Understanding and Approach to M&E Services

Section 4

RFP Reference: 6.3.8.6 Section 4 – Understanding and Approach to M&E Services

The Bidder shall provide a detailed narrative response to the Understanding and Approach topics outlined in Section 5.3. Bidders will respond to the following areas to satisfy or exceed the RFP requirements as described in Section 5 - Requirements, addressing the following topics:

- Sub-Section 5.3.3.1 - Integrated Multi-Contractor Environment
- Sub-Section 5.3.3.2 - Application/Architecture Evolution
- Sub-Section 5.3.3.3 - System Change Requests
- Sub-Section 5.3.3.4 - Innovation
- Sub-Section 5.3.3.5 - Transition-In

By "coloring outside the lines" with Deloitte, the CalSAWS Consortium can deliver County-requested changes faster, provide better integration across vendors, and deliver a next-generation CalSAWS' architecture. To help the Consortium rethink the status quo, we bring a fresh perspective informed by delivering 31 Eligibility and Enrollment (E&E) systems, including California. This experience enables us to implement improved M&E processes via user-centered design and enabling technologies (as we have demonstrated on both the BenefitsCal project and the operation of CalHEERS for Covered California).

SECTION HIGHLIGHTS

- **An M&E approach informed by current E&E delivery projects in 26 states.**
- **An approach that puts humans in the center rather than technology.**
- **An evolved CalSAWS architecture that delivers higher levels of responsiveness while retaining stability.**
- **An approach that helps the Consortium color outside the lines to better serve California Counties.**
- **Processes and tools that accelerate delivery of System Change Requests (SCRs).**

The End Result: The Consortium obtains a responsive, reliable, and innovative vendor that leverages national experience and technical knowledge that supports the Counties with their mission to provide timely health and human services.

4.2.2 Maintaining Legacy Architecture (ME-UA5)

RFP Reference: 5.3.3.2 - M&E Understanding and Approach to Application Evolution

ME-UA5 Describe how you will maintain the legacy architecture during evolution, how platforms and how data will be kept in-sync, how changes will integrate with existing technologies and networks, how changes will be tested, and any other factors to be addressed, including security.

CalSAWS, and more importantly County business continuity, is critical throughout the application / architecture evolution. Our approach enables the legacy CalSAWS and CalSAWS^{CN} to co-exist, while keeping the modernized modules and the legacy codes synchronized and integrated. We selected to migrate the database first to a cloud-native platform to prevent the need for data replication during the changes to the application code. **Modernizing the data first into a cloud native database keeps the data in sync naturally and enables the delivery of a stable and reliable combined system to the Counties with data decoupled from the application through data access layers that are aligned to the target state.** The process for decoupling and migrating the data will also improve the current data quality and provide improved reporting. During the database/data access changes and throughout the application evolution, we use the four pillars in Figure 4.2.2-1 to execute a "no to low disruption" approach while evolving CalSAWS to cloud native, CalSAWS^{CN}.

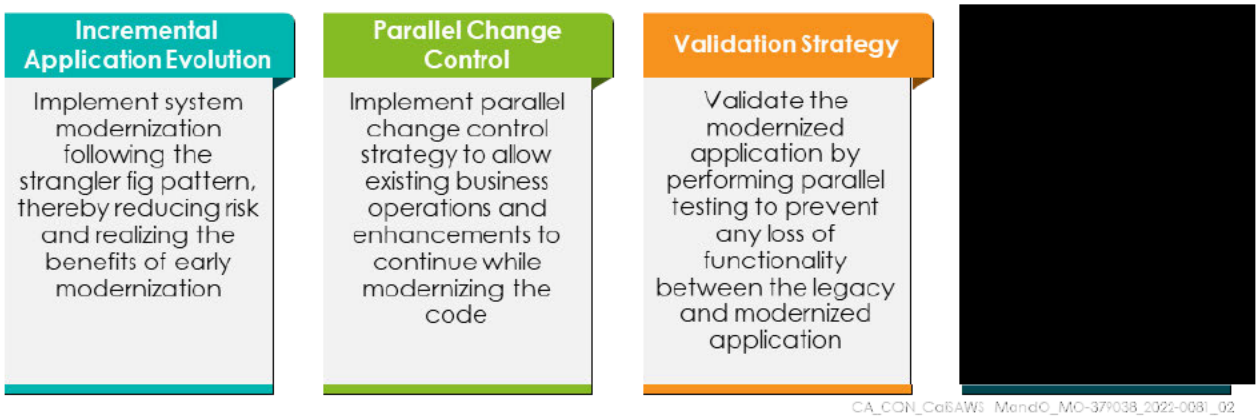

Figure 4.2.2-1. Four Pillars for Evolving and Maintaining Legacy Architecture.

At a high level, these pillars work together to create two independent workstreams, one for new enhancements and one for application evolution, executable by separate teams. Each workstream goes through complete functional and performance test cycles before merging. Then, a third functional and performance test cycle is conducted to verify the combined results of the changes. The emphasis on stability and testing has enabled us to succeed with this approach on multiple projects. We have improved our approach by applying it to real projects. This allows us to enhance functionality while evolving the application's architecture simultaneously with minimal risk. The remainder of this section discusses in more detail how the first three pillars align to, and support, the Consortium's objectives outlined in ME-UA5 (i.e., keeping platforms in-sync, integrating existing technologies, testing changes, and security).

4.2.2.1 Keeping Platforms and Data Synchronized

Synchronizing the legacy and modernized platforms requires an understanding of the end-state architecture and sequence of the application evolution. This approach also

prevents the complication of replicating data between databases for different parts of the application, allowing changes to underlying data structures to occur transparently to the legacy application. Another advantage to the modernized shared database comes with reduced complexity. Fewer database instances require less management. **The only remaining data synchronization uses the ability to automate full or logical replication (table-by-table) to make a replica of the database for read-only access to synchronize to the reporting repository. Using dedicated read-only replicas that can be recreated on demand allows the data to be more rapidly synchronized for reporting by dedicating resources to it.** We take advantage of the automation and improved data quality we create using the modernized database and provide increased velocity in the replication and transfer of data to the reporting system. As we develop each feature module, we apply the first of the four pillars:

Applying Incremental Application Evolution (Pillar 1)

As we transform CalSAWS into microservices, we employ a "Strangler Fig" pattern to the legacy architecture, shown in Figure 4.2.2-2. The "Strangler Fig" pattern is modeled on the Strangler Fig Tree that grows around another tree and eventually strangles it out or replaces it. This pattern allows us to incrementally create the new application services while still using the legacy CalSAWS monolith application.

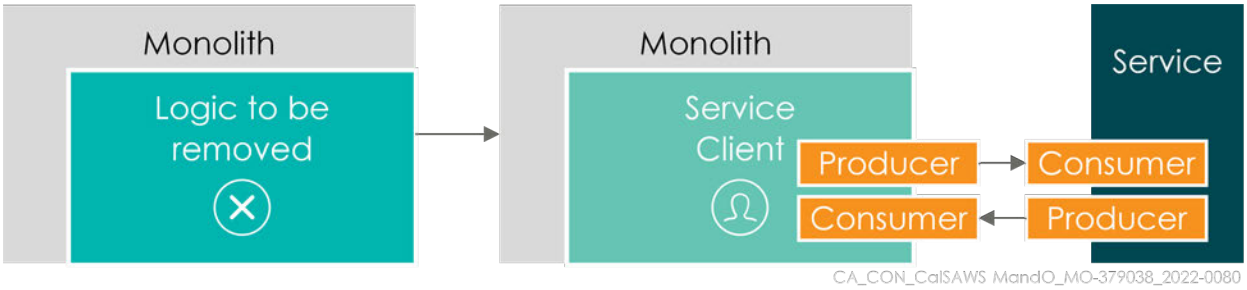

Figure 4.2.2-2. How We Enable Co-Existence During Evolution with the Strangler Fig.

Business functions are refactored as a new service with a well-defined API. In the legacy application, the old logic is replaced by a wrapper directing requests to the new service. Figure 4.2.2-3 provides a before view of using the strangler fig pattern for Application Registration (AR), Data Collection (DC) and Eligibility Determination (ED) capabilities.

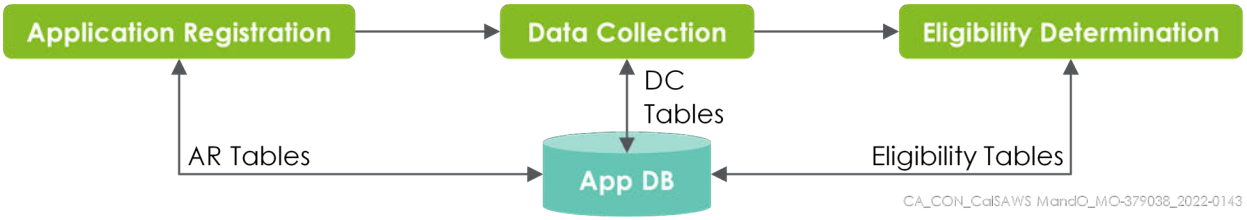

Figure 4.2.2-3. Example: Application Registration Prior To Strangler Fig.

Figure 4.2.2-4 below shows how we balance modernized Application Registration services with Data Collection and Eligibility Determination still running as legacy applications while functionality stays synchronized and integrated using API communications between the legacy and modernized application. The legacy monolith application and the separated microservices share the modernized cloud

native database and take advantage of different schemas and permissions to manage module ownership of the data over time.

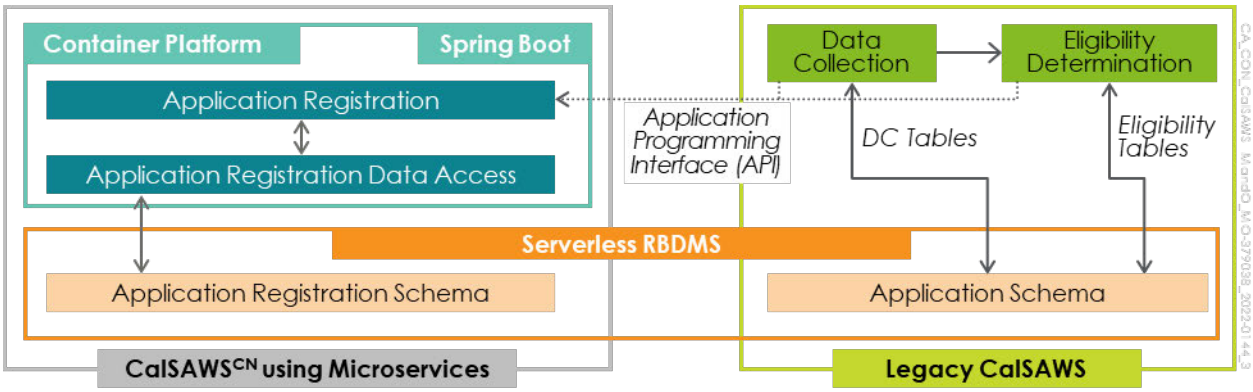

Figure 4.2.2-4. Example: Co-Existing AR Microservices and Legacy Applications.

Implementing incremental change evolution necessitates careful planning and execution. This confirms the prevention of conflicts between business and technical changes, while enabling their harmonious coexistence. While reviewing the plan for new business functionality and identifying the domain areas we are modularizing, we establish the capacity for evolved services to operate completely independently from the legacy application. To initiate the process, we begin by constructing a strong foundational architecture through the design and implementation of core services. These services encompass vital application architecture elements such as exception handling, application logging and tracing, error management, configuration management, and security. These components serve as the building blocks upon which developers construct other elements within the new architecture. We create other utility functions (e.g., cache management, date management, data validations) that can be used across the microservices. The functional microservices leverage this building framework to completely decouple themselves from the monolith throughout the SDLC.

Next, we apply the Strangler Fig pattern by lifting the relevant code for a module out of the monolith to build a standalone service, introducing APIs to access the service and then evolving the **data access layer** to improve the structure of the database. Because we did cloud native database migration first, we have a thorough understanding of the database tables and relationships and know where we will need to introduce APIs in the monolith. We also understand where the **logical responsibilities** for data management and ownership correlate to specific modules. This decreases risk during the evolution because we do not need make business code changes. This initial independent service is tested for parity with the previous release, through the integration with the remaining monolith executable.

In parallel, a different team takes the same starting code base and makes the functional changes necessary to create new features needed by the business. This approach allows each team to make and test their changes on a stable base before combining the changes and retesting.

This implementation requires tight communication and collaboration across teams, disciplined change identification and grouping/sequencing of like changes which are

constantly implemented throughout each sprint. Continuous integration of technical changes allows teams to see working software, not just designs.

This ongoing process is demonstrated in Figure 4.2.2-5 below.

Figure 4.2.2-5. CalSAWS Enhancement Collaboration Plan.

The next key to this process is the parallel change management that we describe in the next section.

4.2.2.2 Integrating with Existing Technologies and Networks

We align with the Consortium's requirement to adhere to the SCR process and regularly deliver SCRs in parallel to CalSAWS^{CN} activities. We evaluate cloud services and open-source software, as well as existing tools and service, to support integration. This includes shared services technologies for accurate logging, monitoring, alerting, and reporting across platforms. To manage integration between existing technologies and networks, and the evolved feature modules, we apply the second pillar: **Parallel Change Management.**

SECTION HIGHLIGHTS

- Application evolution work is planned considering the SCRs prioritized by the Consortium.
- Impacted application components are identified, sequenced, and developed as microservices in a separate code stream.
- Post validation, microservices are merged into the legacy code stream.
- A single integrated release into production per release calendar.

Applying Parallel Change Management (Pillar 2)

Parallel change management enables M&E work to continue while minimizing the impact of CalSAWS^{CN} application evolution activities. Collaboration between teams, change management grouping and sequencing of similar changes, and isolation of features are facets of this pillar. Our holistic view of parallel change management considers facets of an application: the source code, the

database, and the underlying infrastructure (HW, SW and tools). The key tenet here is maintaining 2 parallel sets of environments, code, and database to allow technical changes and business functional changes to run in parallel and then merge successfully for CalSAWS' benefit. This approach promotes stability for both coding activities and allows the consortium to make release decisions without compromising quality. Stable code branches consistently exist for both sets of work and for the combined changes after the merge is complete. This is also particularly important when a change may require significant partner testing that may impact when a change can be released. Below, we provide a description with how we manage the three facets of parallel change management – **Source Code, Database, and Infrastructure.**

Parallel Change Management: Source Code

Parallel change management controls code changes explicitly by leveraging two code streams: a **dedicated legacy code stream** for legacy code and database, and a **CalSAWS^{CN} serverless code stream** modernized code and migrated database. Figure 4.2.2-6 demonstrates how the legacy code stream supports planned changes (e.g., enhancements, patches). In the example, CalSAWS^{CN} creates the new microservice Application Registration (MS_AR) and APIs. At the same time, the team working on the legacy code adds new functionality to the Eligibility module.

After we have successfully completed and tested the new business changes (e.g., in the Eligibility Module) and regression tested the modernized application services (e.g., the new Application Registration microservice), we start through the merge process. This process includes changing the monolith code to call the new microservice, and then merging the new microservice code into the same branch as the updated monolith. The monolith begins to remove parts of its own code by calling the new CalSAWS^{CN} AR module through an API. At this point the data is logically owned by the microservice and only accessed through the API. The new Application Registration service represents an independent business domain that can change and evolve independently with its micro-frontend and its microservice backend.

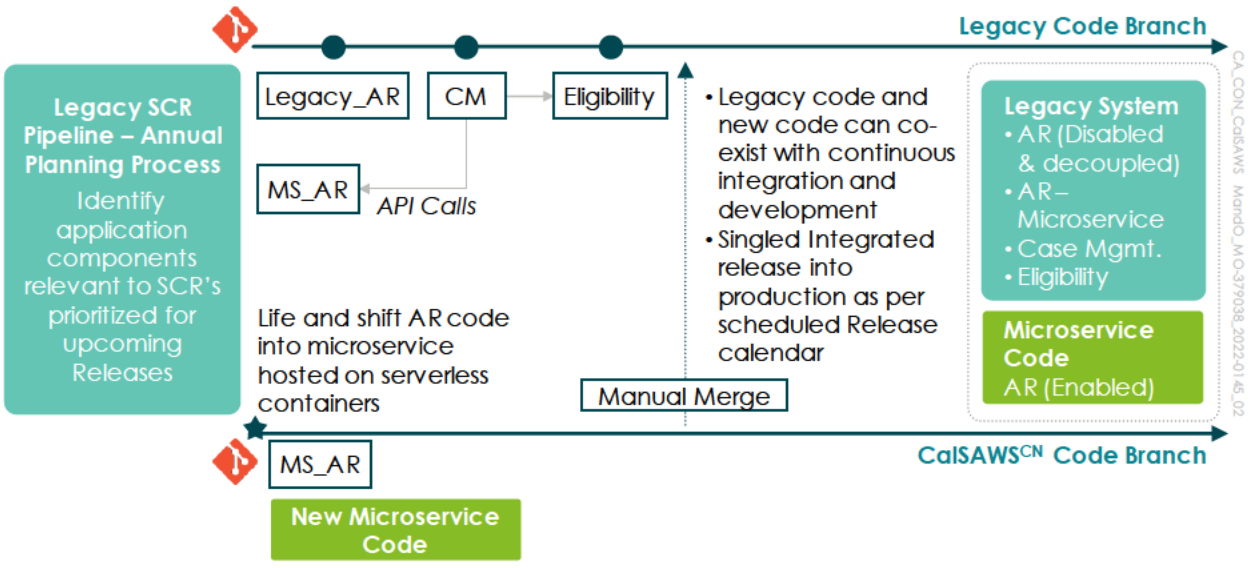

Figure 4.2.2-6. Example: Parallel Change Management between Legacy Code and CalSAWS^{CN}.

Parallel Change Management: Database

We have a proven database change management process that enables us to manage parallel databases (one for legacy environment, one for the modernized environment). Database changes are completed through database scripts that are also checked into the source code repository. The database changes are correlated to specific code versions and tagged into the same baselines and releases. For example, when we embark on the technology modernization journey there will be three parallel streams of work: 1) CalSAWS application modernization into microservices, 2) CalSAWS DB modernization, and 3) CalSAWS functional changes. Our processes merges database changes across the three streams at the right time to allow the application to function smoothly without disruption.

We use logical separation through the evolution process to reduce the need for data replication. When there is a database change needed for a specific phase of the evolution, changes are thoroughly tested with the merged code to verify that data access has been isolated using the data access layer. The industry patterns we use to mitigate the impact of this kind of change includes Natural Keys, Literal Keys, Hexagonal, Façade, Anti-Corruption, and CQRS patterns. Which pattern is used depends on the business domain object that is being addressed, its rate of change, the module responsible for it and the business requirements.

Parallel Change Management: Infrastructure

While the code and database evolve, the underlying infrastructure will also evolve to take advantage of more cloud services and simplified, lighter weight tools. There will be upgrades and patches that get applied to legacy infrastructure, and infrastructure changes that will get applied on the new technology modernization environment. We work with the infrastructure vendor from architecture through design to create the modernized and evolved cloud infrastructure architecture jointly to support the application evolution. Making sure the infrastructure changes are clearly documented and requested, correctly applied and migrated across environments requires close coordination and ongoing collaboration with the infrastructure vendor. Changes will also include security changes as we build and apply zero-trust principles.

We work collaboratively with the infrastructure vendor to make sure the infrastructure changes are sequenced with the corresponding code changes (as applicable) and application release. For example, when the implementation of database modernization is underway, the required legacy DB upgrades would only be applicable to environments that do not have the new upgraded DB code and are scheduled for releases prior to database migration or evolution. This includes upgrading production legacy database as well for any patches that require remediation (e.g., security vulnerability patch). On the flip side, a patch for the evolved DB would only be applied to environments that have evolved DB code or environments slated for later releases after the database evolution. This type of patch will go to PROD, with the evolved DB release.

As we evolve the code, database, and infrastructure we will account for the integration considerations included in Table 4.2.2-1.

Services	Technology and Network Considerations
Interfaces	Existing inbound and outbound interface end points, including virtual private cloud (VPC) peering for RESTful APIs, WebSocket APIs, and SFTP, are updated in lower environments, tested, and validated before updating in production.
Integrations	VPC end points for integrations are updated and tested before the cutover.
Logging	The logging from application services is aggregated to the centralized log files by leveraging existing logging tools (e.g., CloudWatch, Fluentd).
Monitoring	CalSAWS APIs are integrated with CloudWatch to monitor and alert on any application and infrastructure issues.
Reporting	Tools such as Grafana, Kibana, Qlik, or AWS native tools are leveraged.
Security	Authentication and Authorization through ForgeRock; security monitoring through Splunk; and network security through AWS Network Firewall, CloudFront, WAF, NACLs, Security Groups, etc. are integrated to protect CalSAWS from threats.

Table 4.2.2-1. Shared Services Integration Options.

4.2.2.3 Testing Changes

Applying Validation Strategy (Pillar 3)

We use a four-point validation strategy, highlighted in Figure 4.2.2-7. The first applied to the data migration, with the remaining points being applied iteratively throughout the application / architecture evolution journey. Validation is successful if no data loss is sustained during migration, and application functionality remains unimpaired.

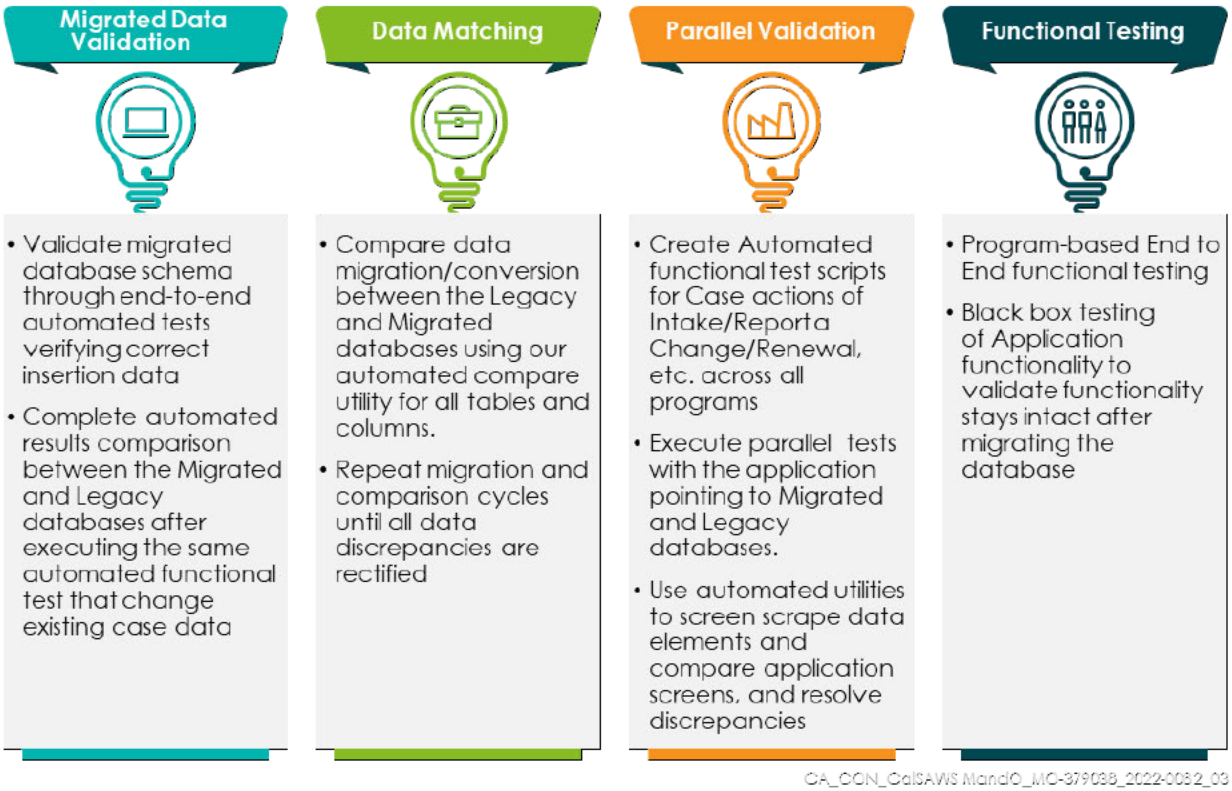

Figure 4.2.2-7. Validating and Testing the Data and Modules.

A robust validation strategy enables the Consortium to discover application challenges early during database migration and application evolution activities. For instance, we uncover non-standard queries, hardcoded data, or undocumented business rules. We have created a set of leading practices in the four-point validation that improve the automated database migrations and evolutions from 70% accuracy to 95% accuracy, leaving only 5% for manual intervention. These processes include the generation of complete data validation test scripts to verify the data migration or change results. Our team leverages GenAI to accelerate the creation of test scripts and also enhance the overall coverage of conditions to be confirmed.

While the four-point validation helps uncover issues and confirms system behavior both pre and post technical changes, it is imperative that the validation is performed ahead of time to minimize any disruption to business-critical changes. To isolate and validate technical changes from the business changes we perform 3 cycles of validation. Each validation cycle includes a full gamut of testing coverage including functional testing, security testing, performance testing and regression testing.

Figure 4.2.2-8 demonstrates the key features of each of the testing cycles.

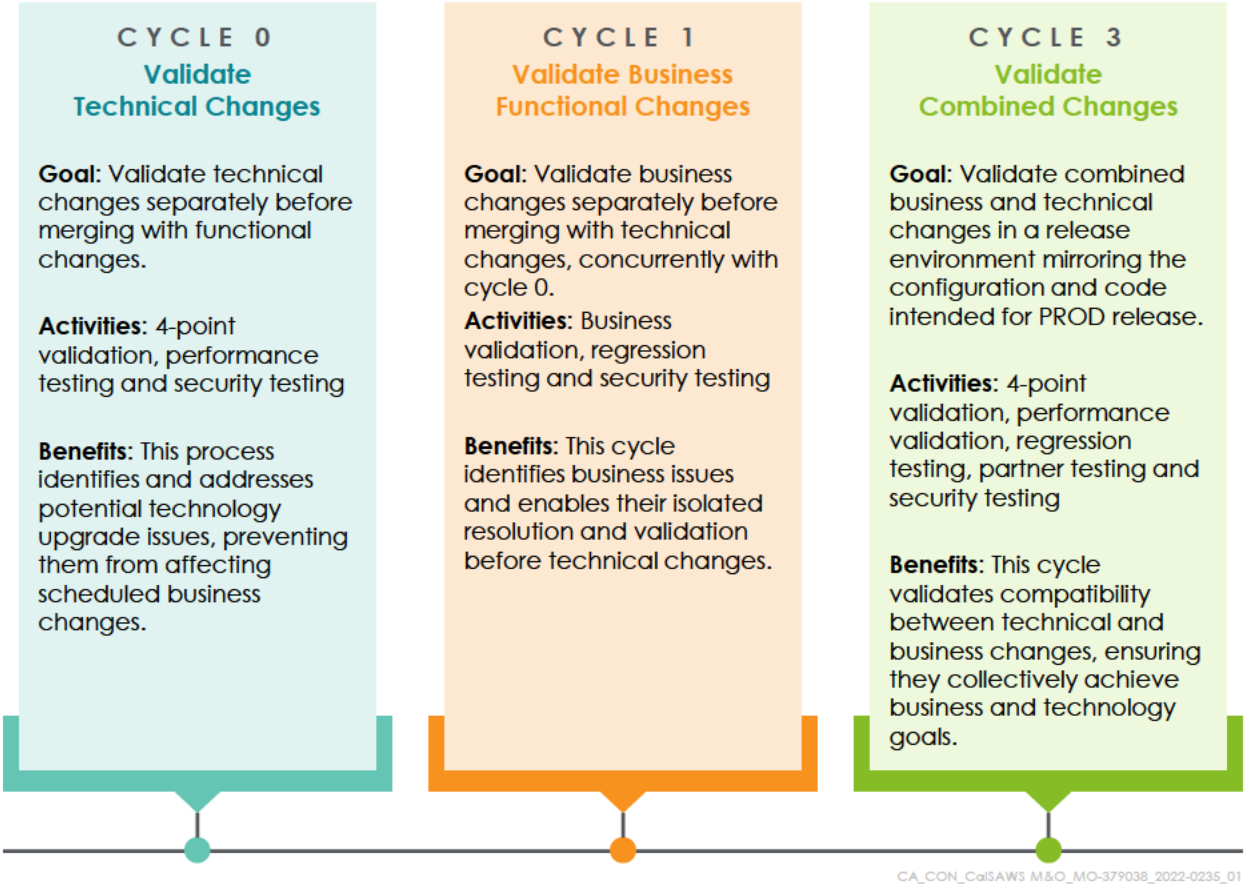

Figure 4.2.2-8. Validation Strategy Testing Cycle Key Features.

By applying this validation approach to testing, we showcase the synchronized functionality of the code, database, and infrastructure, resulting in an enhanced and adaptable system. These methods have elevated the system's quality and are continuously tracked via established quality and performance KPIs. These KPIs are

gathered with each incremental release throughout the year and encompass metrics spanning startup times to response performance of application modules.

4.2.2.4 Other Factors Including Security Controls

Security controls that include data security, network security, and access controls are implemented to mitigate security risks and threats. These include controls such as:

- Encrypting data in-transit and in-use with a minimum Transport Layer Security (TLS) v1.2 and Advanced Encryption Standards (AES) cipher suites using only trusted digital certificates to enable secure communication between the assets.
- Encrypting data at-rest by the selected AWS database services at the disk level using Federal Information Processing Standards (FIPS) 140-2 guidelines.
- Disabling open ports and blocking unsecure ports to minimize the threat surface area.
- Logging and monitoring database activities for suspicious activity (e.g., repeated invalid login attempts, database instance creations, command errors/exceptions).
- Securing customer managed keys (CMK) with AWS Key Management Service (KMS) with a key management lifecycle to monitor and enforce separation of duties.
- Configuring security groups using the principle of least privilege.
- Segregating data between production and non-production databases with controls applicable to an operations copy of the production database.
- Implementing data retention and archive based on Consortium security policies.