Smarter Balanced System Architecture and Technology Report

Assessment System Architecture and Technology Recommendations for the Smarter Balanced Assessment Consortium

11 July 2014

Table of Contents

1. Overview

Assessment System Architecture and Technology Recommendations for the Smarter Balanced Assessment Consortium

Prepared in accordance with RFP SBAC 03 by Measured Progress.

measured progress

1. Overview

This document provides a comprehensive understanding of the Enterprise Architecture of the Smarter Balanced Assessment System. The document is the culmination of a rigorous effort undertaken to identify the business goals and processes that will make up the Smarter Balanced Assessment System, and the technology components and their relationships needed to address these business goals and processes.

The intended audience for this document is the application architects of the teams that will be tasked with designing and developing the eventual software applications that will deliver on the business goals and processes. The document provides a framework to guide these application architects while they make the many decisions necessary to design the individual systems and applications.

It is important to note that the Enterprise Architecture is a definition of what components and relationships are needed to address the business goals and processes, not how the eventual applications will deliver on these. The how will require much further analysis by the application architects and their respective development teams during later phases of this project. The value of the Enterprise Architecture is to provide the necessary frameworks and context that the application architects will need to build a cohesive software system of applications that efficiently work together to solve the business goals and processes.

The Enterprise Architecture definition in this document contains the following:

Architecture Principles
A set of guiding principles to be considered by all teams, systems, and applications. These principles are a set of musts defined by Smarter Balanced teams early on, and should be used to guide all design solutions going forward.

High-Level System Component Diagram
Identifies the individual components and their relationships. Each of these components will result in one or more applications in the final system.

Domain Definition
Looks at the assessment system's domain model, which highlights the scope, attributes, and relationships of a domain. This is followed by a series of high-level system component diagrams that illustrate the various component parts and the functions they perform.

Deployment and Hosting
Presents the deployment and hosting models and their different capabilities in a number of scenarios, demonstrating possible set-ups. Also suggests how the assessment system could function either independently or when incorporating a state's or a district's system(s), where applicable.

Data Architecture Definition

Highlights the principles on creation, management, and delivery of assessments by providing best-practice recommendations and industry benchmark approaches where appropriate.

Interoperability

Discusses the interoperability among components, identifying where interoperability is required and at what stage, as well as providing suggested standards to use to enable such capability.

Non-Functional Requirement Constraints

Addresses concerns with regards to the assessment system. These include: requirements and recommendations on open licensing, interoperability and standards, system high-availability and scalability, accessibility, and technology.

Security

Highlights the security concerns that must be considered when designing and implementing the components. These include requirements for component-to-component communication, user authentication and authorization, and student data areas.

Technical Architecture Definition

Covers server and browser hardware and software requirements, networking requirements, database, data storage and archiving approaches and requirements, middleware and integration software requirements, and the security approach and requirements for applications, data, and end-user access.

Application Development Model

Defines suggested development processes to ensure ease of integration of the software products of the different teams.

Glossary

Provides a reference to the terminology and concepts used in this document.

Release Notes

Details the changes that have been made to this document from one version to the next.

1.1. Assessment Lifecycle

The general model of the assessment life cycle, represented in the diagram below, is designed to include any type of assessment and take into account the iterative nature of assessments. It focuses on the overall assessment processes from conception to post-delivery. Depending on the assessment type, the process can begin and end at any point in the life cycle. The six overarching categories are broken down into subprocesses. Data and information can be exchanged at any point in the process with another assessment, or administrative or instructional application.

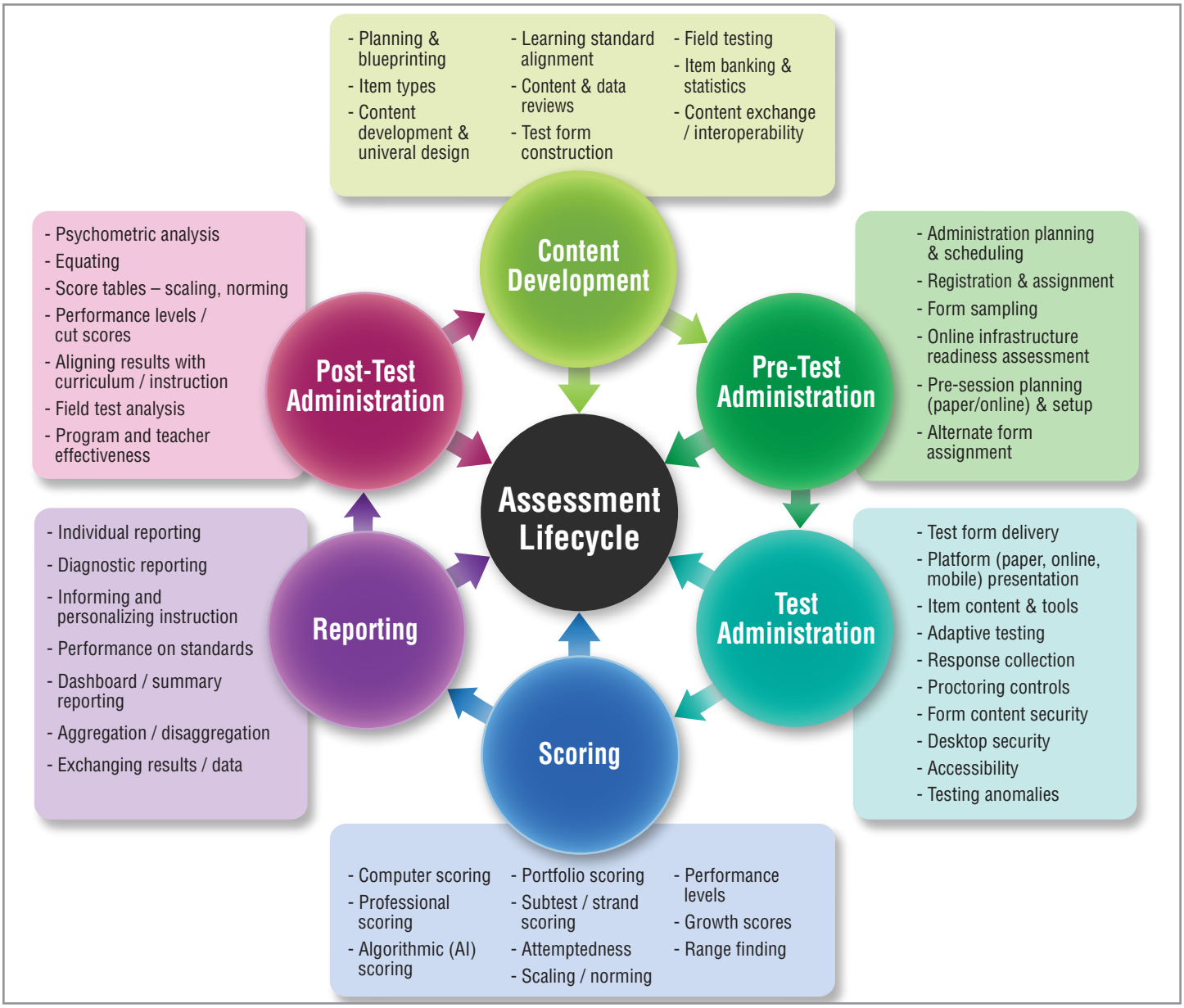

Figure 1.0 Assessment Lifecycle

Descriptions

Content Development

This phase of the lifecycle includes everything involved in developing the assessment content. This includes item and asset development, alignment to learning standards, and field testing.

Pre-Test Administration

Pre-test administration includes the processes necessary prior to test administration. This includes form assignment, registration of students, and scheduling of the assessment.

Test Administration

This phase of the assessment lifecycle includes the actual delivery of the assessment and the subprocesses contained within the phase. This includes proctoring, delivering, and collecting student responses.

Scoring

The scoring phase incorporates the actual scoring of student responses as well as any data needed for item statistics and trending.

Reporting

Reporting occurs after the scoring of the assessments. This reporting could be as formal as summative assessments or as informal as in an instructional setting, providing immediate feedback to teachers and students.

Post-Test Administration

The final phase in the assessment lifecycle includes all processing associated with the finality of the administered assessment, including use of the data and information, equating, and necessary psychometrics.

2. Architecture Principles

Assessment System Architecture and Technology Recommendations for the Smarter Balanced Assessment Consortium

Prepared in accordance with RFP SBAC 03 by Measured Progress.

2. Architecture Principles

Team leads use architecture principles to guide their decisions concerning design.

Some rules of thumb apply when defining these principles:

1. The number of principles needs be kept to a reasonable number (less than 10 is recommended).
2. They need to be maintained to support decision making. They are especially useful when the options presented are equally viable.
3. They need to be maintained by the Architecture Review Board.
4. They also need to be at a level that aids decision making.

The following are the architecture principles of Smarter Balanced, with some rationale and implications:

2.1 Choose Single-Responsibility Systems

Similar to the single-responsibility principle class, this system-level design principle encourages creating or acquiring systems that interact with other systems through standard protocols.

Rationale

- Systems must be upgraded or replaced as business needs change.
- Systems that perform multiple business functions are harder to upgrade and replace.

Implications

- Systems utilize standards for intersystem communication.
- Systems can be replaced with minimal disruption to other systems.

2.2 Design for Emergent Reuse

Emergent reuse is the ability to identify existing systems that can be used in implementing new systems. Utilizing existing systems in a new application is more effective than designing a new application.

Rationale

- Reusing capabilities in different areas reduces the overall system maintenance costs.
- Overly detailed design when designing for reuse is often expensive and ineffective.
- Identifying and refactoring for reuse is less expensive and more effective.

Implications

- Software designers and architects need to be aware of existing system capabilities.
- Systems that use open-standard interfaces that are able to interact with other systems are easier to utilize.

2.3 Develop Homogeneous Systems

Developing homogeneous systems is a prerequisite to emergent reuse. Systems that exchange information through standard protocols are easier to manage and enhance.

Rationale

- Lower maintenance costs.
- Faster on-boarding of new team members.
- Simpler environment configurations.

Implications

- Standards are extensively used.
- Small number of standard operational environments (e.g., Java containers).
- Systems are built using repeatable patterns.

2.4 Demand-Driven Releases

Demand-driven releases indicate that the IT organization is in tune with the business demand. As a result, releases are in step with business changes.

Implications

- Software and system releases are made at a frequency driven by business demand and by the business's capacity to absorb those changes.
- Software updates are available more frequently for UAT.
- Systems are well maintained and can be maintained on an ongoing basis.
- Systems are flexible and amenable to change.

2.5 Business Continuity

Software systems being available at all times to support users.

Implications

- Systems economically scale to accommodate increased business demand.
- Systems are tolerant of infrastructure faults.
- Systems support disaster-recovery scenarios.
- Operational parameters are actively monitored using runtime metrics and dashboards.
- Requirements are traceable to ensure compliance.
- Code is readable and easy to understand.
- Legacy systems are aggressively retired to maintain simplicity of options and lower maintenance costs.

2.6 Low Cost for SEA

A principle where design and implementation designs will strive towards a low cost of ownership for member SEAs, so that an SEA can implement or adopt the Smarter Balanced system with low up-front costs and low ongoing operational costs.

Implications

- Design/implementation decisions that require investments from the SEA must be balanced with benefit, and alternatives need to be considered.

3. High-Level System Component Diagram

Assessment System Architecture and Technology Recommendations for the Smarter Balanced Assessment Consortium

Prepared in accordance with RFP SBAC 03 by Measured Progress.

measured progress

3. High-Level System Component Diagram

3.1. Logical Responsibility Groupings

This section contains the high-level system component diagram of the Smarter Balanced Assessment System. The diagram illustrates the purpose of each component in the assessment system.

The components fall into one of the following logical groupings from a development point of view. The components in each grouping have similar development needs and technologies. Components within each group have a higher degree of interdependency than they do with components from other groups.

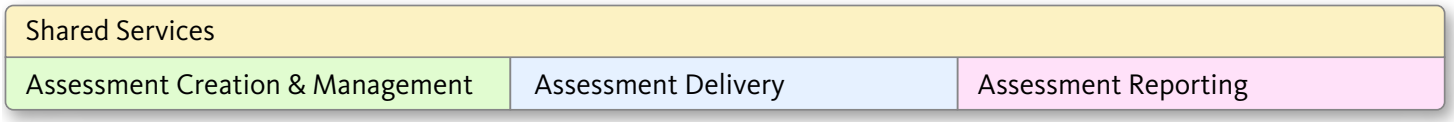

Figure 3.1 Logical Component Responsibility Groupings

Here are the brief descriptions of what each grouping is responsible for:

Shared Services

These components assist the other components and create a cohesive system for the end users.

Assessment Creation and Management

These components manage the process and workflow of the creation and lifecycle of assessment artifacts.

Assessment Delivery

These components deliver the assessment to the students and gather the data and metadata about the assessment.

Assessment Reporting

These components analyze the assessment results and produce reports intended to improve education and benefit student learning.

Definition: Component

In this document, "component" describes a logically separate capability that could (but not necessarily) be separately deployed and managed. This means that any component in the system could be replaced by other implementations of the same component. A vendor solution may entail multiple components that are tightly coupled to give enhanced functionality; parts of that solution should enable the use of other implementations of these components.

As an example, a vendor has an Item & Test Authoring and Banking system with tight integration into its own Test Delivery component. This solution should allow a Smarter Balanced member state the capability to use another Test Delivery component.

3.2. Logical Component Diagram

The diagram below depicts the components of the assessment system. This illustration provides more detail on the concept described above.

As you study the diagram, you will find that the Test Delivery component, under the Assessment Delivery group, has two subcomponents. This is to denote that they are individually deployed. It is required to have a tighter degree of integration between these two subcomponents than the other components in the diagram.

You will also find descriptions of each of the components accompanying the diagram, below.

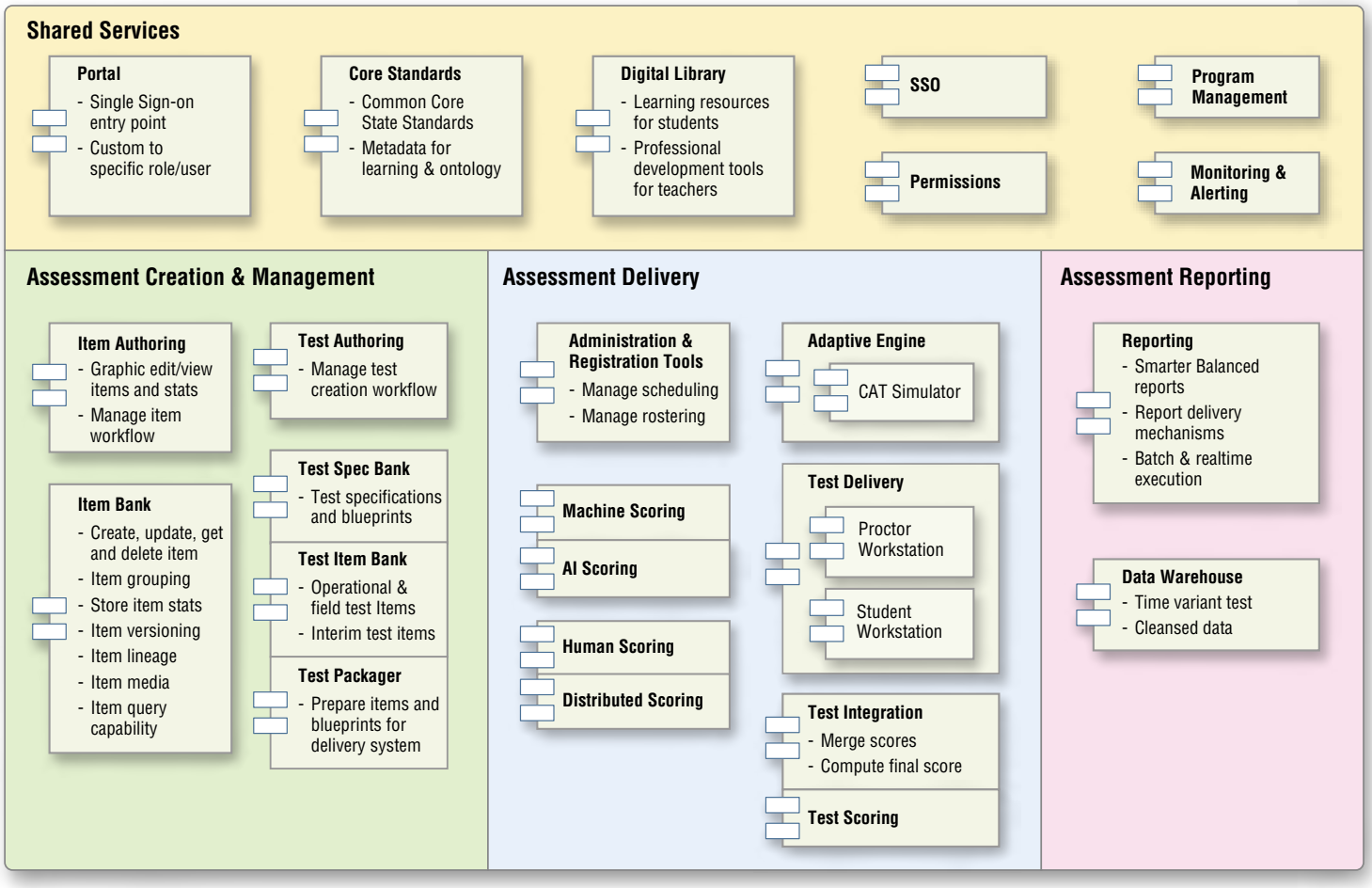

Figure 3.2 Logical Components

Shared Services

Portal

This is the entry point where end users access the components of the Smarter Balanced system. It handles what components to which a user has access to. It allows the display of information and dashboard widgets from the other components.

Core Standards & Metadata for Learning Data and Ontology

This is the component that manages the Common Core State Standards and learning metadata so that other components can reference and use them in the same manner. It is the single version of truth for these standards. When a component needs to reference a core standard, it will use the identifiers and text that have been retrieved from the Core Standards component.

Digital Library

This component is an interactive teacher professional development tool. Teachers will use this component primarily to access resources for their own professional development. This will include resources such as documents, videos, guides with sample summative / interim tests and responses, and forums. Here teachers can customize their content, post their reflections, and monitor their progress on implementing new practices. In addition, it contains a work area where teachers can identify and use the best resources for their needs; the system may also be able to use the teacher's interaction with the system to suggest additional resources.

Single Sign-on (SSO)

Responsible for user authentication from any user interface. The component's interface needs to check for an authentication token on each request. If the token does not exist, the component should redirect to the SSO system for authentication. This allows the user to sign in only once, while still able to use all the components that they are authorized to use.

Permissions

A centralized permissions management component for the system's other components. It requires that components share the same permissioning capabilities in order to reduce permissions management complexity. It also enables a consistent user experience across multiple components developed by different vendors.

Program Management

This is the master data repository and service. It is a set of services to provide data that crosses component concerns. For example, tenancy records, style sheets, Common Core State Standards, etc.

Monitoring & Alerting

A set of services that allow components to send, monitor, and act upon alerts in a consistent way. It also allows vendors to develop add-on applications and features that use and act on these alerts.

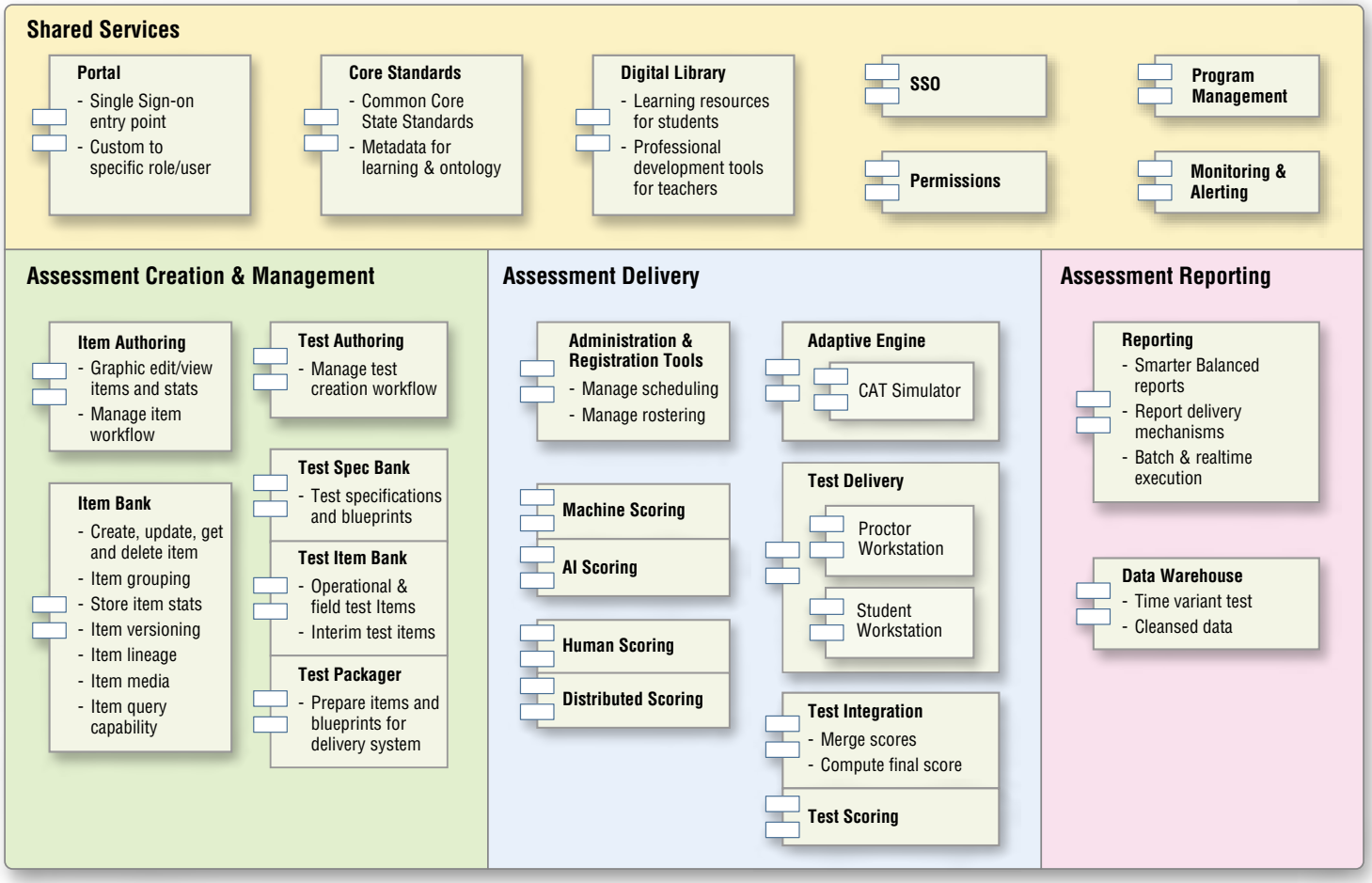

Figure 3.2 Logical Components (repeated)

Assessment Creation & Management

Item Authoring

This is a graphical interface used for the authoring and workflow related to item creation. It interacts with the Item Bank component.

Item Bank

This component is responsible for:

- Storing and retrieving assessment items.
- Storing and retrieving assets and metadata related to the assessment items.
- Tracking item versioning.
- Tracking item lineage. (If an item changes to such an extent that it becomes a new item, the lineage tracks what the item used to be.)
- Providing a robust search and query capability that allows searching on all types of metadata.

The Smarter Balanced instance of the Item Bank will be considered as the system of record for Smarter Balanced items. Items can be moved into other Item Bank and Test Item Bank instances. An item in the Smarter Balanced instance will be considered the definitive source of the item.

Test Authoring

This component is a graphical interface used for creating test blueprints and specifications, and managing the workflow. It will interact with the Test Spec Bank component and the Test Item Bank component.

Test Spec Bank

A repository for test specifications, blueprints, and other data about tests, such as the adaptive algorithm to be used during the test.

Test Item Bank

Similar to the Item Bank, but adapted to handle the load of a live assessment, this component contains items that are in operational, field, or interim tests.

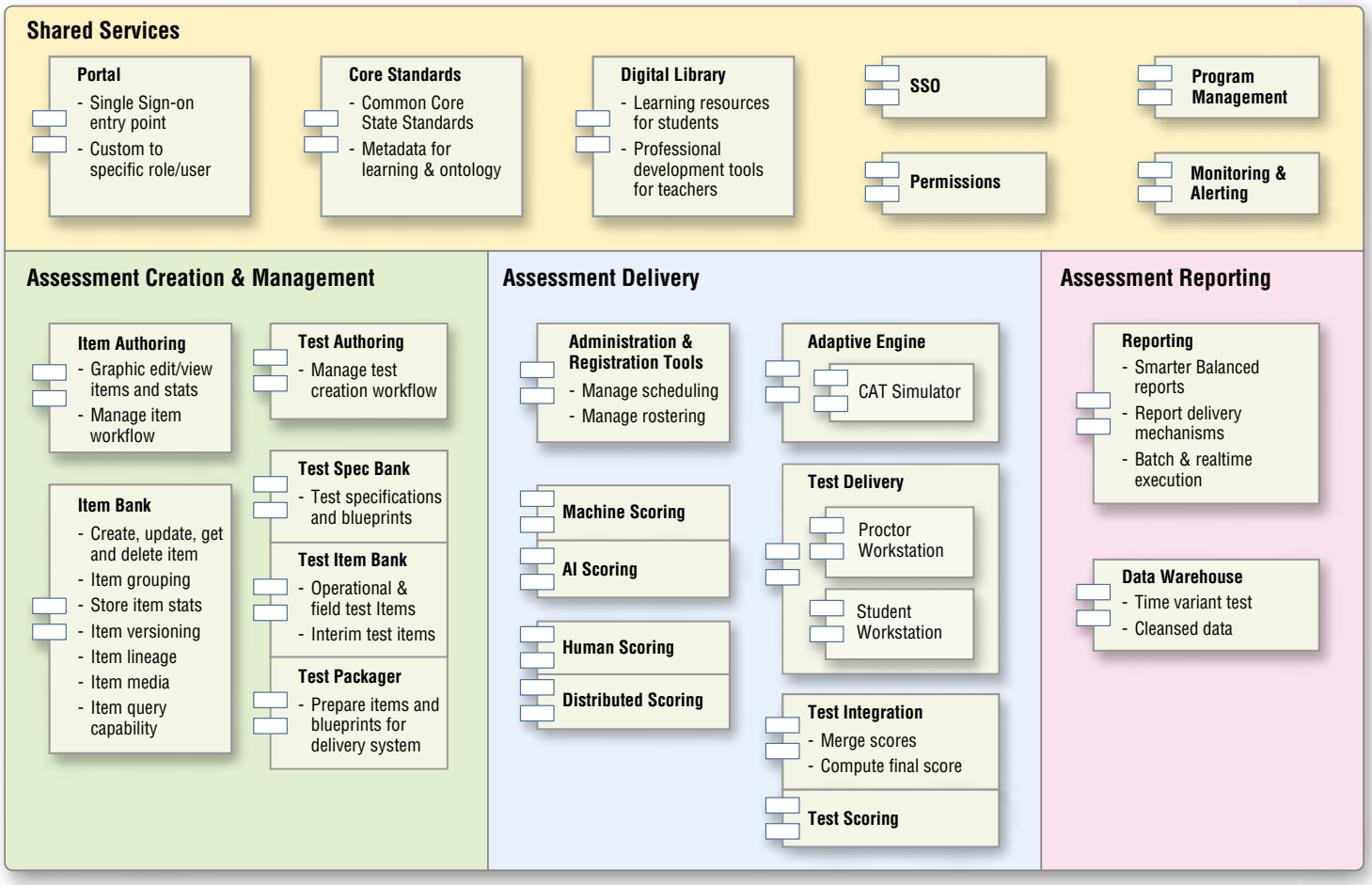

Figure 3.2 Logical Components (repeated)

Test Packager

This component prepares the test items and the test specifications for use by the test delivery system. Test Packager preprocesses assessment assets to make them more efficient for the Test Delivery component. The packager creates the assessment instrument that a Test Delivery component can consume and use to deliver the assessment.

Assessment Delivery

Administration & Registration Tools

This component manages the capabilities and methods required for assessment scheduling, test windowing, room scheduling, proctor assignment, student assignment, and student identification methods. This component also registers the student(s) for assessments and interacts with the Student Information System (SIS) to gather the student information and the accessibility profile. It must also manage staff identification for managing assessment events.

Adaptive Engine

This component is an implementation of an adaptive algorithm. Multiple adaptive engines may be developed. Similar to the AI Scoring component, this component will have performance issues if implemented as a network service instead of as a Test Delivery component plugin. The Test Spec Bank contains metadata that defines the algorithm or engine to use. The Test Delivery component is expected to load the correct algorithm or engine when the test is being administered.

Machine Scoring & AI Scoring

Components that programmatically score items in real time while the student is taking the test. These must be high-performing components. The Test Delivery component must initialize the engine with items so that the engine can preprocess and cache information to most efficiently score assessment items in real time.

Human Scoring & Distributed Scoring

This component provides the interface humans use to score items and view rubrics on how to score the items even when the scorers are not centrally located. It also delivers those scores back to the Test Delivery and Data Warehouse components to be stored with the student responses. It also allows for the development of machine scoring capabilities that are used in conjunction with the human scoring capabilities. It should be able to use the AI Scoring engine(s) that are used in adaptive testing by Test Delivery.

Test Delivery

The overall responsibility of this component is to:

- Securely deliver the assessment to the student.
- Store the student responses.
- Store other information about how the student responded (e.g., time to answer, time to render for the student).
- Deliver the test items in the format appropriate to the student's accessibility needs.

Student Workstation	This subcomponent interacts with the student. It delivers items to the student and gathers the responses and response metadata. It also contains the tools the student needs to take the test. (e.g., calculators, tables, accessibility tooling.)
Proctor Workstation	This is a subcomponent that the proctor uses to manage the test delivery. It allows the proctor to start, stop, suspend, resume, and help students when they are having issues.

NOTE: Test Delivery must be able to provide scalability and allow for deployment of additional servers as required to meet demand. In addition, it must also be designed for the highest level of recoverability, redundancy, and traceability. This component will encompass the largest number of hardware and network differences.

Test Integration

Since hand-scored items go through a different process than machine-scored items, this component takes the machine-scored items of a test and integrates them with the hand-scored items of the same test. After scoring is complete, the data is then uploaded into the Data Warehouse.

Test Scoring

This component is responsible for taking all item scores from a student's test and then scoring the test. This includes scores for any reporting categories, including strands, standards, and benchmarks.

Assessment Reporting

Data Warehouse

This component contains information moved from the Test Delivery components. This data should be temporal in nature so that queries against the data can be executed at different times. While data warehouses at the state level may or may not need that capability, the database schema will support it should the states elect to use it.

Reporting

This component must be able to run Smarter Balanced created reports against the Data Warehouse, and to deliver those reports in multiple formats to authorized users who need to view them. It also must be able to generate and deliver custom-built reports that each state, LEA, etc. may create. Some of these reports may be scheduled to run at a specific date and time, repeating if necessary. These reports may be created when a user makes the request (i.e., in near real time).

3.3. Component Interfaces

This diagram shows the connection points between logical components, including where interoperability standards need to be defined and followed.

1. The shared services box contains components that are required by most of the other components.

2. The dotted arrows indicate connection points between components, and are labeled with either an action or an artifact that exists between the components.

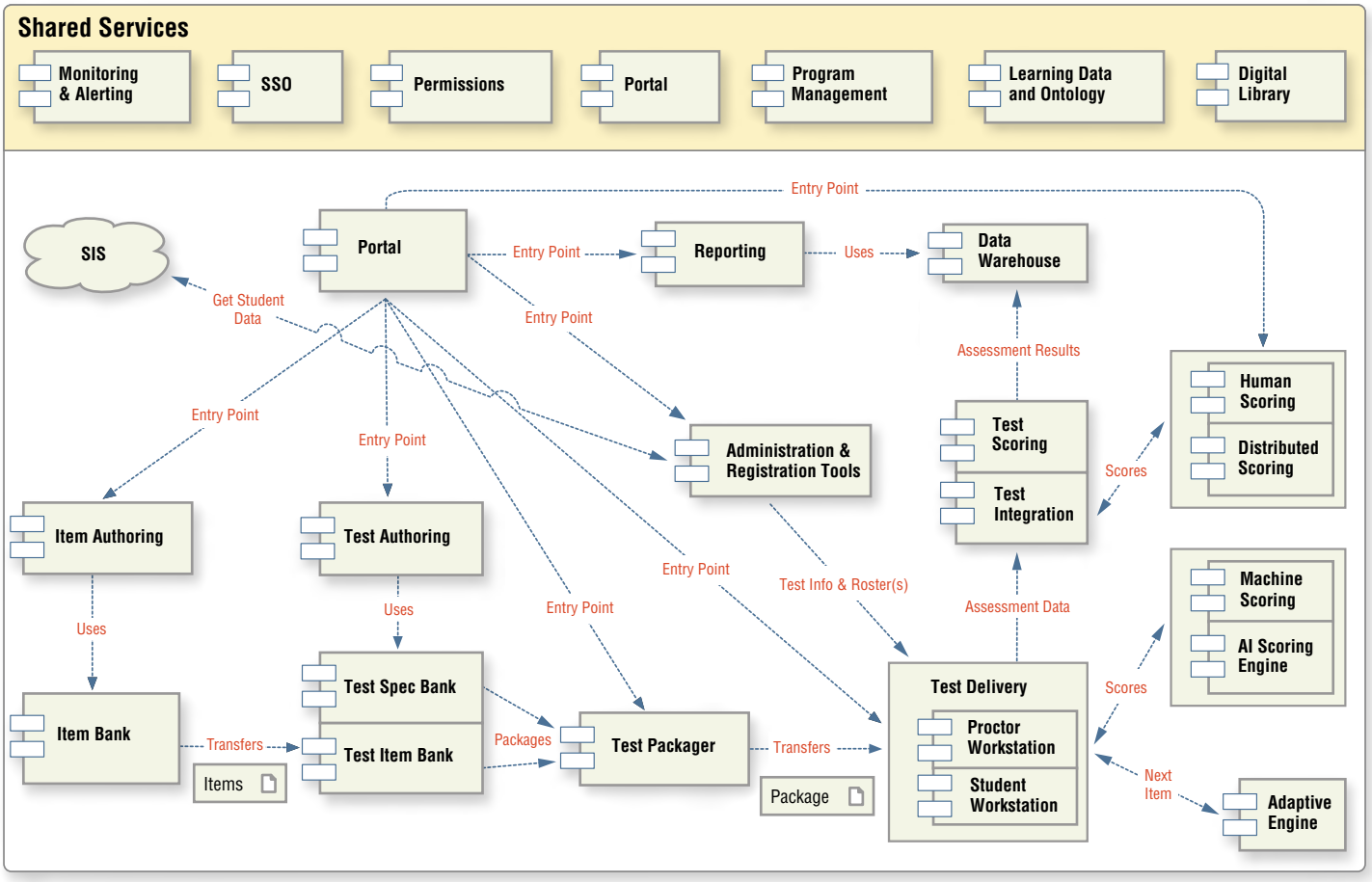

Figure 3.3 Logical Component Interfaces

3.4. Component Transport Path

The component transport path is the path that artifacts will take through the Smarter Balanced components.

The Plugin Binary Transport determines the optimal transport between components. This requires an abstract API to be developed that components can call with a consistent interface. The API uses the data format described by the accepted standard for that asset's domain (e.g., item format, student information, student response.).

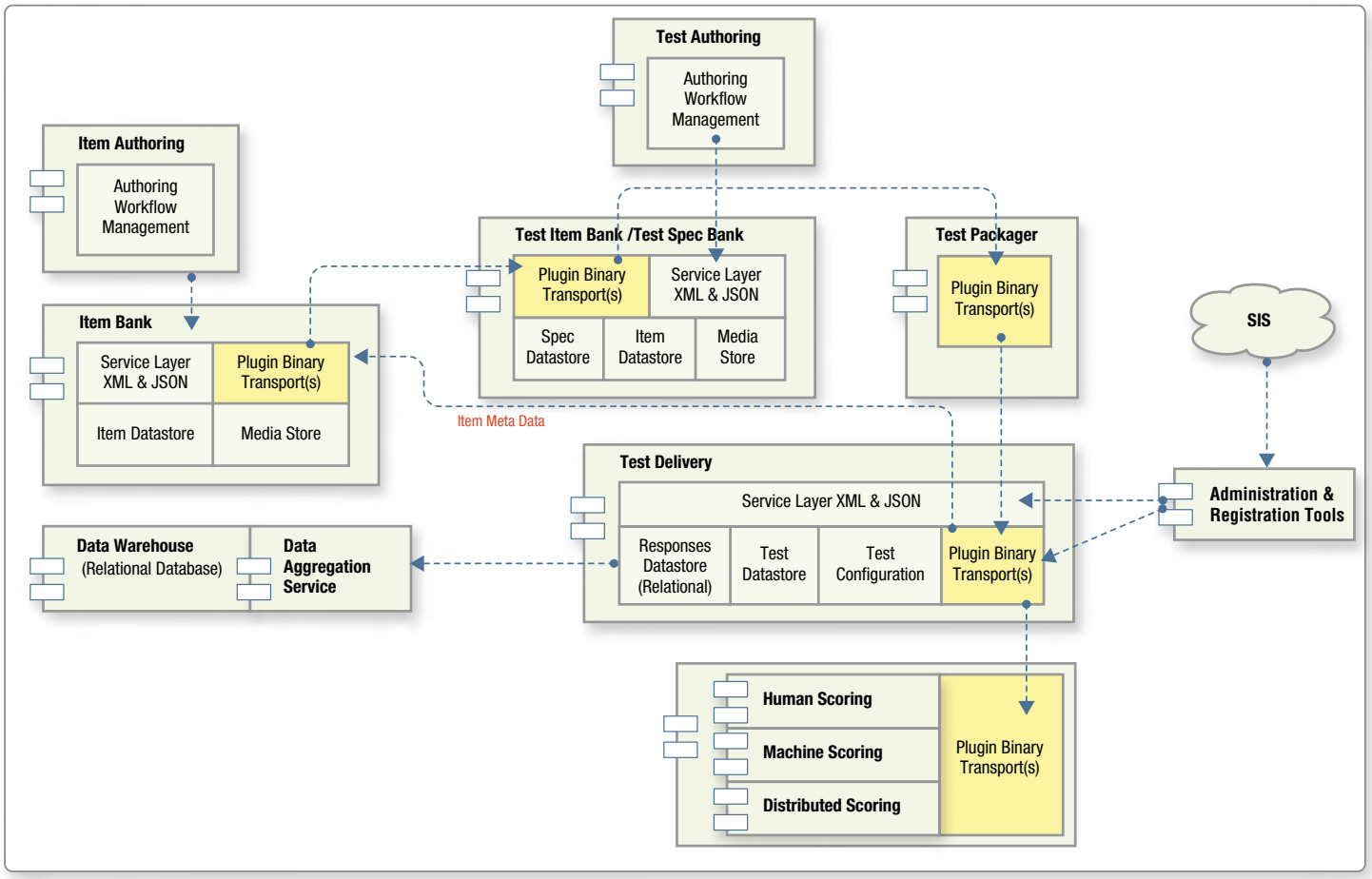

Figure 3.4 Component Transport Path

The following Java™ example shows how a component may use a plugin transport:

```
public void sendItems(List<Item> items,SBACTarget target) {
 SBACWriter itemWriter = SBACInterop.getWriter(SBACType.ITEM,target);
 itemWriter.write(items);

}

public List<Item> receiveItems(SBACSource source, long limit) ) {
 SBACReader itemReader = SBACInterop.getReader(SBACType.ITEM, source,
limit);
 return (List<Item>)itemReader.read();

}
public void fooMethod(List<Item> items)
{

 //export items to the filesystem
 sendItems(items,SBACTarget.EXPORT);

 // retrieve items that have been sent to us from
 // the configured source item banks and save to the datastore
 // limit to 500 at a time so as not to run out of memory
 List<Item> retrievedItems = new ArrayList<Item>();
 while((retrievedItems = receiveItems(SBACSource.ITEM_BANK,500)) != null
)

 ItemRepository.save(retrievedItems);

 //send items to the configured target Test Item Bank
 sendItems(items,SBACTarget.TEST_ITEM_BANK);
 ....

}
```

Following is an example plugin configuration file:

```
<plugins>
 <source name="ITEM_BANK" type="ITEM" description="SBAC item bank">
 <interop-standard name="QTI_2.1"
 provider="org.sbac.interop.provider.qti2_1.itembank"/>
 <transport name="SBAC_TEST_ITEM_BANK"

 provider="org.sbac.transport.hadoop">
 <hadoop-config file="/opt/hadoop-0.20.0/conf/hdfs-site.xml"/>
 <hadoop-dir dir="/sbac/test_items"/>

 </transport>
 </source>
 <source name="ITEM_IMPORT" type="ITEM" description="">

 <interop-standard name="QTI_2.1"
 provider="org.sbac.interop.provider.qti2_1.itembank"/>
 <transport name="ITEM_IMPORT_DIR"
 provider="org.sbac.transport.filesystem">
 <dir name="/home/itembank/import"/>

 </transport>
 </source>
 <target name="ITEM_EXPORT" type="ITEM" description="">

 <interop-standard name="QTI_2.1"
 provider="org.sbac.interop.provider.qti2_1.itembank"/>
 <transport name="ITEM_EXPORT_DIR"
 provider="org.sbac.transport.filesystem">
 <dir name="/home/itembank/export"/>
 </transport>
 </target>
</plugins>
```

These examples show that the Plugin Binary Transport is configured outside of the Java™ program. The SBACInterop returns an object that understands the object type and interoperability standard. The source / target object must deliver the object to the requested format, and both deliver it to and return it from the configured transport (e.g., file system, HTTP, or socket). Components with well-defined sources, targets, and types can be developed independently and can be configured to use different transports and interoperability standards.

There is a case that if using the same platform and language across the components, it would be possible to share the code across these components.

Although most data that needs to be moved can be represented in an XML format, binary assets such as graphics, movies, sound files, etc. also must be moved. These assets can be large and the space to store them grows significantly when multiple instances of these file types are required as part of an assessment. This causes system memory issues and complicates the use of protocols such as HTTP. By creating a pluggable transport capability, the most efficient transport can be used between components. For example, it may be efficient to use an XML REST API to deliver test results from the Test Delivery component to the Data Warehouse component, but the transport among the Item Bank, Test Item Bank, and the Test Delivery component may use an Apache Hadoop™ Distributed File System.

Important

This will also allow multiple vendors to innovate methodologies in order enhance this transport, and incorporate other features at these integration points and plugins.

3.5. Alignment of Logical Components to the Assessment Lifecycle

The following maps components to the corresponding lifecycle area.

Content Development

- Item Authoring
- Item Bank
- Test Authoring
- Test Packager
- Test Spec Bank
- Test Item Bank

Pre-Test Administration

- Administration and Registration Tools

Test Administration

- Adaptive Engine
- Test Delivery
- Monitoring and Alerting

Scoring

- Human Scoring
- Distributed Scoring
- Machine Scoring
- AI Scoring

Reporting

- Date Warehouse
- Reporting

Post-Test Administration

- Portal

Supporting Features

- Program Management
- Digital Library
- SSO
- Permissions

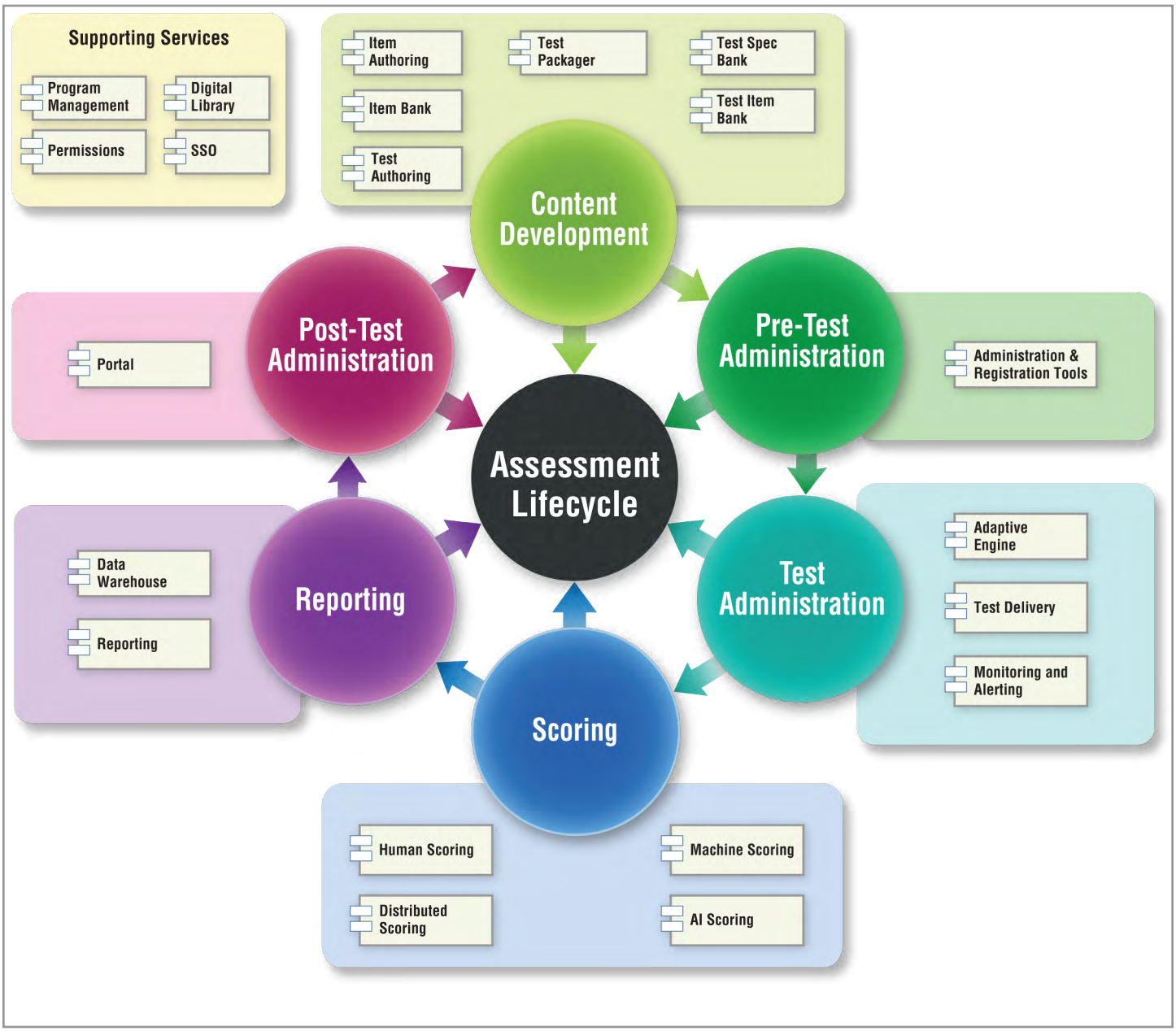

Figure 3.5. Alignment of Logical Components to Assessment Lifecycle

4. Domain Definition

Assessment System Architecture and Technology Recommendations for the Smarter Balanced Assessment Consortium

Prepared in accordance with RFP SBAC 03 by Measured Progress.

4. Domain Definition

This section details the assessment system's key concepts by exploring its domain in various visual representations.

Introduction

The approach used in this section is commonly known as the domain model. A domain model identifies the vocabulary, attributes, and relationships among all the entities within the scope of the problem domain. This section guides readers in understanding the assessment system through a clear depiction of the domain concept. Domain models are also known as Conceptual Entity Relationship Diagrams.

Shown in this section is a series of domain models that expresses the various aspects of the assessment system: assessment creation, delivery, reporting, and shared services.

The following diagrams use crow's foot notation [http://www2.cs.uregina.ca/~bernatja/ crowsfoot.html], which shows relationships and cardinality, or quantities, between domain objects. These diagrams are not intended to show data-level attributes, or to be all-inclusive, but rather to identify a set of critical domain objects of which all components must be aware.

NOTE: Application architecture will define how the domain object attributes will be stored. The attributes must comply with the attributes and names as defined by the interoperability standards for that specific domain object type.

- APIP [http://www.imsglobal.org/apip.html]
- SIF [https://www.sifassociation.org/Resources/Developer-Resources/SIF-3-0/Pages/]

4.1. Assessment Creation Domain

Descriptions:

Item
A composite object that is made up of many item parts and metadata (data providing information about one or more aspects of the item) about that item.

Item Part
Includes things such as the graphics, multimedia, stem(s), item text, option groups, options, etc. that make up an item.

Item Template
A predefined form meant to be used as the starting point for creating an actual item.

Test Template
A predefined test specification form meant to be used as the starting point for creating a test specification.

Testlet
A set of related items that need to delivered together (e.g., items that are part of a stage in a staged adaptive test).

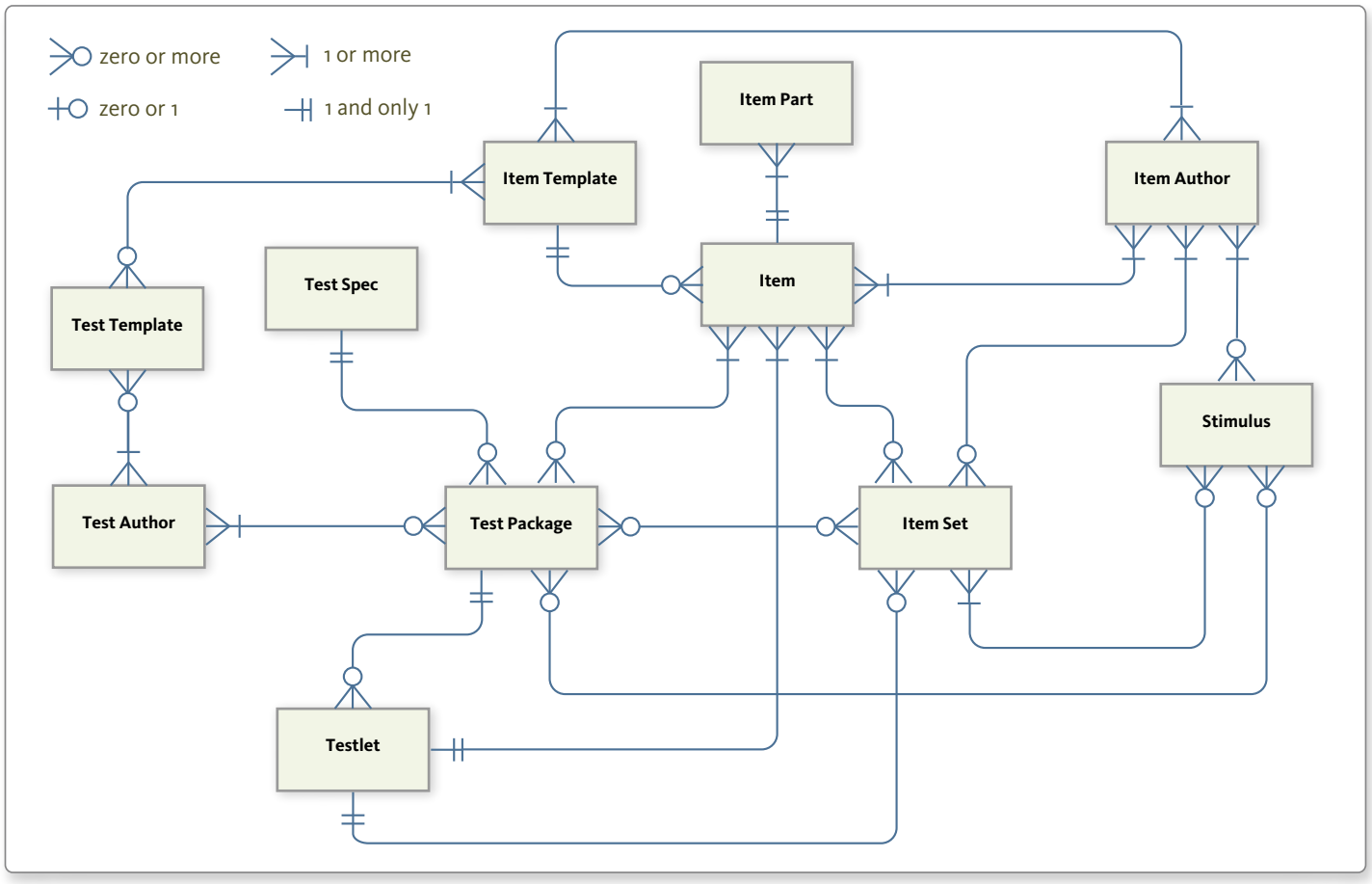

Figure 4.1 Assessment Creation Domain

4.2. Assessment Reporting Domain

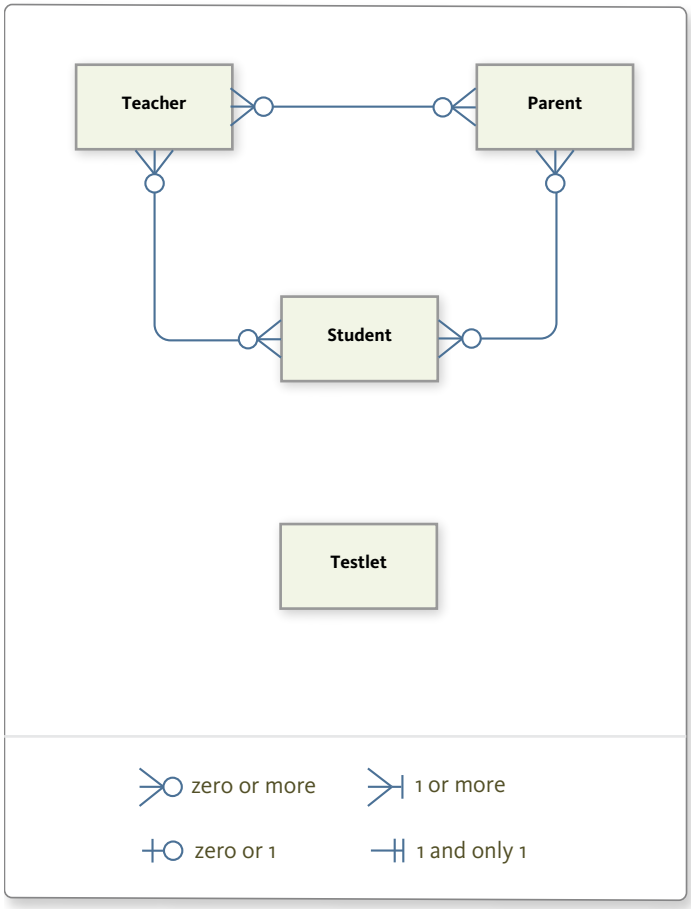

Figure 4.2 Assessment Reporting Domain

4.3. Shared Services Domain

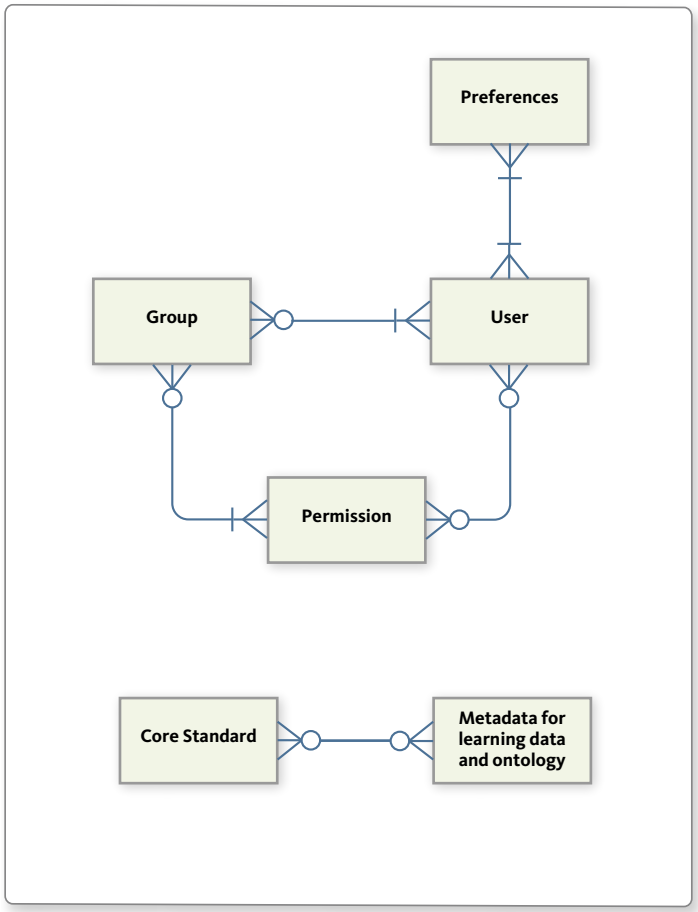

Figure 4.3 Shared Services Domain

5. Deployment and Hosting

Assessment System Architecture and Technology Recommendations for the Smarter Balanced Assessment Consortium

Prepared in accordance with RFP SBAC 03 by Measured Progress.

measured progress

5. Deployment and Hosting

A significant advantage in the architectural design of the assessment system is the capability of being deployed and hosted in a number of different environments. This allows member states, a single state, a district, or a school (each of which has distinct methods of system deployment) to use the same system. The adaptable architecture is effective in each scenario, which we will explore here.

Key considerations for deployment and hosting are:

- The physical location (data center) and its attributes such as network connectivity, clustering, security, availability, and backup facilities
- Application architecture to support data from multiple tenants and partitioning

5.1. Physical Location

The deployment hierarchy, from the highest level to the lowest level, follows:

- Consortium
- Group of States
- State
- LEA (Districts, Counties, etc.)
- School
- Classroom

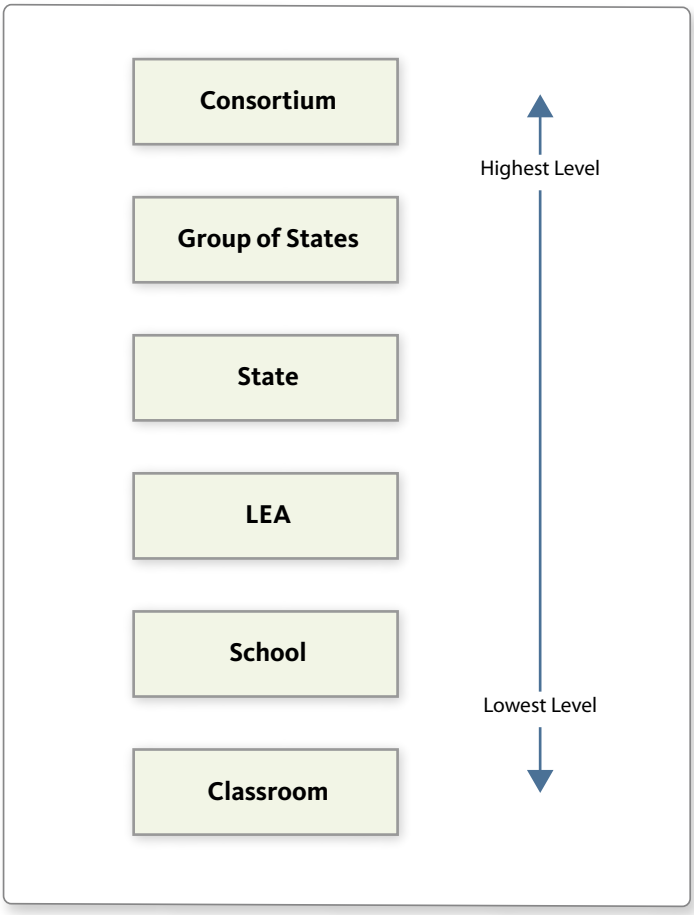

Figure 5.1 Deployment Hierarchy

As a baseline, all components must be deployable at the consortium level. For performance and connectivity purposes, some components may reside at lower levels in the hierarchy. For example, a state or LEA may use their own components. The deployment must consider all other non-functional requirements for these components.

1. Performance and connectivity reasons may force some components to reside lower in the hierarchy.

2. A state or LEA might use their own component.

5.2. Application Architecture

The assessment system is to utilize multitenancy and partitioning in its design, enabling it to implement the flexibility that is required on the above hierarchies.

Multitenancy is the ability for a single instance of a component to host data pertaining to multiple tenants. For example, a single Test Delivery system instance hosted at the consortium level may serve multiple states. The Test Delivery system must be modeled to allow each state access only to data that pertains to the state. (see Figure 5.2) Partitioning creates a smaller, targeted instance of a component or a group of components to be deployed at the lower levels of a hierarchy. Components that are deployed as partitioned must have the ability to synchronize data with the consortium-level component. This is most critical for components such as Test Administration and Registration, Delivery, and Scoring. When components are to be deployed in a partitioned fashion, they will still need to have the ability to synchronize data up to the consortium-level component.

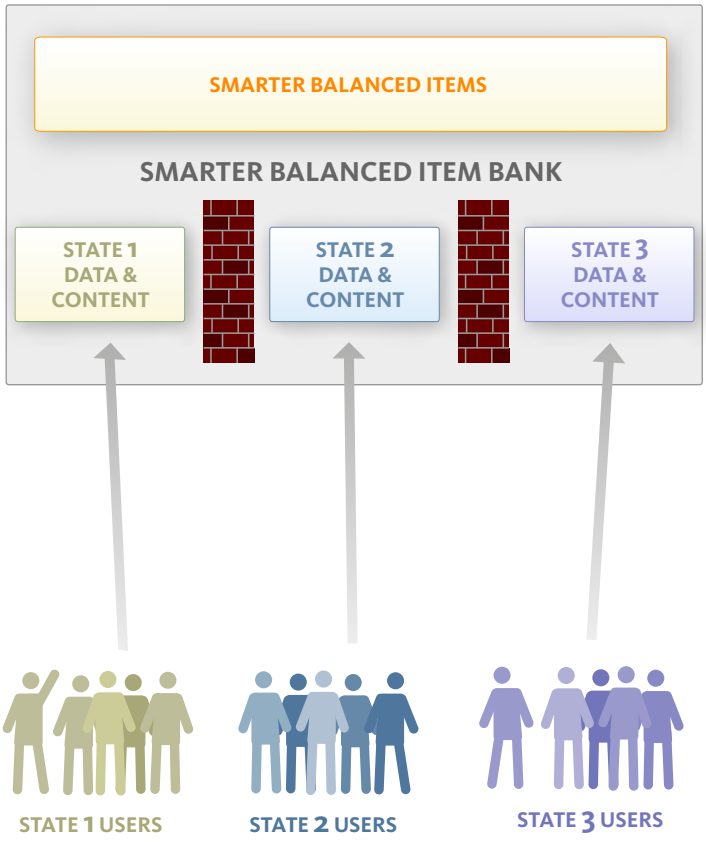

Figure 5.2 Mutitenancy

5.3. Scenarios

Following is a list of some possible scenarios. Please note that the following scenarios are for illustration purposes only; it is not an exhaustive list that shows all possible permutations.

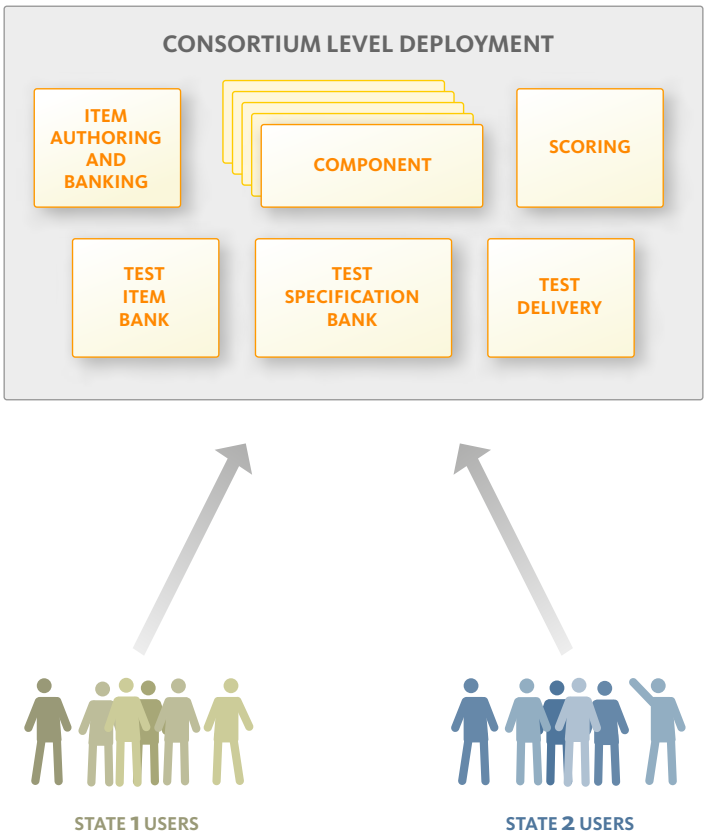

Figure 5.3.1 Deployment Scenario 1

Scenario 1 (Homogeneous): All components are deployed at the consortium level, and one or more states use the components without modification.

NOTE: These are possible deployment scenarios that the open-source platform has been designed to support. At this time, Smarter Balanced has decided to go with a deployment model most similar to Scenario 2. The Test Delivery and Scoring components, however will not be available as a service at the consortium level.

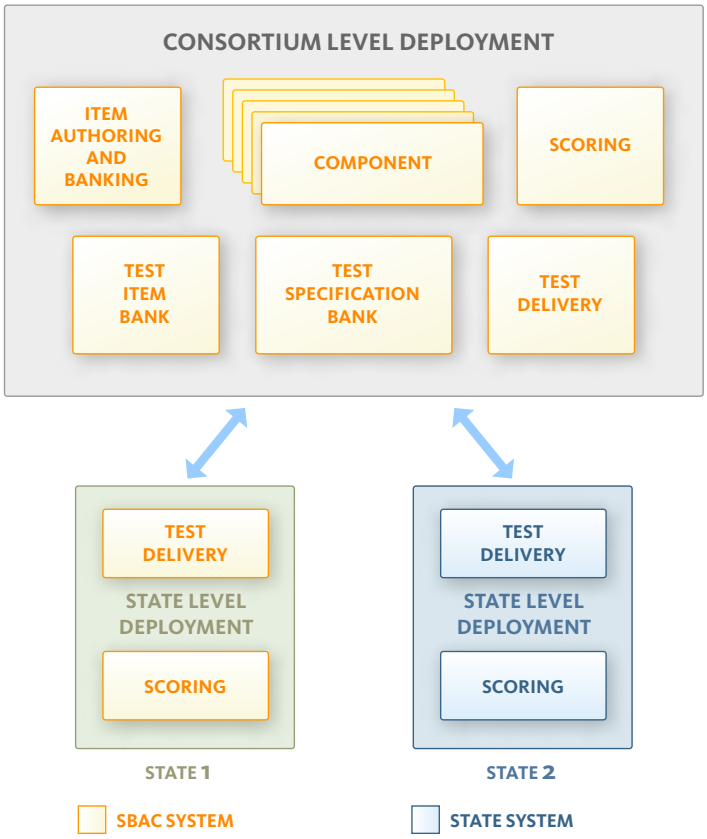

Figure 5.3.2 Deployment Scenario 2

Scenario 2 (Heterogeneous): Some states deploy the Smarter Balanced Test Delivery and Scoring system components at the state level for performance reasons, while others deploy the Test Delivery & Scoring system at the state level and deploy all other components at the consortium level.

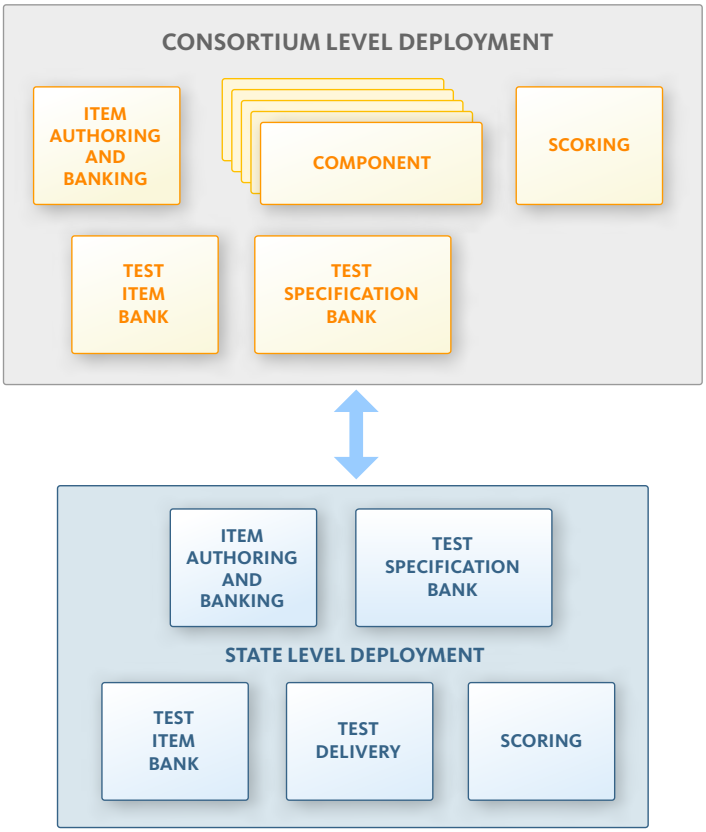

Figure 5.3.3 Deployment Scenario 3

Scenario 3 (Heterogeneous): States may add their own, state-specific items and test with the Smarter Balanced Item and Test authoring components, or use their own authoring tool to add items and tests.

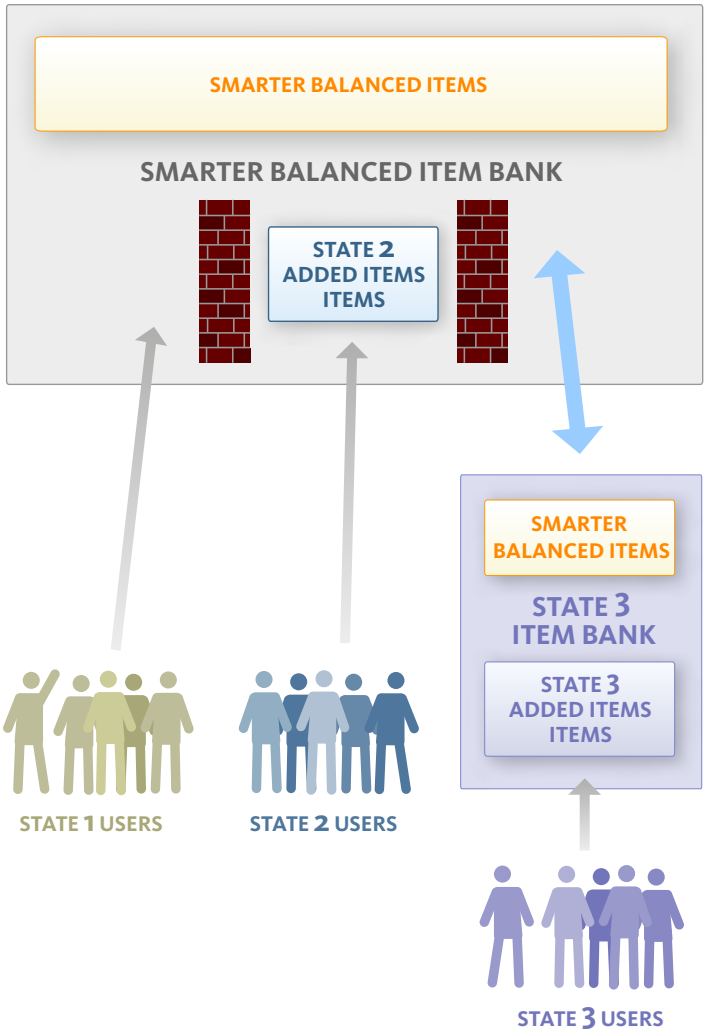

Figure 5.3.4 Deployment Scenario 4

This scenario shows some common models of how the Item Bank can be used. In this scenario:

- State 1 is using the Smarter Balanced Item Bank as is, and has access only to the Smarter Balanced Items.
- State 2 is using the Smarter Balanced Item Bank and have added its items, which are accessible only to State 2.
- State 3 is using its own Item Bank and has the Smarter Balanced items and its own items in it.

5.4. Deployment and Hosting Requirements

This list details the deployment and hosting requirements for the assessment system. They act as guiding principles to the design and implementation of the assessment system regardless of which way the system is to be deployed or hosted.

1. All components must be deployable at the consortium level and available for use.

2. Architecture and its implementation must support installations at the consortium level, state level, or the LEA level. It is conceivable that there might be some components that are used at a consortium or state level while some other components are closer to the end user.

3. Architecture and its implementation must support cloud-based hosting services, as well as traditional hosting options.

4. Components must support multitenancy at the state level and above. (The architecture does not need to support at the LEA level or lower.)

5. Components need to support partitioning.

6. When components are deployed at the lower levels of the hierarchy, they must still support all security requirements and other non-functional requirements, such as Item Security, Test Security, and Student Data Security.

7. When states or other entities on the hierarchy choose to deploy the Smarter Balanced components deployed at the state level or a lower level, Smarter Balanced items and Test Data Security cannot be compromised. Only approved personnel may have access to the items and data.

8. Smarter Balanced items can be exported only to Smarter Balanced approved state systems. Items cannot be exported to LEAs or other organizations lower in the hierarchy.

9. If applicable, the Smarter Balanced instance of the Item Bank must support state-specific items.

6. Data Architecture Definition

Assessment System Architecture and Technology Recommendations for the Smarter Balanced Assessment Consortium

Prepared in accordance with RFP SBAC 03 by Measured Progress.

6. Data Architecture Definition

Having described the ways that the assessment system could be deployed and hosted across the hierarchy, we now turn to details of the data architecture of the assessment system—in particular how its design and execution serve the intended purposes of the system.

This section will explore the various aspects of the assessment system in terms of data architecture, from creation, delivery, reporting and management. At this point, it is necessary to highlight some conventional schools of thought with regards to database architecture and design, for contextual purposes:

In the past, all data was assumed to be stored in a relational database. This caused developers to build applications using object relational mapping tools, which in turn led to an impedance mismatch between how an application stored and used its data. In order for an application to use a rich-object hierarchy, the relational database often represents the hierarchy using multiple tables. Unfortunately, this necessitates multiple joins to enable the application to instantiate the objects it needs, consuming many computing cycles to little benefit.

There has recently been a movement, however, driven by the explosion of Web 2.0 applications, to move away from using relational databases. This movement is referred to as "NoSQL" [http://en.wikipedia.org/wiki/NoSQL]. There are many different styles of NoSQL databases. This website, NoSQL-database.org [http://nosql-database.org/], lists many types of NoSQL databases, and the kind of datastore they represent.

Some of the more interesting databases that may be applicable (but are not limited) to the Smarter Balanced system are:

- Document-Oriented Database [http://en.wikipedia.org/wiki/Document-oriented_database]
- XML Database [http://en.wikipedia.org/wiki/Xml_database]
- Graph Database [http://en.wikipedia.org/wiki/Graph_database]
- Key Value Stores [http://en.wikipedia.org/wiki/Distributed_hash_table]

The NoSQL databases generally provide for better horizontal scaling than traditional RDBMS databases, and can be placed in multiple data centers, allowing the data to be synchronized or replicated with other nodes. Most of the NoSQL solutions are open-source, and have a great community of users supporting each other; this will make the solution inexpensive for schools and school districts. The NoSQL solutions are also better at running on commodity hardware, not requiring investments in special hardware.

Another noteworthy concept here is that the data from one component should not be directly exposed to another component. Drawing from the current best practices, this kind of information and data exchange should be accomplished by using APIs. This is a fundamental change from conventional concepts, and is discussed extensively in this section and throughout this document.

6.1. General Data Architecture Principles

The following list presents the guiding principles of the assessment system's data architecture. These principles apply to the creation, delivery, reporting, and management aspects of the assessment system.

1. Components should not directly access other components' data stores. A component that must make its data available to other components should implement services to expose the data for other components' use. This reduces dependencies on how the data is stored, allows components to evolve independently, and allows each component to store data in the format best suited for that component.

2. Use a storage mechanism that fits the intended use of the data.

3. Storage mechanisms must allow for multitenancy.

4. Each domain data element is owned by some component, and that component must be the source of truth for that data element. For example, the Test Authoring system owns the tests, and hence will be the source of truth for tests. It will generate all the new tests, and assign them keys that can be referenced by other systems or components. No other system or component should be able to create new tests or modify them. This will ensure that all tests conform to the proper standards and data rules.

5. Determine if a history of the domain object needs to be maintained, so that point-in-time data is maintained. For example, if students take the test "123", then the version of the test that they took must also be saved so that it can be referenced later. Alternatively, the version of the test taken should be saved (as in embedded) in the score or results.

6. All services and/or databases should not accept data to be stored that does not follow "minimum data needed" rules. For example, if creating a user requires, at a minimum, a username, password, and email, then the service should not accept anything less than such.

7. Each entity must have keys generated such that they are unique across the system, such as UUID.

8. When using relational databases, tables must be normalized to the third normal form unless there is a compelling reason not to do so.

For further reference, please see this article by Martin Fowler, Chief Scientist at ThoughtWorks: Polyglot Persistence [http://martinfowler.com/bliki/PolyglotPersistence.html?utm_source=feedburner&utm_medium=feed&utm_campaign=Feed%3A+PlanetTw+%28Planet+TW %29].

6.2. Assessment Creation and Management

The banking components in this group fit more into a document style of datastore. When querying the datastore, the usual intent is to return the full item, group of items, or test specification. If the datastore was relational, multiple joins would be necessary to return those objects. This is not to say that users do not need to query the document database for objects that have certain characteristics (i.e., "Give me all items that have a P value = x"), but that the result of those queries are always well-known document types (e.g., item, blueprint).

Another benefit of these styles of databases is that they do not enforce schema compliance. This allows each document to contain only the parts that pertain to that document; other documents can contain different parts. The document schemas are then able to grow and change over time, which simplifies versioning those documents. In a relational datastore, supporting this capability expands the number of tables necessary and/or creates table data sparseness. Further, when making a change to the structure of the object, usually the database

schema would need to change. With a document database, the database does not actually care what the structure is.

6.3. Assessment Delivery

A relational database may not be the most efficient means of storing items and the assets that support them. If items are processed properly before the test is delivered to the student, they will behave more like static web pages. This then allows the Test Delivery component to leverage the scaling capabilities of HTTP servers and content delivery networks. If the user-interface controller is browser based, there is no need for dynamic web page creation (e.g., JSP, ASP), so the only dynamic storage need during the test is the student test-taking session information (i.e., the response data necessary for the Adaptive Engines).

This server-side portion must deal only with storing responses, calling the scoring and adaptive engines, and returning the static URL of the next item. This simplifies the horizontal scaling needs of the Test Delivery component. It also allows the component to use a fire-and-forget resilient queuing of student responses, enabling a slower relational update process to place responses in a relational database (resilient queuing/guaranteed message queuing software like ActiveMQ, rabbitMQ, MQSeries, etc.).

The test session state could be stored using a distributed cache in order to facilitate horizontal scaling and durability (e.g., Ehcache [http://ehcache.org/], Memcached [http://memcached.org/]).

6.4. Assessment Reporting

This functionality set ideally suits the support capabilities of the relational database. The data warehouse must have SQL-based query capability. It will also be beneficial if the warehouse also supports Online Analytical Processing (OLAP) [http://en.wikipedia.org/wiki/Online_analytical_processing]. This will allow complicated data mining capability and the support of Pivot Tables [http://en.wikipedia.org/wiki/Pivot_table].

There are two standards that the warehouse should use for OLAP support:

- XML for Analysis (XMLA) [http://en.wikipedia.org/wiki/XML_for_Analysis]
- Multi-Dimensional Expressions (MDX) [http://en.wikipedia.org/wiki/MultiDimensional_eXpressions]

By supporting these standards, Smarter Balanced users will be able to choose from a range of reporting tools with differing capabilities.

7. Interoperability

Assessment System Architecture and Technology Recommendations for the Smarter Balanced Assessment Consortium

Prepared in accordance with RFP SBAC 03 by Measured Progress.

measured progress

7. Interoperability

7.1. Interoperability and Standards

Interoperability Intention

Smarter Balanced will build all the components in the architecture and provide clear interfaces into every component. This will enable states to add or substitute their own component/s (or vendor-supplied component/s). Beyond component-level replaceability, the system also provides integration into other systems, such as state data systems and Student Information Systems (SIS).

Interoperability Requirements

1. Any component that is built in accordance with the Smarter Balanced architecture must be replaceable, such that a state can substitute their own component(s).

2. Intercomponent communication must use current standards (e.g., SIF, APIP) where possible. In the event that a current standard does not cover the need, new extensions must be created.

3. Smarter Balanced architecture must plan for the communication to SIS using prevailing industry standards.

Process Flow Diagrams

This section identifies the points that demand interoperability. Please note the process flow diagrams below. In them, swim lanes depict the components involved in the flow. The points that require interoperability are the lines that cross from one swim lane to another.

Standards Alignment

The following standards should be followed when defining data elements:

- Assessment Interoperability Framework (AIF) https://ceds.ed.gov/aif.aspx
- Common Education Data Standards (CEDS) https://ceds.ed.gov/
- Schools Interoperability Framework (SIF) http://www.sifassociation.org
- Accessible Portable Item Protocol (APIP) http://www.imsglobal.org/APIP/

There is a strong possibility that the eventual assessment system will have to bridge significant gaps in the existing standards. These gaps will be identified during detailed application design (but in some cases, not until implementation). There should be no expectation that all of these gaps will be identified during the architecture definition phase.

Format Specifications

Detailed format specifications will be published on the SmarterApp website, along with the code assets and all other documentation associated with the open-source work of Smarter Balanced.

http://www.smarterapp.org

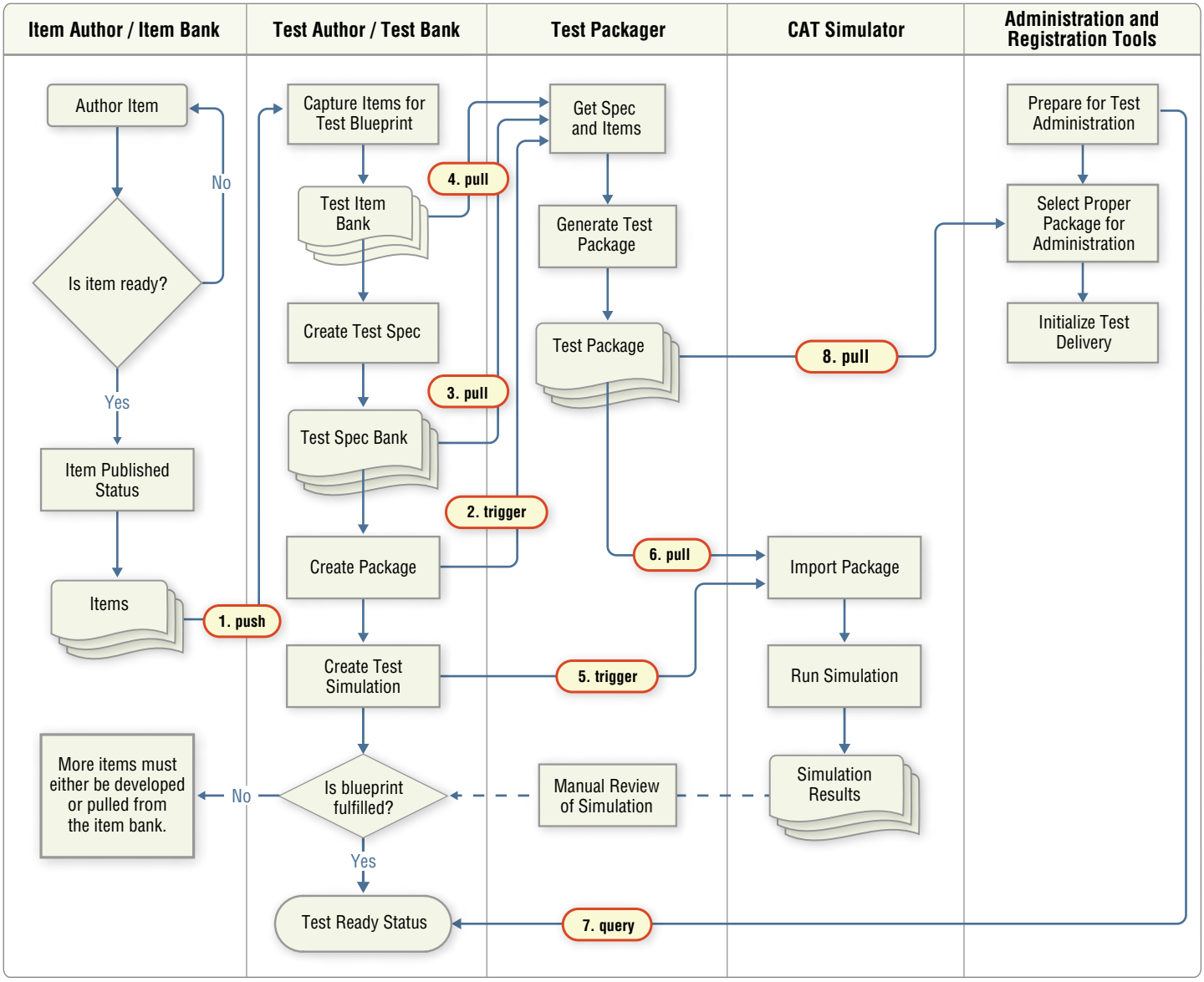

Figure 7.1.1 Item Authoring to Test Administration Flow of assessment content through the system.

Line#	Source Component	Target Component	Domain Objects(s) or Trigger	Suggested Standard
1	Item Authoring	Test Item Bank	Items (push)	APIP
2	Test Authoring	Test Packager	Trigger start packaging	RESTful API
3	Test Packager	Test Spec Bank	Test Specs	APIP
4	Test Packager	Test Item Bank	Items (pull)	APIP
5	Test Authoring	CAT Simulator	Trigger for adaptive simulation	RESTful API
6	CAT Simulator	Test Packager	Test Package (pull)	APIP
7	Administration and Registration Tools	Test Spec Bank	Query for available tests	RESTful API

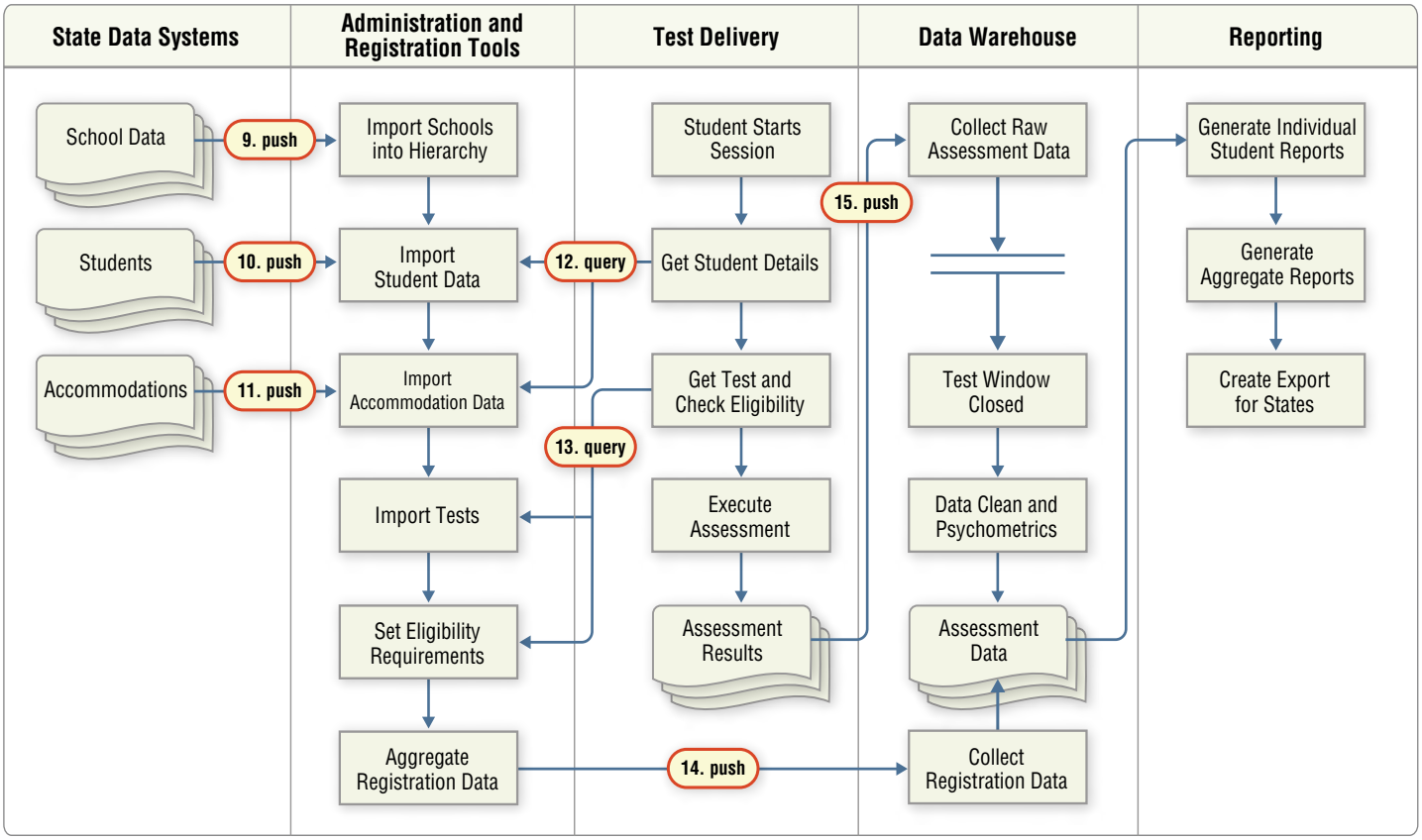

Figure 7.1.2 SIS to Reporting

Flow of student information and assessment data through the system.

Line#	Source Component	Target Component	Domain Objects(s) or Trigger	Suggested Standard
9	State Data Systems	Administration and Registration Tools	District & School Hierarchy	File elements must align to CEDS and ideally map to the SIF data structure.
10	State Data Systems	Administration and Registration Tools	Student data	File elements must align to CEDS and ideally map to the SIF data structure.
11	State Data Systems	Administration and Registration Tools	Accessibility profiles	APIP
12	Test Delivery	Administration and Registration Tools	Query student profile and accommodations	RESTful API
13	Test Delivery	Administration and Registration Tools	Query for available tests and confirm eligiblity requirements	RESTful API
14	Administration and Registration Tools	Data Warehouse	Registration data	File elements must align to CEDS and ideally map to the SIF data structure.

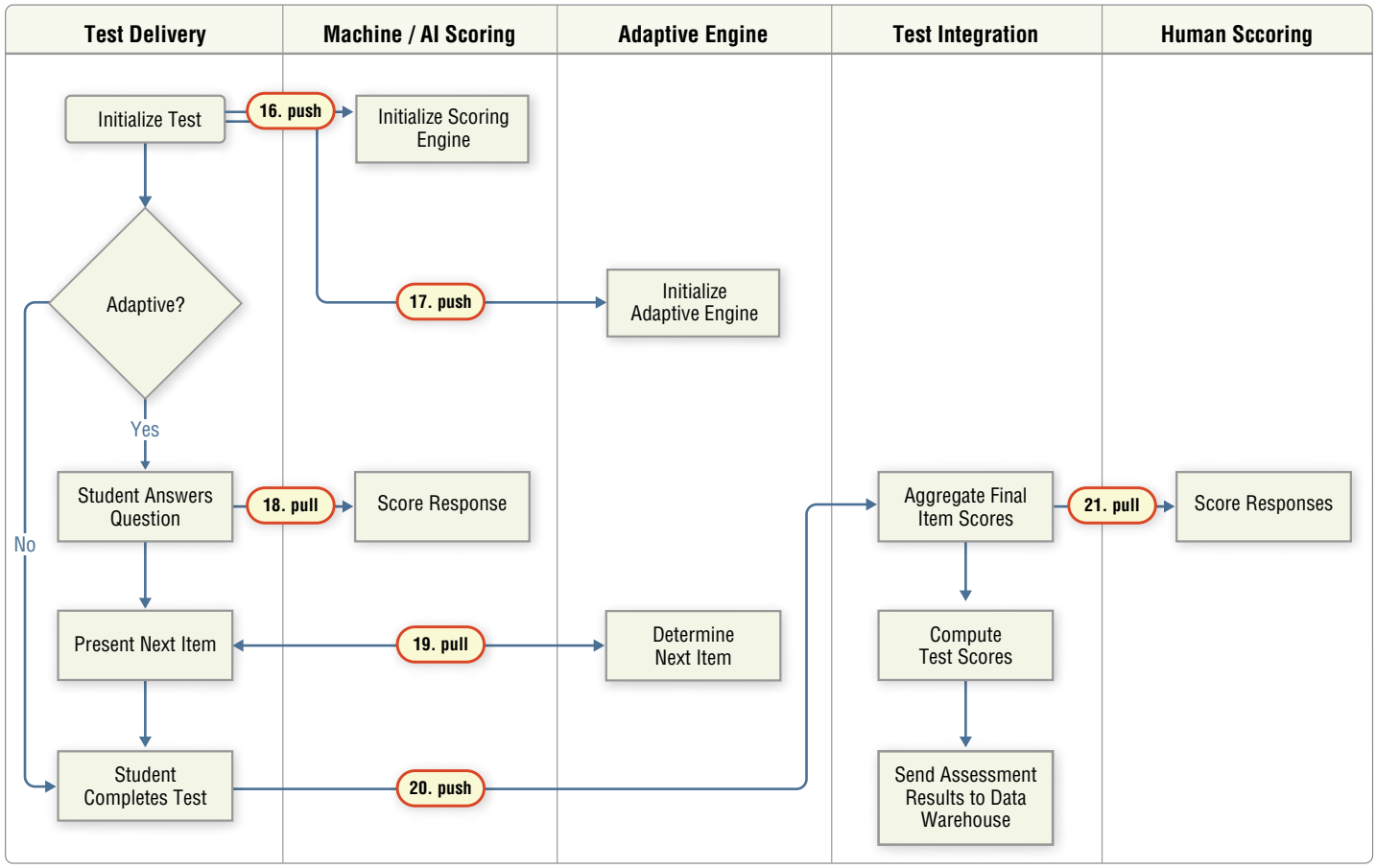

Figure 7.1.3 Test Delivery to Test Integration

The interaction of Test Delivery with AI Scoring, Adaptive Engine, and Human Scoring.

Line#	Source Component	Target Component	Domain Objects(s) or Trigger	Suggested Standard
16	Test Delivery	Machine / AI Scoring	Test Package with rubrics and scoring parameters	Standard needs to be created.
17	Test Delivery	Adaptive Engine	Test Package with adaptive specifications	Standard needs to be created.
18	Test Delivery	Machine / AI Scoring	Item responses and scores	SIF
19	Test Delivery	Adaptive Engine	Scored items and next item selection	Standard needs to be created.
20	Test Delivery	Test Integration	Partial assessment records	SIF
21	Test Integration	Human Scoring	All responses and scores	SIF

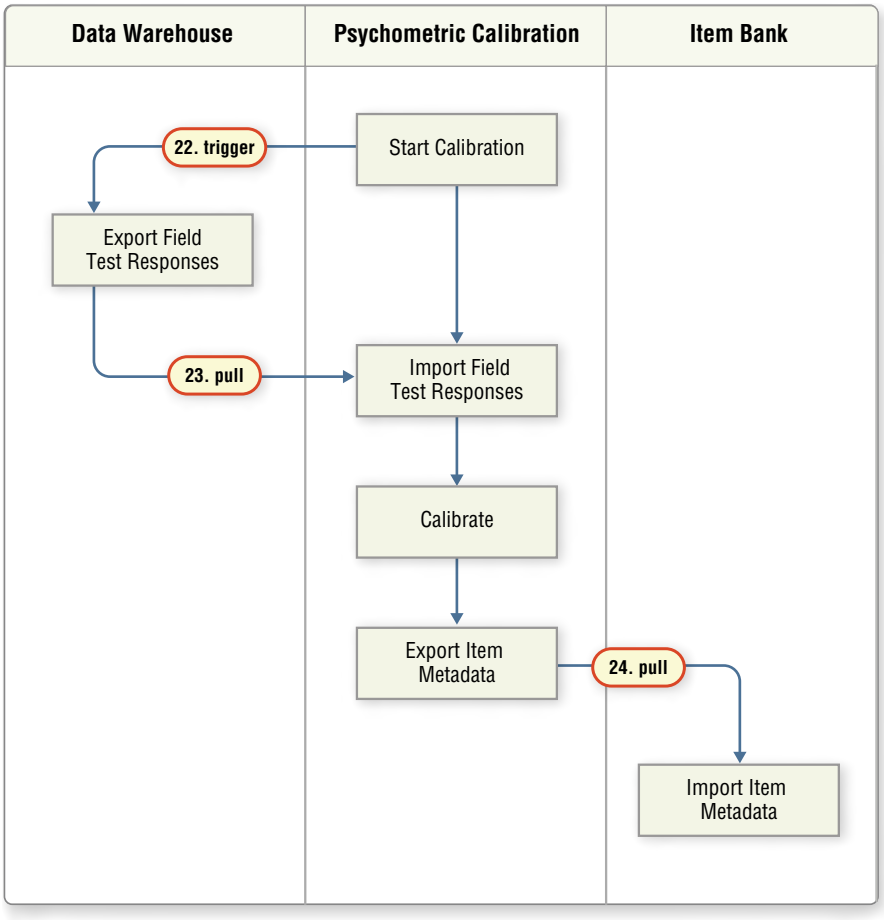

Figure 7.1.4 Data Warehouse to Item bank

Flow of psychometrician data from the pull of response data to the updating of item parameters in the Item Bank.

Line#	Source Component	Target Component	Domain Objects(s) or Trigger	Suggested Standard
22	Psychometric Calibration	Data Warehouse	Trigger field test responses report	SIF (where data standards exist) or CSV (Comma Separated Value)
23	Psychometric Calibration	Data Warehouse	Item response data	SIF (where data standards exist) or CSV (Comma Separated Value)
24	Psychometric Calibration	Item Bank	Item metadata	APIP (where data standards exist) or CSV (Comma Separated Value)

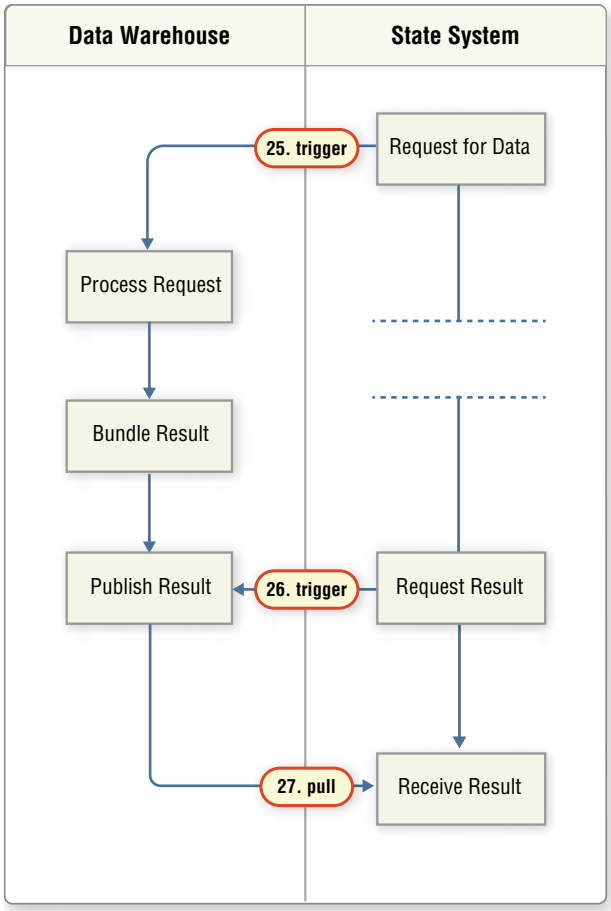

Figure 7.1.5 Data Warehouse to State system

Export of student assessment results from the Data Warehouse.

Line#	Source Component	Target Component	Domain Objects(s) or Trigger	Suggested Standard
25	State System(s)	Data Warehouse	Trigger request for data	RESTful API
26	State System(s)	Data Warehouse	Trigger request for data result	RESTful API
27	Data Warehouse	State System(s)	Assessment result data	SIF (where data standards exist) or CSV (Comma Separated Value)

7.2. Interoperability Matrix

Line#	Source Component	Target Component	Domain Objects(s) or Trigger	Suggested Standard
1	Item Authoring	Test Item Bank	Items (push)	APIP
2	Test Authoring	Test Packager	Trigger start packaging	RESTful API
3	Test Packager	Test Spec Bank	Test Specs	APIP
4	Test Packager	Test Item Bank	Items (pull)	APIP
5	Test Authoring	CAT Simulator	Trigger for adaptive simulation	RESTful API
6	CAT Simulator	Test Packager	Test Package (pull)	APIP
7	Administration and Registration Tools	Test Spec Bank	Query for available tests	RESTful API
8	Administration and Registration Tools	Test Packager	Test Package (pull)	APIP
9	State Data Systems	Administration and Registration Tools	District & School Hierarchy	File elements must align to CEDS and ideally map to the SIF data structure.
10	State Data Systems	Administration and Registration Tools	Student data	File elements must align to CEDS and ideally map to the SIF data structure.
11	State Data Systems	Administration and Registration Tools	Accessibility profiles	APIP
12	Test Delivery	Administration and Registration Tools	Query student profile and accommodations	RESTful API
13	Test Delivery	Administration and Registration Tools	Query for available tests and confirm eligiblity requirements	RESTful API
14	Administration and Registration Tools	Data Warehouse	Registration data	File elements must align to CEDS and ideally map to the SIF data structure.
15	Test Delivery	Data Warehouse	Assessment results	File elements must align to CEDS and ideally map to the SIF data structure.
16	Test Delivery	Machine / AI Scoring	Test Package with rubrics, and scoring parameters	Standard needs to be created.
17	Test Delivery	Adaptive Engine	Test Package with adaptive specifications	Standard needs to be created.

Line#	Source Component	Target Component	Domain Objects(s) or Trigger	Suggested Standard
18	Test Delivery	Machine / AI Scoring	Item responses and scores	SIF
19	Test Delivery	Adaptive Engine	Scored items and next item selection	Standard needs to be created.
20	Test Delivery	Test Integration	Partial assessment records	SIF
21	Test Integration	Human Scoring	All responses and scores	SIF
20	Adaptive Engine	Test Delivery	Next Item choice	Interoperability standard definition needs to be created.
21	Test Delivery	Scoring	All responses and scores	SIF
22	Psychometric Calibration	Data Warehouse	Trigger field test responses report	SIF (where data standards exist) or CSV (Comma Separated Value)
23	Psychometric Calibration	Data Warehouse	Item response data	SIF (where data standards exist) or CSV (Comma Separated Value)
24	Psychometric Calibration	Item Bank	Item metadata	APIP (where data standards exist) or CSV (Comma Separated Value)
25	State System(s)	Data Warehouse	Trigger request for data	RESTful API
26	State System(s)	Data Warehouse	Trigger request for data result	RESTful API
27	Data Warehouse	State System(s)	Student assessment result data	SIF (where data standards exist) or CSV (Comma Separated Value)

Bulk Import and Export

There is a need to import and export data between components. It is expected that components format the exported data to the standard defined for that data type.

- Items : APIP
- Item metadata : the defined standard when developed (most likely extensions to IMS QTI / APIP)
- Student info : SIF
- Student responses : SIF
- Scores : SIF
- Specs / blueprints : the defined standard when developed
- Test Package : the defined standard when developed
- Test Registration : SIF

Usually, import and export implies a filesystem transport between systems (i.e., System A exports to a file, and that file is moved by LAN, WAN, FTP, email, or USB drive to import into System B). This can be supported by implementing a system based on Plugin Binary Transport (PBT). It is also possible to use other transport technologies. It is important that these bulk files be secured, and it is also suggested to use Pretty Good Privacy (PGP) [http://en.wikipedia.org/wiki/Pretty_Good_Privacy] encryptions on these bulk files.

8. Non-Functional Requirement Constraints

Assessment System Architecture and Technology Recommendations for the Smarter Balanced Assessment Consortium

Prepared in accordance with RFP SBAC 03 by Measured Progress.

8. Non-Functional Requirement Constraints

8.1. Open Licensing

This section describes licensing, requirements, and recommendations. The actual licenses can be found on SmarterApp.org [http://www.smarterapp.org]

Types of Open Licensing

Open Access(OA)

Open access licensing allows licensing access through the internet without any restriction. This type of licensing can be used for artifacts produced from the architecture phase and other creative content. For a more extensive definition, see the Wikipedia definition of Open Access. Open access [http://en.wikipedia.org/wiki/Open_access_(publishing)].

Open-source Software(OSS)

Open-source software [http://en.wikipedia.org/wiki/Open-source_software] licensing can be used for all software components developed as part of Smarter Balanced architecture.

Open Licensing Requirements

1. All artifacts describing the architecture must be under an Open Access license.

2. All software artifacts produced must be under an OSS license. If a vendor is selling a proprietary solution, that solution must be made available with an OSS license.

3. Where available, OSS components must be used for building the software systems. This includes, but is not limited to:

   - Operating systems
   - Tools used for authoring, building, and testing the software components
   - Database software
   - Messaging systems

Recommendations

Not all software licenses are compatible. This article [http://en.wikipedia.org/wiki/List_of_software_licenses] lists some of the caveats of software licensing. In particular, GPL [http://en.wikipedia.org/wiki/GNU_General_Public_License] licensing has known incompatibilities with other licenses like Apache [http://en.wikipedia.org/wiki/Apache_License]. This article [http://en.wikipedia.org/wiki/Comparison_of_free_software_licences] compares various licenses.

1. In period B, during construction and/or customization of components, use a consortium-level closed-source license, but allow components to view each other's source for integration and troubleshooting purposes. When a component is ready for production, decide on the most appropriate OSS licensing.

2. Use an open-access license such as Creative Commons [http://creativecommons.org/] for the artifacts that describe the architecture.

Future OSS Project Governance

When Smarter Balanced moves from closed-source to an open-source model, it becomes necessary to formally manage the maintenance and enhancement of the produced software. Therefore, each component must have a management process or structure. Although numerous processes have been used in the open-source community, the following structure tends to surface:

Project Management

A single person or small group manages the project. They are responsible for deciding priorities and determining additions, and for removing people who are given commit capability to the source-code repository. They also define timelines for releases of the project.

Committers
Project management approves these people to make changes and additions to the source. They usually have shown a keen grasp of the domain and understanding of the software, and have followed good programming practices in the view of project management.

Contributor
This would be a developer, motivated by his or her own needs and wishes, who makes enhancements to the source code, and contributes the changes to the project for possible inclusion. Contributors who make many contributions that project management accepts may become committers.

User
A user is an individual who uses the software, and contributes to the project by submitting bug reports, beta-testing the software, or suggesting additional features.

This type of structure is usually sufficient for single projects that work on a single component, but the Smarter Balanced system is a series of many separate components that work not only independently but also in combination with the other components of the system. A controlling structure of project governance is necessary to make sure that individual components don't drift from the goal of working with one another. A group should be tasked with coordinating subprojects to ensure that those efforts interoperate correctly, follow constancy in architecture, follow standard development practices, and grow together with a constant vision.

It is also possible that one or more existing open-source organizations may sponsor these projects or form a new foundation to support the continuation of these projects. Such organizations include:

- The Apache Software Foundation [http://apache.org/foundation/]
- Eclipse Foundations [http://www.eclipse.org]

8.2. High-Availability and Scalability

Areas of high availability, scalability, and performance to be considered are:

- System performance as perceived by a single user
- Scalability with large volumes of concurrent users
- Resiliency and recoverability of the system
- System data capacity with large volumes of data
- High availability of systems

System Performance

1. While no specific requirements were identified during the initial planning, test delivery systems must be the most resilient. These systems are highly susceptible to burst modes of operations, with large numbers of concurrent users accessing the system. The combination of adaptive testing and AI scoring will have a significant impact on system performance, and sufficient architecture planning is needed to support this.

2. When tests are delivered, a significant amount of student data must be collected (e.g., item, answer, score, comments). The architecture must consider data volumes and purging strategies.

3. Network bandwidth and reliability must be considered, and the architecture must make appropriate recommendations for critical components.

Requirements that need to be identified in the application architecture for each component are:

- Total number of users for each component
- Minimum number of concurrent users that each component must support individually
- Maximum number of concurrent users that each component must support individually
- Amount of data that each component will store

Principles

- Components should scale horizontally. The use of NoSQL technologies and distributed caches will better enable this in the data storage area.
- When components log events or send BI events, it must do so a in fire-and-forget fashion, such that there are no delays to the critical path functions.

8.3. Accessibility

Items must be accessible to fulfill the requirements of special-needs students. The system must move and process items in ways that conform to accessibility standards. The Item Authoring and Test Delivery components must allow for creating and rendering accessible items, respectively. Accessibility must be taken into consideration also for special-needs users other than students. For example, alternative ways are necessary to render colored pie charts for those who are color blind.

Profiles for accessibility needs may be stored in a SIS system. The Test Delivery component uses this and other student data, and therefore the delivery system must be able to retrieve the profile from outside systems.

The Consortium is also considering other types of test delivery devices (such as Braille).

8.4. Technology

Although the initial planning did not raise any technology constraints, LEAs typically have limited budgets and may have outdated technology. It is difficult to innovate using older technologies. Because the system will not be in place until 2014, assuming a target technology of 2012-era technologies may be reasonable.

9. Security

Assessment System Architecture and Technology Recommendations for the Smarter Balanced Assessment Consortium

Prepared in accordance with RFP SBAC 03 by Measured Progress.

9. Security

Multiple types of security are required for the system.

9.1. Component-to-Component

Each architecture component must be capable of connecting to other system components, and to allow authorized components to use its services. The communication channel between components must ensure that those components cannot be surreptitiously monitored or spoofed. Some techniques that need to be considered are:

- IP filtering
- SSL
- PGP (Pretty Good Privacy)
- SAML

9.2. User Authentication and Authorization

The system must be able to confirm that users are who they say they are, and to confirm that they are authorized to use the features and functionalities of the components that their authorization allows. This user authentication and authorization must be standardized across components to allow seamless access, or users and system administrators will have difficulty maintaining the components. User authentication and authorization must also be configured in each system to prevent mismatches among components and difficulties in troubleshooting.

This necessitates a Single Sign-on (SSO) solution and component support for the chosen solution. Among the open-source implementations that will be considered is OpenAM [http://en.wikipedia.org/wiki/OpenAM].

There are typically two types of authorization:

Role-based

Each end user is given a role or set of roles, and each component permits specific roles to use only specific features of the component. This requires a finite list of roles to which all components code. The downside of this is that all roles must be defined before or during implementation; identifying new roles can require code changes across all components.

Permissions-based

In this type of authorization, components define permissions (such as view item, edit item, create test, etc.). These permissions are associated with groups, and users are placed within these groups. This method allows the creation of an unlimited number of groups without component code changes, but requires that each component poll an external system for the permissions associated with a user.

Student authentication in the Test Delivery component is a specialized mechanism that requires a verification that each student is who the system thinks he or she is.

9.3. Item-level Security

Because item exposure is critical, item security must consider the following:

- How are items stored, and who has access to the data? It is critical that only authorized users, with the correct level of privileges, are able to operate on the items.
- How are items transmitted to other systems and how are those systems authenticated and authorized?
- Summative items must be given the highest level of security.

9.4. Student Data Security

Student data security must comply with:

- FERPA and COPPA.
- State laws regarding data breaches. For example, California has enacted the "California Data Breach Notification Law" (SB 24 – Sep 2012), which requires companies, institutions, and government agencies to provide key details in data-breach notification letters, and to notify the state attorney general of the data breach.

9.5. Data at Rest

Any confidential/sensitive data that is at rest (e.g., password field in the database, export file, or SSN in an XML file) must be encrypted so that it cannot be mistakenly or surreptitiously viewed.

10. Technical Architecture Definition

Assessment System Architecture and Technology Recommendations for the Smarter Balanced Assessment Consortium

Prepared in accordance with RFP SBAC 03 by Measured Progress.

10. Technical Architecture Definition

10.1. Server Hardware and Software Requirements

Software Requirements: Platform

There are many programming languages and platforms in use today. The two most popular platforms are Oracle Java Platform (JVM) and Microsoft's .NET Platform (CLR). While C and C++ are still frequently used, the JVM and CLR platforms support for multiple programming languages and hosting multiple target platforms make them the logical choices for this application. They allow programming an application in multiple languages while running on the same platform, which enables developers to use the most efficient language when solving specific issues. For example, Scala [http://www.scala-lang.org] could be used for parts of a component that require concurrent processing. Clojure [http://clojure.org], however, is good for concurrency, and also works well for functional style programming, which supports more mathematical features.

Many consider .NET to be available only for Microsoft platforms, but the open-source project Mono [http://www.mono-project.com/Main_Page] allows .NET technologies to run on Linux and OSX operating systems. Commercial versions of Mono are also available for Apple's iOS and Google's Android. Both platforms are suitable for components of the Smarter Balanced system. Some components can be built in one platform, and others can be built in the other.

Mono does extend the platforms on which the application may be deployed, but it does not support all features of the .NET 4.0 platform, may introduce increased support costs, and lags behind .NET updates and new features. Until recently, Mono was supported by Novell. Xamarin, a company founded in May of 2011 by some of the originating Mono developers, now supports the product.

Sun introduced JVM in the mid 1990s, and Oracle acquired it in April of 2009 when they bought Sun. Most platforms other than iOS support it, and it has been a dominant enterprise platform since the beginning of the millennium. Most cloud technologies also support JVM well. .NET is still competitive, although it is not as broadly supported.

JVM is the preferred platform for component development, but this does not necessarily mean that the Java programming language is preferred. Instead, using JVM will allow the use of many languages that interact with each other. Using JVM allows innovation and deployment to non-Windows operating systems, and enables the output to be deployed to many servers. In addition, Oracle will continue to support these operating systems for the foreseeable future.

Software Requirements: Browser

Browser technology advances at a rapid pace. Recently, they have begun to provide standards support, reducing the incompatibilities between them. Support of older browsers (especially Internet Explorer 6) requires custom code and additional quality assurance testing.

Supporting earlier browser versions may become necessary as component development begins. The Progressive Enhancement development design technique [http://en.wikipedia.org/wiki/Progressive_enhancement] will allow adding features supported by newer browsers without inhibiting the older browsers' ability to use their basic features.

Toward that end, the following draft specifications serve as a starting point for the discussion regarding system requirements. Smarter Balanced will consult with member states and districts about how to balance the advantages of new technologies against the pragmatic budget and logistical issues that face schools and districts. After further analysis of the results from the IT Readiness Tool, Smarter Balanced will make final recommendations for hardware, browser versions, and operating systems.

NOTE: For the most up-to-date device and browser requirements, refer to the Technology Strategy Framework and Testing Device Requirements. [http://www.smarterbalanced.org/smarter-balanced-assessments/technology/]

Hardware Requirements

If the components use a JVM (or .NET / Mono), then specific hardware and operating systems are not required (e.g., Solaris running on Sparc hardware). Any hardware or operating system that runs the JVM is possible. A component's application architecture will determine that component's hardware requirements, but the following guidelines will enable flexible choices.

A component's server parts should be as stateless as possible. Where state is necessary, the server should use a distributed cache (e.g., Ehcache [http://ehcache.org/], Memcached [http://memcached.org/], etc.). This will facilitate horizontal scalability.

When developing parts of the component that can take advantage of concurrency, use the Actor Model [http://en.wikipedia.org/wiki/Actor_model], so that the component can make more effective use of multicore server hardware.

Hosting environment and Recommendations

The Smarter Balanced assessment system needs to support many different hardware and network topologies. The recommended architectural direction will enable this capability. Because of the requirement for ultimate flexibility, this limits the ability to predetermine the precise hardware and network requirements. Each components' application architecture will be better able to determine its physical needs once the requirements are fully fleshed out, but we can give a logical hosting representation toward which each component can develop.

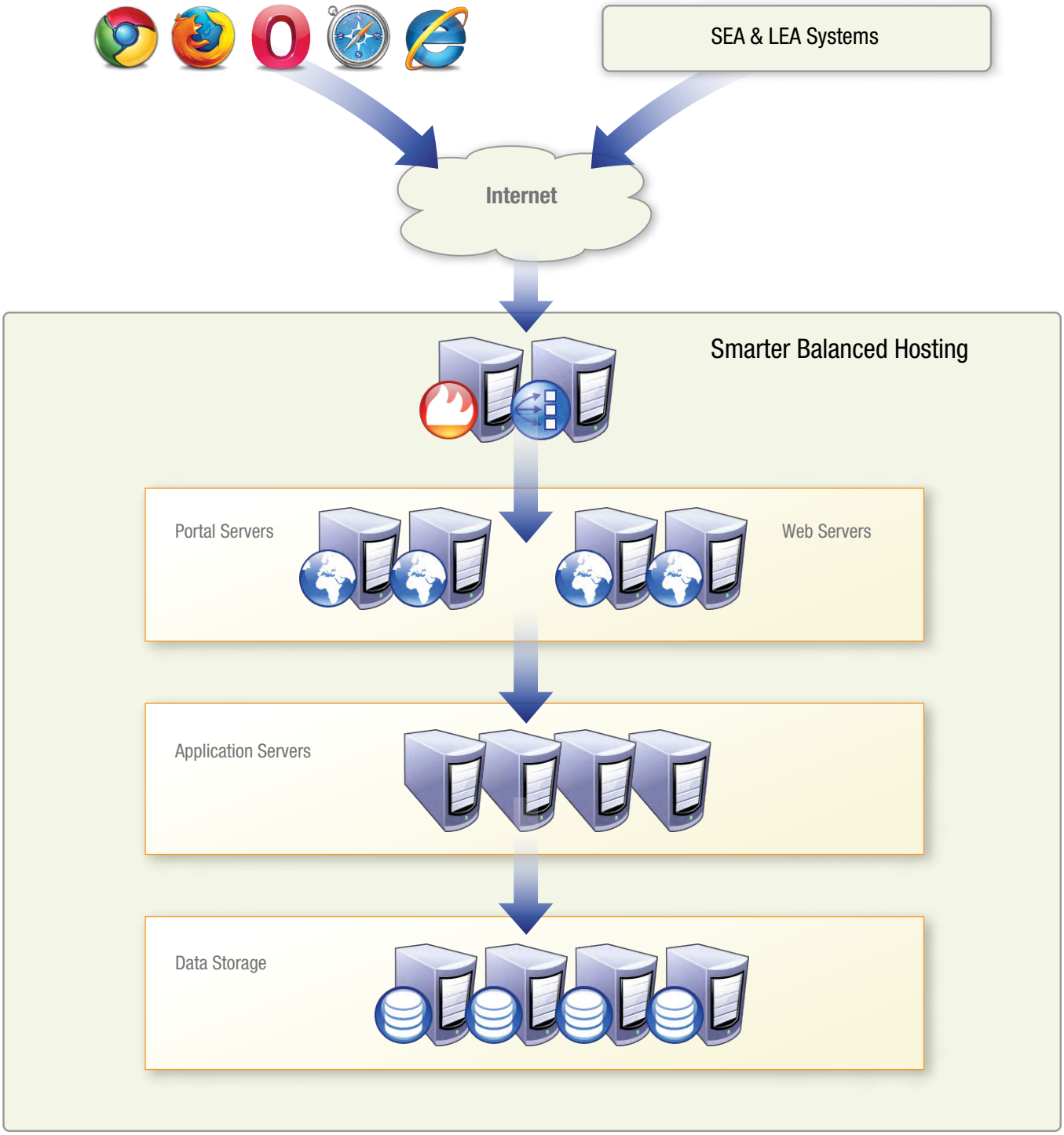

Figure 10.1.1 Logical Hosting Environment

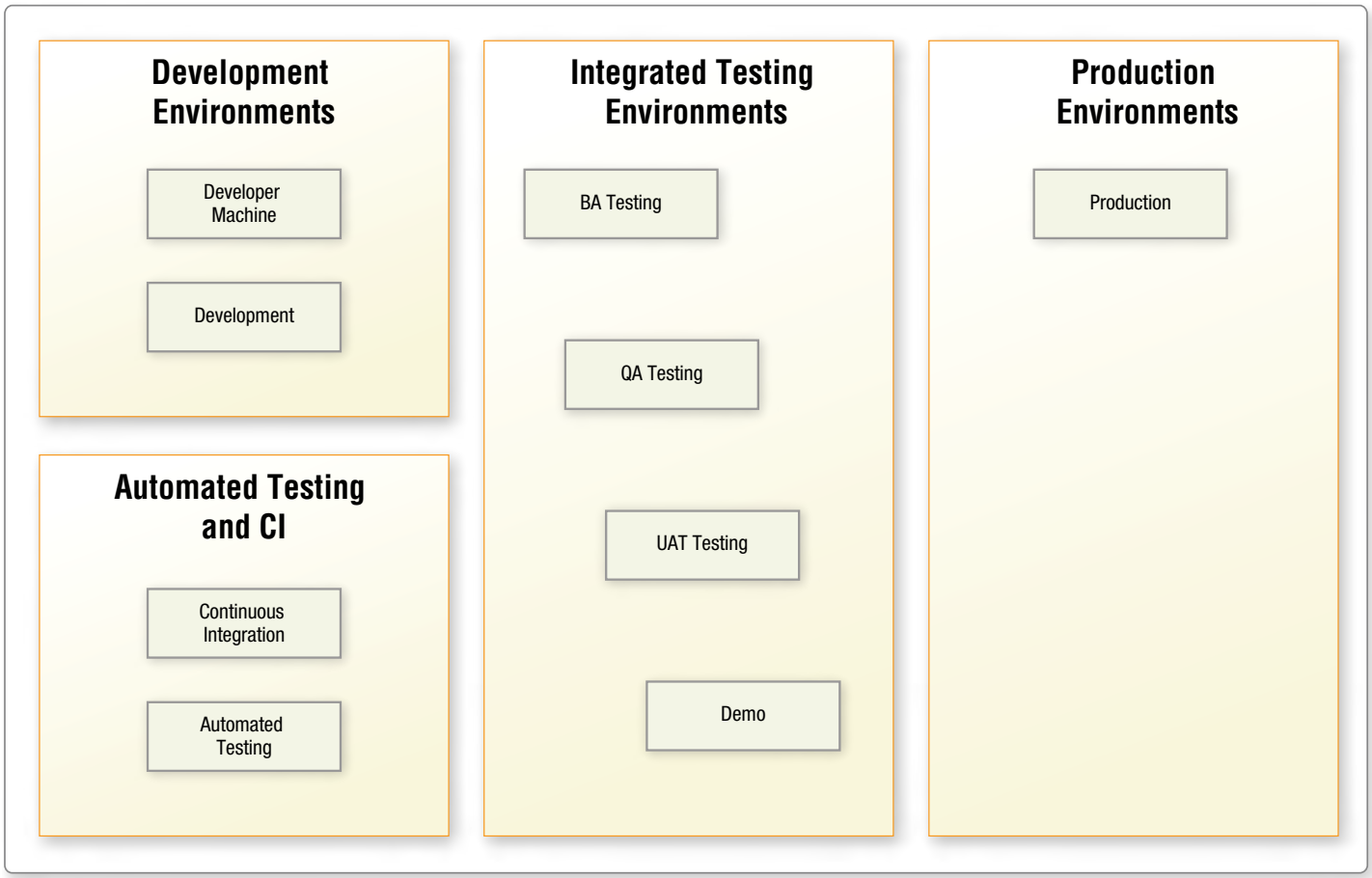

Figure 10.1.2 Development Environments

When a component has this style of hosting application architecture, it will be flexible enough for deployment to multiple hosting solutions.

Component Server Matrix

Component	Minimum Component Server Count	Minimum Data Server Count
Item Authoring / Item Bank	2	2
Test Authoring / Test Item & Spec Bank	2	2
Administration / Registration	2	2
Test Delivery	2-N (based on max concurrent usage expectation)	2-N
Scoring	2-N (based on max concurrent usage expectation)	2-N
Data Warehouse	2-N (based on max concurrent usage expectation)	2-N
Reporting	2	0 (Uses Data Warehouse store)
Portal	2	(Depends on application architecture. May be able to share data store of SSO, et al.)
SSO / Permissions / Program Management / User Preferences / Identifier Management	2	2
Monitoring & Alerting	2	(Depends on application architecture)
Digital Library	2-N	2-N

The rationale behind the matrix is that each component server and data server will need to have a minimum of 2 instances, in order to maintain the up-time expected. Meeting the above minimum

Recommendation for Virtual Machine (VM) based Hosting

System VM [http://en.wikipedia.org/wiki/Virtual_machine] / Virtual Private Server (VPS) [http://en.wikipedia.org/wiki/Virtual_private_server] based hosting will allow the Smarter Balanced assessment system to bypass the usual hardware procurement needs. Estimation of hardware and network requirements before significant progress in software development is usually inaccurate, leading to an under- or overestimation of needs. By VM hosting through a hosting provider, the assessment system will be able to procure the required server and data storage in a just-in-time manner. VM providers also usually have a higher-speed network backbone to support interserver communications. They can also support the hardware management and network requirements.

The following are some VM hosting providers to consider:

Rackspace [http://www.rackspace.com/cloud]

Amazon [http://aws.amazon.com/ec2]

10.2. Requirements and Approach for Database, Data Storage, and Archiving

By following the recommended architecture, each component can have individualized storage and archiving requirements. It is expected that the application architecture of those components will need to define those requirements. The following is a list of principles that should be observed:

- Data storage needs to be point-in-time recoverable. The time resolution is dependent on the criticality of the respective data object.
- Student assessment responses must never be lost. If a student has submitted an answer to an item and is presented with another item or test / section completion page, the system must be able to recover all responses including that response.
- Item / test authoring requirements should be based on the Smarter Balanced policy. Smarter Balanced must define what is an acceptable loss in case of system failure (e.g., one day, one hour, fifteen minutes, etc.). It should be noted that the shorter the time recovery point, the higher the development and support costs.
- Student responses and other Data Warehouse data needs to be kept for longitudinal use, and Smarter Balanced needs to define the retention lengths for this data.
- The delivery / warehouse components should guarantee delivery of Item Metadata to the Item Bank. The Item Bank should be able to accept resent metadata, and be able to gracefully handle redundant data.
- Components should be able to seamlessly recover from single data-node failures. When this occurs, components should be able to switch to other data nodes. This means that data storage for a component must have a minimum of two nodes to support single node failure.
- Smarter Balanced policies must be explicit about archiving lengths for specific data objects. Some of these policies may be driven by state and federal law.

Master Data Management

[http://en.wikipedia.org/wiki/Master_data_management]

It is the responsibility of the Program Management component to manage and maintain any data and reference data that is needed across components. This is to be considered as the definitive source for those data. Other components are expected to retrieve those data from the Program Management component. They may store their own copies of the data, but are responsible for updating from the data in Program Management when their copy becomes stale. The components should access the data through a REST API [http://en.wikipedia.org/wiki/Representational_state_transfer] provided by the Program Management component.

The Smarter Balanced architecture segregates data by where it is modified. The general rule is that only the component that owns the data may modify the data. For example: Item Bank owns all Item-related data, and therefore is the only component that may update this data. Other components consume parts of this data, but may never update it. It is recommended that a simple custom solution be used instead of a commercial MDM product, since the Smarter Balanced requirements do not demand a sophisticated system with features like "single version of truth" [http://en.wikipedia.org/wiki/Single_version_of_the_truth] and data governance.

10.3. Systems Management and Monitoring Requirements

All components should use a logging framework that is configurable outside of the component. This will allow components to write log and tracing information in a consistent and configurable way. Here are the suggested tools:

- JVM - log4j [http://logging.apache.org/log4j],slf4j [http://www.slf4j.org]
- .NET - Log4Net [http://logging.apache.org/log-4net]

For components built on the JVM, the component should use Java Management Extensions (JMX) [http://www.oracle.com/technetwork/java/javase/tech/javamanagement-140525.html]. Applications can expose information about performance, load, and other information through a standard interface. Many management solutions support JMX through direct support or through JMX to Simple Network Management Protocol (SNMP) [http://en.wikipedia.org/wiki/Simple_Network_Management_Protocol] adapters.

Similar to JMX, components on the Windows .NET platform should implement Windows Management Instrumentation (WMI) [http://msdn.microsoft.com/en-us/library/windows/desktop/aa394582(v=vs.85).aspx]. It performs the same capabilities for Windows components, and is built into the operating system. Windows itself uses this protocol, so all tools capable of monitoring Windows should be able to monitor the components.

Cloud vendors usually offer monitoring capabilities to their solutions. By following JMX / WMI and SNMP standards while implementing components, Smarter Balanced will be able to choose management and monitoring solutions without being tied to a specific vendor.

Possible vendors include, but are not limited to:

- Nagios [http://www.nagios.org]
  openNMS [http://www.opennms.org/]
  Hyperic [http://www.hyperic.com]
- CA Technologies
- HP Openview
- IBM Tivoli [http://www-01.ibm.com/software/tivoli/solutions]

10.4 System Management Categories

High Availability

Components that fit in this category need to be highly available and redundant. They are:

- SSO
- Test Delivery
- User Preferences
- Permissions
- Identity Management
- Portal
- Monitoring and Alerting

These components need to expose information concerning the status of the component (e.g., Test Delivery component needs to expose the number of connected students). These components also must be monitored for preventative issues. The machine or VM that they run on must be monitored for low-memory issues, disk-full issues, processor overloading issues, and exceptions. These must cause alerts in the system management software, notifying support personnel of possible issues.

Medium Availability

These are all components that are not in the above list. Components in this category must be available, but are not as time critical, and do not need to be as redundant.

Although these components must be available, the criticality of their up-time has less impact on the assessment system processes. These components can expose information for the management system to monitor, but alerting could be reduced. The high availability / redundancy needs are reduced; this could in turn reduce cost.

NOTE: The management data that each component needs to expose will need to be defined at the application architecture level, as this is the point at which critical capabilities will be fleshed out. It is at this point that the alerting and operational procedures to mitigate the issues should be determined.

10.5. Middleware and Integration Software Requirements

This section details the main integration patterns and technology recommendations for messaging, communication, and data transfer to data warehouses.

Recommendations relied on the following principles:

1. Favor lightweight integration and frameworks over centralized hub-and-spoke models or messaging systems. These are easier to test, integrate, and have very low requirements for hardware and software.

2. Resist adding business logic in centralized service buses, since they are harder to test and troubleshoot.

3. Favor lightweight RESTful services over integrating at the database level, which stifles emergent changes to the database.

Publish and Subscribe Using ATOM Feeds

ATOM [http://en.wikipedia.org/wiki/Atom_(standard)] feeds are a lightweight, XML-based syndication format, and use HTTP(S) for transport, which is secure and reliable. This is best used in scenarios where an application wishes to share its data with other applications, and instead of using queuing systems, it publishes an ATOM feed. The recent book, *REST in Practice* [http://restinpractice.com/book/] has some great examples of using ATOM for this purpose.

This pattern should be used when data is pushed to a data warehouse or other areas where data is to be published. The interoperability matrix (section 8.2) enumerates the areas where this recommendation is relevant. The image below shows this pattern at work in the scenario of the Test Delivery system and other components publishing data to the Data Warehouse.

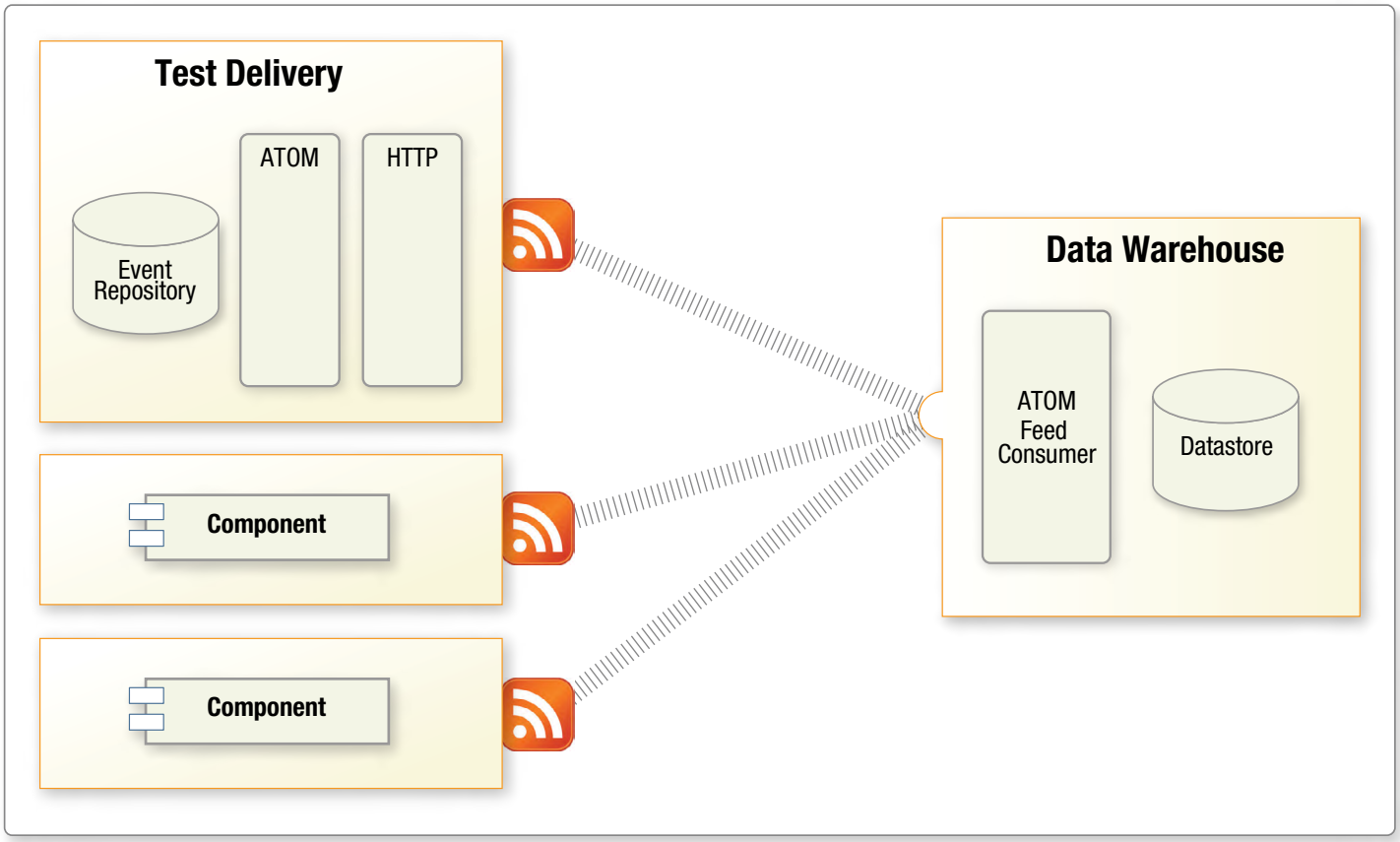

Figure 10.4.1 Publish and Subscribe using ATOM feeds

Point-to-Point Communication Using RESTful Services

REST [http://en.wikipedia.org/wiki/Representational_state_transfer] is recommended for use where point-to-point communication is needed between components, either in a fire-and-forget mode or in a request-response mode. REST uses HTTP(S) for transport, and message formats can use XML, JSON, and the standard HTTP methods. The aforementioned book [http://restinpractice.com/book/] provides an excellent deep dive into the subject.

The image below shows this pattern at work in the scenario of the Test Authoring system querying the Test Item Bank component for available items matching a specification.

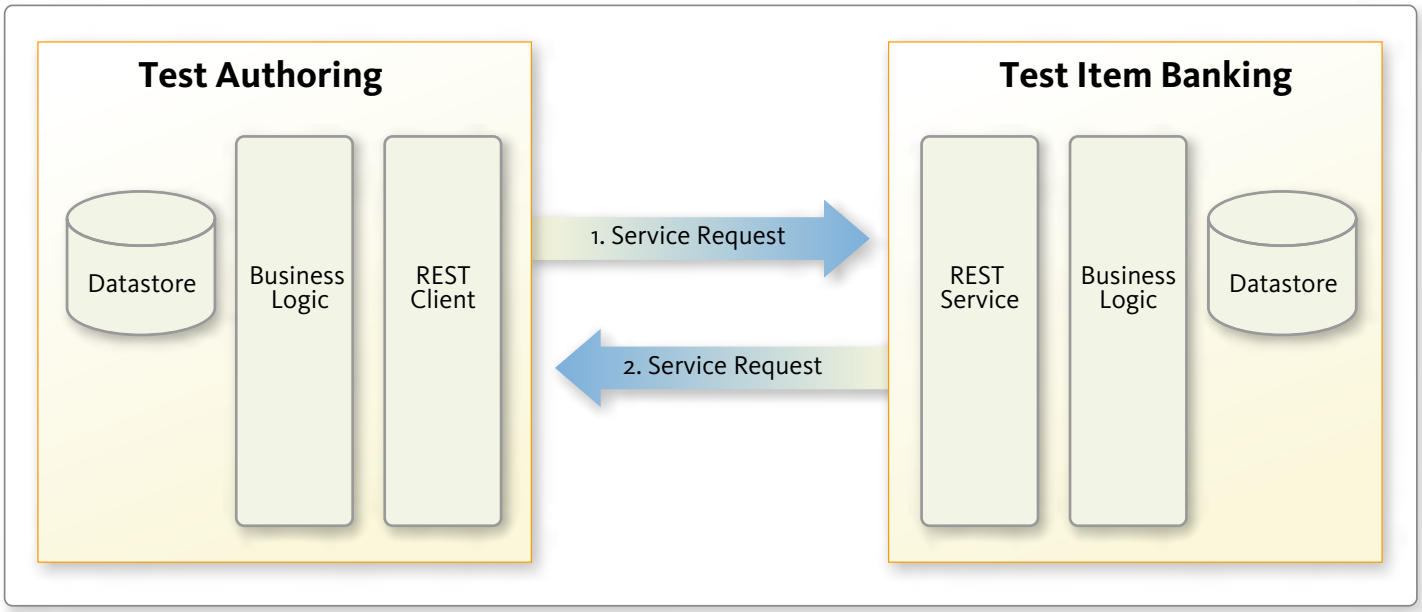

Figure 10.4.2 Point-to-point Communication using REST

10.6. Security Requirements and Approach for Applications, Data, and End-user Access

Also see Security. End-user access should be controlled using a Single Sign-on solution. This solution should support OAuth [http://oauth.net]. Users enter all components through the Portal component. If a component determines that the user is not authenticated, the component should be redirected to the SSO provider.

All web-based traffic should use Secure Sockets Layer (SSL) [http://en.wikipedia.org/wiki/SSL]. Any intercomponent network communication should also use SSL. Such communication should use nonstandard ports, and be firewalled to accept only connections from defined, static IP addresses. Any data that is exported to a file needs to be encrypted. It is recommended to use Pretty Good Privacy (PGP) [http://en.wikipedia.org/wiki/Pretty_Good_Privacy] to do so.

11. Application Development Model

Assessment System Architecture and Technology Recommendations for the Smarter Balanced Assessment Consortium

Prepared in accordance with RFP SBAC 03 by Measured Progress.

11. Application Development Model

11.1. Objective

The Application Development Model outlined below supports Smarter Balanced system development by multiple vendors. Clear leadership and vendor constraints are required to achieve the envisioned timetable. These principles will allow vendor teams to work as independently as practical while reducing integration risk.

11.2. Principles

Early and frequent integration

Rationale

- Early integration exposes and resolves inherent ambiguities, reducing the risks in overall system delivery. Frequent integration facilitates the early discovery of implementation issues.

Implications

- Use automated integration tests, which minimize the costs of integration testing.

Component Versioning

Rationale

- A consistent versioning scheme across components facilitates the communication of dependencies between components.

Implications

- Maintain a component version dependency matrix.

Vendor Collaboration

Rationale

- Support early and frequent integration by enabling clear and convenient communication channels among vendor teams.

Implications

- Vendors must communicate and coordinate feature development, a requirement that may be unnatural to some vendors' culture. Smarter Balanced will facilitate this communication, and resolve conflicts between vendors.

11.3. Development Practices

- All components use short synchronized iterations (2-3 weeks). All teams' iterations should start and end on the same days.
- Use a common version control tool. (Recommended Git [http://git-scm.com/])
- Leverage Test Driven Development [http://en.wikipedia.org/wiki/Test-driven_development] including the use of Mock Objects [http://www.mockobjects.com].
- Follow Behavior Driven Development [http://en.wikipedia.org/wiki/Behavior_Driven_Development] (Recommended Cucumber Framework [http://cukes.info]).
- Implement Continuous Integration (CI) [http://martinfowler.com/articles/continuousIntegration.html]. All vendors should use a common CI infrastructure to facilitate cross-component integration testing.
- Use Continuous Delivery [http://en.wikipedia.org/wiki/Continuous_Delivery], Each component should be "one-click deployable" to its target environment.
- YAGNI [http://en.wikipedia.org/wiki/You_aren't_gonna_need_it] (You aren't gonna need it) Do not add functionality until it is needed.

11.4. Evolutionary Database Design

Database development should follow the same practices that software development does. It is necessary that certain practices be followed:

1. Configuration Management: All database artifacts must be in the revision control system to allow the component code and the database it interacts with to be in the same revision control system. This enables the concurrent development of the component and the database.

2. Database Sandbox: Create an automated process to create the database sandbox, so the component database can be created with all the base, or seed, data that the component needs. This enables anyone who needs a database sandbox to create one without manual intervention.

3. Database Behavior: Like code, database objects have behavior, and that behavior must be enforced by the application layer so the database provides the same behavior. Here is an article about Behavior-driven Database Design [http://www.methodsandtools.com/archive/archive.php?id=78].

4. Tracking changes to the database design and schema: Every change to the database must be coded as migration scripts, and checked into the Configuration Management System to allow for automated deployment of database changes in different environments.

5. Continuous Integration: Database changes must be part of the Continuous Integration cycle, as any changes made to the database must be tested and verified with the component. Without this verification, neither the code nor the database changes should be published. After the code is verified with the database, the changes may be published as artifacts to be deployed in other environments. Here is an e-book about Continuous Integration with databases [http://www.informit.com/store/product.aspx?isbn=032150206X].

6. Database Design: Database designers and developers should work together during component development for cohesive design and development.

12. Glossary

Assessment System Architecture and Technology Recommendations for the Smarter Balanced Assessment Consortium

Prepared in accordance with RFP SBAC 03 by Measured Progress.

12. Glossary

This section contains two glossaries for the Smarter Balanced Systems Architecture initiative. The first, Inception Glossary, defines some of the terms and items used in the inception period. The second, Architecture Glossary, captures the terms used for the architecture documentation that follows the inception period.

12.1. Inception Glossary

Below, for reference, are some of the acronyms and terms used in the inception period. In addition, the Architecture Glossary may also be used as a secondary reference.

AI
(Artificial Intelligence) The ability of a computer and software to score assessment items.

Application
Computer software designed to help the user perform specific tasks.

ARB
(Architecture Review Board) Group responsible for the ongoing governance and assurance that the architecture is periodically reviewed and updated; that the standards, practices, patterns, and policies are followed; and that solution approaches that further the goals and objectives of Smarter Balanced are reviewed.

Asset
Digital text, multimedia, or images.

ATP
(Association of Test Publishers) A nonprofit organization representing providers of tests and assessment tools and/or services.

APIP
(Accessible Portable Item Profile) A technical standard that focuses on accessibility in assessment items.

AYP
(Adequate Yearly Progress) A term defined in the No Child Left Behind Act (NCLB) as: "An individual state's measure of yearly progress toward achieving state academic standards. 'Adequate Yearly Progress' is the minimum level of improvement that states, school districts, and schools must achieve each year."

Blueprint
The design for a test. The test blueprint indicates the number of test questions or points related to each competency on the test, and the relative emphasis placed on each competency.

Charter
A statement of the scope, objectives, and participants in a project.

Cog Lab
(Cognitive Lab) A method of studying the mental processes one uses when completing a task, such as solving a mathematics problem or interpreting a passage of text. An environment where new or modified items are evaluated for their effectiveness.

DLM
(Data Lifecycle Management) Managing the flow of a system's data throughout its entire life cycle.

DNU
(Do Not Use) Describes a state of an item.

DOK
(Depth of Knowledge) The complexity of knowledge required by standards and assessments. Four DOK levels have been defined: 1) recall, 2) skill/concept, 3) strategic thinking, and 4) extended thinking.

Epic
A large feature, or a grouping of smaller features or stories. See also "Story."

FERPA
(Family Educational Rights and Privacy Act) Federal law protecting the privacy of student data.

Field Test
Test made up of test items intended to develop and calibrate new assessments.

IEP
(Individual Education Plan) Mandated by the Individuals with Disabilities Education Act, a document intended to help children reach educational goals more easily than they otherwise would. Each IEP must be tailored to the individual student's needs as identified by the IEP evaluation process, and must especially help teachers and related service providers understand the student's disability and how the disability affects the learning process.

Item Pool
Collection of items or test questions.

LEA
(Local Education Agency) An educational unit within the state. For example, a school district, a charter school, or a special needs school.

Monitoring
The overall process of supervising the administration of an assessment, including scorers.

MSL
(Master Story List) A list of requirements that evolves over time, usually found towards the end of the development process.

NFR
(Non-functional Requirement) A criterion that defines how a system is supposed to be, as distinguished from a functional requirement, which defines what a system should do. NFRs can include constraints, attributes, or processes.

PARCC
(Partnership for the Assessment of Readiness for College and Careers) One of two consortia that received Federal Race to the Top Assessment monies to develop a comprehensive assessment system.

Performance Task
A form of testing that requires students to perform one or more tasks.

Pilot Test
A trial series of new or modified items given to a select group of students.

Platform
The composite of a computer's elements, including architecture, operating system, programming languages, and related user interfaces.

PLP
(Personalized Learning Plan) Learning goals and objectives designed to meet the needs of an individual learner. It may include academic, career, and personal interests.

PNP
(Personal Needs Profile) A Profile to define individual student needs.

QTI
(Question and Test Interoperability) An IMS standard that defines interoperability for assessment items.

Requirement
An expression of certain characteristics or behavior that software should have. Also see Story.

Retired Item
An item that will no longer be used for an assessment.

Service-oriented
A set of principles and methodologies for designing and developing software in the form of interoperable services.

SIIA
(Software and Information Industry Association) A nonprofit organization for software and digital content industries.

SIF
(Schools Interoperability Framework) A non-profit organization that produces open technical standards for interoperability in the education ecosystem, including student information systems, assessments, and learning resources.

SIS
(Student Information System) A software system that houses and manages data pertaining to students.

SLA
(Service-level Agreement) Levels and standards of service defined within a contract.

SLDS
(Statewide Longitudinal Data System) A U.S. Department of Education program that provides grants to states to design, develop, and implement statewide longitudinal data systems to capture, analyze, and use student data from preschool to high school, college, and the workforce. Administered by the Institute of Education Sciences and the National Center for Education Statistics.

Specifications
(as in item specifications) Definition of the required elements of an item at a low level of granularity.

SRC
(Score Reporting Category) Assessment category of student understanding in specific content and learning standards.

Story
A required, granular element that accomplishes a specific goal in software development.

12.2. Architecture Glossary

The following list defines ambiguous terms that directly reference the architecture document and are commonly found in both the educational and technology fields.

Application Architecture
The design of an application's internal structure.

Application Development
The development of a software product.

Architecture
The practical art of selecting and interconnecting hardware components to create computers that meet functional, performance, and cost goals, also to formally model those systems.

API
(Application Programming Interface) A source-code-based specification intended as an interface for communication among software components. An API may include specifications for routines, data structures, object classes, and variables.

ASP
(Active Server Pages) A web-scripting interface by Microsoft.

Bandwidth
The rate of data transfer, bit rate, or throughput, measured in bits per second (bps).

Binary Transport
A transport implementation suited for distributed applications.

Cardinality
In database design, the defined cardinality explains how each table links to the others.

Component
One of the high-level functional units that make up a system.

Concurrency
A property of systems in which several computations are executing simultaneously, and are potentially interacting with each other.

Data Warehouse
A database used for reporting and analysis.

Database
An organized collection of data for one or more purposes, usually in digital form. The data are typically organized to model relevant aspects of reality.

Deployment
The process of making a software system available for use.

Domain
A set of common requirements, terminology, and functionality for any software constructed to solve a problem.

Hosting
A facility where software and data are kept.

Identifier
A unique name given to a specific object or a specific class of objects.

Interface
A tool and concept that refers to a point of interaction between components. The concept on an interface is applicable at the hardware level and at the level of software elements.

JSON
(JavaScript Object Notation) A lightweight, text-based, open standard designed for human-readable data interchange.

JSP
(Java Server Pages) Technology that helps software developers serve dynamically generated web pages based on HTML, XML, and other document types.

LGPL
(Lesser General Public License) A free software license published by the Free Software Foundation (FSF).

NoSQL
A broad class of database management systems that significantly differ from the classic relational database model.

Tenant
In architecture design, an instance of the software that runs on a server, serving a single client organization. Multitenancy is an instance of the software that runs on a server, serving multiple client organizations (tenants).

XML
(eXtensible Markup Language) A set of rules for encoding documents in machine-readable form.

13. Release Notes

Assessment System Architecture and Technology Recommendations for the Smarter Balanced Assessment Consortium

Prepared in accordance with RFP SBAC 03 by Measured Progress.

measured progress

13. Release Notes

The following is a history of the Smarter Balanced System Architecture and Technology Report since its initial release in January of 2012.

Version 2.0.1 – Released on March 21, 2012

This update includes a clarification on system requirements and a few minor fixes.

NEW Added CAT Simulator to High-Level Component Diagram.

CHANGE Replaced instances of SBAC with Smarter Balanced.

CHANGE Add clarification to Browser Requirements in 11.1.

FIX Title correction to Figure 8.1.3.

FIX Fixed split bullet in Section 13.3.

FIX Correction to Diagram 4.1.

Version 2.0.2 – Released on April 25, 2014

Includes several updates to the High-Level Component Diagram and Swim-lane diagrams.

CHANGE Removed User Preference and Identifier Management components. [Chapter 4]

CHANGE Merged Test Administration and Registration into a single component. [Chapter 4]

CHANGE Added Test Integration component. [Chapter 4]

CHANGE Test Packager now shown as tightly coupled with Test Bank. [Chapter 4]

CHANGE Updates to swim-lane diagrams to reflect component changes. [Chapter 8]

FIX Minor edits and corrections. [Chapter 4 & 8]

Version 2.0.3 – Released on July 11, 2014

Final Phase II updates to make the document more complementary to the SmarterApp.org assets.

CHANGE Removed several chapters that were relevant only to the initial development.

CHANGE Relabeled Administration and Registration Tools component.

CHANGE Cleaned up and added clarifications across the document.

FIX Minor edits and corrections.