use for each array, and so on. Similar subsystem design tasks are required for CPU, memory, and network subsystem sizing.

Finally, the activities of *system management* also fall within the system architect's domain. System management activities include all of the operational tasks required for the system to provide ongoing functional service and system performance to its users. These tasks include system performance management, online and offline data storage management, fault management, configuration management, and software distribution.

To summarize the architect's key tasks:

- *Configuration planning*. Configuration planning is the process of determining the appropriate architectural topology to meet the system's defined functional integrity, performance, availability, manageability, and flexibility requirements. The architect's involvement in the configuration planning process begins with requirements analysis and includes any configuration decision that affects system performance.
- *Capacity planning*. Capacity planning is the creation of a detailed plan to ensure that adequate software, procedure, personnel, and hardware resources will be available to meet the future workload demands in a cost-effective manner while meeting the performance objectives. The initial capacity plan is sometimes called *hardware sizing* because a prominent output of the process is the initial specification for hardware.
- *Service level requirement specification*. Service level requirement specification is the process of documenting the functional integrity, performance, availability, manageability, and flexibility

requirements of a proposed new workload, resulting in the creation of a service level agreement (SLA). Service level requirement specification is inextricably linked to the workload forecasting process.

- *Workload forecasting*. Workload forecasting is the process of forecasting a system's workload using information about the system's service requirements. Workload forecasting and service level requirement specification must occur before configuration planning, capacity planning, or hardware sizing can occur. Workload forecasting is an important part of an overall discipline of workload management.
- *Workload optimization*. Workload optimization is the process of adjusting hardware, software, or usage to consume the minimum capacity necessary to meet a business need. Workload optimization is a superset of *performance tuning*, which is the process of optimizing the purely technical components of a system. Workload optimization is another important part of an overall discipline of workload management.
- *Workload management*. Workload management is an overall discipline in which workload components are continually evaluated for their impact upon system performance, and in which workload components are prohibited from driving system performance in violation of required levels.
- *Hardware subsystem design*. Hardware subsystem design is the task of planning the configuration of a CPU, I/O, network, or memory subsystem that will meet the system's throughput, response time, reliability, and storage capacity requirements.
- *System management*. System management activities include all of the operational tasks required for a system to provide ongoing functional integrity, performance, availability, manageability, and flexibility to its users, including system performance management, online and offline data storage management, fault management, configuration management, and software distribution.

A system consists of more than hardware and software components. The people who manage and use the system, the way they are organized administratively, and the procedures they use are also vital system components to which the system architect

must attend. In the design and construction of an application system, the architect must guide the acquisition and construction of system management tools and procedures. The architect must work to ensure that the system management staff and users are properly trained in techniques that optimize system performance, and that they are sufficiently well-rehearsed and ready to execute them.

The system architect's tasks often blur together during the course of a project. Because the architect shoulders most of the technology risk in an implementation project, he often takes an iterative approach to his job that results in many small cycles through the entire architecture process, such as in the example in the box on page 11. We will study this iterative approach in the next section.

Example: Iterative Architecture Design Steps

1. Specify requirements.
2. Forecast workload.
Learn from this process that the requirements violate physical laws.
3. Modify requirements specification.
4. Plan configuration.
Learn from this process that the planned configuration will exceed the cost requirements.
5. Modify requirements specification.
6. Modify configuration plan.
7. Forecast workload.
Learn from this process that the workload will exceed the configuration's attainable capacity.
8. Optimize workload.
9. Modify workload forecast.
10. Plan capacity.
Learn from this process that the workload still exceeds attainable capacity.
11. Modify configuration plan.
12. And so on.

7. Approach

The best results in large, complex implementations have come from adhering to the following simple guidelines:

- Test performance.

- Use models to reduce the cost of forecasting.
- Prove concepts in small steps.
- Use talented and experienced architects.
- Give the system architect authority in the decision-making process at the enterprise level.

Performance tests are spectacularly effective in their ability to motivate implementation project teams to solve performance problems early in a project while time is still available to fix them at a reasonable cost. Performance testing also has the equally important advantage of requiring your system management team and your users to become intimate with the specification, construction, and operation of your system. Companies who execute performance tests in their implementations have more effective system managers and users than companies that do not test adequately. Through the experience gained in performance testing, the system managers become better trained and better prepared to deal with the real issues inherent in their specific implementation. And the users become more efficient consumers of the functional services and system performance capabilities of their system.

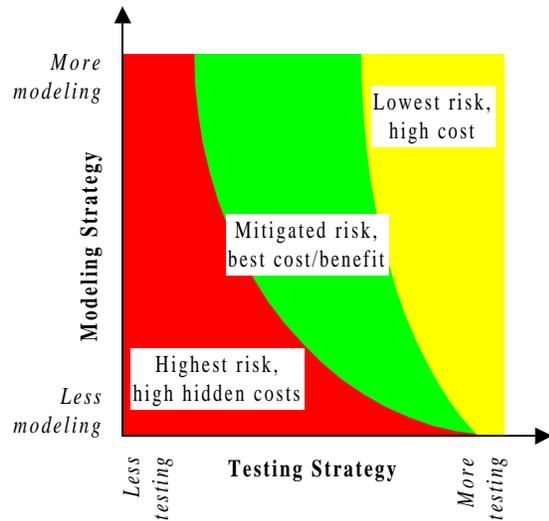
If you had the benefit of a literal full-scale test of your implementation, then you would have the most complete information possible to use in your project. However, the cost of 1:1 full-scale testing is enormous because it requires all of the following elements:

- Full production-sized, fully configured hardware
- Fully configured production-ready software
- Fully loaded production data volume
- Complete production workload
- All users online during testing

Despite the risk mitigation advantages of full-scale testing, the technique is often simply infeasible because of the cost.

Adding *models* to a risk mitigation strategy provides risk reduction at significantly reduced cost. For example, forecasting system throughput capacity with a queueing theory model is significantly less costly than executing even a reduced-scale test with real workload executed by real users upon real data. Modeling is economically attractive because it is much easier to plug numbers into a spreadsheet than to execute a real-life workload. However, models carry the inherent risk that the modeler may have made invalid assumptions about the behavior of the system. To mitigate this risk, you must use talented,

experienced modelers who build valid models, who can determine when testing is necessary, and who can build valid tests.



Next, *iterative* design and construction techniques provide better information much sooner in a project than the traditional waterfall project execution approach. Iterative design is the engineer's method of building a system in modules, testing each one as he progresses, before integrating the tested components into the general architectural framework, which itself has been tested during its design. Iterative project execution, the method of constructing a complex system in small steps, delivers the excellent advantage of allowing the project team to identify and repair small inexpensive mistakes before they grow into big expensive ones. Iterative techniques allow users and system managers to incorporate their reactions to the system's functional and performance behavior into the system's design, long before they're required to live with it. The flowchart in Appendix A illustrates the overall iterative nature of the successful implementation project.

Finally, the architect needs access to information at the enterprise level in an implementation project—especially access to information about business requirement drivers. Without this level of information, an implementation team will often make decisions that may be optimal within a limited decision-making scope, but are suboptimal when viewed in the corporate executive's context.

There are often several ways to accomplish a specific business desire. Given information about the *basis* of a business desire, the good system architect is skilled at determining the most efficient means of fulfilling

it. Users tend to specify new system requirements in terms of services that their former system provided. But in new environments, opportunities often exist to provide equal or superior services in new and different ways that consume less system workload, making the application faster, more reliable, more manageable, and more flexible for a given price.

For example, users often specify requirements for the same detailed paper reports that their legacy system provided. However, in a modern open systems environment, it is often possible to replace a big, slow paper report with a fast, tightly focused online query. In spite of executing potentially millions of times less expensively, an online query often provide better targeted and actually more useful results more quickly to the end user than the legacy report did. By equipping the system architect with information and authority at the enterprise level of decision making, you will maximize your project's opportunities to benefit from dozens of such examples of "more for less" without necessarily resorting to a full-blown business process reengineering effort.

8. Conclusion

The architect's goal is a system that performs satisfactorily with respect to functional integrity, performance (including availability), manageability, and flexibility. The architect's means to this goal is to obtain more complete information about the future performance of a system earlier in the project. If something is going to go wrong, you want to learn about it quickly and inexpensively so that you'll have time and resources to correct it without jeopardizing your deadlines or your budget. A cost/benefit balanced portfolio of testing and modeling, using talented and experienced people, and using an iterative execution method provides this information in your project early at a reasonable cost.

References

- BOEHM, B. 1981. *Software Engineering Economics*. Prentice-Hall, Englewood Cliffs NJ.
- GRAY, J.; REUTER, A. 1993. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Francisco CA.
- GUNTHER, N. 1998. *The Practical Performance Analyst*. McGraw-Hill, New York NY.

Acknowledgments

I extend my gratitude to the following colleagues for their valuable input into the creation of this docu-

ment: Curtis Bennett, Ronald Carnahan, Hal DeLong, Dominic Delmolino, Kimberly Free, Alastair Newson, Mahesh Pakala, Robert Rudzki, and Hugh Ujhazy.

About the Author

Cary Millsap is the founder and senior director of Oracle Corporation's System Performance Group, a team of experienced professionals dedicated to providing system architecture services for Oracle-based systems. The System Performance Group provides premium system architecture services to customers in Oracle's vertical markets segment worldwide. Mr. Millsap is also Oracle's global service line leader for the company's entire line of system architecture and system management services.

Appendix A

