

> FINDING THE RIGHT
SOFTWARE FOR
MANAGING YOUR
BUSINESS-CRITICAL
TRANSACTIONS

TABLE OF CONTENTS

Introduction: Optimizing Business-critical Transactions	1
What Makes a Successful Solution?	2
Asset Management	7
Asset Management Requirements.	8
Automatic Discovery	8
Consumer Discovery	9
Dependency Tracking.	10
Always On	10
Enables Communications Across Teams.	12
Policy Management	14
Performance and Scalability	14
Support for Multiple Protocols and Applications	17
Support for Business Process Tracking	18

INTRODUCTION: OPTIMIZING BUSINESS-CRITICAL TRANSACTIONS

Organizations constantly strive to maximize revenue and minimize costs. From an IT perspective, the way to contribute to these business objectives is to optimize your distributed business-critical transactions. These are the transactions that are vital to increasing revenue or productivity and minimizing costs, yet often the underlying systems lack transparency. Today, these transactions are increasingly service- or SOA-based and distributed—i.e., composed of services that can be quickly reused in other transactions to achieve agility in aligning the IT with new business opportunities and goals. Because of the vital role the underlying services and business processes play and their ability to be reused to great advantage, they should be considered corporate assets and managed as such.

Unfortunately, managing these assets is easier said than done. Business-critical transactions and the services that comprise them typically span systems, databases, and middleware from multiple vendors and are built using a variety of technologies. These service assets are also often distributed, residing in different organizations, under different ownership—and, importantly, not always under the control of the application owner. It's often unclear how changes in one component will affect the overall system (until it's too late). Finally, these systems often demand high performance and scalability to meet the increasingly high demands of around-the-clock, global business. As a result, they are hard to manage. For example:

- > How can you get complete visibility into every step of an application, and how can you track, *in production*, the quality of service for different customers in this diverse infrastructure—and still have a low total cost of ownership (TCO)?
- > How can you detect issues proactively before they occur? And when they do occur, how can you perform root cause analysis quickly before end users experience the issue or business is interrupted?
- > How can you efficiently establish and enforce policies that align your IT infrastructure assets with your business goals, again while achieving a low TCO?

It's worth taking a moment to define "optimize" in the context of distributed transactions. Though specifics vary by vertical market, ask yourself a few questions:

1. *What percentage of your transactions require manual intervention?*
2. *Do you ever have a situation where a transaction simply "gets lost"?*
3. *Do you have revenue-leakage?*
4. *Do you have common process errors, like credit cards being double-charged, or products shipping twice for the same order?*
5. *Does data loss occur (or data flow stop) even when applications are up and running just fine?*

If none of these situations occur in your environment, you're lucky. The examples above are common across many vertical markets. In fact, a recent telco survey discussing changes to the mobile industry in this economic downturn pointed out that revenue assurance projects are as valuable as launching a new service. In addition, they're probably less risky and can be achieved incrementally with less investment than a new service requires.

This buyer's guide explains the core asset management capabilities required to address these challenges *in production* and optimize critical application performance and availability. It should be used as a point of reference for assessing and evaluating vendors and developing a comprehensive request for proposal (RFP) for selecting the appropriate tools and infrastructure to successfully build a component-based infrastructure for delivering mission-critical applications.

First, however, let's look at the characteristics that lead to easy adoption and successful use of a solution.

WHAT MAKES A SUCCESSFUL SOLUTION?

Keeping in mind the focus on the *mission-critical nature* of the distributed application, solutions will have success criteria that go beyond feature/functionality. These criteria, often thought of as solution architecture, should be obvious, but often get "lost in the shuffle." We strongly recommend evaluating value against these high-level success criteria, in addition to key feature requirements, in order to ensure the ability to grow and expand the solution over time. In fact, key features are perhaps less important because, with the right architecture and vendor relationship, features are easy to add. However, the wrong product architecture is often difficult to compensate for and presents problems when addressing enterprise-scale mission criticality.

In fact, it seems that the biggest gap in current offerings is in the inability to run solutions to this "space" in a production environment without changing the application and without having a disastrous impact on the performance and scalability of the applications themselves.

These common or architecture-level success criteria are:

1. Easy implementation
2. Shallow learning curve
3. Low cost of ownership
4. Support for a mission-critical environment with the right scalability, performance, and survivability

5. Robust and meaningful policy capabilities

Each of these criteria is standard for evaluating products, yet there remains no consistent way to measure vendor claims in these areas. Let's discuss a few that are essential.

Easy implementation. A solution is easy to implement if:

1. It can be implemented equally well at any phase of the project.
2. It doesn't require any application-level architecture changes.
3. It is not necessary to configure the solution on a service-by-service basis.

Shallow learning curve. A management product should provide an intuitive visual interface to enable operators to realize benefits quickly. The visual interface should clearly represent the distributed application, provide drill-down into the underlying technical components, and enable key information about dependencies and performance to be easily exported into executive dashboards. The high-level representation of the business-critical application can be used by both business and technical teams to communicate with each other, while the drill-downs and policies enable each team to take action on what they are visualizing.

The visual display should be capable of handling thousands of nodes and services, giving administrators the ability to view the end-to-end application at the highest level and to explore it simply by drilling into the UI. Unfortunately, many products can't scale the UI to provide the enterprise-wide, global, holistic overview that puts everything in context with everything else.

Low cost of ownership. Many features impact the cost of ownership. Here are a few that are critical to look for:

1. Application changes are reflected automatically in the management system; administrators do not need to constantly update service information or business dependencies in the console each time application-level changes are implemented.
2. Performance overhead is low on the managed system, so application platform costs are not increased due to the addition of management.

Also, it is important that the performance/scalability curve is not negatively impacted by the addition of management. Application behavior and architecture should be the same regardless of whether management is present or not.

3. Policies are decoupled from services and applications and, therefore, can be managed independently. Policies that are tightly coupled to applications need to be updated anywhere they are implemented, adding significant costs to managing the policies themselves.

Mission criticality. Admittedly, different applications have different mission-critical metrics. When choosing a solution, you should look for one that won't "top off" before your applications. That refers to performance as well as scalability. Fundamentally speaking, the solution architecture must be consistent whether there is one service or 1,000. "Mission critical" behaviors that should be tested in any evaluation include:

1. **Performance.** Can the solution perform to and beyond the application performance expectations?
2. **Scalability.** How does the solution scale? Will application growth or, worse yet, growth in the shared infrastructure affect the solution's architecture? If so, then the corollary is that the solution architecture will ultimately limit what the application can do, and that's bad. When testing scalability, all aspects of the solution must be explored. One of the most overlooked aspects of scalability to test is understanding how many nodes a single server can manage. If a 100-node network requires five servers, there comes a point when you need a management solution for the management solution! Also, how does the UI scale? It is not uncommon to have 1,000 services, or even 50,000 dependencies. How does the UI represent these components? How are end-to-end dependencies configured and tracked? Remember: a high administrative overhead affects scalability just as the technical architecture does. As "services" grow beyond Web services and JMS to include JDBC, RMI, EJBs and so on, the number of services the server and UI must handle increases greatly, and the number of interrelationships grows exponentially! These are often

hidden bottlenecks that buyers fail to properly evaluate. A simple test is to determine if the server is involved in policy evaluation. It is highly likely that communications from agent to server and back decrease application performance to such an extent as to make the management application impossible to run in production under any circumstances. Therefore, any solution in which the server is involved in run-time policy evaluation should be automatically excluded from consideration. In fact, the fastest proof-of-concept evaluation would be to install the product in production and see what happens. If the application continues to work, the solution can be taken to the next level of evaluation. If not, nothing else should matter. It really is that simple.

3. **Survivability.** Will a management failure impact the application? What happens when the agent fails? Or the intermediary? What about the server? These seem like obvious questions, but are often overlooked.

Robust and meaningful policy capabilities. With all the focus on services in the market, policies and the importance of a robust policy infrastructure are often overlooked. There are three policy management criteria that are critical to the success of creating a distributed application management infrastructure.

1. **Flexible policies.** At a minimum, compound policies must be supported. These are policies that can combine multiple classes of metrics and multiple classes of services into a single policy. For example, a single policy can check to see if the average response time is slow or any individual message response time is slow. Furthermore, the policy should be able to specify a customer or business process to which this compound policy applies. However, true flexibility is the result of more than just the representational language used for formulating policies. Policies must be completely abstracted from the services and applications they manage. In this way, they can have their own lifecycle, be managed independently of the application, and be easily shared and maintained. As you evaluate

solutions, you should explore areas where policy is decoupled and the semantics used for connecting decoupled policies to the applications is managed.

2. **Policy lifecycle management.** Just as the components of the application have a lifecycle, so do the policies. A solution must have a way for managing policy lifecycles that reduces complexity and improves consistency of policy across applications. There must also be a way to version policies and implement new versions in a production environment, just as there are ways to version services.
3. **Policy optimization.** Writing “good” policy is about the consumer, i.e., protecting the consumer experience when back-end or downstream infrastructure components go “bad.” Many solutions will require policy to be placed on every node, regardless of its proximity to the customer and independent of the customer experience. The policy is based upon a technical metric that may (or may not) have something to do with the customer experience. Policies should be about the customer experience. As such, they should be enforced at the customer access point only. In this way, the evaluation of policies is optimized (they’re evaluated only once), and meaningful policies can be written (i.e., because they express the perspective of the customer and are not about technology abstracted from the business context).
4. **Policy on agent-less (unmanaged) nodes.** Unfortunately, not all nodes can be controlled enough to have an agent installed. However, it remains important to place policy on such nodes and track service levels on processes in which agent-less nodes participate. Agent-less nodes should be supported without breaking the scalability model, limiting management features, or introducing the server into the critical path of the application.

Once you establish success criteria for evaluating a solution, you can evaluate features in the context of specific platform goals. There are four areas—asset management, risk management, change management, and business alignment—for which you can present high-level value propositions for justifying the implementation and prioritizing the solution. Asset

management includes the core capabilities required for the others and will be explored in this guide.

ASSET MANAGEMENT

You must be able to visualize and monitor your services to gain control of your results. How can you achieve your objectives if you don't have any benchmarks? Where are you starting? Where are you going? And how can you make sure you never lose a single business transaction?

Knowledge is power. In a distributed application model, one team no longer owns the entire end-to-end application. Yet they continue to be responsible for the end-to-end experience—and for solving problems when they occur.

Time is money. Lost transactions and slow response times impact more than application performance. They result in lost business opportunities and frustrated customers. The closer the technology aligns with the business, the more important it becomes to be able to answer questions about what is happening or has happened.

In other words, you must know. And you must know quickly.

Consequently, the first critical factor in managing the assets of a distributed application is to know what those assets are and how they're being used. Just as important, you should know this without having to manually keep up with changes in the services or the environment. Once the management software is installed, there should be no configuration required to visualize and monitor the application. Automatic "learning" by the environment lowers the cost of ownership while increasing accuracy.

Understanding services is just the beginning. A solution should automatically discover (and categorize if appropriate) consumers of the service as well as service interdependencies. It should do so across multiple protocols (SOAP, HTTP(s), JDBC, ADO.NET, RMI, EJB, ESB, etc.) and present the results visually so that it is easy for operators to understand what they are seeing and to act on the information presented.

With this understanding, application owners and infrastructure administrators can use policies to collect, control, and share application status. Done correctly, distributed-application management enables the business and technology teams to be able to communicate properly.

ASSET MANAGEMENT REQUIREMENTS

Automatic Discovery

“Automatic” is the keyword for this requirement. Automation lowers the TCO and ensures users that data is accurate because its freshness does not depend upon manually updating the management system. Discovery has often been confused with searching. Many vendors promote discovery as a method for looking in a registry/ repository for items placed there by administrators/developers. However, there are three key items to be discovered without any a priori knowledge about the environment that are necessary to have fully automated discovery and full visibility into an end-to-end critical business process:

1. **Services.** All services on an instrumented platform should be discovered, as should services one-hop away on un-instrumented platforms as they are used.
2. **Consumers.** Service consumers should be discovered, even without an explicit configuration or software installation on the consumer. Consumers should be able to be published to a registry/repository for tracking; policies can be enforced on a per-customer basis, even for a shared service.
3. **Dependencies.** Automatically tracking dependencies, from consumer to producer—through however many hops, across whichever protocols—gives distributed application owners a holistic view of their application that enables rapid problem resolution. It is important that dependencies are tracked automatically and across multiple hops without affecting application performance or scalability.

BENEFITS	POC TEST SUGGESTIONS
Easy to install and fast time to benefit	Instrument the software, including any agents, intermediaries, and management servers. Send traffic. Determine if the servers, consumers, and dependencies appear on the visual display without any per-service configuration. If not, perform the work necessary to display the infrastructure and note the time and effort required.
Always up-to-date picture of shared infrastructure	Using the environment configured in the previous test, and with the management system running, add a new service or service consumer. Watch the display visually update to represent the new element as traffic is detected. If it is not automatically detected, note the steps and effort required for detection, including any mandatory platform rebooting.
Lower cost of ownership through automation	Cluster services using two (or more) application servers and a hardware or software load balancer. Tell the server about the cluster. Send traffic. Can you expose a node-detail panel to view each message and which balancing is working as expected. If not, how much time and effort does it take to get this view?
Anticipate potential conflicts that might arise as a result of a shared environment	Instrument several supported platforms and leave them running over a period of time. Examine the distributed application flow of messages across systems, if possible. Do you notice unexpected connections between systems? Unexpected consumers? If there is an historical data view to look at traffic across the whole monitoring period, use it.
One-hop-away visibility	Use an instrumented platform to access a remote service. For example, write a Web client app (that can be instrumented) that accesses Amazon, Google, or Salesforce.com. Access the Web client and watch the display. Is the remote service placed on the display even though the remote system is not instrumented? If not, perform functions necessary to display remote services. Note what is required.

Consumer Discovery

Automated discovery of consumers, including those one-hop away, is important. It allows consumers to have a view of the shared infrastructure from their perspective so they can see if they are getting the service levels they've been promised by the producers. The producers benefit as well. They can see who is using their services as well as how much they are using them.

As you will see in evaluating management products, most require coding or configuration. Look for any capabilities that automate the discovery of consumers, including one-hop-away service consumers. Also, see if there are ways to consolidate various consumer groups to provide an easy-to-understand view of the consumer relationships you build with your distributed applications. Grouping in the UI should not preclude the ability to drill down into customer-specific information, which is crucial to troubleshooting problems.

BENEFITS	POC TEST SUGGESTIONS
Know which consumers are using what applications	Use multiple consumers to access a service. Is there automatic consumer discovery even though software is not installed on the consumer? Can you discover new customers easily and automatically, including unauthorized consumers, due to the visual nature of the display? If not, test the facilities for finding new consumers. How much work is involved in doing this?
Create consumer-specific policies to monitor and control your application	Set up a policy with performance requirements specific to an individual consumer. Determine if a service shared by multiple consumers reports statistics on a per-consumer basis, enabling alerts to be raised for individual customer groups. Or is the solution limited to displaying "service-wide" statistics? For example, can the solution report that for EMEA customers, service A's response time is X, or can it only report that service A's response time is Y.
Track service consumers in standard registry/repository product	Set up and configure a registry/repository product. If possible, configure the server to publish discovered consumer dependencies into the registry. Use the registry UI to browse services and examine service-consumer dependencies.

Dependency Tracking

A distributed application adds the requirement of visibility between nodes that make up the application. As a result, it is not enough to know how each machine in the flow is behaving; you need to know how they are functioning together to deliver the application to the consumer.

BENEFITS	POC TEST SUGGESTIONS
Holistic view of the overall service infrastructure and how it's delivering from the perspective of the distributed application	Set up a test environment that has at least two instrumented service platforms. In this way, there is a chain from consumer, to service, to another service. Send test messages. Determine if you can drill down to a more granular level to see how messages across multiple hops are tracked end-to-end. Note how much time and work are required to do this, if it is possible.
Eliminate the need to trace log files to track problems across the network	Create a policy to audit messages when responses are slow. Trigger a slow message so that an alert is raised. Go to the alerts functions, and drill into the alert. Is it possible from a single interface point to explore all the audit messages across the whole flow of the alert? Is there a view that filters the alerts to the particular message that triggered the policy failure across all nodes that services a request in the flow?

Always On

Typically, with most management tools distributed application problems are resolved by analyzing log files across the distributed systems. Unfortunately, deep logging often reduces system performance and, therefore, is only turned on when administrators are looking for something.

This creates a situation where application troubleshooting becomes an extended procedure, summarized in the following steps:

1. Alert on problem; take standard steps to resolve it; realize more information is needed.
2. Turn on additional logging across all systems, thereby impacting application performance.
3. Wait for problem to recur! Note: the new environment, with logging turned on, has different performance characteristics from the original, and the problem may not recur or may not recur quickly.
4. When the problem recurs, look across all system log files and manually correlate events to look for the root cause.

This procedure presents the following challenges:

1. The problem must recur a minimum of twice.
2. The original problem message—and the context around it—are never captured.
3. During the investigation process, the performance of production systems is decreased by additional logging levels.

With a distributed application, in which many applications share common services, these limitations are unacceptable. A management system must be on all the time without affecting performance. It must catch the problem the first time it happens and deliver the context around the problem so it can be resolved quickly.

An interesting way to experience this capability is to look at a clustered service. Often, problems in a cluster are difficult to identify because it is hard to tell if the problem is cluster-wide or just localized to a specific machine in the cluster.

BENEFITS	POC TEST SUGGESTIONS
Tracks all messages	Install in the pre-production or QA environment and run performance comparison against an un-instrumented application. Apply policies, including auditing policies, and track message latency and CPU overhead. The impact on performance of this exercise will show you if you can allow the solution to run in production all the time. It will also show the additional costs required to run in production because a management system with high CPU utilization will require more powerful servers for the applications and services being managed.
Tracks messages with minimal impact on application performance or scalability	See “Performance and Scalability” (section 7) below under “POC Test Suggestions” for more detailed performance and scalability tests than were shown in the prior test.
Identifies the root cause of a problem the first time the problem is identified, eliminating both the need to turn on debugging to catch the second problem and the performance hit taken as a result of more logging activity	Create a policy on message response time and apply it near the customer interaction. Run a process that triggers a violation. Now look at the network view of the process and drill down. Is the information about what happened in the message in this first incidence captured? If so, you don’t have to wait for the problem to occur again in order to begin to analyze the root cause. Because information on a policy violation is captured the first time, you don’t need to turn on debugging to have the information you need to start to troubleshoot a transient problem. In fact, turning on debugging changes performance conditions by adding an additional processing load, obscuring the precise problem even were it to recur under the new performance conditions.

Enables Communications across Teams

One of the challenges of shared services and distributed applications is meeting the requirement for better coordination among teams. When one team owns an application from end-to-end, there is a confidence that problems can be solved by that team. In a shared environment the teaming dynamic changes, and it’s easy to point fingers instead of accepting shared responsibility. Too much finger-pointing or failure to meet service-level expectations (or the perception thereof) will create a hesitation among application owners to collaborate and use a distributed application infrastructure.

In addition to the requirements above of having detailed “what’s happening” information in the shared infrastructure, the UI must enable sharing and filtering of relevant information so that application owners know how performance of the shared infrastructure impacts them.

BENEFITS	POC TEST SUGGESTIONS
Transparency prevents finger-pointing and will improve service	Create a per-consumer-based dimension (for example, by geography, customer, customer class, or business process) and share a performance information portlet on this per-consumer information with the application team. You are showing the end user that even though it is a common infrastructure, their mission-critical business concerns can be tracked independently of how others are using the infrastructure.
Supports multiple business dimensions and views without affecting performance and scalability	Building on the test above, create a second dimension to track service calls by a different business aspect. Run test messages and view distributed application behavior by each dimension. Notice if the software can handle both dimensions at once without affecting performance. If so, that means you can collect information on one dimension (e.g., a customer segment) today and create other or sub-dimensions in the future without having to re-do all of your dimensions. Run tests with many dimensions to see the scalability and flexibility of the solution.
Leveraging shared and existing infrastructure (email, SMS, other management platforms) will reduce the learning curve, overall implementation costs, and total cost of ownership	Create a policy that alerts via SNMP and share the alert with the existing management system to understand how you can integrate alerts with the systems operators already understand. Alternatively, or in addition, create a policy that alerts via email to see the options for distributing alerts outside the management systems.
Customization is available through documented alert notification API	Create a custom alert notifier, for example, one that sends an instant message or an SMS.
Enables it and business groups to communicate around functional business areas instead of artificial technical boundaries—enabling them to align the technical infrastructure with the way the company does business	Create a business process view, and share with the team responsible for the business process.
Ties application-level logging into message flow context within the distributed transaction	Create a policy that raises an alarm and set it to capture Log4J application-level logs when the alarm condition is raised. Trigger the alarm. Explore the alarm flow map, and see how you can explore the root cause of the problem, as well as drill into both message level audits and application level log messages.

Policy Management

Policies are the tool through which users manage their environment. It is critical that they be fully decoupled from the services themselves so they may be managed more easily over time. Decoupling policy ensures that

you can separate policy ownership from service ownership. In addition, since policies don't version on the same timeframe as services, decoupling enables policies to have their own lifecycle. Policies should also be evaluated at the point of management, not at some central server, for optimal performance and scalability. Finally, the policy definition language should be sufficiently robust to enable compound policies—i.e., policies that support both the business and underlying technical infrastructure—and automatic policy inheritance.

BENEFITS	POC TEST SUGGESTIONS
Support for proactive and reactive policies	Create a policy warning level below the level at which customers would experience performance problems. Trigger the warning, and explore how this can be used to set “service stabilizers” or any other mechanism (if available) to control message flow and avoid problems in the event of a service slowdown.
Policy inheritance	Create a policy and apply it to a node, dimension, or business process. Modify the node/ dimension/process, and see if the newly installed components inherit the policy without any policy changes or manual mapping between policy and service. This demonstrates policy inheritance and shows how policy can be implemented consistently and automatically.
Policy optimization	Create a policy to manage service performance. Apply the policy to ONE place in the flow (close to the customer). Trigger the policy. Is the complete message flow captured for analysis? If so, this demonstrates policy optimization and shows that though the policy is implemented in only one spot, it protects the entire flow.
Policy lifecycle management	Choose an existing policy and create a policy revision. How does this affect message flow? Does a revision enable multiple changes to be implemented and provisioned without impacting (or stopping) existing message flow? If so, should a policy revision have errors, it also becomes simple to fall back to the prior version. In addition, explore how policy changes are provisioned and logged for accountability.
Support for complex policy expressions	Create a compound policy that involves measuring performance to get both an individual and an aggregate statistic simultaneously. For example, when the average response time is greater than 6s, but any individual message is greater than 9s, raise an alert. Understand that this sort of policy is common and should be executed at the managed node, not at the server, to assess performance and scalability.
Policy enforcement on unmanaged nodes	Create an unmanaged node policy, and deploy it into the network. Notice that you have no requirement to install software on the unmanaged node, nor do you require that the server insert itself into the message flow in order to evaluate policy. Send messages through the network, trigger policies, and explore how unmanaged nodes can raise alerts.

Performance and Scalability

Determining the performance and scalability of a management product is not a simple matter of how many messages or services you have.

Rather, assessing this issue manifests itself in four areas that are important over the lifecycle of the mission-critical infrastructure supporting distributed application deployments:

1. **CPU impact.** Installing management software will add CPU overhead to a node. You need to know what impact to expect to properly size application platforms. When more CPU cycles are used by the management product, fewer are available to the application. In other words, management will impact the number of CPU licenses necessary for the application and increase application software licensing costs (for applications, like app servers, licensed by CPU). A light impact is very important to a low total cost of ownership (TCO).
2. **Message latency.** Adding management software can also affect message processing latency. It is important to identify and rule out products that can achieve either high performance or full functionality, but not both simultaneously.
3. **UI scalability.** This is harder to gauge. Some products can handle 30 or 40 nodes, but degrade when there are 1000s or even 100s of nodes (or even services). While a simple project may not put stress on the software, you need to know if a couple of projects (especially, multiple projects using the same service) will affect message throughput, cause a slowdown, or even bringing down a shared service. Also, you need to see if a product can scale—but must turn off functionality to do so. A product should be able to maintain its full functionality under increasing loads without a change in the deployment architecture.
4. **Feature growth and expansion of product use.** This is a matter of how simple or complex it is to upgrade the product (for example, to move from operational management to business process tracking). Specifically, do you need to re-instrument your application? Do you need a different type of “agent,” one that has more functionality? Do the changes require a new software installation on the instrumented node? In some organizations, if you change a node, you have to retest every application on the node; a simple upgrade then demands weeks

of integration testing. Other questions concern the architecture of the installation: does a feature upgrade require a change in the management system's deployment architecture?

Not surprisingly, then, this issue is ultimately about the architecture of a product. Positive results to the tests listed below—and the ease with which you can achieve higher performance and scalability increases in these areas—will help you assess whether a product is designed from the ground up to be scalable and high-performing. If it isn't, the product may have various work-arounds to increase performance and scalability. But it will require time and work to do so, or it will reach a point where performance or feature growth requires a forklift upgrade.

Of course, performance and scalability are essential for supporting business-critical applications and the core processes they execute. They are equally important for a low total cost of ownership (TCO) over the lifecycle of the product and the network of distributed, business-critical applications.

BENEFITS	POC TEST SUGGESTIONS
CPU impact on the managed node is low	Run your traditional test for CPU impact on a business-critical application before the software is installed and measure the results. Then install the software on a server and application nodes and run the test using the same application and measure the results. Compare the "before" and "after" results.
Message latency is minimal	Run your traditional test for performance/message throughput on a business-critical application before the software is installed and measure the results. Then install the software on a server and application nodes and run the test using the same business-critical application and measure the results. Compare the "before" and "after" results. When running these tests, it's always better to run them from multiple consumers simultaneously, to simulate a shared environment. It's also important to run them with some "real" policies—like policies that segment by business dimension—so that actual performance under policy-load can be evaluated. It is also possible to evaluate the architecture of the product to understand message latency expectations. Look for double processing on the instrumented node or for agent communication to the server. When an agent communicates to the server for policy evaluation, it should immediately disqualify the product from consideration due to the high overhead such communication adds to business-critical applications.
UI scalability is high	This can be difficult to test in a small test environment or in the initial deployment for a small project. Check references, and ask for large-scale deployments in production. Make sure to determine if references are in production, or just center-of-excellence configurations. Also, often it is possible to instrument a large test environment or use a vendor's API's to simulate a large-scale service network implementation.

BENEFITS	POC TEST SUGGESTIONS
<p>Feature growth and expansion of product use cases don't require a "forklift upgrade"</p>	<p>The test to run here is really vendor-dependent. Some vendors will have one product with all functionality. In this case, cost and performance are an issue. If you get all functionality, do you have to pay for it all even if you are not using it? Does implementing a "full" product affect performance?</p> <p>Other vendors will have different products to address various use cases in terms of deployment architecture or feature use cases. For these vendors it is important to examine what is involved in an upgrade between product versions or to add capabilities from one product to the next. For these vendors, it is important to look at each component in the solution and determine which features it provides and how to migrate from one to the next. Determine how adding components impacts the deployment architecture and scalability. Determine if new server components are needed, how they are configured as part of the overall design and if policies from one server component to the next integrate well. Notice that if multiple servers are needed to extend functionality, and then multiples of each server are needed for scalability, each server added is a multiplier (rather than an aggregate) in terms of costs for hardware, complexity, and total cost of ownership of the solution.</p> <p>Two examples follow:</p> <ol style="list-style-type: none"> 1. Some vendors have multiple types of "agent." Determine which features (and scalability/ performance characteristics) belong to each agent. Then, examine the process for upgrading from one to the next. Does it involve installing new software on the instrumented node? Does it involve reconfiguring the services being managed? Is it necessary to reboot the instrumented system? 2. Some vendors also have platform functionality broken into components. Often these components separate operational functionality from security and from business functionality. Set up the basic configuration, and work through the steps to upgrade additional features to determine the effort and intrusiveness of the upgrade.

Support for Multiple Protocols and Applications

This functionality is essential for tracking, managing, and troubleshooting business-critical, distributed applications across the entire message flow. Many enterprise applications are based upon a variety of technologies, encompassing Web services, Java, or .NET components, databases, service buses, and orchestration engines (and more). If a distributed application crosses multiple protocols, shouldn't the system being used to manage it do so, too?

As a result, you need to look for a product with end-to-end visibility across many hops for mission-critical applications. A complete picture of the end-to-end message flow is important in order to achieve comprehensive policy enforcement and rapid problem resolution.

To provide the flexibility to manage a diverse environment from protocol to protocol and platform to platform, a product must have an architecture that allows you to integrate new protocols and platforms without requiring a fundamental design change. And, importantly, performance and scalability characteristics should not change from protocol to protocol.

BENEFITS	POC TEST SUGGESTIONS
End-to-end visibility	<p>Testing end-to-end visibility requires a multi-protocol application. If you don't have one, you can create a simple one by writing and deploying a Web service that accesses a database via ADO.NET or JDBC. You can then create a Web page that calls the Web service. Install each component on its own host, and make sure that the Web page is called from a separate consumer machine for the maximum effectiveness of this test.</p> <p>Instrument the Web server and Web service platforms. Access the Web page, and perform a transaction.</p> <p>Are you able to visualize the HTTP call, to the SOAP Web service call, to the database?</p> <p>Do you see each of the four nodes (consumer, Web server, Web service, and database)? Are you able to track each message from end-to-end?</p> <p>Of course, it's best to test with the protocols that are used in your environment.</p>

Support for Business Process Tracking

In a SOA or network of distributed services and applications, individual business processes run on a shared infrastructure. To help business and IT work more closely together, it is important to (1) be able to communicate about the business, rather than about the technology infrastructure; and (2) be able to track the infrastructure by the business it delivers. Often, companies expect that business process tracking is the end-result of implementing an intrusive business-process orchestration tool into production. Orchestration engines are powerful, but limited in that they only visualize processes that have been pre-configured, and they often don't cross technology boundaries.

Tracking a business process in a distributed, mission-critical infrastructure is different from orchestration of the process. Tracking the process enables visualization even when no orchestration is present. This visualization is, therefore, non-intrusive and can be implemented through discovering existing flows and, in a simple step, naming the processes to track. In fact, often organizations will first

track their existing processes and use the information collected to re-orchestrate their business processes using an orchestration engine.

Additionally, tracking (versus orchestration) delivers a high-level view of the infrastructure, so that both technologists and business analysts can communicate more efficiently. Using the high-level view, it becomes easy either to look at a business process and drill into the supporting technology or to look at the technology and visualize the impact to the business. Better communication and the ability to create policies around business events and priorities improve technology efficiency and customer services.

BENEFITS	POC TEST SUGGESTIONS
<p>Automatic business-specific process discovery, tracking, and reporting without any special instrumentation (e.g., orchestration, or business process management [BPM] implementation of the business process)</p>	<p>Using an environment with shared and distributed services, create a business process: perhaps, something like “order entry” or “fulfillment.”</p> <p>Run test traffic through the system to see the business process visualization and automatic discovery of the steps in the process, even though no orchestration was performed.</p> <p>Using the product interface, you should be able to examine the business process metrics as well as the components of the process and how those components serve the process. For example, let’s say that “order entry” has a “check stock” step in the process. You can imagine that other processes or applications might use the “check stock” service. You should be able to write policies and see metrics on how “check stock” is servicing the “order entry” process specifically. Be aware that many management solutions not designed for a shared infrastructure will only provide metrics on the “check stock” service, but won’t be able to differentiate how it’s servicing various consumers. (In our case, one of the consumers is the “order entry” process.)</p> <p>You should also be able to share performance information with business analysts and incorporate information into existing business dashboards via URL.</p>
<p>Enable different business groups to use a shared infrastructure but monitor, control, and police their own specific business process.</p>	<p>This builds on the test above. Have a business analyst create several processes. These processes should share some components.</p> <p>Create policies for the business process. Notice how business analysts for the process could create their own policies, and even though the policies run on a shared infrastructure, they can control the experience users have with their process independently from each other.</p> <p>In fact, even within a process, you should be able to create different dimensions of service, so that different users of the same process can be managed with different service-level agreements (SLAs).</p> <p>Note: here it becomes very important, as mentioned above, to have scalability and high performance. As you start to monitor many business processes, with many business aspects (region, customer, product, etc.), you must have the flexibility to abstract the business aspects clearly, and not be limited to either a small number of aspects or aspects that are pre-configured by the vendor.</p>

As you will notice as you evaluate various vendors, Progress® Actional® Web service and SOA management software fulfills the key RFP requirements that enable you to ensure the reliability of service-based, distributed applications and adherence to SLAs. As a leader in innovation and technology, Actional provides robust, mission-critical infrastructure for delivering applications that can be governed by policy.

With Actional you can:

- > Manage all applications with end-to-end visibility across the services network including systems that are one-hop away. Actional's patented automated discovery and mapping requires no coding and little configuration. Additionally, Actional can apply policies to a business process (e.g., "audit all steps of a financial process") or type of data (e.g., private customer information) without first knowing the specific services involved.
- > Rapidly address services delivery issues (before they affect customers) with automated root cause analysis—even if the issue is sporadic or only affects one customer, partner, or channel.
- > Implement service-level policies that align with your business requirements—for example, providing the best service to premium customers to maximize revenue or customer satisfaction

Finally, Actional can do all of this with negligible performance impact and is, therefore, able to provide full functionality in a production setting without affecting the application. Actional monitors and controls end-to-end processes with only microseconds of latency on the network and without server bottlenecks. It requires less than two percent of CPU capacity per network node, and one Actional server can support more than 1,000 managed systems and over 50,000 service dependencies.

For more information on Actional and other Progress products, visit www.actional.com/contactus/index.html.



PROGRESS SOFTWARE

Progress Software Corporation (NASDAQ: PRGS) is a global software company that enables enterprises to be operationally responsive to changing conditions and customer interactions as they occur. Our goal is to enable our customers to capitalize on new opportunities, drive greater efficiencies, and reduce risk. Progress offers a comprehensive portfolio of best-in-class infrastructure software spanning event-driven visibility and real-time response, open integration, data access and integration, and application development and management—all supporting on-premises and SaaS/cloud deployments. Progress maximizes the benefits of operational responsiveness while minimizing IT complexity and total cost of ownership.

WORLDWIDE HEADQUARTERS

Progress Software Corporation, 14 Oak Park, Bedford, MA 01730 USA
Tel: +1 781 280-4000 Fax: +1 781 280-4095 On the Web at: www.progress.com

For regional international office locations and contact information, please refer to the Web page below:

www.progress.com/worldwide

Progress, Actional and Business Making Progress are trademarks or registered trademarks of Progress Software Corporation or one of its affiliates or subsidiaries in the U.S. and other countries. Any other trademarks contained herein are the property of their respective owners. Specifications subject to change without notice.

© 2009-2010 Progress Software Corporation and/or its subsidiaries or affiliates. All rights reserved.

Rev. 9/10 | 6525-127999