Software Development & Re-Engineering Guidelines

for

Cloud Ready Applications

Version 2.1

Prepared by:

Department of Electronics & Information Technology

Ministry of Communications & Information Technology, New Delhi

Table of Contents

1 Introduction 3
1.1 Need for Software Development & Re-Engineering Guidelines 3
1.2 Evolution of eGov App Store 3
2 Software Development & Re-Engineering Guidelines 5
2.1 Solution Architecture 5
2.2 Standards Adoption & Solution Engineering 8
2.3 Integration & Interoperability 12
2.4 Quality Certification, Release Management & Documentation 13
2.5 Solution Sizing & Scalability 14
2.6 Language & Interface 15
2.7 Legacy Integration – Digitization & Migration 15
2.8 Intellectual Property Rights (for Center & State owned applications) 15
3 Cloud Enablement of Applications 17
3.1 Application Migration to Cloud 17
3.2 Software as a Service Characteristics 23
4 Annexure I - Application Self-Assessment Checklist 27
5 Definitions 30

1 Introduction

Productized and Cloud enabled applications are ideal solutions that can be utilized by various departments at centre and states without having to invest time, cost and effort in development of the same. This would enable re-use and deployment of applications rapidly across several states/departments.

1.1 Need for Software Development & Re-Engineering Guidelines

The basic need for Software Development and Re-engineering Guidelines is to ensure development of Common Application Software (CAS) which can be configured as per different states / departments requirements without the need of modifying the core code of the application for a faster deployment so that time, effort and costs in developing applications are saved and to obviate duplication of efforts. It is therefore imperative that applications are developed in conformity to guidelines that makes them standardized and compatible for hosting and running across states. This need has translated in the conceptualization, development and roll-out of productized cloud enabled application which can be centrally run & hosted and are available to states for configuring them as per their relevant processes with minimal customization for rolling out the services in shortest time possible.

It is envisioned that an application which is centrally run as a SAAS is easy to roll out to all interested parties at the same time and therefore such application’s architecture and design should be compliant to common minimum practices / considerations that will convert it to standard product.

1.2 Evolution of eGov App Store

The productized and cloud enabled application for states / departments will be made available on the eGov AppStore. The eGov AppStore was launched by the Hon’ble Minister of Communications & Information Technology, on 31st May 2013. The eGov AppStore is a national level common repository of customizable and configurable applications, components and web services that can be re-used by various government agencies/departments at Centre and States, with the vision to accelerate delivery of e-services as envisaged under NeGP and optimizing the ICT spending of the government with the following objectives:

- Speeding up the development and deployment of eGov applications
- Easy replication of successful applications across States
- Avoid duplication of effort and cost in development of similar applications
- Ensure availability of certified applications following common standards at one place

The key benefit for Stakeholders is that they need not reinvent the wheel and an application which is successfully running in another state can be made available to them expeditiously with requisite customization. Core and common applications that have high demand and are replicable across the central and state levels are the likely candidates for the eGov AppStore, which shall be hosted on the National Cloud. The eGov AppStore will include the setting up of a common platform to host and run applications (developed by government agencies or private players) at National Clouds under Meghraj, which are easily customizable and configurable for reuse by various government agencies or departments at the central and state levels without investing effort in the development of such applications.

2 Software Development & Re-Engineering Guidelines

2.1 Solution Architecture

The solution architecture is key differentiator for product like solutions. A well architected solution gives it robustness for reusability (in code, configurations, databases, services etc.), enhancements and interoperability.

The following should be adopted as good architecture principles:

- **<u>Well established Service Contracts</u>**
  A contractual agreement between the Application Owner (Govt. Department at Centre/State or any Private Player) and the Application Provider (Govt. Department or independent entities which host & provide services through eGov AppStore) over the period of Application Lifecycle (for example: Productization + Replication + Hosting + Operation & Maintenance). The contracts related to licenses, source code etc. will also be a part of such agreements.

- **<u>Loose Coupling of Services</u>**
  This is one of the fundamental concepts of Service Oriented Computing. Loose coupling ensures that application components are treated individually and dependencies are reduced. This further ensures that addition, removal, failure or update of one component has a minimum impact on other components.
  Effort should be made to develop components separately and then their integration/ interaction mechanism could be defined in a separate component. For example, while developing a component that calculates the order of a commodity should not start calculating the total cost of the order placed. Order should be calculated separately and the cost should be calculated separately so that any change in costing structure should only affect the cost calculation code and not the order placement component.

- **<u>Service Reusability</u>**
  For the purpose of reusability, services should be written in such a way that they can be automated for testing. Test automation is necessary to ensure services can be upgraded, re-factored, etc. without breaking other services that use this.
  Further, all services should be inherently versioned and all invocations must specify the version of service. Efforts should be made to ensure that new versions of services should be backward compatible with at least one or two previous versions so that users of the service can start using new version of the service without mandatorily making changes to their code.
  Rapid Replication and productization of successful applications running across different States/UTs would ensure that these applications are also reusable in other states with appropriate built-in configurations which can be undertaken by concerned seeker state / department. The solution

should also support minor customization if so essentially required by the seeker state / department. A repository of re-usable components is to be maintained and made available on eGov AppStore. Software components can often be classified according to reusability levels:

- **Foundation Components**
  Examples of foundation components are classes such as Money, Date, List, Person and Number. These can be reused in almost any application
- **Domain Components**
  Examples of domain-specific components include classes like Customer, Account, and Transaction
- **Architectural Components**
  Examples of architecture-specific components include event notification mechanisms; user interfaces components, and message passing systems
- **Application Components**
  Examples of application-specific components include message handlers, exception handlers, and views

➢ **Service Abstraction**

Abstraction provides control on what part of the service logic of a particular application are private (hidden) and what parts are made public (consumable). The public or consumable parts of the service logic can be designed in a generic manner to ensure that they encourage reusability as discussed in the point above. Abstraction also supports the loosely coupled principle discussed above. In a three tier (database, business and presentation) software application, necessary abstractions should be done in each layer so as to achieve loose coupling and to keep the code modular so that addition of any logic could easily be done at any tier. For example, in application development for scholarship disbursement system, a function to fetch beneficiary details may be designed to interact with database layer and gives the information to presentation layer. How the database layer performs the operation to fetch details should be abstracted from the business layer. Similarly how the presentation layer represents the information should be abstracted from the business layer.

➢ **Service Discoverability**

While productizing the existing application or designing a new application for hosting on the eGov AppStore, it is important that accidental creation of redundant services or implementation of redundant logic is avoided. Service discoverability makes this happen by ensuring that metadata attached to a service and describes overall purpose of the service and its functionality, which makes the services easily discoverable. A repository of re-usable business logic components is to be maintained and made available on eGov AppStore. For example an existing service or business logic already available at the data center should not be recreated to save duplicity.

- **<u>Service Autonomy</u>**

  In addition to the principle of Reusability discussed above, it is important to ensure that services which are delivered do not just possess reusable logic, but they are also autonomous to be reused. This Autonomy will also facilitate adaptation to changing constraint in terms of scalability, service levels adherence, availability etc. For example only loosely coupled services or service components can be reused, therefore autonomy becomes an important parameter to efficiently design solutions.

- **<u>Service Location Transparency</u>**

  This refers to ability of the Service Consumers to use a service regardless of its actual location, for example being available on a cloud.

- **<u>Service Granularity</u>**

  Service Granularity means identification of optimal scope of business functionality in a service operation. Each service operation should ideally perform single transaction to simplify error detection, error recovery, and simplify the overall design (this means that particular Service operation is granular). In addition, each service operation maps to a single business function, although if a single operation can provide multiple functions without adding design complexity or increasing message sizes, this can genetically reduce implementation and usage costs (here each service operation is generalized enough and interoperable for multiple functions, making it granular).

- **<u>Platform & Database Agnostic</u>**

  From an architectural perspective, it would be required that the productized solutions should be not only be modular in nature, but be adaptive to converse with other technology components such as platforms and databases, complete with management suites or with the induction of adaptors and interfaces or even smaller bespoke solutions to support the same. It would also be required that the application provider should be able to deliver application on latest IT Infrastructure & system software components available at National Cloud and at SDCs under Meghraj. This would ensure that the applications developed can overcome the technology dependences and be available to a variety of seeker states.

- **<u>Application design for occasionally connected systems</u>**

  For the small percentage of functionality that requires “occasional disconnected/offline” operations, applications may be designed to use a local persistent store/cache just for the purposes of offline capability and later sync as and when connectivity is restored. As connectivity becomes ubiquitous, less of such offline capabilities are needed.

2.2 Standards Adoption & Solution Engineering

There are a number of standards available on software engineering lifecycles which ensure quality product development and scope of continuous improvements. The standards are to be followed as per the Government of India issued policies and guidelines promulgated from time to time.

The proposed solutions should be adaptable to the following as good software engineering practices:

- **Domain / Sector specific Meta Data Standards**
  Each sector or domain has its unique challenges in standardization of Meta-Data. It is important that any solution being developed to provide services in the domain or sector adhering to the Meta-Data standards for that particular sector or domain. This would ensure seamless integration between solutions developed for domain or sector. The GOI has also come out with Meta data standards which can be seen at www.egovstandards.gov.in

- **Software Engineering Standards**
  It is important that software engineering standards are adopted during the initial stages of the development lifecycle to ensure that the developed solution is able to meet quality certifications and security testing. Recommended testing requirements will be provided by STQC / empanelled agencies.

- **Usage of Open Standards technologies**
  As part of the software engineering, it is important to use technologies developed in open standards. As part of the overall software development lifecycle, a minimum customization and maximum configuration approach should be adopted. There should not be any hard-coding in any aspect of the development and release lifecycle of the proposed application. The following section articulates areas (no limited to) that should be available as configurable parameters, while overall software having the ability to be customized so as to meet the local requirements of the user state / department / agency. e-Governance application should preferably be developed using open source tools and components.

2.2.1 Configurable Components

An important facet of product like solution is its ability to be configurable to meet the business requirements. The following should be available as configurable components:

- **Master Data**
  Master data should be available in parameterized format. It should be based on the Meta data standards for the industry / domain / sector. They should not be hard-coded in the application.

- **Screen Labels**
  Screen labels may differ between solutions owing to the localization requirements for a solution proposed to be implemented. Configuration of screen labels should be made available through resource files. They should not be hard-coded in the application.

- **User Alerts & Messages**
  Based on the user departments business requirements, alerts and messaging services need to be pushed or pulled to the end user. Allowing for alerts and messages to be available as a configurable component would ensure that unwanted alerts and messages are not routed through to all workflow entities.

- **Reports**
  It is generally required from solutions to be able to prepare various kinds of reports for various levels of officers in the hierarchy, along with aggregation and data sorting features. Available as a configurable component, it would ensure that the reports are localized to the needs of a user, rather than being generic to business function or sub-unit.

- **Workflow Management**
  Common business functions in two similar organizations may have different processes related to approvals, escalations, reviews, recommendations etc.; therefore it is important that workflows are available as configurable components to allow the solution to be configured to the business requirements of that organization.

- **Multi Language Support**
  Government departments operate in multiple languages depending on their region. Product like solutions should be adaptable, to allow through configuration, selection of language in which the user wishes to operate the system. Product like solutions should at least be bi-lingual, with English as one of the languages.

- **Business Rules (if - then - else)**
  Business rules are at the core of workflow processes and allow for information, interaction and transaction services to be communicated. Product like solutions should ensure that business rules are configurable to allow the organization to localize the solution to their business requirements. They should not be hard-coded in the application.

- **Dashboards**
  As a management tool, most senior officers require dashboards to review service progress, service levels, escalations, alerts and reminders, messages etc. As an operational tool it is required by the office staff for work-list detailing, alerts, reminders and messaging. As configurable component, it would ensure that the user is able to see his or her, role based dashboard for summary of tasks and activities to be completed.

- **Online Help & Feedback**
  As a feature in most standard products it would be required that online help and feedback mechanism should be available as configurable parameters to assist the users in functioning of the application. This could include context sensitive help, user manuals etc. In online feedback mechanism, feedback on technical aspects as well as service delivery should be given to the users.

2.2.2 Customizable Components

A solution may be required to be customized to meet specific business requirements of an organization. The following should be kept in perspective while customizing core solutions:

- **Ability to add additional features without compromising the core code**
  The solutions should be developed in modular format, or should allow for modular integration or interfacing with other solutions, without the need of editing existing core code. Solutions should allow for the development of new features, functionalities, changes to done through interfaces external to the existing code base.

- **Ability to interface with other independent sub-applications**
  It may be required that a product like solution is required to interface with other bespoke smaller applications, unique to an organization. There should be minimal effort required for such activities, and should be made available through external adaptors interfacing with the core application.

Methods of customization could include:

1. Implementing a plug-in architecture so that tenants could upload their own code through defined interfaces without changing the core application or;
2. using some form of rules engine that enables process customization through configuration

3. Another alternative to consider is enabling application to call a service endpoint provided by the tenant, which performs some custom logic and returns a result.

In addition, application may also require providing ways to extend the application without using custom code. To achieve this application must implement a mechanism for customizing the UI, and a way of extending the data storage schema.

Methods of extending schema can be:

- Single fixed schema with a set of columns available for custom data
- Single fixed schema with separate tables holding custom data

2.2.3 Mobile Enablement

The reach of mobile technology and devices has percolated beyond the last mile of connectivity into the households of the most unreachable terrain in India. Therefore it becomes important that government service delivery is undertaken through this medium to increase the scale and reach of government services throughout the nation.

As a resultant it is required that the applications that planned to deliver these services use the mobile medium to provide services. There are three means through which applications can be engineered to provide services through mobile enablement:

1. Accessing application on a mobile device
2. Accessing a mobile version of the application through a mobile device (m.website)
3. Accessing a mobile application through a mobile device

In the first means the accessibility of the application through the medium changes translating from a system based access to a mobile device. The second means assumes the redevelopment or reconfiguration of the application to suite a mobile based delivery platform, (m.website) which follows best practices in providing applications with limited or complete functionality to be accessible over a variety of mobile service delivery resolutions (such as in case of smartphones, tablets, key interface phones). The third means assumes the redevelopment of an additional application or app, which can be downloaded and run on the mobile device.

Furthermore the application access can be given through multiple means over mobile devices in formats such as UDDS, SMS, App etc. It is predominantly decided the services being offered by the parent domain department / agency to select the means through which the services can be provided.

In case of native mobile application development wherein business layer is planned for deployment on remote tier, a separate service layer can be designed. Services should be designed for maximum reusability by not assuming any specific details of client. For improving interoperability, REST based protocols and transport mechanisms can be implemented.

From an application development and re-engineering guidelines perspective it is required that the applications are developed to meet mobile device service delivery platform requirements while at the same time ensuring security of data, ease of use of the application and continuation of the citizen experience as over traditional access mechanism.

2.3 Integration & Interoperability

A key requirement for any product is its ability to interface, integrate and more importantly be interoperable with other technology suites. The application should be developed in a manner that it should support flexible, modular and extendable services. The proposed solution should have the following:

- Clear input and outputs should be defined
- Ability to perform business validations
- Clearly defined error codes
- Support (i) Asynchronous (ii) Synchronous (ii) Batch mode, models of integration
- Support Web Services
- Support File Transfer
- Support SMTP
- Support Mobile (SMS) service delivery
- Support API based integrations
- Support Push & Pull Integration
- Support Published / Subscribed methods such as Java Messaging Service, RSS etc.
- Support integration on open standards
- All major validations / constraints such as primary & foreign keys can be at the database level and others such as business logic at the context level
- Access should be compatible with external devices such as hand-held devices, tablets & smartphones

2.3.1 List of Open APIs proposed to be published

The eGovernance projects are linked to each other because they service a common list of beneficiaries, i.e. the citizen. It is required that new applications developed and those re-engineered should capture and process data limited to their agency / department. Any data which can be furnished or exchanged through an external department / agency should be done through the use of APIs (Application Programming Interface). This will allow the application to source data through a unified data pool and will marginalize errors in data entry for the same record. API invocation must allow platform neutral and language neutral way of calling. For example, a service written in Java should also be usable within an application developed in .NET environment.

The proposed application should list all the APIs that it intends to provide to be consumed by other departmental applications (including citizen interfaces) and should also list data elements which it needs to be sourced from other departments. Prime examples of this can be UID for personal information, Vahan data for vehicle data etc.

2.4 Quality Certification, Release Management & Documentation

2.4.1 Quality Certification

It is important for product like solutions to adhere to quality certification processes to ensure that solutions being given for replications to other stakeholders, meets minimum quality benchmarks. To ensure a quality product it would be required that the solution:

- Should qualify defined functional testing through STQC / empanelled agencies
- Should qualify defined performance testing through STQC / empanelled agencies
- Should qualify defined security testing through empanelled agencies
- Should have well documented development & testing process artifacts
    - Business Requirements Document (BRD)
    - Functional Requirement Specifications (FRS)
    - Software Requirement Specifications (SRS)
    - Software Design Documents (including HLD, LLD etc.)
    - Requirements Traceability Matrices (RTM)
    - Test Plan, Test Cases & Test Reports
    - Code Review Reports
    - Database Review Reports
    - Project Implementation Plan User Manual
    - Deployment Guide

Detailed testing requirements will be provided by STQC.

2.5 Solution Sizing & Scalability

Since the solution will be required to be hosted on various deployment models, it is important for the solutions to be able to scale up to meet increasing usage requirements. Although an initial estimation of the hardware specifications (quantity and model / version) would be required to size the solution based on system interaction, to increase capacities the solution should adaptable to scaling. The following should be kept in perspective:

- **<u>Able to scale up to meet increasing load</u>**
  Solution should be able to handle increasing number of first time users, transactions, data sharing processes etc.

- **<u>Able to demonstrate stress levels exerted</u>**
  Solution should be able to handle increasing number of concurrent users, concurrent transactions, synchronous data sharing with other systems etc.

- **<u>Able to perform on throttled bandwidth environments</u>**
  Solution should be able to perform to the agreed service levels regardless of the bandwidth available or in multiple bandwidth availability scenarios

- **<u>Should have low technical & infrastructure resource consumption</u>**
  Solution should optimally use technical resources such as memory, processor (CPU), storage etc. In addition should optimally use data center resources on available bandwidths.

- **<u>Should be interoperable to newer technology upgrades</u>**
  The solution should be able to harness the advantages of legacy technology (servers, software, devices etc.) while be able to upgrade to newer systems. This would enable low cost – optimal utilization of resources.

- **<u>Horizontal Scalability</u>**
  1. Scalability of an application is aided through designing services as granular as well as loosely coupled. Use of distributed data stores and sharding also aid application scaling. If the service uses database/datastore, it must ensure database layer can also span multiple database nodes
     a. This can be achieved either by using a distributed data store; or
     b. If using traditional RDBMS systems, this can be achieved by ensuring application level sharding (partitioning) is implemented to partition data across many RDBMS nodes. Each shard has the same schema, but holds its own distinct subset of the data. A shard is a data store in its own right, running on a server acting as a storage node.

2.6 Language & Interface

A key requirement for government application being available nationally is their ability to provide the user a local interface and support local language. Therefore the proposed solutions should:

➢ **<u>Be developed on Unicode Compliant Code practices</u>**
The development should be undertaken using Unicode compliant practices.

➢ **<u>Support open standards on language interfaces</u>**
The solution should support open standards on language interfaces.

➢ **<u>Should support multiple language (Indian & Foreign) APIs</u>**
Solution should at least be bi-lingual, but should possess capabilities to be multi-lingual.

➢ **<u>Should support self-learning data dictionaries</u>**
The solution should be support APIs that enable building of transliterated data dictionaries, with preemptive text, so that the user is given the choice to select the nearest match.

2.7 Legacy Integration – Digitization & Migration

The proposed solution should be able to acquire, sort and store the data that has been accumulated for the service being provisioned through multiple legacy ICT solutions. Therefore the proposed solutions should be:

➢ **<u>Able to migrate data through offline user interfaces</u>**
The solution should provide for manual data entry of legacy data (allow for conduct of digitization activities)

➢ **<u>Able to migrate data through be-spoke / product utilities</u>**
Solution should support migration legacy data through be-spoke utilities which allow for data entry, extraction and submission of data into the proposed solution

2.8 Intellectual Property Rights (for Center & State owned applications)

➢ The Intellectual Property Rights for the developed product should invariably reside with the Government Department. This should include the source code, release management artifacts and all other technical and domain related documentation for the developed solution. The licenses procured for the implementation of the existing application may be provided.
    - Release Management Artifacts should include, but not be limited to the following:

- Core Application
- Packaged Installation
- Application Code
- Code Review
- Unit Test Results (Multilingual)
- Test Suites
- UAT Scripts & Test Cases (Multilingual)
- User Interface Testing Results (Multilingual)
- Performance Test Results
- Security Test Results
- Requirement Traceability Matrix
- Deployment Scripts
- Deployment Manual
- User Manual
- Technical Manuals
- Release Notes
- Standard Operating Procedures
- Application Customization Guidelines
- Quality Assessment Report
- UAT Acceptance Benchmarks
- Mapping sheet for defects/functionality and system test cases
- Non-Functional Requirements Compliance sheet
- Backup of the Database before executing the incremental Script
- Incremental Script
- Release note for Database changes done between builds
- DB Code Review Report

- The IPR for the developed product / solution should not be restricted / compromised through any legal interpretation. The solution should clearly be the property of the government department.

3 Cloud Enablement of Applications

3.1 Application Migration to Cloud

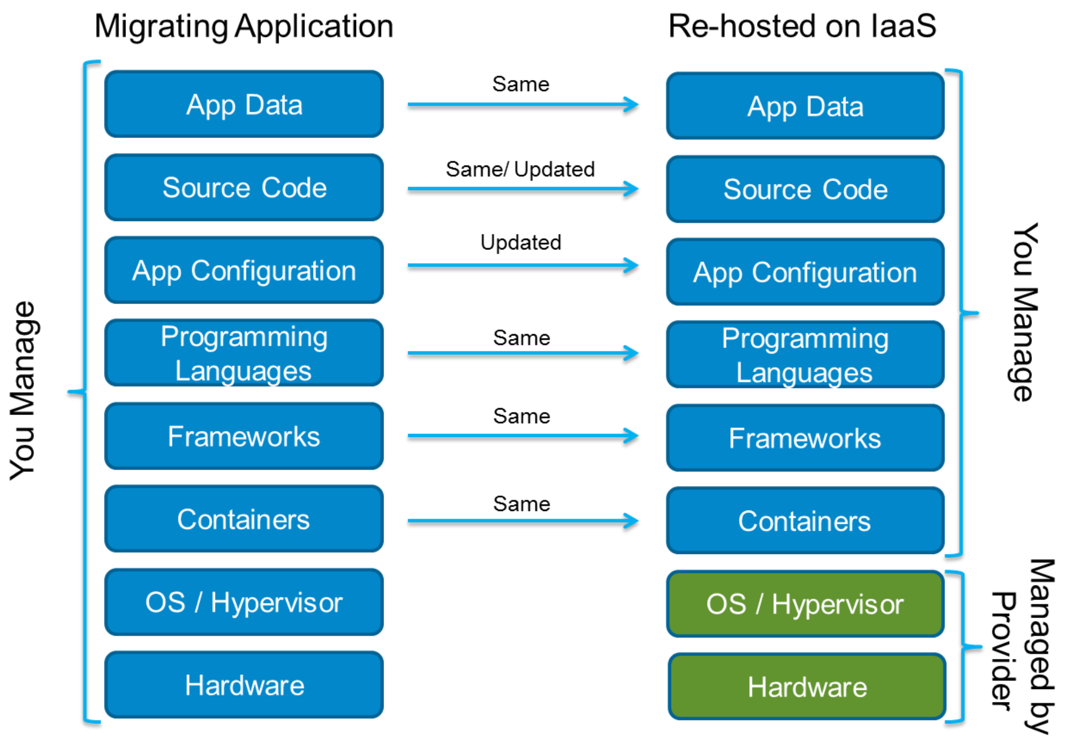
There are five well established approaches to migrate traditional applications to the cloud, these include:

1. **REHOST** on Infrastructure as a Service (IaaS)
2. **REFACTOR** for Platform as a Service (PaaS)
3. **REVISE** for IaaS or PaaS
4. **REBUILD** on PaaS
5. **REPLACE** with Software as a Service (SaaS)

3.1.1 Rehost on IaaS

This approach involves the re-hosting of the application from the existing infrastructure to the cloud infrastructure without making any significant changes to application code-base or the application configuration files. The application Operating System / Hypervisor and the Hardware in addition is managed by the cloud provider.

The following diagram depicts the re-hosting of the application on Infrastructure as a Service.



Rehosting solutions vary from a hosting infrastructure to application virtualization.

Example:

There are ways by which server applications are moved rapidly to and across the cloud, without code change or lock-in. Use of toolkits may be made that allow cloud integrators to handle migrations on behalf of their enterprise accounts.

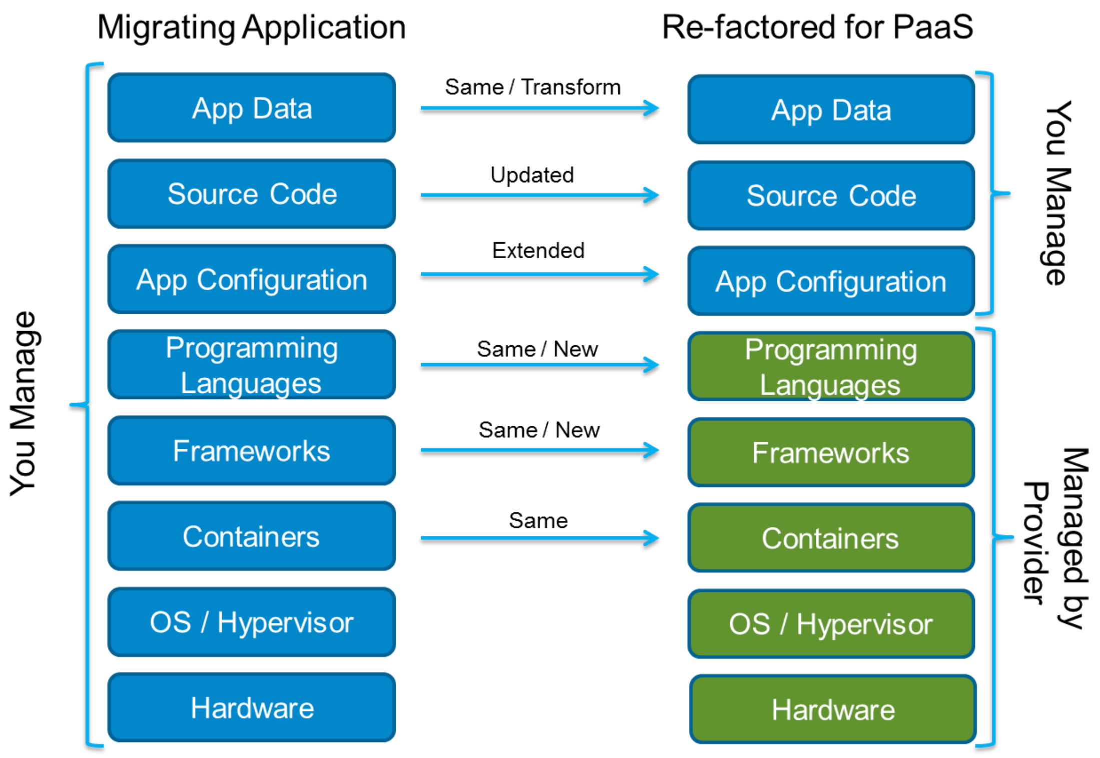
Taking an application-centric approach in moving Server applications, use of application images rather than server or machine images is considered more efficient i.e. encapsulating an application and its dependencies in what is called a “virtual application appliance” (VAA), without a virtual machine (VM). The result is application flexibility that is hypervisor-agnostic, cloud independent, and fast.

3.1.2 Refactor for PaaS

This approach involves the refactoring of the application to use the platform provided by the cloud provider to migrate the application. In this method the programming languages, development frameworks, containers, operating system / hypervisor and the hardware are all managed by the cloud provider.

In addition application data is kept the same or transformed upon migration, application source code is updated, application configurations are extended to service the customer and programming languages and development frameworks are either kept as the same or new ones provided by the platform are used.

The following diagram depicts the refactoring of the application for Platform as a Service.



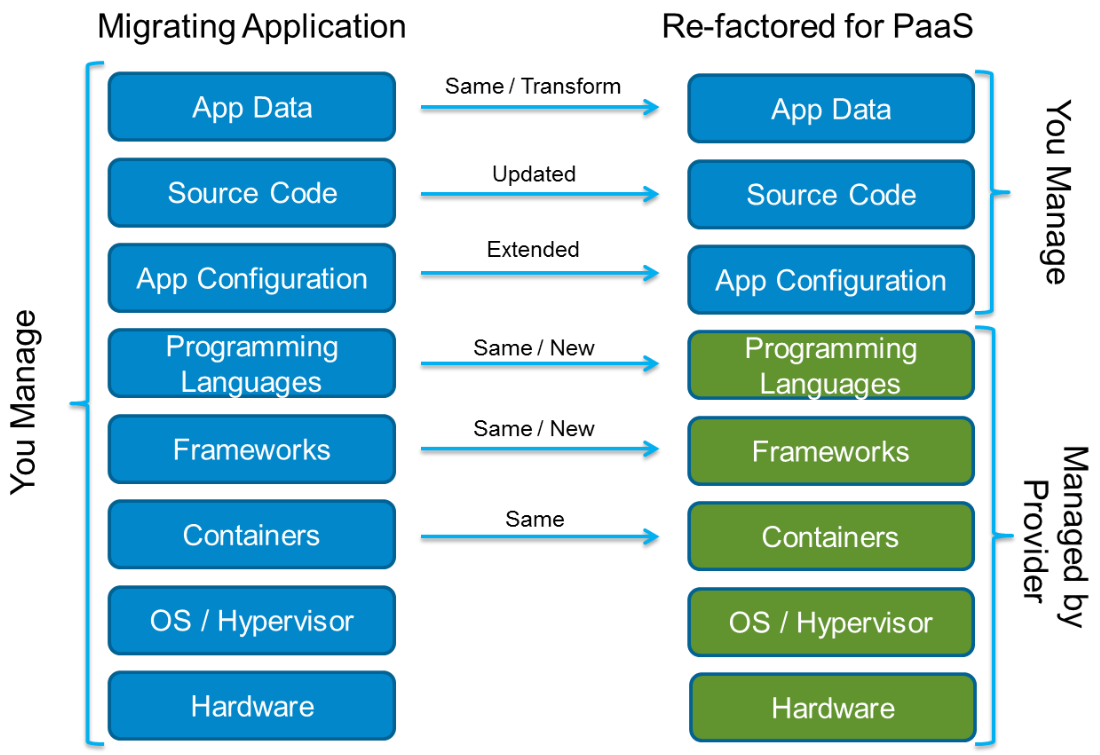
In simple terms, refactoring means doing just enough to migrate an application to a platform-as-a-service cloud offering. It is not merely lifting an application onto a PaaS, because the way the vendors handle security, authentication and data access is generally very different which leads to break open the code in order to use the new frameworks and libraries in the platform.” So application code needs to be refactored to leverage the benefits of the PaaS frameworks

3.1.3 Revise for IaaS or PaaS

This approach involves the migration of the application requiring rebuilding the application utilizing either the infrastructure components of the cloud or utilizing the platform components of the cloud. In this approach similar to the Refactor approach, programming languages, development frameworks, containers, operating system / hypervisor and the hardware are managed by the cloud provider; while the application data, source code and application configuration is managed by the development agency.

In addition the application data is kept as a same or transformed, the source code is updated, new application configurations are required, same or new programming languages, development frameworks and containers are used for revising applications.

The following diagram depicts the revising of the application for Platform as a Service or Infrastructure as a Service.



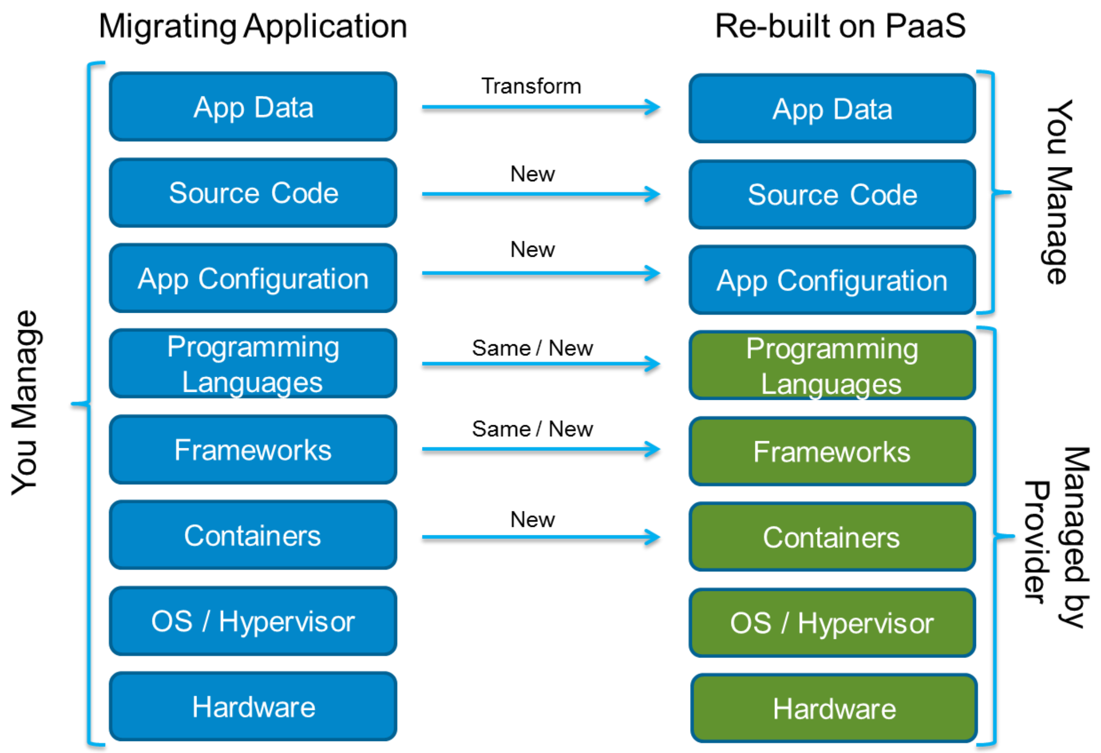
Refactoring does a minimalistic change in the application which is required to move it onto a PaaS system. But in order to reap the maximum benefits from the scalability of cloud infrastructure, the application has to undergo more fundamental changes to the architecture. The development team has to do very significant work in the application to make it cloud optimized. The guiding principle on whether this kind of revision is worth paying for should be the value of the code in question.

3.1.4 Rebuild on PaaS

This approach involves the redevelopment of the application to suite cloud based deployment. Similar to the revise approach, in this approach also the programming languages development frameworks, containers, operating system / hypervisor and the hardware are managed by the cloud provider; while the application data, source code and application configuration is managed by the development agency.

In addition the application data is transformed from the existing infrastructure to the new environment, the source code and application configurations are written / configured anew. The existing or new programming languages and development frameworks from the cloud platform are used.

The following diagram depicts the rebuilding of the application on Platform as a Service.

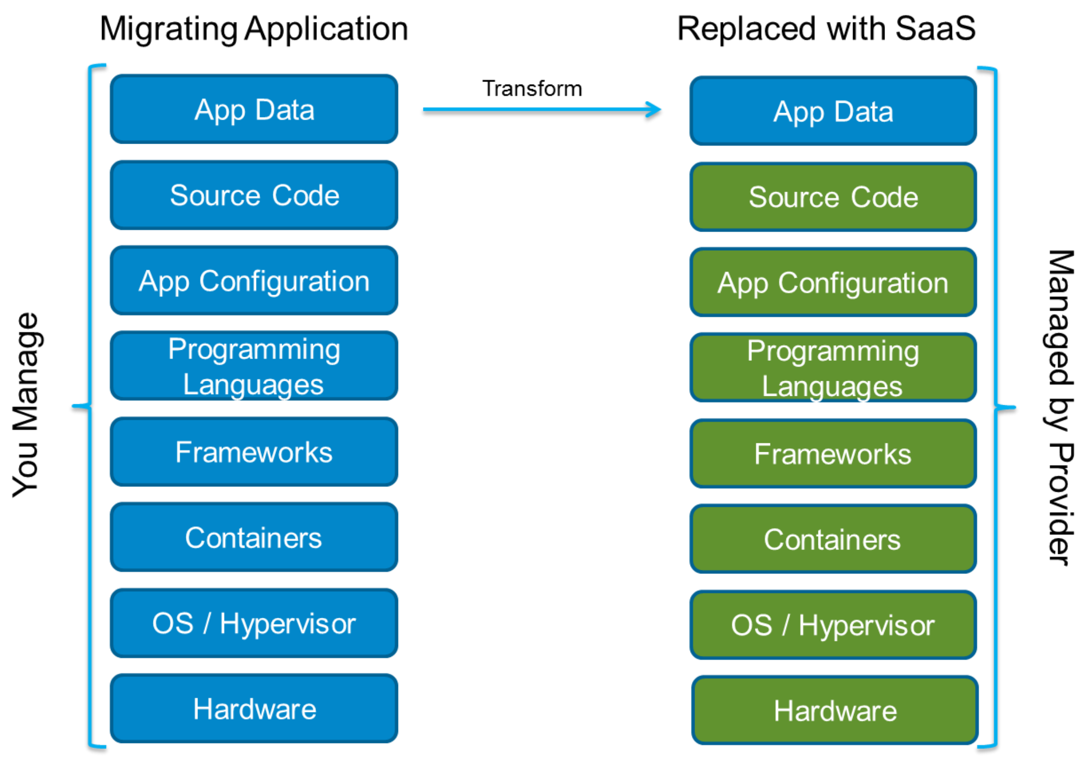


PaaS offerings include facilities for application design, application development, testing, and deployment as well as services such as team collaboration, web service integration, and marshalling, database integration, security, scalability, storage, persistence, state management, application versioning, application instrumentation, and developer community facilitation.

3.1.5 Replace with SaaS

The last approach for migration of the applications to cloud involves replacing the existing application with a new application, which is completely managed by the cloud provider, and is available to the customer on Software as a Service Model. In this approach only the application data from the existing application is transformed to the new application; while all other aspects such as source code, application configuration, programming languages, development frameworks, containers, operating system / hypervisors and hardware are managed by the cloud provider.

The following diagram depicts the replacing of the application with Software as a Service.



In contrast with the other approaches, the replace approach suggests to use the SaaS solutions instead of building the application.

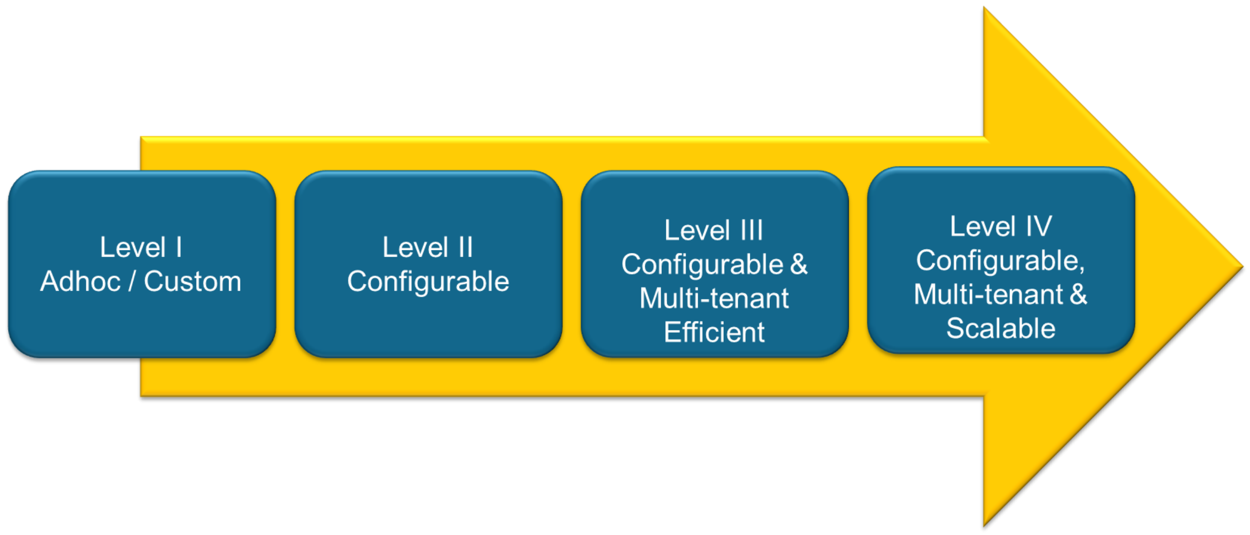
3.2 Software as a Service Characteristics

The following should be considered as key aspects for development of applications planned for deployment on SaaS model:

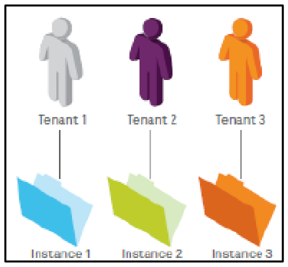
- The application should support **Multi-Tenancy**
- The application should have certain level of **Self-Service Sign-Up**
- The application should be **Scalable** in nature
- The application should be **Stateless** in nature
- The application should support mechanisms to **Measure Service**
- There should be a mechanism in place to support **Unique User Identification & Authentication**
- There should be a mechanism in place to support **Configurability** (UI, Business Logic, Workflow etc.) for each tenant
- There should be functions in place to **Monitor, Configure, & Manage** application & tenants

3.2.1 SaaS Maturity

The following diagram depicts the various stages of SaaS maturity model:

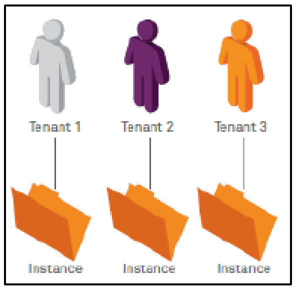


SaaS Maturity Level I



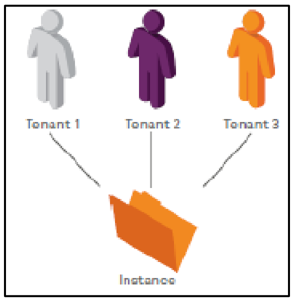
Level I of the SaaS maturity model implies that custom development is undertaken for each tenant and managed separately. In such instances the application development agency has to manage the changes being suggested for each tenant along with version control for each deployed version of the application.

SaaS Maturity Level II



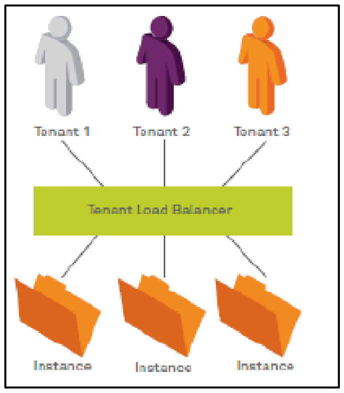
Level II of the SaaS maturity model implies that multiple copies of the same instance are run separately each tenant. In such a model the application development agency has to manage configuration files for each tenant separately and make changes based on the business requirements.

SaaS Maturity Level III



Level III of the SaaS maturity model implies that the single instance of the application is used by all the tenants. Different configuration files are managed through a multi-tenant efficient architecture. The configurations are managed by the tenant themselves and can be changed in run-time.

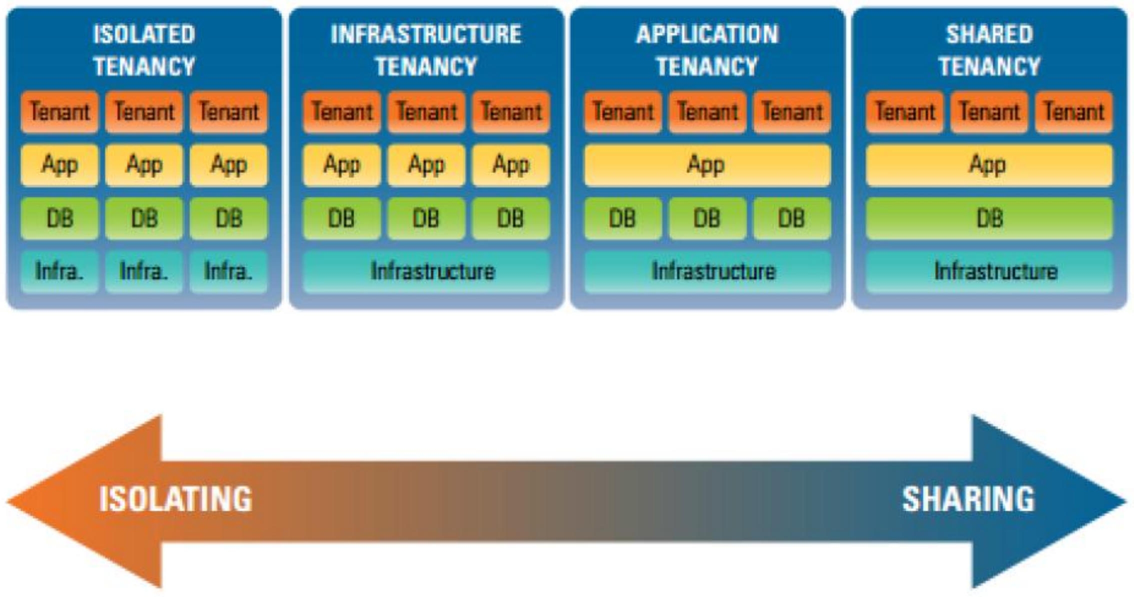
SaaS Maturity Level IV



Level IV of the SaaS maturity model implies that multiple instances of the application can be managed through a tenant load balancer, which creates additional instances based on the load on the application. These instances serve a number of tenants and are based on the configuration files defined for the application.

3.2.2 Multi-tenancy

Multi-tenancy is defined as an architecture in which a single instance of an application serves multiple customers. The following diagram depicts the multi-tenancy continuum from isolated tenancy to shared tenancy utilizing the cloud resources.

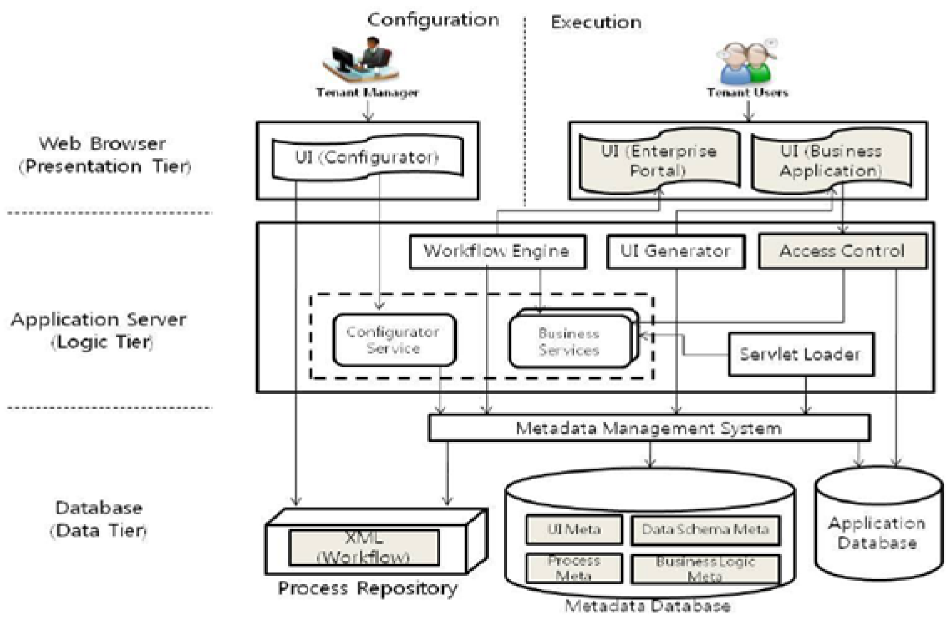


In order to develop multi-tenant application the architecture should be designed and developed in a manner so as to have the following:

- Identification of most granular functionality
- Implementation of functionality as a web service
- Orchestrating each functionality to configure the desired workflows
- Configuration of application workflow using workflow designer
- Execution of application workflow using workflow engines
- All configurable aspects of application should be stored in separate tenant specific metadata database
- UI (User Interface) Customizability
- Segregated data storage (tenant-wise) for protecting access and data isolation amongst various tenants

3.2.3 Designing Configurable Application

The following diagram depicts architectural maturity for all application development envisaged to be made available under the national eGov AppStore. The architecture proposes applications to be built as component stores, which will allow the application to be readily re-usable and scalable, two of the key design aspects for cloud enablement.



Application Tiers

The application should be developed on a multi-tiered architecture to benefit for distributed computing and scaling advantages brought onboard by cloud enablement. These should be primarily split into Presentation, Application and Database Layers. These distinct layers can subsequently be divided into multiple sub-tiers which will allow for greater ease of simultaneous computing capabilities to be made available on persistent infrastructure.

Application Design

The application should be designed in a manner that provides clear distinction between configuration components and execution components. This will allow the application to scale-up in run-time to handle more simultaneous service requests, while at the same time allow the administrators to manage configurations for multiple tenants. Both configuration and execution processes should be developed as stores loosely coupled to allow better access to processes within. Process stores of architecture design should have User Interface Configurator, Service Configurator, Workflow Configurator, Business Logic Configurator, Meta-data Management System, Access Controllers, Run-time Engine & Application Database.

4 Annexure I - Application Self-Assessment Checklist

The following check-list needs to be self-assessed by Application Development / Providing / Provisioning Agencies to be eligible for the proposed national level App Store.

Application Details	
Application Name	
Application Current Version	
Released on	
Certification (if any, done by)	
Provider department details	
Initial cost of application development	
Proposed effort and cost of application customization for Cloud enablement (in case the application is not cloud enabled)	Effort: Man Months Cost: INR

Rating Criteria
Rating 1 – The application is non-compliant and cannot be changed
Rating 2 – The application is presently non-compliant and would require more than 50% of the original development effort to change the application
Rating 3 – The application is presently non-compliant and would require between 30% to 49% of the original development effort to change the application
Rating 4 – The application is presently non-compliant and would require between 10% to 29% of the original development effort to change the application
Rating 5 – The application is presently non-compliant and would require less than 10% of the original development effort to change the application
Rating 6 – The application is fully compliant on the component

S No	**Compliance Component**	**Rating (between 1 – 6)**	**Weight**	**Provide Details**
1.	Is master data, screen labels, user alerts, reports, dashboards configurable (not hardcoded in the application)		4	
2.	Are business rules configurable (managed using rule-set engines and not hardcoded in application)		4	
3.	Are workflows configurable (not hardcoded in the application)		4	
4.	Is user interface (application screens - look & feel) customizable		4	
5.	Can the application be integrated with SMS Gateway and/or Messaging Systems		5	

6.	Can the application be integrated with other external applications / components / services, implies: (i) Application can be integrated with payment gateway, third party applications etc. (ii) Application can be integrated with third party components such as Identity & Access Management Tools etc.		4	
7.	Can the application be integrated with other external applications / components / services, implies: The application has defined interface points and mechanism for data exchange		4	
8.	In case required, is the application developed in such a manner that it can support offline data entry & synchronization mechanisms.		2	
9.	Is the application designed on a ‘Multi-Tiered’ architecture, implies: (i) Are tiers configurable with minimal effect to other tiers (ii) Is there clear segregation of duties between presentation, business and database layers		4	
10.	Is the application scalable		5	
11.	Is the application deployable on multiple platforms		1	
12.	Is the application developed on Service Oriented Architecture		3	
13.	Is the application deployable on multiple databases		3	
14.	Can the application be deployed as packaged installation and creates verification log for the installation		2	
15.	Does the application have multilingual capabilities, implies: that the application is UNICODE compliant		3	
16.	Can new features be added in the application from a remote central location, implies: application supports automated patch management		3	

17.	Is the application developed in a manner that in case newer versions of the core application are released, it does not affect the integrated components to the core application.		4	
18.	Are the application release management, configuration management and version management clearly articulated with well-defined policies, implies: project artifacts such as SRS, FRS, RTM etc. are available.		5	
19.	Is the application developed in a manner that it is multiple browser compatible including backward compatibility of bowsers		3	
20.	If required, is the application accessible through multiple clients including handheld devices, tablets, smartphones etc.		3	
21.	Does the application support Multi-Tenancy?		5	
22.	Is the application designed to store configuration files outside the application & allowed to be changed in run-time?		4	
23.	Is the application designed to be Scale-Out or Scale-In?		5	
24.	Is the application designed to be Stateless?		5	
25.	Does the application assume any specific infrastructure dependency?		3	

5 Definitions

S No.	Keyword	Definition
1	**Product**	A well-developed product is defined as an integrated packaged solution which is available to the end user for ready to use. This proposed solution may require configurations to adapt to the business processes of the end user. Also the product should only allow for minor customizations to localize the solution for end user department / agency.
2	**Service Contract**	A service contract is defined as a physical contract which is signed between the service provider and service consumer. In context of the project service provider would imply, the App providing department / agency / stakeholder and the service consumer would imply the department / agency that would be using the product.
3	**Loose Coupling of Services**	It is explained as concept wherein the individual services designed, developed and integrated as part of the solution are loosely coupled in the solution, so that in case another solution, service wants to use / re-use the service they are able to do so.
4	**Service Reusability**	It is explained as a concept wherein the individual services designed under a solution for a particular business unit can be reused by configuring certain parameters to suite the business requirements of another business / functional unit.
5	**Service Abstraction**	It is explained as a concept wherein the application development takes account of the reusability factor and allows for the developed service and its components to available for other services / solutions. This would also ensure that there is transparency in the development of the application and that features can be reused as part of other services.
6	**Service Discoverability**	It is explained as a concept wherein services loosely coupled under a solution are easily identifiable, so that other services / solutions do not replicate the logic defined in another service. This would help in weeding out redundant logic in the developed application code and make its performance optimized.
7	**Service Autonomy**	It is explained as a concept wherein the services which are loosely coupled, discoverable and can be easily abstracted are single as far as their development is concerned. It implies that the functional logic that has been developed for the service is not required to be replicated in as part of another development in the same solution.
8	**Service Location**	It is explained as a concept wherein a service that has been developed under a solution, is not only discoverable to other solutions / services, but is also

S No.	Keyword	Definition
	Transparency	denotes clearly its location, such as locally available, available on the networked data center or available on the cloud. This would ensure that the end user / solution are able to use the service independent of its physical location.
9	**Service Granularity**	It is explained as concept wherein each service is developed in a manner that it easily understood, used by other services, and that the actions performed under the services are transparent. This would include the maintaining the activity logs – including user information, delta change, time stamps,
10	**Platform & Database Agnostic**	It is explained as concept wherein the solution is developed in a manner that it has ability to integrate with other systems, developed diverse platforms, in addition to being interoperable with multiple databases available. This would ensure that service delivery is the prime forte of the solution, while technology and infrastructure support its delivery.