Choosing between traditional WebSphere and Liberty

Alex Mulholland
IBM Senior Technical Staff Member

Originally published June 2015
Updated November 2016 for Liberty 16.0.0.4

Table of Contents

1 Introduction

There are a number of aspects to consider when choosing between the traditional WebSphere Application Server (aka WAS full profile, or WAS Classic) and WebSphere Liberty for deployment of applications. If you have a hard requirement which is only available on one or the other (such as a particular API) then the choice is easy, but as the Liberty function has grown, it has become more common to need to weigh the pros and cons of each more closely; to consider topology choices, operational capabilities and security options. Over one hundred and fifty IBM products have shipped with Liberty as their internal application server and an ever-growing number of IBM SaaS offerings, such as BPM Workflow and Watson Analytics, are running on Liberty. As you compare traditional WAS and Liberty for deployment of your own applications, I hope this paper will help you to choose with confidence.

WAS has provided a robust, high performing Java EE application server for many years. In 2012, the Liberty profile was added to the product, providing a lightweight option that is very fast to set up and more dynamic to use, initially with a subset of the WebSphere programming model and administration capabilities. Both are profiles of the same application server, in the sense that they run a (mostly) common set of components on different kernels. So which one is right for your application?

Traditional WAS remains fully supported and strategic, and if it meets your needs there is no reason to move applications out of their existing environment; however, if you are starting a new project, or you need a lighter-weight or more flexible server, if you want to exploit some of the new cloud environments like IBM's Bluemix PaaS, or perhaps if you are stretching the practical limits of the WAS – Network Deployment (WAS-ND) cell size and need even higher scale management, then Liberty may be a good choice. Much has been written on the benefits of Liberty for application development; this article will not revisit those (which are many, and obvious if you have ever watched a demo or tried it out) but will provide a comparison of the aspects that are important for application deployment in production systems. This comparison is based on the features of the WAS traditional 9.0.0.2 and Liberty 16.0.0.4 product versions.

2 General Architecture

The most obvious difference between traditional WAS and Liberty is the runtime architecture. Traditional WAS has a fairly fixed set of application services that load and initialize on startup, including the full Java EE platform, with a few extensions that are more configurable (and thus result in some flexibility in memory footprint). This provides a runtime with the full programming model available by default, with all services, applications and resources fully initialized when server startup completes.

Liberty has a small kernel, which is all that will load and initialize by default at startup; the user configures, at a fine-grained level, exactly which services (*features*) are needed, with the intent that each server instance closely matches the needs of the application(s) it runs, providing 'just enough application server' in each case. This model is better suited to resource-sharing environments like cloud. Initialization of services, applications and resources is generally delayed until they are used, in a 'late-and-lazy' model that provides a faster server start and further reduces the memory footprint.

These basic differences in runtime architecture are often the most important factor in choosing between traditional WAS and Liberty, especially as more applications move into cloud environments and resource sharing becomes increasingly important.

3 Programming Model (APIs)

A critical aspect to consider in placing your application is which programming models (APIs) it uses, and where they are available. If your application uses an API that is not available as a Liberty feature, then you may have an easy decision to use traditional WAS. Before making that decision though, do consider whether it would be a good time to modify your application to use more modern, strategic programming models; if the answer to that question is no, however, then you have your decision: deploy on traditional WAS. For new applications, however, Liberty has the full set of strategic Java APIs, and updates those APIs faster than traditional WAS, so is generally a better choice for new applications.

Both runtimes have the Java EE7 Full Platform set of APIs. With Liberty you can choose which of those APIs to include in each server instance (to match the needs of your application); in traditional WAS they are all present in all instances. Liberty also has some of the proprietary WebSphere APIs, where they are still useful, but those older WebSphere APIs that have been rendered redundant over the years by enhancements to the public Java specifications have not been replicated on Liberty. The remaining programming model gap, therefore, is mostly comprised of APIs that are rarely used in new applications, although there are still a few strategic gaps in OSGi Application support. APIs which have been deprecated by the Java EE specification or traditional WAS are unlikely to appear on Liberty unless there is high customer demand. Figure 1 provides a high-level view of the remaining API gap between traditional WAS and Liberty.

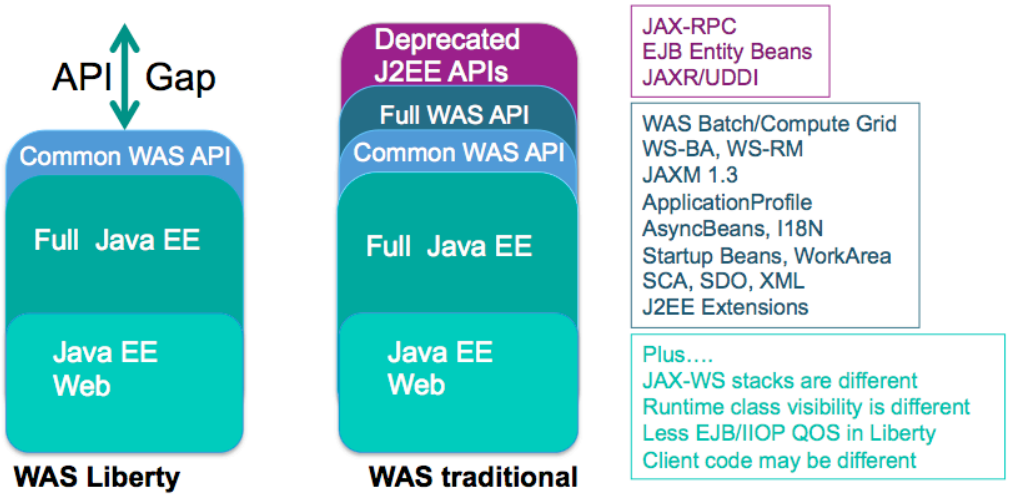


Figure 1. API differences between traditional WAS and Liberty.

3.1 WAS APIs that are not in Liberty

- OSGi Application support
  - Blueprint Security
  - Blueprint Resource References
  - EJB bundles
  - Remote Services
- APIs deprecated by the Java EE specification
  - JAX-RPC
  - JAXR / UDDI

    - EJB Entity Beans
- WebSphere extension APIs superseded by Java EE capabilities
    - WAS Batch (Compute Grid API)
    - Async Beans, Scheduler
    - Startup Beans
    - Remote Request Dispatch
- APIs used mostly by extending products
    - WS-BA, WS-RM, WS-RF, WSIF, JAXM
    - Application Profile
    - WorkArea, I18N, Activity Sessions
    - SDO, SCA, XPath, XQuery, XSLT
- Other WAS APIs
    - Object Pools
    - Cross component Trace
    - Common Business Events
    - Dynamic Query

Applications can be scanned for their use of these APIs with the WebSphere Application Migration Toolkit, which is available both as an eclipse plugin and as a command-line tool. The eclipse plugin scans application source files in eclipse projects, while the command-line tool operates on application binaries (ears, wars etc.) individually or in directories, and both will identify the use of any APIs that are not available in Liberty. They provide detailed information in each case of the options available; for example changing the code to a newer technology or packaging a copy of the older library in your application.

If your application uses an API that is not available in Liberty, there are several options:
1. deploy the application to traditional WAS,
2. if the API implementation is publicly available (e.g. JAX-RPC), package it in your application, or as a shared library, and deploy to Liberty,
3. if the API is non-strategic (for example if it has been deprecated by the Java EE specification), consider converting your application to use a strategic API that is available on Liberty (eg update JAX-RPC web services to use JAX-WS).

You may also want to keep an eye on new features being delivered into the Liberty Repository and into the beta drivers as the API gap between WAS and Liberty has been rapidly shrinking.

3.2 Earlier availability of new APIs with Liberty

While some older APIs are only available in WAS traditional, new APIs are generally being delivered faster on Liberty, so if you are keen to get hold of those APIs sooner, Liberty may be the better choice. The delivery of the Java EE 7 features is a good example of this pattern. The pluggable feature architecture of Liberty allowed the first set of Java EE 7 features, including servlet-3.1 and websocket-1.0, to be delivered in version 8.5.5.4, before the rest of the Java EE 7 stack was ready. This was also a very tangible demonstration of the continuous delivery model employed with Liberty, which provides new capabilities in net new features, which can be optionally installed and configured. The second set of Java EE features came in 8.5.5.5 and then full certification was completed in 8.5.5.6: the first generally available, fully supported Java EE 7 product in the market, and a full year before the WAS traditional Java EE 7 stack was available in WAS 9.0. This demonstrated not only how much faster IBM is able to develop and deliver features for Liberty, but reflects the speed and agility of Liberty as a general development platform for applications as well. It was also the first very visible demonstration of the Liberty zero-migration policy: in this case the fact that the new Java EE 7 features were available on the existing Liberty kernel and that they could be added to an existing Liberty installation without forcing an upgrade of either configuration or applications.

3.3 API flexibility: mix-and-match across Java EE versions

Delivery of the Java EE 7 features in three stages for Liberty also demonstrated the flexibility of the feature architecture to allow a 'mix and match' approach to the existing Java EE 6 features and the new Java EE 7 features. This approach allowed use of the early EE 7 features, like websocket-1.0, in the same application as existing Java EE 6 features like cdi-1.0, so that a complete stack was available to the developer even when using the early EE 7 features. This flexibility remains useful even with the full set of Java EE 7 features available: the new technologies can be used where they are needed, but they can still be combined with the EE 6 features where upgrading is an unwanted cost. The supported combinations of Java EE 6 and Java EE 7 Liberty features are documented here.

In WAS 9.0 traditional, there is more limited support for mixing versions of Java EE technologies. Where particular upgrades could require a lot of rework to an application, it is possible to configure the WAS 9.0 server to load the older implementation. This has been provided for JPA and JAX-RS; in both cases the underlying implementations were changed in WAS 9.0, and since both of those specifications tend to 'leak' implementation details up to the application, it was valuable to offer the older components to minimise the migration effort to the new product version.

4 Administration and Topology Choices

While there are a few differences in the APIs provided by WAS traditional and Liberty, they essentially support the same applications. The administration of Liberty, however, is intentionally different and (in the view of its creators) much improved. The Liberty design allows for easy integration into lightweight, flexible systems of continuous delivery and operations. Its small size, flexible packaging and radically simplified configuration provide many more choices of operational models and integration with provisioning and management tools.

4.1 Configuration Files

Traditional WAS has a large number of configuration files spread over several directories for each server instance. The files are not designed to be edited directly, but to be created and manipulated through administrative tools like *wsadmin* and the WAS Admin Console in order to maintain configuration consistency. As a result the product tools need to support fine-grained operations on the configuration, and to ensure that operations that cause changes across multiple files are properly coordinated.

The configuration for Liberty is very different: the user can design their configuration structure, from the single mandatory server.xml file to any number of logically nested, included XML files, and a small number of optional properties files. Updates do not need to be coordinated across multiple files. The XML content of the Liberty configuration files has been carefully designed to be human-readable and human-editable, using plain text editors if desired. The Liberty kernel and features each provide a full set of useful default values for their configuration attributes, so the user configuration remains minimal, or *sparse*, consisting of the list of required features, overrides to any feature attributes, and instances of resources such as applications or datasources. Support for variables, includes and overrides (*dropins*) makes the Liberty configuration very flexible; it is easy to version in a source control system and to have a high degree of sharing of configuration files across server instances and host machines.

The simple and flexible nature of the Liberty configuration reduces the need for tools to perform fine-grained updates. While traditional WAS configuration is focused on resources, Liberty configuration is really a file-based model. For production environments, the addition or replacement of whole files is recommended. Use of the *include* mechanism, or the monitored configuration *dropin* locations, make it practical to contain changes to single files. There is a file transfer mbean provided to easily move such files around the system, and the server will process updates dynamically by default, or can be configured to wait for an mbean prompt or a server restart before acting on configuration changes. The developer tools for Liberty (WDT and RAD) provide a visual configuration editor in the eclipse IDE; the Liberty Admin Center (web administration tool) has a similar configuration editor currently in beta.

4.2 Deployment models

The deployment of an application on traditional WAS typically involves:
- Product installation using IBM Installation Manager (IM)
- Server profile creation using the Profile Management Tool (PMT) or *manageprofiles(bat/sh)* command line script
- Configuration of the server using *wsadmin* commands
- Application installation using *wsadmin* commands (or the Admin Console)

All of those operations can be automated through Jacl or Jython scripting to make them efficient and repeatable. Subsequent updates are generally applied as deltas to the existing system, again using IM to update the product, and *wsadmin* scripts or Admin Console to update the configuration or the application.

The flexible nature of Liberty, and its file-centric configuration, have led to a different predominant deployment pattern, that of 'rip-and-replace'; the complete, configured stack is generated as a single deployment artefact (ideally as part of a DevOps flow), and is completely replaced with each update. There are several ways the deployment package can be generated: directly from the application development environment, from a build process, or from a combination of steps.

Liberty has a utility to package a server; this command operates on a configured server, with application(s) installed, and produces a zip file containing the application(s), user configuration and resources, and, optionally, the runtime (product binaries) required by that server configuration. This customized, configured package can then be very quickly deployed onto a host machine through file transfer and unzip, guaranteeing clean, identical clones on each host. This pattern works well for an application developer to generate a self-contained deployment package, which is increasingly common with the growth of micro services. The package can be generated to contain only the Liberty binaries that are needed by the packaged configuration, making it very small and fast to transfer over a network.

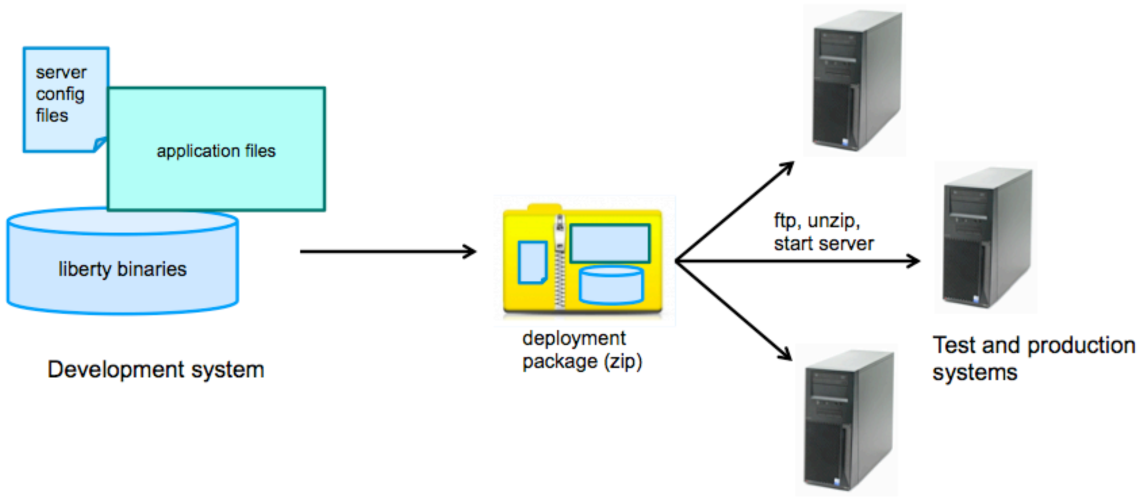


Figure 2. Deployment of server package created on development system.

Another common approach is to keep the Liberty configuration files with the application source files in a source code management system, and to deploy all of them through an automated dev/ops pipeline. The Liberty binaries can be held in the same system, but it is more common to provide them from a different location, or to pre-install them onto the host systems (this is an audit requirement in some organizations). The configuration files and application source files are extracted from the source control system by the build process, and can be published to a location that is monitored by a deployment automation tool, for distribution to the host machines, as shown in Figure 3. There are major advantages to this approach: all the deployment assets are carefully controlled, with a complete audit trail, and it is easy to identify and backout changes that cause problems.

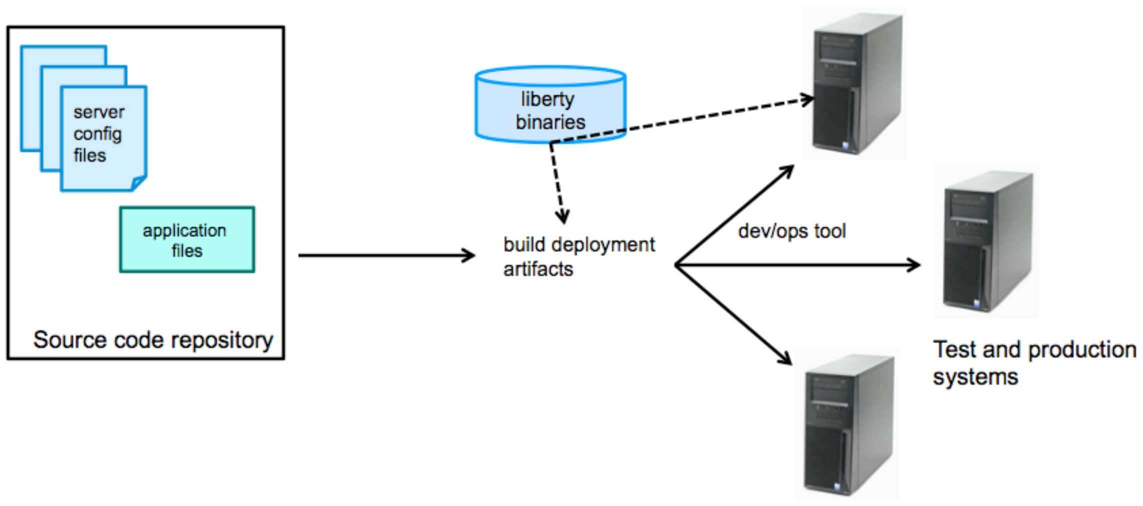


Figure 4. Deployment of applications and configuration through an automated pipeline.

Where multiple teams need to contribute to the configuration, the design of the Liberty configuration really shows its value. Using the 'config as code' approach, each team contributes their part of the configuration as XML files into source control. Critically, this approach allows each team to deliver their configuration independently, and makes it simple for multiple applications to share common pieces of configuration. In this way, an infrastructure team can contribute their pieces of configuration into the source control system on their own schedule, and not delay the development team in making updates to their applications.

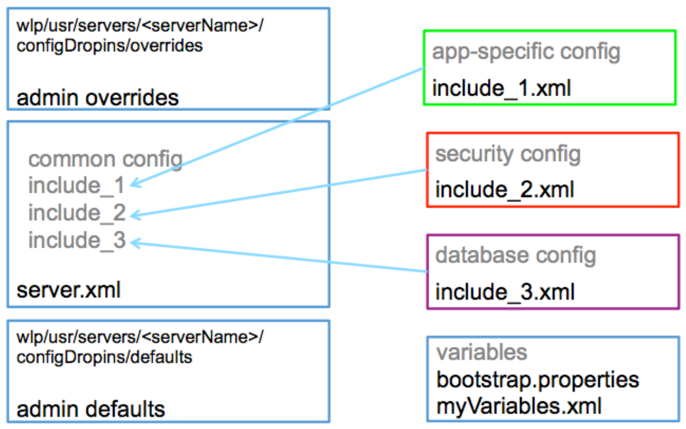


Figure 3. Independent contribution of configuration from multiple teams

Liberty deployment is very well suited to DevOps flows, where builds can be triggered when a change is commited. The build output may be a new server package or multiple changed files, which can be monitored by tools like UDeploy or Chef, which then distribute the updated files. Overrides can be applied to individual hosts: configuration variables can be set for values like port numbers, and overrides to the packaged configuration can be enforced through use of the 'config overrides' directory locations.

The difference between these deployment patterns is often an important factor in choosing between traditional WAS and Liberty. If you have a large investment in existing WAS automation, and that system meets your current needs, you may decide to use Liberty solely for efficient development and then to subsequently deploy those new applications into your existing WAS topology. Conversely, if you want to modernize your application infrastructure and change to a more agile DevOps flow, Liberty will provide much greater flexibility.

4.3 Centralized Management

Traditional WAS uses the *cell* as the management scope. A cell defines all the processes controlled by a single management process. In WAS (base), cells are individual servers each with their own integral management process; in WAS-ND a cell is controlled by a Deployment Manager (DMgr), and each managed node runs an agent process. The DMgr is a singleton process, which, in conjunction with its ‘shared everything’ architecture and tight coupling between processes, limits its scalability for very high-volume administrative operations and is potentially a single point of failure for management operations. The practical limits of cells size vary with hardware, and with the load of administrative operations in each individual system, but typically are reached when there are a few hundred application servers in a Core Group, which is a High Availability Domain associated with the cell. The DMgr owns the configuration of all the managed nodes and synchronizes the configuration across all the nodes. This provides a tightly managed system, which will, for example, ensure that all members of a cluster have matching configuration. It is also supported by a mature set of tools that allow fine-grained operations through the Admin Console or *wsadmin* (which can be scripted using Jacl or Jython).

The Liberty profile has a new management model, the Collective, with a Collective Controller as the central management point. A collective may have a single controller, or can have a clustered ‘replica set’ of controllers which, in conjunction with the ‘shared nothing’ architecture of a Liberty collective, allows much higher scale of management operations. A set of controllers will share the load of administrative operations, and also provide high availability of management as they can failover for each other. Again, the limits of collective size will vary with hardware and operations, but collectives as large as ten thousand application servers, using a replica set of five controllers, have been successfully tested. There are no local agent processes in a collective, so resource use is lower. Collective members retain ownership of their own configuration; there is no synchronization of configuration, which also increases the scalability of the system. Facilities are provided to help the user to keep configuration consistent across members and clusters, but this is not enforced by the collective logic so responsibility lies with the user for keeping the configuration across cluster members sufficiently matching. This is a fairly common pattern with Liberty; you have greater choice and flexibility but also greater responsibility for getting things right; this is a compelling reason to automate operations (for anyone who hasn't yet been convinced of the value of doing so). The collective controller acts as a proxy for JMX operations on the member servers, including transfer of configuration (and other) files.

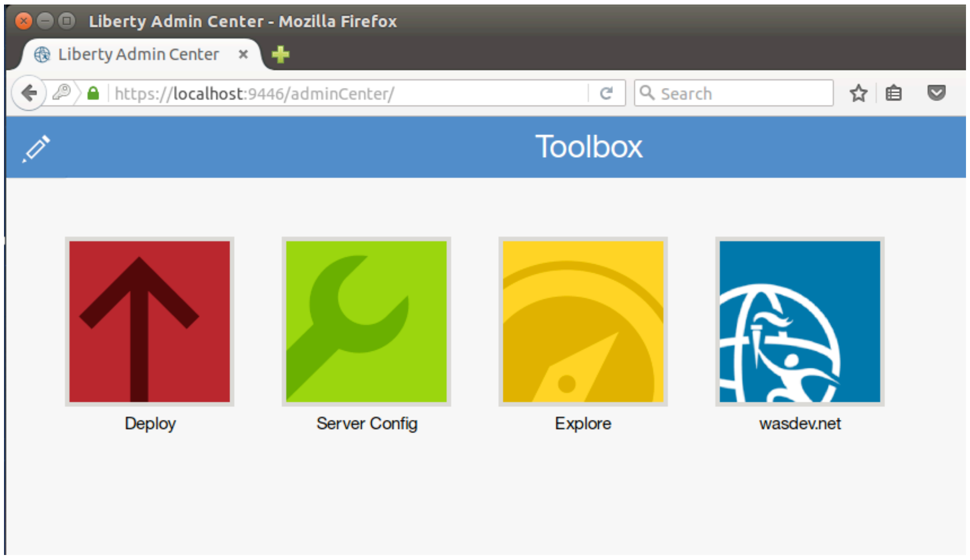
Another benefit of the Liberty collective is in the licensing options. A number of management features can be configured in the controller and the member servers, which provide varying levels of management capability. The controller feature itself is only provided by the WAS-ND product, but the basic collective member feature is included in all WAS products, including the low-end Liberty Core edition. This means that Liberty servers that are licensed for Liberty Core or WAS (base) can still be part of a collective, and can be managed from the central collective controller. In addition, Dynamic routing (see Intelligent Management) can also be used in this mixed-license configuration.

Overall, the Liberty collective is a more scalable, robust and flexible management model, and provides for more choice in licensing; however for existing WAS users it requires investment in learning the new system and creating new automation scripts.

4.4 Administration Console (Web Admin GUI)

WAS provides a rich web admin GUI (the WAS Admin Console) for browser-based control and monitoring of traditional WAS application servers, clusters and cells. Fine-grained updates to configuration and applications are fully supported.

Liberty has an Admin Center feature: a web-based graphical interface to monitor and manage Liberty servers and topology. It has been designed around a toolbox model, so users can select tools in a customized Admin Center instance.


Liberty Admin Center - Mozilla Firefox
Liberty Admin Center
https://localhost:9446/adminCenter/
Search
Toolbox
Deploy
Server Config
Explore
wasdev.net

The Admin Center tools allow operations against both standalone servers and collectives; the administrator can view the whole collective topology and can control servers and applications, as well as deploying server packages to host machines that are registered to the collective. The

Admin Center has a greater focus on monitoring and control than on fine-grained updates, as many users are adopting DevOps tools for automated system provisioning and configuration. Deployment and update of individual applications is not supported in the current release, which may be an inhibitor for users who want to perform manual delta-updates rather than following the better practice of automated server package deployment.

The Admin Center provides graphical views of the monitoring data available in Liberty, and the WAS-ND Job Manager console can be used for high-level control of standalone Liberty servers, but not, for example, to make fine-grained configuration updates. The traditional WAS Admin Console can also be used to manage Liberty servers under the assisted lifecycle model (see **Error! Reference source not found.**).

4.5 Intelligent Management

In WAS ND 8.5, the intelligent management capabilities of the WebSphere Virtual Enterprise product were integrated with traditional WAS, providing useful additional value in the product. The Java ODR Proxy can be used to provide request routing based on service policies and workload; servers can be started and stopped by the system as resources like CPU and memory approach configured thresholds; health monitoring can trigger automatic responses to a variety of conditions, and applications can be updated without any downtime. In WAS ND 8.5.5 the requirement for a separate ODR Proxy process was relaxed by adding the ODR logic into the WAS web server plugin, so the Intelligent Management functions can now be used with a topology that comprises the IHS or Apache web server, a WAS ND Deployment Manager, and the endpoint application servers.

There are Liberty features that provide some of those intelligent management capabilities. Dynamic routing feeds updated routing information to the web server plug-in as application endpoints change, and autoscaling causes servers to be started and stopped automatically in response to changes in resource use. Health policies, added to Liberty in 8.5.5.7, allow pre-defined actions to occur in response to breaches in policy configuration based on the percentage of requests timing out, the average response time, the total memory use or a memory leak. Other functions including service policies, custom health policies and application update management are not yet available for Liberty.

4.6 Hybrid Management and Assisted Lifecycle

Hybrid management is the ability to manage multiple types of endpoint (such as different application server types) from a single management process. The WAS traditional DMgr can control various different types of application server, including Liberty servers, through its Assisted Lifecycle support. The Liberty Collective Controller has a plug point for deployment and control of processes, so it can manage any process type that you provide a Deploy rule for. Out of the box, it can manage Liberty and Node.js (StrongLoop) servers, either of which can be running in a Docker container.

If you want to introduce Liberty servers into your existing WAS topology, there are two options for managing Liberty servers in traditional WAS cells. These servers will not have the same 'look and feel' as traditional WAS servers, for example you cannot perform fine-grained updates to the Liberty server configuration through the WAS Admin Console (although you can import and export a Liberty profile server.xml); however these options can provide a faster way to start managing Liberty servers from a central point without diving fully into collectives if you have an existing WAS-ND cell.

When Liberty was first introduced in WAS 8.5.0, it was possible to manage Liberty server endpoints through the WAS ND Job Manager process. The Job Manager was introduced in WAS Version 7 as an administration point for a geographically separated and looseley-coupled collection of cells and stand-alone application servers. The Job Manager is a server type that was added to support flexible management. In contrast to a Deployment Manager, a Job Manager supports a looseley-coupled, asynchronous style of management. As denoted by its name, the function of the Job Manager is to queue jobs to application servers. These queued jobs are pulled from the Job Manager by the local administrative process and distributed to the appropriate application server or servers for execution and configuration change. The jobs can be submitted as available immediately or at a scheduled time and date.

With Liberty, a Job Manager can be used to:
- Deploy (and remove) Liberty server packages on registered host machines
- start, stop, and view status (running/not running) of Liberty servers
- generate and merge web server plugin configuration files for Liberty servers

In WAS 8.5.5, the WAS ND Assisted Lifecycle model was extended to include Liberty servers. Under this model, existing Liberty servers can be started, stopped and monitored (running/not running status) from a DMgr process. Server configuration and log files can be uploaded to and downloaded from the DMgr process, and viewed in a text editor in the traditional WAS Admin Console. Unlike the Job Manager model, Assisted Lifecycle can also manage Liberty servers in dynamic clusters, meaning that the servers can be automatically stopped and started in response to changes in workload. Servers cannot be created using this model, and node agents are required on the Liberty host machines.

The Job Manager is not as commonly used as the DMgr, but it does retain the advantage that local agent processes are not required and thus the managed Liberty nodes do not require WAS ND licenses. With both models, the liberty servers are represented in the WAS Admin Console, and operations on the servers can be performed (and scripted) through *wsadmin* commands, but those commands (and scripts) are not the same as those that can manage traditional WAS servers, and can't be reused if you later move to the Liberty collective model.

4.7 Automation Through Scripting

The traditional WAS *wsadmin* command-line tool has a number of command objects with a wide range of administrative capabilities, including fine-grained configuration updates. Use of *wsadmin* can be scripted using JACL or Jython (the JACL support is no longer strategic and is stabilized). Use of *wsadmin* on a machine without a full traditional WAS installation requires the Administration Thin Client library. A library of Jython scripts is provided with WAS to perform many common administrative functions; users can copy and customize those scripts. Jython is packaged with WAS at a version that works with *wsadmin* and the product scripts.

Liberty is administered through JMX calls to the server MBeans, and by direct modification or replacement of the configuration files; no command-line tool (equivalent to *wsadmin*) is required. This means that the client-side requirements for standalone administration clients are minimal and any scripting language can be used. A small restConnector library allows Java clients to easily establish secure connections to the JMX server running in the Liberty kernel.

There are a number of sample scripts for Liberty available on the WASdev community site. Most of the scripts are written using Jython, but users are free to download the Jython version of their choice or to use a different scripting language. There is also a Jython wrapper for the restConnector library included with Liberty.

4.8 Monitoring and Logging

Traditional WAS can output monitoring data (such as usage statistics) from most components, and has a set of MBeans to provide access to that data. These MBeans require a client side library as they return custom types. Visualization of the monitoring data is built into the Tivoli Performance Viewer (TPV) tool in the WAS Admin Console, and data can be saved to the local filesystem.

Liberty produces similar statistics but so far from a smaller number of components. MBean access is easier, though, since they all return simple Java types and there is no client-side library requirement. This also means that standard JMX clients like JConsole can be used to access the data. The Liberty Admin Center *Explore* tool has graphical views of this data

A number of monitoring solution products work with both traditional WAS and Liberty, including IBM Application Performance Diagnostics (APD), CA Application Performance Management (Wily Introscope), AppDynamics Application Performance Management and New Relic Application Performance Monitoring.

Both runtimes produce messages, trace and other output data for operation monitoring and problem diagnosis. High performance binary logging is available for both. Liberty has a feature to output log data, in a JSON format, to a separate process such as Logstash, which is the collection process for the popular open source *ELK* stack (Elasticsearch, Logstash, Kibana).

4.9 Containers and Cloud

Docker images for Liberty are available on Docker Hub, and Docker files to help you build your own WAS Docker containers are available for both Liberty and traditional WAS. These containers can be used for development, testing and production, whether deployed to the IBM Containers service or any other Docker engine. The Liberty collective controller can manage Liberty docker containers.

Both WAS cells and Liberty collectives move nicely to Infrastructure-as-a-Service (IaaS) clouds like SoftLayer, and PureSystems provides efficient deployment of cells and collectives both on-premise and off-premise. Architecturally, WAS and Liberty are equally suited to running in an IaaS, but such environments do have the particular benefit to existing WAS cells in allowing then to move to the cloud largely unchanged, thus preserving what may be a large and long-standing investment in applications, automated administration scripts, and administration skills.

Liberty has advantages in PaaS environments, where its smaller size and greater flexibility make it more suited to the rapid provisioning used in instance management in those systems. Liberty is the Java runtime in Bluemix, so is the better choice when you are writing new applications (or decomposing existing ones) to take advantage of the application services in those environments.

Pre-deployed instances of both traditional WAS and Liberty can be rented in Bluemix, Softlayer, Azure and AWS clouds. Bluemix has a choice of instance 'sizes' as well as standalone and managed instances (traditional WAS cells and Liberty collectives).

5 Other Operational Characteristics

5.1 HTTP Traffic

WAS and Liberty servers handle HTTP traffic in pretty much the same way. Both profiles allow the use of reverse proxies using common configurations for both WAS and Liberty; configuration of virtual hosts can be done on both and the same browser types and versions are supported as HTTP clients.

If additional capabilities are required in the proxy, HTTP requests can be routed through a web server with the WebSphere plugin, in the same way for both profiles. The web server plugin configuration can be generated and merged by product tools. WAS has additional tools to automatically install the plugin configuration on the web server, otherwise the two profiles provide similar capabilities for HTTP routing, session affinity and failover, as well as for automatic, dynamic updates to the routing end points in the plugin configuration if they change on the server. DataPower can be used with both profiles, however the Java ODR Proxy server only works with traditional WAS.

5.2 HTTP Session Distribution

Both profiles allow HTTP session distribution via persistence to a database, and to WebSphere eXtreme Scale (WXS) for distributed memory-memory caching. Entitlement to use WXS for HTTP Session caching with either profile is included with WAS 8.5.5. Liberty does not provide built-in memory distribution (in WAS this is based on DRS), but it can be achieved through use of WXS. In both profiles, session persistence is configured at the server level with no changes needed to application files.

5.3 Web Service Requests

Stateless web service requests are handled the same way in both profiles and can be routed through a web server using the WebSphere plugin. Liberty does not support the failover of stateful web service requests since that requires routing through the Java Proxy server. Service Mapping (new in version 8.5.5) is only available in traditional WAS.

5.4 Remote EJB Requests

Both profiles are compliant with the full EJB specification, but the capabilities provided beyond the specification are different.

WAS provides the following capabilities for remote EJBs that are not provided by Liberty:
- Workload management (WLM) and failover
- Transaction context propagation (so XA transactions can span multiple WAS JVMs)
- Security attribute propagation (although full CSIv2 support is provided per the EJB specification)

The absence of WLM and failover for remote EJBs may limit the applications that can be deployed in Liberty.

5.5 Java Messaging Service (JMS) Support

The same JMS clients for WAS and WMQ messaging engines are provided in both profiles. Messaging engines can also be configured on both profiles.

JMS engines on the Liberty profile are always standalone and cannot be clustered in the way that WAS messaging engines can, to scale to high workloads with failover capability. The Liberty JMS engine is therefore better suited to development and test, or low-volume production use. The message store in the Liberty message engine is always file-based and can be easily restored from a backup and accessed by a different server if necessary for disaster recovery.

5.6 Transaction Integrity

Both profiles contain the same WebSphere Transaction Manager, which provides high integrity for XA transactional work and automatic resolution of transactions when the server restarts after a system failure. Both also allow the transaction log files to be stored in the filesyste, or in a relational database.

Liberty does not provide automated *peer* recovery of transactions, but it is easier to restore the original server configuration (on a new host, if necessary) so the logs can be recovered. This is not really a gap in function, since transaction logs can always be restored, but can result in a longer delay before recovery can be performed.

Liberty has support for the WS-AtomicTransaction protocol since 8.5.5.9, which provides propagation of XA (global) transactions between stateful web services running in different server processes.

5.7 Web Response Caching

The dynacache component from WAS is available in Liberty as the distributedMap and webCache features. The distributedMap feature provides an API to access an in-memory cache that is local to the server (JVM) process; it is named after the existing WAS API that it exposes and does not provide cache distribution without the addition of a distributed cache service like WXS.

Web response caching can be configured for applications in both profiles through addition of a cache-spec.xml file to the application. Web service response caching is not provided by Liberty.

5.8 JDBC Connection Pooling

The same high-performance WebSphere Connection Manager component is available in both profiles.

5.9 Security Services

Both profiles provide servers that are secure out-of-the-box; there are no default passwords and remote administration is always secured. The OAuth, OpenID and OpenIDConnect (client) protocols are supported by both profiles, as are SPNEGO and LTPA tokens. Liberty also has support for OpenIDConnect server/provider, and for the production and consumption of Java Web Tokens (JWT).

Some security capabilities that are provided by WAS are not yet available with Liberty. They include security auditing (e.g. login failures) and enhanced key and certificate management. SAML web SSO has been delivered in Liberty 8.5.5.7 and SAML web services in 8.5.5.8.

In Liberty, a simple user registry can be added to the server.xml file, which is useful for quick testing; LDAP, SAF and custom user registries are supported and recommended for production use; LDAP can be federated, and federation across the other registry types has been added in 8.5.5.9. Apart from SAF, no local OS registries are supported by Liberty. There is a REST API (SCIM) to query and manage registry content (users and groups) in Liberty. The traditional WAS file-based registry can't be used by Liberty servers.

All of the security required by the full Java EE platform is provided by both WAS and Liberty. Application security can be added incrementally to a Liberty server configuration with the appSecurity feature. This means you can test applications without the security constraints being applied, then configure the server to apply them when you are ready to do security testing; this incremental approach can also be applied to JMS and web services transport security.

Liberty provides a single administrative user role with no fine-grained control around resource access. On the face of it, this seems like a serious restriction for Liberty administration, but when you consider the agile dev/ops flows being built with Liberty, it becomes less of a concern. While the primary administrator of the system retains the *administrator* role, and the associated authority to administer the deployed server configuration and application, anyone else who needs to contribute configuration and application files now does so through the dev/ops flow, ideally by delivering their updates into a source code repository which then triggers a build and subsequent application of the build artifacts (application and configuration files) onto the deployment system. Thus it it only the administrator who needs to have direct access to the deployed system, and others can contribute their changes in a way that is inherently audited and controlled (and easy to back out in the event of a problem).

5.10 Java Batch Management

Both runtimes have the new Java Batch (JSR-352) API that is part of the Java EE7 specification. Management capabilities for those new batch applications, though, are only available with Liberty.

6 Other Considerations

6.1 Performance Comparisons

Liberty starts servers and applications faster than WAS. This is largely due to its composable architecture; also through *late and lazy* design that defers loading and initialization of services and applications until they are used. Application start is also generally faster on Liberty, and some application initialization may be delayed until the application is first accessed, although that behavior is configurable.

The performance of request processing is very similar on the two profiles in most scenarios, because the application request paths are mostly common code (channels, transports, containers etc). On the z/OS platform, the single-process architecture of Liberty provides significantly higher request throughput than the split-process architecture of WAS on z/OS; about 30% higher for the web services SOA benchmark on z/OS Liberty for example. Production performance is therefore not a differentiator between the profiles except on z/OS.

The two profiles have similar overheads for application security, SSL, monitoring, and both have the same high-performance binary logging (HPEL) component.

6.2 Upgrades and Migration

The different architectures of WAS and Liberty largely dictate how new function can be delivered. WAS is a monolithic Java EE runtime and, while minor functions can be delivered at any time, major enhancements like new Java EE versions need to be delivered in a single, major release upgrade. Liberty's feature architecture is more flexible, and allows independent delivery of individual Java EE technologies as well as any other capability. This flexibility is the basis of the continuous delivery model now being used to provide new features to Liberty users at regular intervals; these features can be downloaded from the online Liberty repository and configured into servers as and when they are needed.

Moving to a new version of WAS typically involves three aspects of migration; changes to the system configuration, changes to the application that are required by the WebSphere code, and changes to the application required by a Java version upgrade. The tightly coupled nature of the WAS cell also dictates that a DMgr must be upgraded before any managed nodes. There have been major investments to minimize the number of changes that users have to make in order to upgrade their WAS systems, and to provide tools to automate most of the work. However, a major version upgrade remains a significant effort.

Liberty was designed to have minimal migration requirements, and to make it easy to use the same configuration and applications with multiple versions of the runtime. User configuration remains valid for all service and release levels of the runtime, with no migration required. A single environment variable can be used to 'point' the runtime at the user data, which can be kept in a distinct location (this also makes it easy to control read/write access for the product binaries, the user data, and the server output area, all of which can be controlled with single environment variables). The loosely-coupled nature of Liberty Collectives means that the different processes can be upgraded in any order, or run at different versions indefinitely.

The feature architecture of Liberty allows for new versions of features to be added to the product while the old ones are retained, so behavior can evolve in the new features while preserving backward compatibility in the original features. Upgrades of Java EE specification levels have traditionally forced WAS (and other Java EE application servers) to introduce application-breaking changes in order to be compliant, because there is a single implementation of each Java EE technology in that runtime. Liberty can have features for different versions of any given technology, only applying the new behavior to the new features, so existing applications are protected if the configuration (the features being used in the server) is not changed. Users can configure the feature versions that their application needs into each server instance.

The one remaining area that may require application changes for a Liberty version upgrade will be if a Java version upgrade is required (for example when support for running with Java 6 is removed); in that case there may be changes required for an application to be compatible with Java 7; however, version to version migration with Liberty is trivial compared with migration for other Java EE application servers.

6.3 Installation and Service Application

The WAS install has a few optional features, but in general the whole product is installed to the local system, providing a single version of the Java EE Platform and WAS APIs. Liberty has a modular install; features (or sets of features called *addons*) can be independently installed onto the Liberty kernel, providing custom installs as needed. IBM hosts an online repository of Liberty content which can be accessed using command line and development tools to download and install content. A local repository can also be established, within a company firewall, to enable and control access to specific content.

Both profiles can be installed using Installation Manager (IM), which will also install a supported, WebSphere-provided JDK (this is optional for Liberty). IM can be used to apply and remove iFixes and Fix Packs to both profiles.

Liberty also offers an archive-based install; the same iFixes and Fix Packs are provided for both install types. iFix removal from an archive install is a manual process. Fix Pack archives are complete replacements and the best practice is to perform a parallel, non-destructive install of the new Fix Pack to allow easy switching between old and new service levels. IBM Java is now included in archive installs on limited platforms.

Liberty servers can be packaged, with or without the product runtime and Java, into archives that contain the user configuration and applications; these are known as server packages (or packaged servers). These archives can then be deployed for testing or production use, and are fully supported and serviceable. The server package is proving to be popular for continuous deployment, where host machines can be updated with a complete replacement of the whole stack.

The general service and support policies and practices are identical for the two profiles; the same Level 2 and Level 3 support teams handle service problems.

6.4 Supported Software

Both profiles currently support the same broad range of operating systems, with the one additional platform for Liberty only being Mac OSX. On certain desktop and client platforms, such as Windows 10, WAS traditional can only be used for development and test; Liberty is supported for all purposes on all those platforms. The following platforms will be removed from support in a future release of Liberty: HP-UX, Solaris, SUSE Linux, and RHEL/Power; after that time you may continue to use Liberty on those platforms and receive support until such time that the unsupported operating system is identified as a probable cause of the problem or a contributing factor, at which time you may be asked to recreate the problem on a supported operating system.

Liberty supports Java 6, Java 7 and Java 8; the Liberty features for Java EE 7 will not run on Java 6 since the Java EE 7 specification requires the use of Java 7 as a minimum level. Traditional WAS 8.5 also supports Java 6, 7 and 8; WAS 9.0 supports only Java 8. Traditional WAS uses the IBM Java that is installed and serviced as part of the product; Liberty can be used with the supplied/supported IBM Java but is also supported on any compliant Java implementation and is agnostic of the 32/64 bitness of the Java being used.

Both profiles support the same wide range of JDBC drivers and LDAP servers. Both have JMS clients for WebSphere Messaging and WMQ. Both support the same web browsers and web servers.

6.5 Code and product Maturity

When choosing a deployment platform for what may be a mission-critical application, it is perfectly reasonable to consider the maturity of the software itself, the team who produce and support it, and the company that sells it. Is this the first release of the code? Does this team understand your support needs? Will this company still be in business for long enough after you have invested in this platform? With Liberty the last two questions are easy to answer; Liberty is delivered by the same team who have delivered and supported WAS for many years, and who have a clear understanding of the needs of application server users, and IBM has been in business for over one hundred years and is a thriving business built largely on its reputation for solid product support.

When it comes to the code, it is useful to understand how Liberty has been built. The kernel is mostly new code for WAS, but relies largely on OSGi components that have been used for many years in Eclipse. The Liberty features however consist almost entirely of components that are reused from traditional WAS. These components are modified ‘around the edges’ to achieve three things:

- component lifecycle is managed by the new Liberty kernel
- component configuration is injected by the kernel
- dependencies between components have been minimized.

The core code of these components however remains largely unaltered; so the code that interacts with applications in the containers for example, or that processes application requests in the transport layer, retains the same behavioral and performance characteristics, as in traditional WAS. The same fixes are applied to the code running in both profiles, maintaining common behavior, in both WAS and Liberty. The result is that most of the feature code has been tested and maintained for many years in WAS and now continues to be tested and fixed across both profiles.

7 Summary

As stated in the introduction, if you are using traditional WAS and it does what you need, you should not feel compelled to move to Liberty. Many companies have large investments in traditional WAS infrastructure and automation, and IBM will continue to serve and support those customers. It is expected that there will be some applications that remain on WAS indefinitely, because it is impractical or cost-prohibitive to adapt them to run on Liberty. These applications have a path to the cloud by moving WAS servers into containers, or deploying servers or even complete clusters into a VM or IaaS service where existing administration scripts can be reused.

There are a number of reasons though to choose Liberty for application deployments. The main motivations described by current Liberty adopters are immediate cost reduction and increased flexibility of the application infrastructure.

Cost reductions are achieved through a more flexible license mix, faster deployment of applications and reduced resource use. License flexibility comes from the ability to centrally manage Liberty servers of all product editions as collective members. Application deployment is faster when using the same runtime for development and production, and by use of modern DevOps tools and flows. Resource use is reduced with Liberty by having right-sized server instances and an agentless management system.

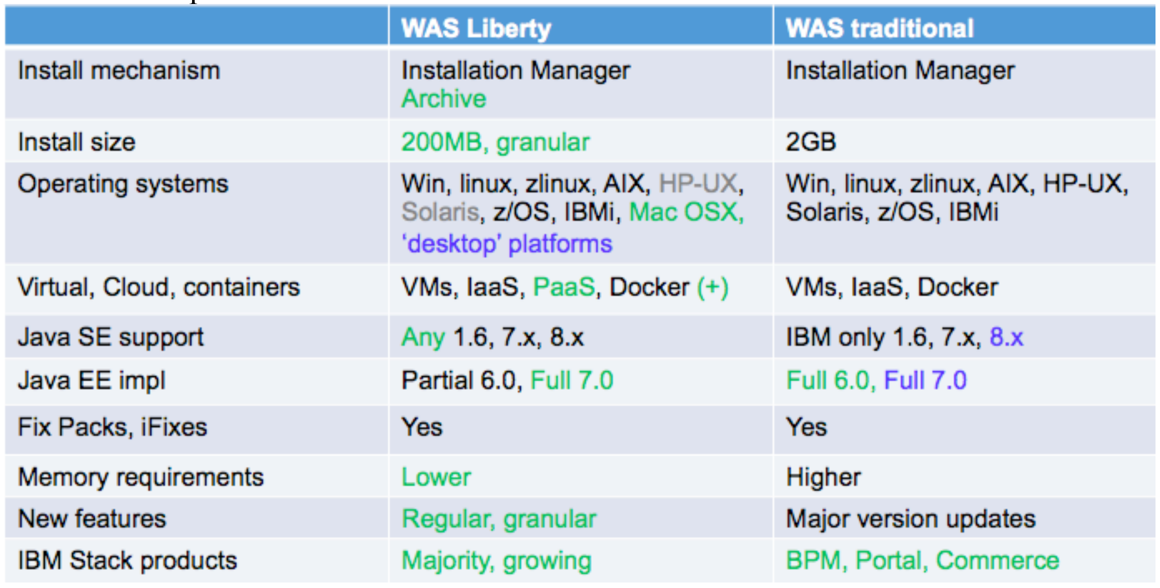
Next-generation application infrastructures are being designed around the Liberty runtime to achieve highly flexible DevOps models. Application developers version control the server configuration in a source code repository, and the output of the build is a server package archive. These archives may be deployed by the collective controller, a third-party tool like UDeploy, Chef or Puppet, or home-grown scripts. The infrastructure team will then set appropriate values for configuration variables, and if required, override the development-provided configuration with production settings using the configuration dropins locations.

Whichever WAS profile you choose to run your applications, you are using a reliable, robust application server with world-class performance and the full backing of IBM, so either way you have made a great choice!

Thanks to Tom Alcott and Don Bourne for input and review.

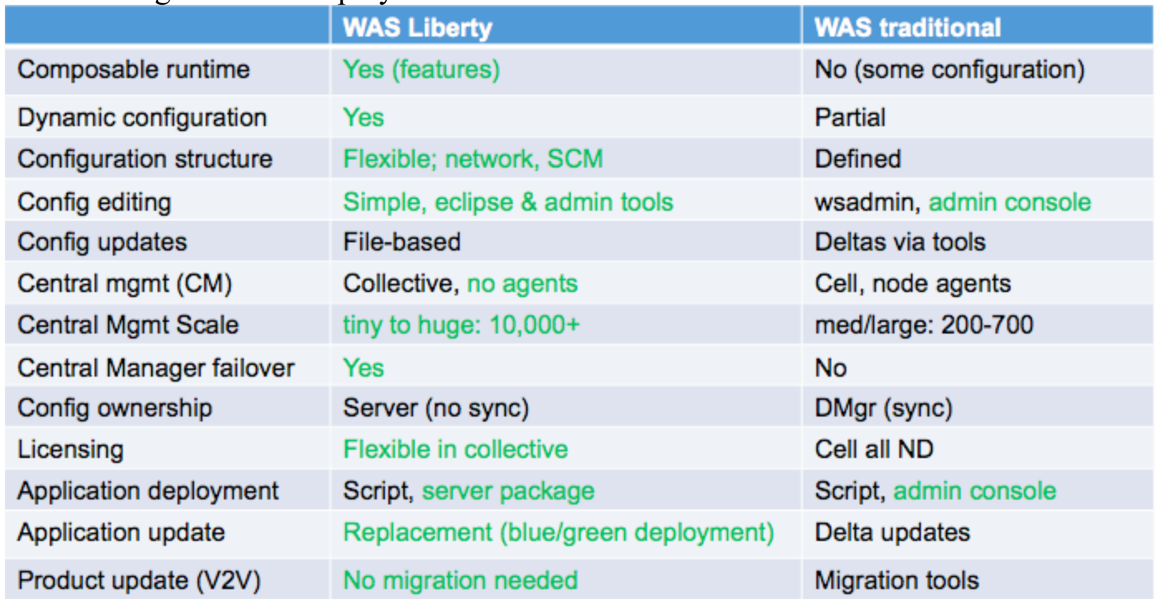
8 Appendix

8.1 General product differences



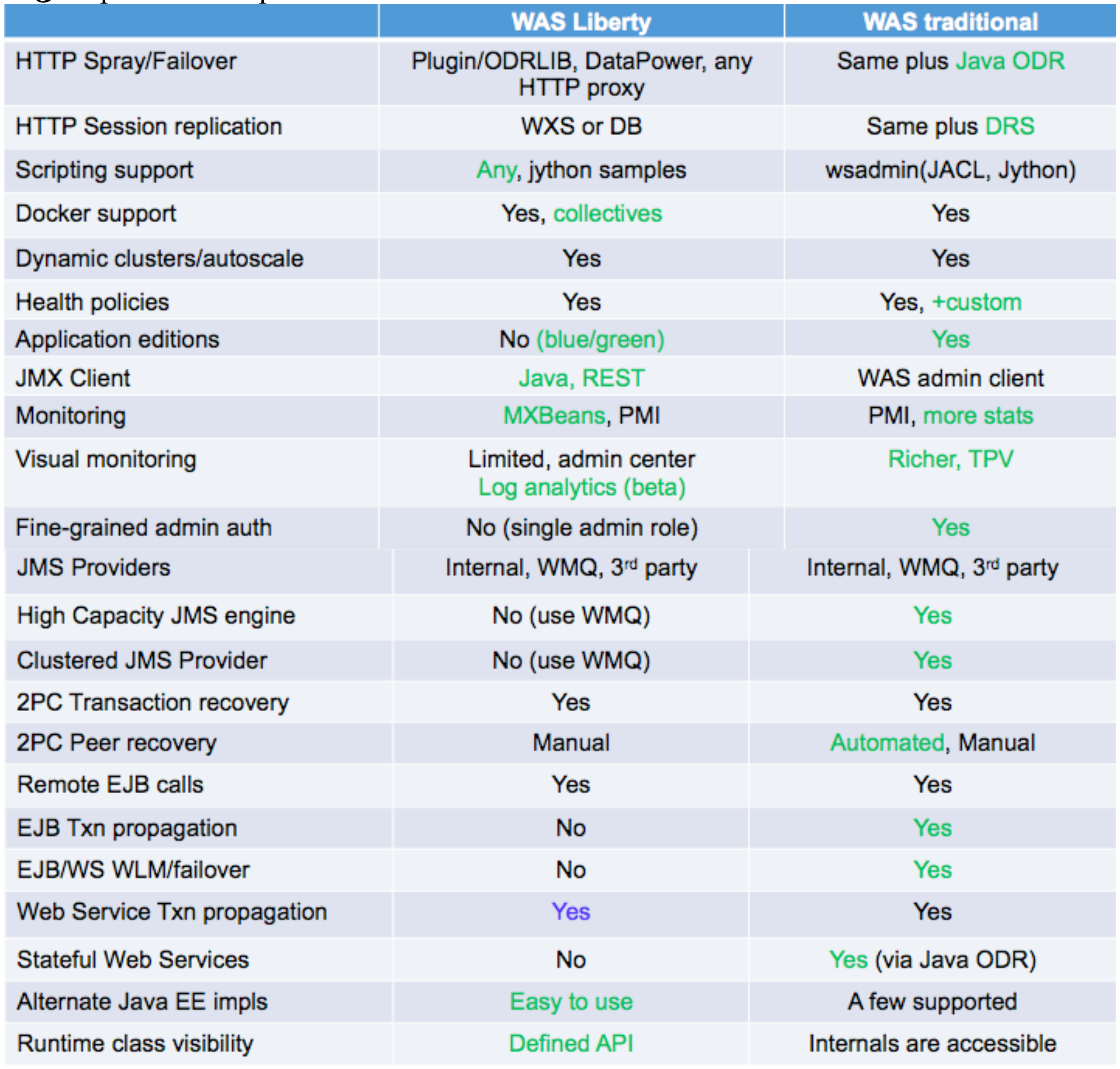
	WAS Liberty	WAS traditional
Install mechanism	Installation Manager Archive	Installation Manager
Install size	200MB, granular	2GB
Operating systems	Win, linux, zlinux, AIX, HP-UX, Solaris, z/OS, IBMi, Mac OSX, 'desktop' platforms	Win, linux, zlinux, AIX, HP-UX, Solaris, z/OS, IBMi
Virtual, Cloud, containers	VMs, IaaS, PaaS, Docker (+)	VMs, IaaS, Docker
Java SE support	Any 1.6, 7.x, 8.x	IBM only 1.6, 7.x, 8.x
Java EE impl	Partial 6.0, Full 7.0	Full 6.0, Full 7.0
Fix Packs, iFixes	Yes	Yes
Memory requirements	Lower	Higher
New features	Regular, granular	Major version updates
IBM Stack products	Majority, growing	BPM, Portal, Commerce

8.2 Configration and deployment



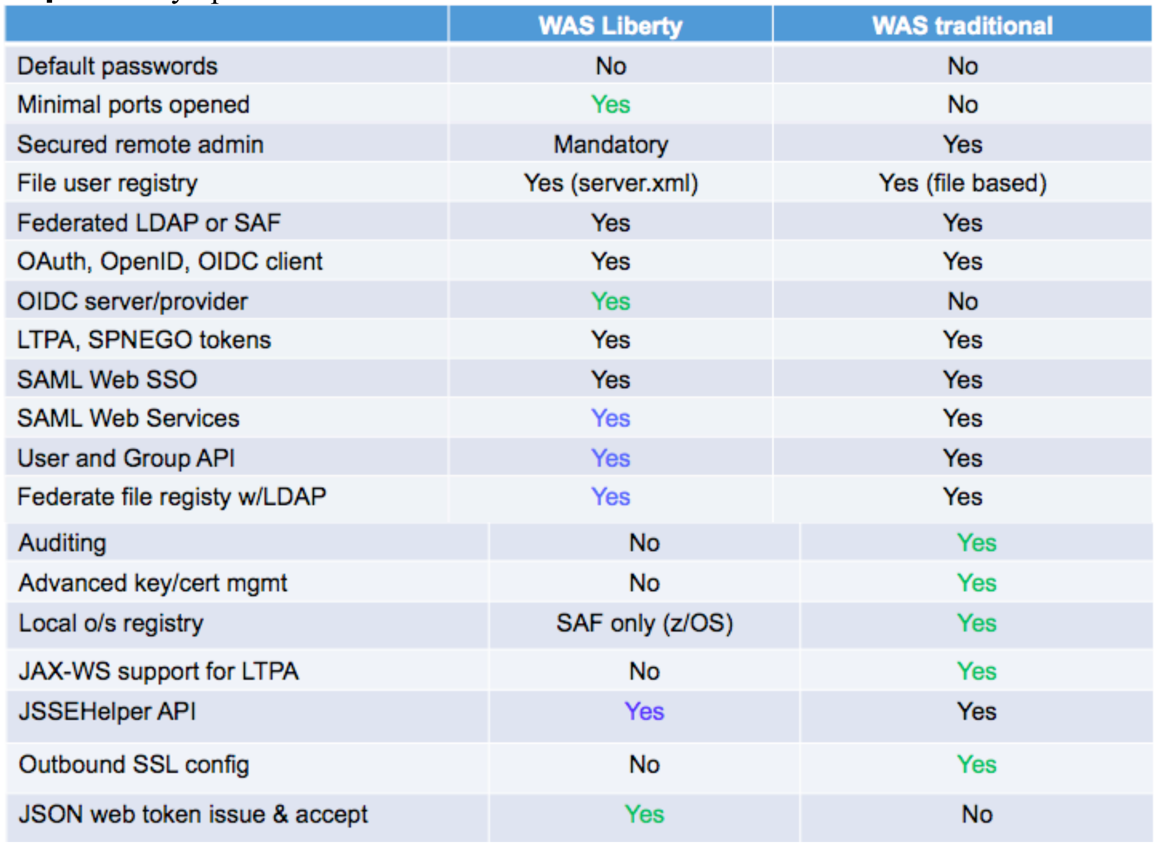
	WAS Liberty	WAS traditional
Composable runtime	Yes (features)	No (some configuration)
Dynamic configuration	Yes	Partial
Configuration structure	Flexible; network, SCM	Defined
Config editing	Simple, eclipse & admin tools	wsadmin, admin console
Config updates	File-based	Deltas via tools
Central mgmt (CM)	Collective, no agents	Cell, node agents
Central Mgmt Scale	tiny to huge: 10,000+	med/large: 200-700
Central Manager failover	Yes	No
Config ownership	Server (no sync)	DMgr (sync)
Licensing	Flexible in collective	Cell all ND
Application deployment	Script, server package	Script, admin console
Application update	Replacement (blue/green deployment)	Delta updates
Product update (V2V)	No migration needed	Migration tools

8.3 Operational capabilities



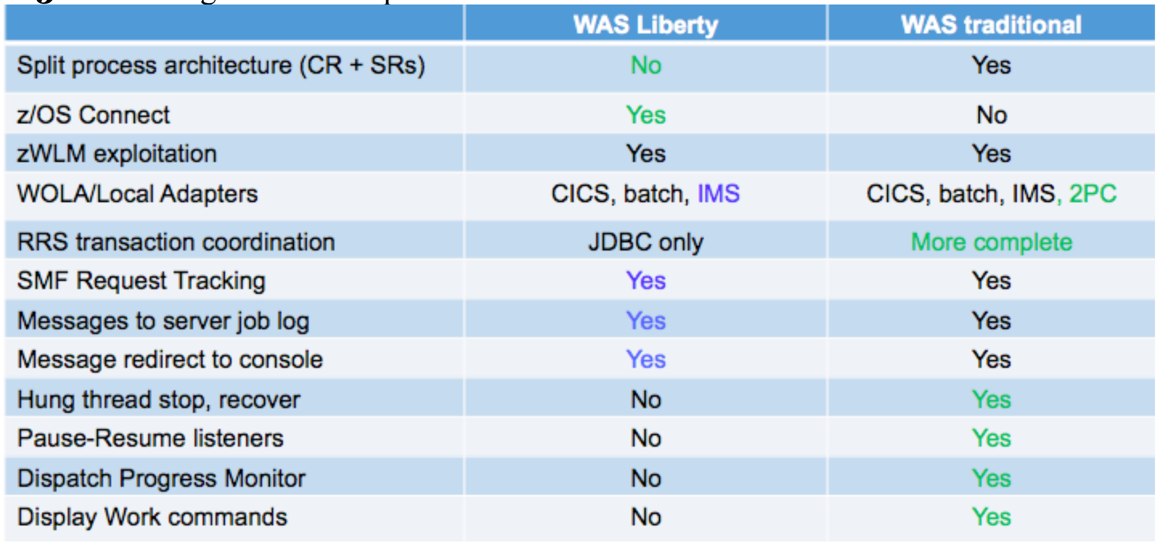
	WAS Liberty	WAS traditional
HTTP Spray/Failover	Plugin/ODRLIB, DataPower, any HTTP proxy	Same plus Java ODR
HTTP Session replication	WXS or DB	Same plus DRS
Scripting support	Any, jython samples	wsadmin(JACL, Jython)
Docker support	Yes, collectives	Yes
Dynamic clusters/autoscale	Yes	Yes
Health policies	Yes	Yes, +custom
Application editions	No (blue/green)	Yes
JMX Client	Java, REST	WAS admin client
Monitoring	MXBeans, PMI	PMI, more stats
Visual monitoring	Limited, admin center Log analytics (beta)	Richer, TPV
Fine-grained admin auth	No (single admin role)	Yes
JMS Providers	Internal, WMQ, 3rd party	Internal, WMQ, 3rd party
High Capacity JMS engine	No (use WMQ)	Yes
Clustered JMS Provider	No (use WMQ)	Yes
2PC Transaction recovery	Yes	Yes
2PC Peer recovery	Manual	Automated, Manual
Remote EJB calls	Yes	Yes
EJB Txn propagation	No	Yes
EJB/WS WLM/failover	No	Yes
Web Service Txn propagation	Yes	Yes
Stateful Web Services	No	Yes (via Java ODR)
Alternate Java EE impls	Easy to use	A few supported
Runtime class visibility	Defined API	Internals are accessible

8.4 Security options



	WAS Liberty	WAS traditional
Default passwords	No	No
Minimal ports opened	Yes	No
Secured remote admin	Mandatory	Yes
File user registry	Yes (server.xml)	Yes (file based)
Federated LDAP or SAF	Yes	Yes
OAuth, OpenID, OIDC client	Yes	Yes
OIDC server/provider	Yes	No
LTPA, SPNEGO tokens	Yes	Yes
SAML Web SSO	Yes	Yes
SAML Web Services	Yes	Yes
User and Group API	Yes	Yes
Federate file registy w/LDAP	Yes	Yes
Auditing	No	Yes
Advanced key/cert mgmt	No	Yes
Local o/s registry	SAF only (z/OS)	Yes
JAX-WS support for LTPA	No	Yes
JSSEHelper API	Yes	Yes
Outbound SSL config	No	Yes
JSON web token issue & accept	Yes	No

8.5 z/OS integration and exploitation



	WAS Liberty	WAS traditional
Split process architecture (CR + SRs)	No	Yes
z/OS Connect	Yes	No
zWLM exploitation	Yes	Yes
WOLA/Local Adapters	CICS, batch, IMS	CICS, batch, IMS, 2PC
RRS transaction coordination	JDBC only	More complete
SMF Request Tracking	Yes	Yes
Messages to server job log	Yes	Yes
Message redirect to console	Yes	Yes
Hung thread stop, recover	No	Yes
Pause-Resume listeners	No	Yes
Dispatch Progress Monitor	No	Yes
Display Work commands	No	Yes