[AUDIT 2](#_Toc16493833)

[Vocabulary 2](#_Toc16493834)

[Inshoring SOX functions 2](#_Toc16493835)

[RPA 2](#_Toc16493836)

[SKILLS 2](#_Toc16493837)

[AWS 2](#_Toc16493838)

[HADOOPP 2](#_Toc16493839)

[MONGODB 2](#_Toc16493840)

[SPARK 2](#_Toc16493841)

[Core 2](#_Toc16493842)

[SQL 2](#_Toc16493843)

[Stream 2](#_Toc16493844)

[STREAM 2](#_Toc16493845)

[APACHE KAFKA 2](#_Toc16493846)

[DJANGO 3](#_Toc16493847)

[SQL 3](#_Toc16493848)

[Oracle PL/SQL (Procedural Language for SQL) 3](#_Toc16493849)

[Microservices 4](#_Toc16493850)

[API 5](#_Toc16493851)

[12 FACTORS 5](#_Toc16493852)

[REST 5](#_Toc16493853)

[DOCKER 5](#_Toc16493854)

[KUBERNETES 5](#_Toc16493855)

[PYTHON 6](#_Toc16493856)

[LUIGI 6](#_Toc16493857)

[DASK 6](#_Toc16493858)

[PANDAS 6](#_Toc16493859)

[SQL ALCHEMY 6](#_Toc16493860)

[R 6](#_Toc16493861)

[JAVASCRIPT 6](#_Toc16493862)

[SCIKIT-LEARN 6](#_Toc16493863)

[TENSORFLOW 6](#_Toc16493864)

[PYTORCH 6](#_Toc16493865)

[MACHINE LEARNING 6](#_Toc16493866)

[Regression 6](#_Toc16493867)

[Boost 6](#_Toc16493868)

[Deep Learning 6](#_Toc16493869)

[POWER BI 6](#_Toc16493870)

[ORACLE 6](#_Toc16493871)

[SQL SERVER 6](#_Toc16493872)

[EXHIBIT 7](#_Toc16493873)

[Audit Lifecycle 7](#_Toc16493874)

[AGILE AUDIT 7](#_Toc16493875)

[APPLICATION DEVELOPMENT AGILE 9](#_Toc16493876)

[NOVA Target State 10](#_Toc16493877)

[IDR Data Governance 11](#_Toc16493878)

[IDR Data Quality Lifecycle 11](#_Toc16493879)

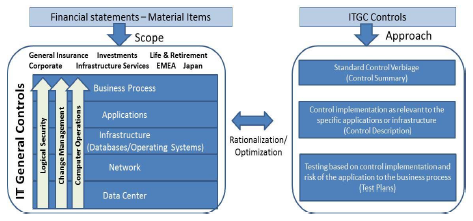
[IDR State Architecture 13](#_Toc16493880)

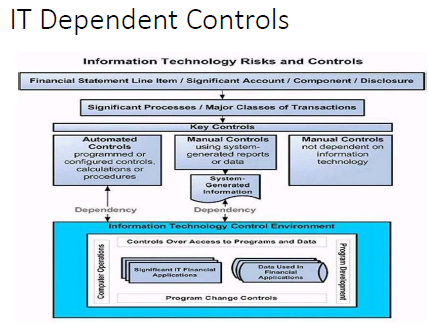
# AUDIT

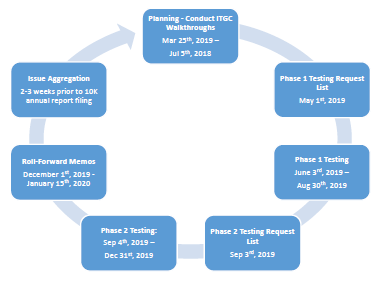
## Vocabulary

⬩**ELC** (Entity Level Control) ⬩**OSP** (Outside Service Providers) Oversight ⬩**LU**= Least (privileged) User Access ⬩**NTE**= nature, timing and extent ⬩**SSAE16 SOC1, SOC2** ⬩**Audit Writing 5C’s**: Criteria (what should be), Condition (the current state), Cause (the reason for the difference), Consequence (effect), Corrective action plans/recommendations.

## Inshoring SOX functions







## RPA

⬩Initiated from Accounts Payable (NJ), DBA <Tax, FIS Billing, FP&A Planning&Analysis, Comptrollers> ⬩Consultant: GENPACT ⬩Process 1: Batch creation + Monies moving ⬩Process 2: VOID/STOP Payment (Reversal) ⬩Systems AWD (Automated Work Distributor Imaging & Workflow), OASYS PrC (Fixed annuity Admin) ⬩ RPA: OPENSPAN PEGASYSTEMS

# SKILLS

## AWS

•**Elastic Compute Cloud (EC2)**, a service for provisioning computing resources on

demand •**Simple Storage Service (S3)**, online storage for opaque data •**Elastic Block Store (EBS)**, persistent disk-like storage for EC2 instances, in 2008 •**Elastic MapReduce (EMR)**, a service providing Hadoop-like clusters for running MapReduce (and later Apache Hive and Apache Pig) jobs, in 2009 •**Relational Database Service (RDS)**, a service for managing relational database server instances running in AWS, also in 2009

⬩**Instance types**: heavy compute capability, vast storage, economy, or simply general-purpose use ⬩**Availability zones** independent within a region, but faster interconnections ⬩**Temporary instance** can disappear after some time ⬩**Images** what instances are running: operating system type and version, the software packages that are available, and applications that are installed. These considerations are all bundled up into images ⬩**CIDR** (Classless Inter-Domain Routing) [notation](https://erikberg.com/notes/networks.html)

## HADOOPP

## MONGODB

## SPARK

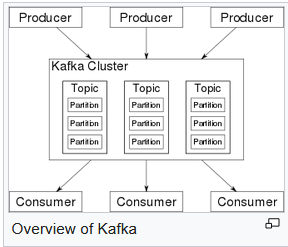
#### Core

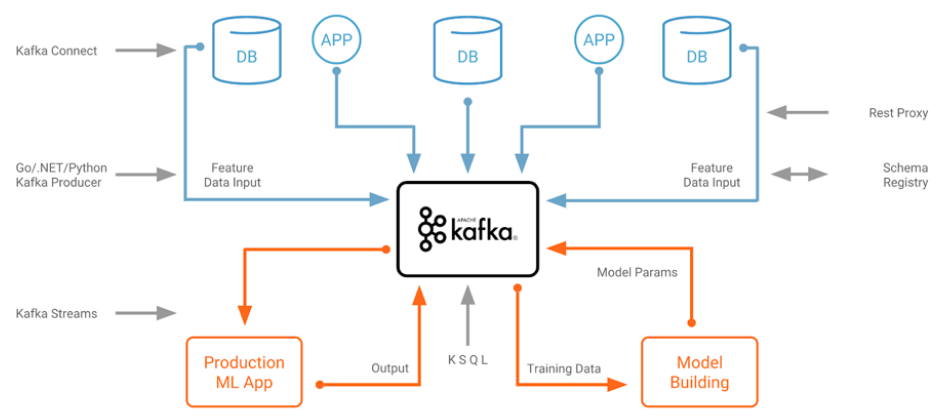
#### SQL

#### Stream

## STREAM

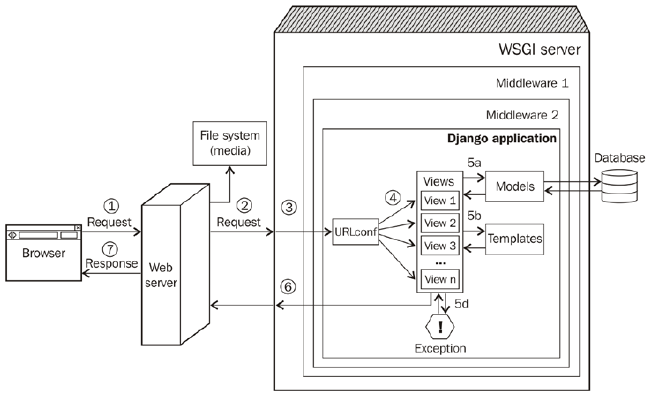
#### APACHE KAFKA

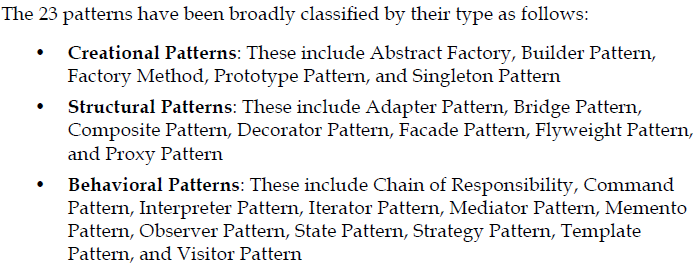


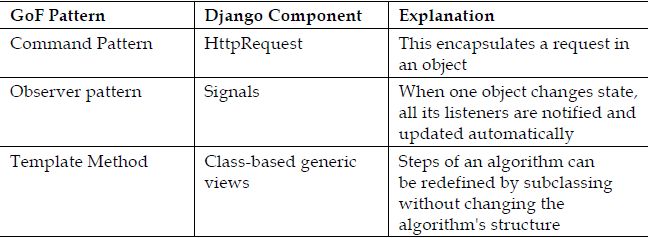


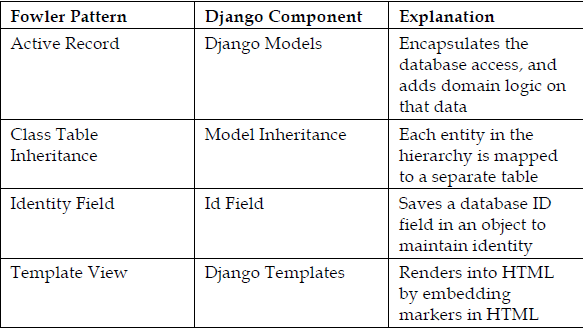
Kafka stores key-value messages that come from arbitrarily many processes called *producers*. The data can be partitioned into different "partitions" within different "topics". Within a partition, messages are strictly ordered by their offsets (the position of a message within a partition), and indexed and stored together with a timestamp. Other processes called "consumers" can read messages from partitions. For stream processing, Kafka offers the Streams API that allows writing Java applications that consume data from Kafka and write results back to Kafka. Apache Kafka also works with external stream processing systems such as [Apache Apex](https://en.wikipedia.org/wiki/Apache_Apex), [Apache Flink](https://en.wikipedia.org/wiki/Apache_Flink), [Apache Spark](https://en.wikipedia.org/wiki/Apache_Spark), and [Apache Storm](https://en.wikipedia.org/wiki/Apache_Storm). Kafka runs on a cluster of one or more servers (called brokers), and the partitions of all topics are distributed across the cluster nodes. Additionally, partitions are replicated to multiple brokers. This architecture allows Kafka to deliver massive streams of messages in a fault-tolerant fashion and has allowed it to replace some of the conventional messaging systems like [Java Message Service](https://en.wikipedia.org/wiki/Java_Message_Service) (JMS), [Advanced Message Queuing Protocol](https://en.wikipedia.org/wiki/Advanced_Message_Queuing_Protocol) (AMQP), etc. Since the 0.11.0.0 release, Kafka offers *transactional writes*, which provide exactly-once stream processing using the Streams API. Kafka supports two types of topics: ➊*Regular topics* can be configured with a retention time or a space bound. If there are records that are older than the specified retention time or if the space bound is exceeded for a partition, Kafka is allowed to delete old data to free storage space. By default, topics are configured with a retention time of 7 days, but it's also possible to store data indefinitely. ➋For *compacted topics*, records don't expire based on time or space bounds. Instead, Kafka treats later messages as updates to older message with the same key and guarantees never to delete the latest message per key. Users can delete messages entirely by writing a so-called tombstone message with null-value for a specific key. Four major APIs: ➊ **Producer API** Permits application to publish streams of records ➋ **Consumer API** Permits application to subscribe to topics and processes streams of records ➌ **Connector API** Executes the reusable producer and consumer APIs that can link the topics to the existing applications ➍ **Streams API** Converts the input streams to output and produces the result. The consumer and producer APIs build on top of the Kafka messaging protocol and offer a reference implementation for Kafka consumer and producer clients in Java. The underlying messaging protocol is a [binary protocol](https://en.wikipedia.org/wiki/Binary_protocol) that developers can use to write their own consumer or producer clients in any programming language. This unlocks Kafka from the [Java Virtual Machine](https://en.wikipedia.org/wiki/Java_Virtual_Machine) (JVM) eco-system. A list of available non-Java clients is maintained in the Apache Kafka wiki.

## DJANGO









**Structural patterns**: ⬩normalized models ⬩model mixins ⬩user profiles ⬩service objects

**Retrieval patterns**: ⬩property field ⬩custom model managers **View patterns**: ⬩acces controlled ⬩context enhancers ⬩services **Template patterns**: ⬩inheritance ⬩active link

## SQL

#### Oracle PL/SQL (Procedural Language for SQL)

##### PL/SQL anonymous block

The basic unit of a PL/SQL source program is the block, which groups together related declarations and statements. A PL/SQL block is defined by the keywords DECLARE, BEGIN, EXCEPTION, and END. These keywords divide the block into a declarative part, an executable part, and an exception-handling part. The declaration section is optional and may be used to define and initialize constants and variables. If a variable is not initialized then it defaults to [NULL](https://en.wikipedia.org/wiki/Null_(SQL)) value. The optional exception-handling part is used to handle run time errors. Only the executable part is required. A block can have a label.

**<<label>>** -- this is optional

**DECLARE**

-- this section is optional

number1 NUMBER(**2**);

number2 number1%TYPE := **17**; -- value default

text1 VARCHAR2(**12**) := 'Hello world';

text2 DATE := SYSDATE; -- current date and time

**BEGIN**

-- this section is mandatory, must contain at least one executable statement

**SELECT** street\_number

**INTO** number1

**FROM** address

**WHERE** **name** = 'INU';

**EXCEPTION**

-- this section is optional

**WHEN** OTHERS **THEN**

DBMS\_OUTPUT**.**PUT\_LINE('Error Code is ' || TO\_CHAR(sqlcode));

DBMS\_OUTPUT**.**PUT\_LINE('Error Message is ' || sqlerrm);

**END**;

The symbol := functions as an [assignment operator](https://en.wikipedia.org/wiki/Assignment_operator) to store a value in a variable.

Blocks can be nested – i.e., because a block is an executable statement, it can appear in another block wherever an executable statement is allowed. A block can be submitted to an interactive tool (such as SQL\*Plus) or embedded within an Oracle Precompiler or [OCI](https://en.wikipedia.org/wiki/Oracle_Call_Interface) program. The interactive tool or program runs the block once. The block is not stored in the database, and for that reason, it is called an anonymous block (even if it has a label).

##### Function

PL/SQL function is generally used to compute and return a single value. This returned value may be a single scalar value (such as a number, date or character string) or a single collection (such as a nested table or array). User-defined functions supplement the built-in functions provided by Oracle Corporation.

**CREATE** **OR** **REPLACE** **FUNCTION** <function\_name> [(**input**/output variable declarations)] **RETURN** return\_type

[AUTHID <**CURRENT\_USER** | **DEFINER**>] <**IS**|**AS**> -- heading part

amount number; -- declaration block

**BEGIN** -- executable part

<PL/SQL block **with** **return** **statement**>

**RETURN** <return\_value>;

[**Exception**

**none**]

**RETURN** <return\_value>;

**END**;

Pipe-lined table functions return collections[[4]](https://en.wikipedia.org/wiki/PL/SQL#cite_note-4) and take the form:

**CREATE** **OR** **REPLACE** **FUNCTION** <function\_name> [(**input**/output variable declarations)] **RETURN** return\_type

[AUTHID <**CURRENT\_USER** | **DEFINER**>] [<**AGGREGATE** | PIPELINED>] <**IS**|**USING**>

[declaration block]

**BEGIN**

<PL/SQL block **with** **return** **statement**>

PIPE **ROW** <**return** **type**>;

**RETURN**;

[**Exception**

**exception** block]

PIPE **ROW** <**return** **type**>;

**RETURN**;

**END**;

A function should only use the default IN type of parameter. The only out value from the function should be the value it returns.

##### Procedure

Like functions, procedures are named program units that can be invoked repeatedly. The primary difference is that **functions can be used in a SQL statement whereas procedures cannot**. Another difference is that the procedure can return multiple values whereas a function should only return a single value.

The procedure begins with a mandatory heading part to hold the procedure name and optionally the procedure parameter list. Next come the declarative, executable and exception-handling parts, as in the PL/SQL Anonymous Block.

**CREATE** **PROCEDURE** create\_email\_address ( -- Procedure heading part begins

name1 VARCHAR2,

name2 VARCHAR2,

company VARCHAR2,

email **OUT** VARCHAR2

) -- Procedure heading part ends

**AS**

-- Declarative part begins (optional)

error\_message VARCHAR2(**30**) := 'Email address is too long.';

**BEGIN** -- Executable part begins (mandatory)

email := name1 || '.' || name2 || '@' || company;

**EXCEPTION** -- Exception-handling part begins (optional)

**WHEN** VALUE\_ERROR **THEN**

DBMS\_OUTPUT**.**PUT\_LINE(error\_message);

**END** create\_email\_address;

The example above shows a standalone procedure - this type of procedure is created and stored in a database schema using the CREATE PROCEDURE statement. A procedure may also be created in a PL/SQL package - this is called a Package Procedure. A procedure created in a PL/SQL anonymous block is called a nested procedure. The standalone or package procedures, stored in the database, are referred to as "[stored procedures](https://en.wikipedia.org/wiki/Stored_procedure)". Procedures can have three types of parameters: IN, OUT and IN OUT.

1. An IN parameter is used as input only. An IN parameter is passed by reference, though it can be changed by the inactive program.
2. An OUT parameter is initially NULL. The program assigns the parameter value and that value is returned to the calling program.
3. An IN OUT parameter may or may not have an initial value. That initial value may or may not be modified by the called program. Any changes made to the parameter are returned to the calling program by default by copying but - with the NO-COPY hint - may be passed [by reference](https://en.wikipedia.org/wiki/Call_by_reference).

PL/SQL also supports external procedures via the Oracle database's standard ext-proc process. [[5]](https://en.wikipedia.org/wiki/PL/SQL#cite_note-5)

##### Package

Packages are groups of conceptually linked functions, procedures, variables, PL/SQL table and record TYPE statements, constants, cursors, etc. The use of packages promotes re-use of code. Packages are composed of the package specification and an optional package body. The specification is the interface to the application; it declares the types, variables, constants, exceptions, cursors, and subprograms available. The body fully defines cursors and subprograms, and so implements the specification. Two advantages of packages are:

1. Modular approach, encapsulation/hiding of business logic, security, performance improvement, re-usability. They support [object-oriented programming](https://en.wikipedia.org/wiki/Object-oriented_programming) features like function overloading and encapsulation.
2. Using package variables one can declare session level (scoped) variables since variables declared in the package specification have a session scope.

##### Trigger

A [database trigger](https://en.wikipedia.org/wiki/Database_trigger) is like a stored procedure that Oracle Database invokes automatically whenever a specified event occurs. It is a named PL/SQL unit that is stored in the database and can be invoked repeatedly. Unlike a stored procedure, you can enable and disable a trigger, but you cannot explicitly invoke it. While a trigger is enabled, the database automatically invokes it—that is, the trigger fires—whenever its triggering event occurs. While a trigger is disabled, it does not fire. You create a trigger with the CREATE TRIGGER statement. You specify the triggering event in terms of triggering statements, and the item they act on. The trigger is said to be created on or defined on the item—which is either a table, a view, a schema, or the database. You also specify the timing point, which determines whether the trigger fires before or after the triggering statement runs and whether it fires for each row that the triggering statement affects. If the trigger is created on a table or view, then the triggering event is composed of DML statements, and the trigger is called a DML trigger. If the trigger is created on a schema or the database, then the triggering event is composed of either DDL or database operation statements, and the trigger is called a system trigger. An INSTEAD OF trigger is either: A DML trigger created on a view or a system trigger defined on a CREATE statement. The database fires the INSTEAD OF trigger instead of running the triggering statement.

###### Purpose of triggers

⬩Generating derived column values automatically ⬩Enforcing referential integrity ⬩Event logging/ storing information on table access ⬩Auditing ⬩Synchronous replication of tables ⬩Security authorizations ⬩Preventing invalid transactions

##### Array handling

PL/SQL refers to [arrays](https://en.wikipedia.org/wiki/Array_data_type) as "collections". Three types: ➊ [Associative arrays](https://en.wikipedia.org/wiki/Associative_array) (Index-by tables) ➋Nested tables ➌Varrays (variable-size arrays). Programmers must specify an upper limit for varrays, but need not for index-by tables or for nested tables. The language includes several collection [methods](https://en.wikipedia.org/wiki/Method_(computer_science)) used to manipulate collection elements: for example FIRST, LAST, NEXT, PRIOR, EXTEND, TRIM, DELETE, etc. Index-by tables can be used to simulate associative arrays, as in this [example of a memo function for Ackermann's function in PL/SQL](https://en.wikipedia.org/wiki/Ackermann_function#cite_note-10).

###### Associative arrays (index-by tables)

With index-by tables, the array can be indexed by numbers or strings. It parallels a [Java](https://en.wikipedia.org/wiki/Java_(programming_language)) *map*, which comprises key-value pairs. There is only one dimension and it is unbounded.

###### Nested tables

With nested tables the programmer needs to understand what is nested. Here, a new type is created that may be composed of a number of components. That type can then be used to make a column in a table, and nested within that column are those components.

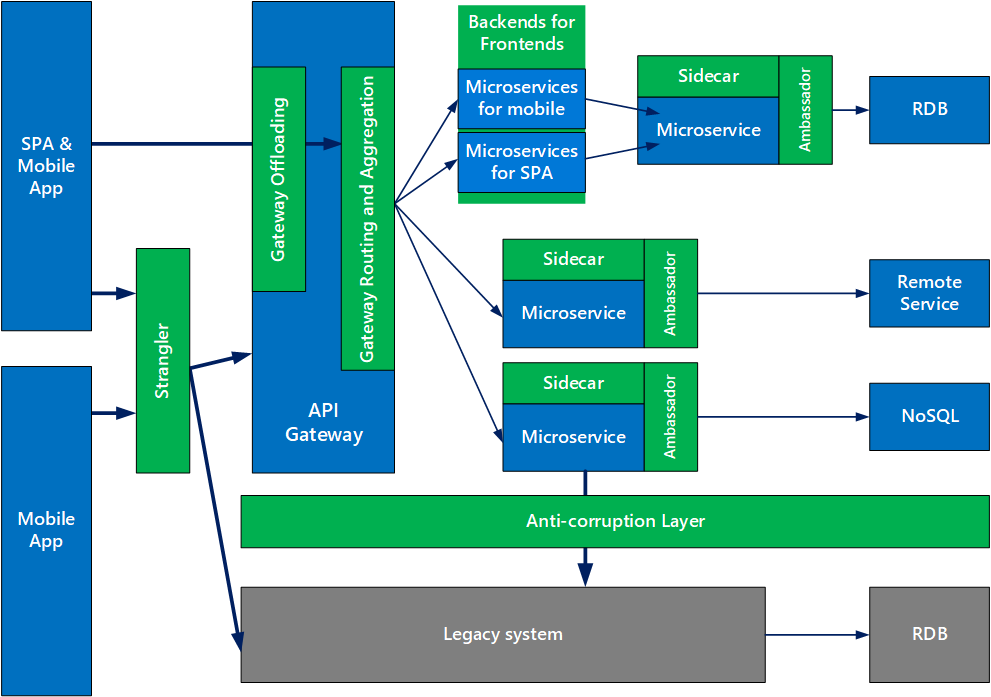
###### Varrays (variable-size arrays)

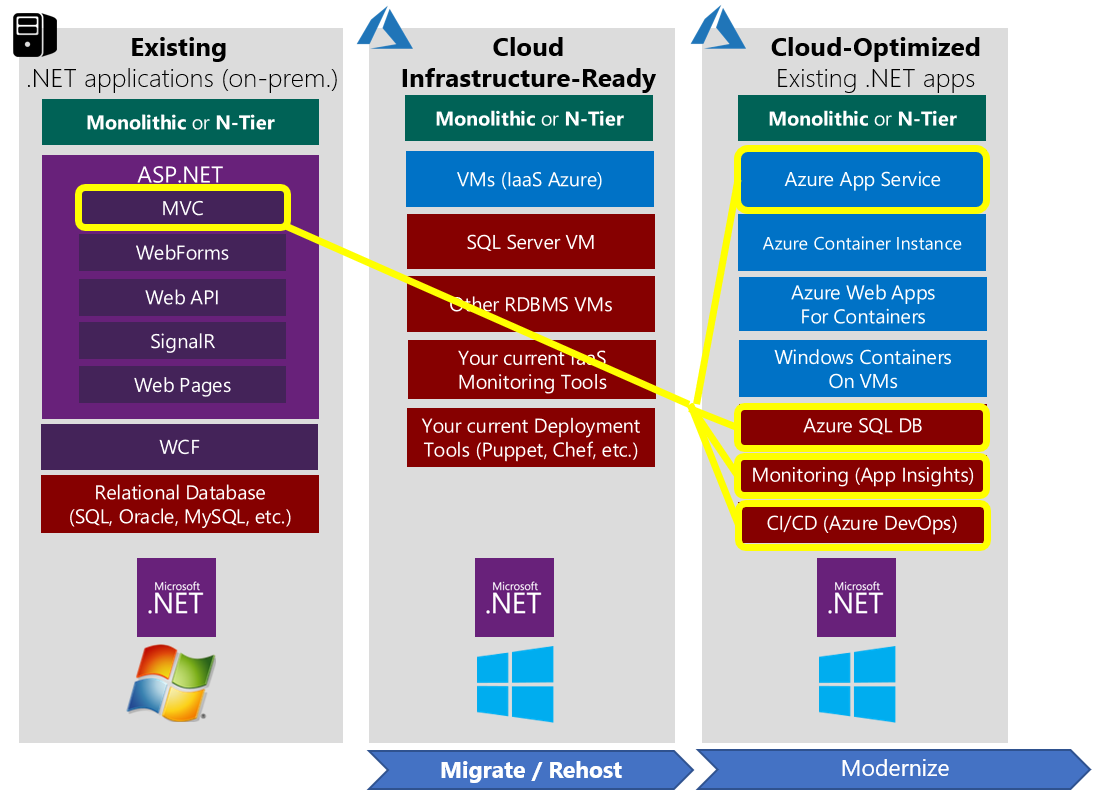
With Varrays you need to understand that the word "variable" in the phrase "variable-size arrays" doesn't apply to the size of the array in the way you might think that it would. The size the array is declared with is in fact fixed. The number of elements in the array is variable up to the declared size. Arguably then, variable-sized arrays aren't that variable in size.

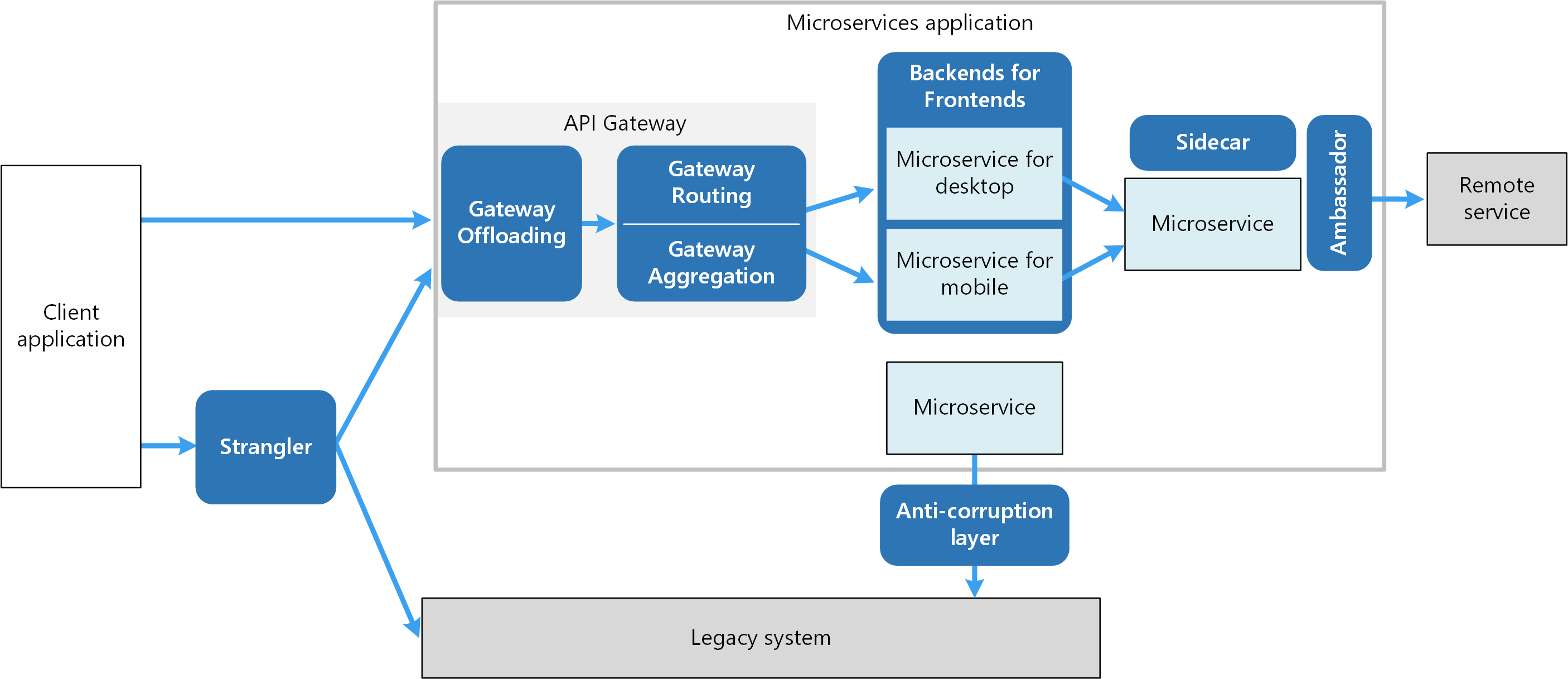
##### Cursors

A [cursor](https://en.wikipedia.org/wiki/Cursor_(databases)) is a mechanism, pointer to a private SQL area that stores information coming from a SELECT or data manipulation language (DML) statement (INSERT, UPDATE, DELETE, or MERGE). A [cursor](https://en.wikipedia.org/wiki/Cursor_(databases)) holds the rows (one or more) returned by a SQL statement. The set of rows the [cursor](https://en.wikipedia.org/wiki/Cursor_(databases)) holds is referred to as the active set.[[7]](https://en.wikipedia.org/wiki/PL/SQL#cite_note-7) A [cursor](https://en.wikipedia.org/wiki/Cursor_(databases)) can be explicit or implicit. In a FOR loop, an explicit cursor shall be used if the query will be reused, otherwise an implicit cursor is preferred. If using a cursor inside a loop, use a FETCH is recommended when needing to bulk collect or when needing dynamic SQL.

## Microservices







* [**Ambassador**](https://docs.microsoft.com/azure/architecture/patterns/ambassador) can be used to offload common client connectivity tasks such as monitoring, logging, routing, and security (such as TLS) in a language agnostic way.
* [**Anti-corruption layer**](https://docs.microsoft.com/azure/architecture/patterns/anti-corruption-layer) implements a façade between new and legacy applications, to ensure that the design of a new application is not limited by dependencies on legacy systems.
* [**Backends for Frontends**](https://docs.microsoft.com/azure/architecture/patterns/backends-for-frontends) creates separate backend services for different types of clients, such as desktop and mobile. That way, a single backend service doesn’t need to handle the conflicting requirements of various client types. This pattern can help keep each microservice simple, by separating client-specific concerns.
* [**Bulkhead**](https://docs.microsoft.com/azure/architecture/patterns/bulkhead) isolates critical resources, such as connection pool, memory, and CPU, for each workload or service. By using bulkheads, a single workload (or service) can’t consume all of the resources, starving others. This pattern increases the resiliency of the system by preventing cascading failures caused by one service.
* [**Gateway Aggregation**](https://docs.microsoft.com/azure/architecture/patterns/gateway-aggregation) aggregates requests to multiple individual microservices into a single request, reducing chattiness between consumers and services.
* [**Gateway Offloading**](https://docs.microsoft.com/azure/architecture/patterns/gateway-offloading) enables each microservice to offload shared service functionality, such as the use of SSL certificates, to an API gateway.
* [**Gateway Routing**](https://docs.microsoft.com/azure/architecture/patterns/gateway-routing) routes requests to multiple microservices using a single endpoint, so that consumers don't need to manage many separate endpoints.
* [**Sidecar**](https://docs.microsoft.com/azure/architecture/patterns/sidecar) deploys helper components of an application as a separate container or process to provide isolation and encapsulation.
* [**Strangler**](https://docs.microsoft.com/azure/architecture/patterns/strangler) supports incremental migration by gradually replacing specific pieces of functionality with new services.

#### API

#### 12 FACTORS

Cloud-friendly applications embrace elastic scalability, ephemeral filesystems, statelessness, and treating everything as a service ⇨ Can scale and deploy rapidly ⬩**Codebase** One codebase tracked in revision control, many deploys ⬩**Dependencies**. Explicitly declare and isolate dependencies ⬩**Configuration**: Store configuration in the environment ⬩**Backing Services**: Treat backing services as attached resources ⬩**Build, release, run:** Separate build and run stages ⬩**Processes**: Execute app as one or more stateless processes ⬩**Port binding:** Export services via port binding ⬩**Concurrency**: Scale out via the process model

⬩**Disposability**: Maximize robustness with fast startup and graceful shutdown ⬩**Dev/prod parity:** Keep development, staging, and production as similar as possible ⬩**Logs**: Treat logs as event streams ⬩**Admin processes**: Run admin/management tasks as one-off processes

#### REST

REST constraints are design rules that are applied to establish the distinct characteristics

of the REST architectural style. Each constraint is a pre-determined design decision that can have both positive and negative impacts. The intent is for the positives of each constraint to balance out the negatives to produce an overall architecture that resembles the best features of the Web. The formal REST constraints are:

• **Client-Server** {393} requires that a service offer one or more capabilities and listen for requests on these capabilities. A consumer invokes a capability by sending the corresponding request message, and the service either rejects the request or performs the requested task before sending a response message back to the consumer. Exceptions that prevent the task from proceeding are raised back to the consumer, and the consumer is responsible for taking corrective action

• **Stateless** {395} The communication between service consumer (client) and service (server) must be stateless between requests. This means that each request from a service consumer

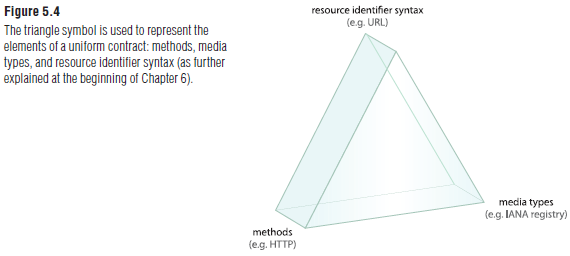
should contain all the necessary information for the service to understand the meaning of the request, and all session state data should then be returned to the service consumer at the end of each request. Statelessness is one of the primary influences over service contract design in REST-style architecture. It imposes significant restrictions on the kinds of communication allowed between services and their consumers in order to achieve its design goals. The application of the Cache {398} and Layered System {404} constraints helps to compensate for limitations resulting from Stateless {395}.

• **Cache** {398} Response messages from the service to its consumers are explicitly labeled as cacheable or non-cacheable. This way, the service, the consumer, or one of the intermediary middleware components can cache the response for reuse in later requests.

The Cache {398} constraint builds upon Client-Server {393} and Stateless {395} with a requirement that responses are implicitly or explicitly labeled as cacheable or noncacheable.

Requests are passed through a cache component, which may reuse previous responses to partially or completely eliminate some interactions over the network. This form of elimination can improve efficiency and scalability, and can further improve user-perceived performance by reducing the average latency during a series of interactions. However, a common reason for incorporating caching as a native part of a REST architecture is as a counterbalance to some of the negative impacts of applying the Stateless {395} constraint

• **Interface/Uniform Contract** {400} The Interface {400} constraint (also known as “Uniform Interface”) states that all services and service consumers within a REST-compliant architecture must share a single, overarching technical interface. As the primary constraint that distinguishes REST from other architecture types, Interface {400} is generally applied using the methods and media types provided by HTTP and other Internet standards



• **Layered System** {404} A REST-based solution can be comprised of multiple architectural layers, and no one layer can “see past” the next. Layers can be added, removed, modified, or reordered in response to how the solution needs to evolve. The Layered System {404} constraint builds on Client-Server {393} to add middleware components (which can exist as services or service agents) to an architecture. Specifically, Layered System {404} requires that this middleware be inserted transparently so that interaction between a given service and consumer is consistent, regardless of whether the consumer is communicating with a service residing in a middleware layer or a service that represents the ultimate receiver of a message. Similarly, a service does not need to be aware of whether its consumer sent its request message directly, or whether the message passed through one or more service agents along its delivery path. This form of information hiding simplifies distributed architecture and allows individual architectural layers to be deployed and evolved independently of specific services and consumers.

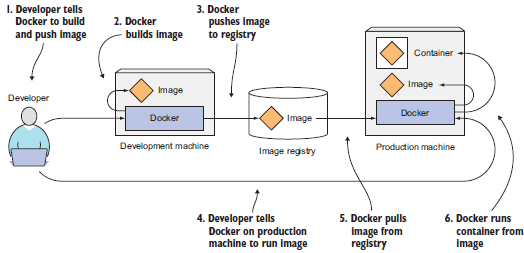
• **Code-On-Demand** {407} This optional constraint is primarily intended to allow logic within clients (such as Web browsers) to be updated independently from server-side logic. Code-On-Demand {407} typically relies on the use of Web-based technologies, such as Web browser plug-ins, applets, or client-side scripting languages (i.e. JavaScript). Code-On-Demand {407} can further be applied to services and service consumers. For example, a service can be designed to dynamically defer portions of logic to service consumer programs. For example, this type of functionality can be used in support of Stateless {395}, which dictates when session state should be deferred back to the service consumer. Code-On-Demand {407} can also build upon this by further deferring the processing effort. This approach may be justifiable when service logic can be executed by the consumer more efficiently or effectively

#### DOCKER

Docker = platform for packaging, distributing, and running applications. Allows you to package your application together with its whole environment (libraries that the app requires or files usually available on the filesystem of an installed operating system). Docker makes it possible to transfer this package to a central repository from which it can then be transferred to any computer running Docker and executed there. 3 concepts: **Images:** Docker-based container image is something you package your application and its environment into. It contains the filesystem that will be available to the application and other metadata, such as the path to the executable that should be executed when the image is run. **Registries**: A Docker Registry is a repository that stores your Docker images and facilitates easy sharing of those images between different people and computers. When you build your image, you can either run it on the computer you’ve built it on, or you can push (upload) the image to a registry and then pull (download) it on another computer and run it there. Certain registries are public, allowing anyone to pull images from it, while others are private, only accessible

to certain people or machines. **Containers**: Docker-based container is a regular Linux container created from a Docker-based container image. A running container is a process running on the host running Docker, but it’s completely isolated from both the host and all

other processes running on it. The process is also resource-constrained, meaning it can only access and use the amount of resources (CPU, RAM, and so on) that are allocated to it.

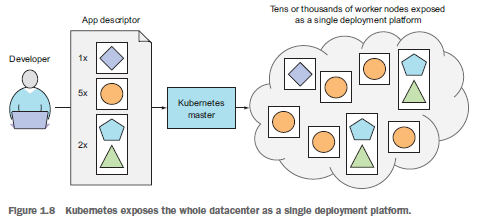


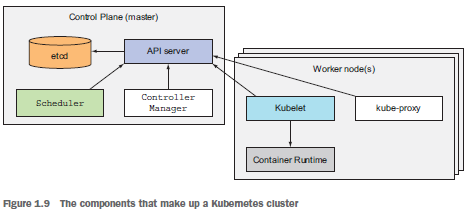
#### KUBERNETES

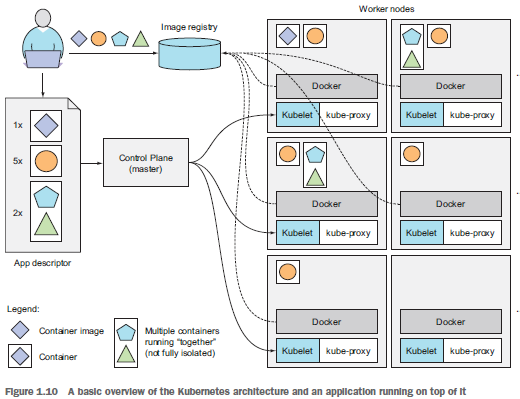
Kubernetes is a software system that allows you to easily deploy and manage containerized

applications on top of it. It relies on the features of Linux containers to run heterogeneous

applications without having to know any internal details of these applications and without having to manually deploy these applications on each host. Because these apps run in containers, they don’t affect other apps running on the same server, which is critical when you run applications for completely different organizations on the same hardware. Kubernetes enables you to run your software applications on thousands of computer nodes as if all those nodes were a single, enormous computer. It abstracts away the underlying infrastructure and, by doing so, simplifies development, deployment, and management for both development and the operations teams. Deploying applications through Kubernetes is always the same, whether your cluster contains only a couple of nodes or thousands of them. The size of the cluster makes no difference at all. Additional cluster nodes simply represent an additional amount of resources available to deployed apps.







## PYTHON

#### LUIGI

#### DASK

#### PANDAS

#### SQL ALCHEMY

## R

## JAVASCRIPT

## SCIKIT-LEARN

## TENSORFLOW

## PYTORCH

## MACHINE LEARNING

#### Regression

#### Boost

#### Deep Learning

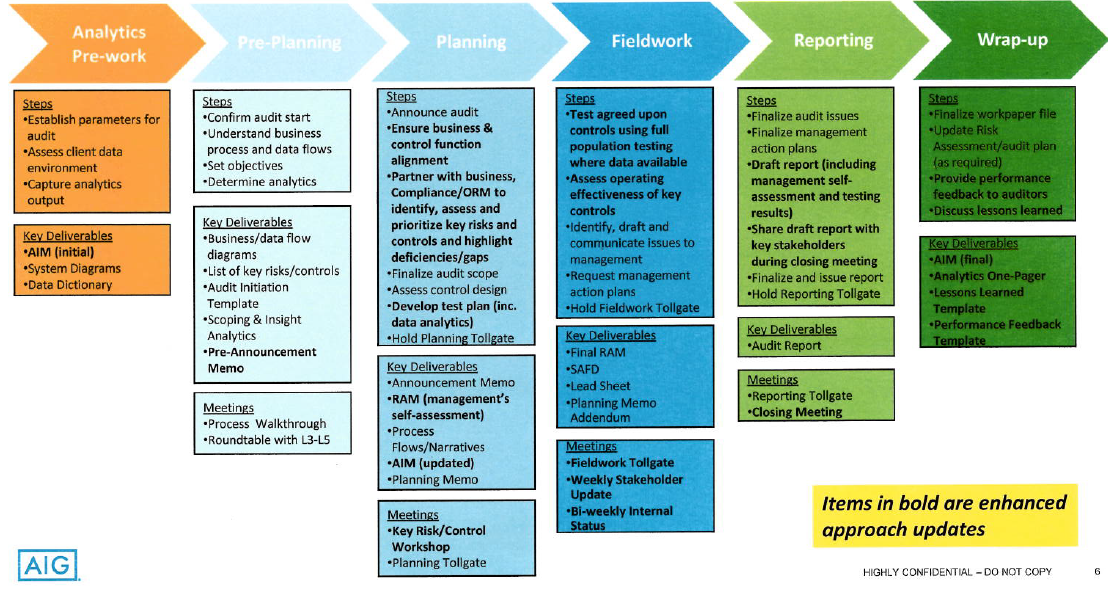
## POWER BI

## ORACLE

## SQL SERVER

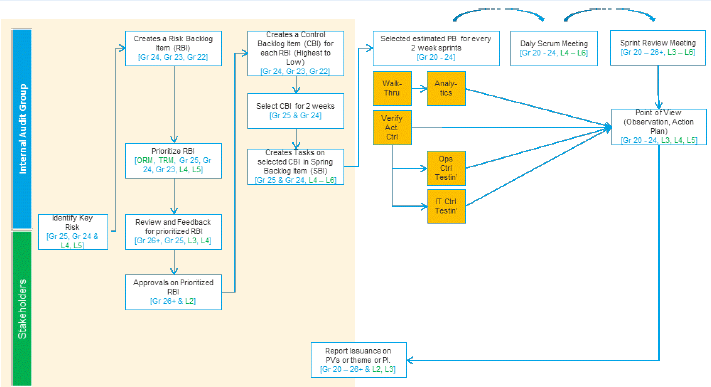
# EXHIBIT

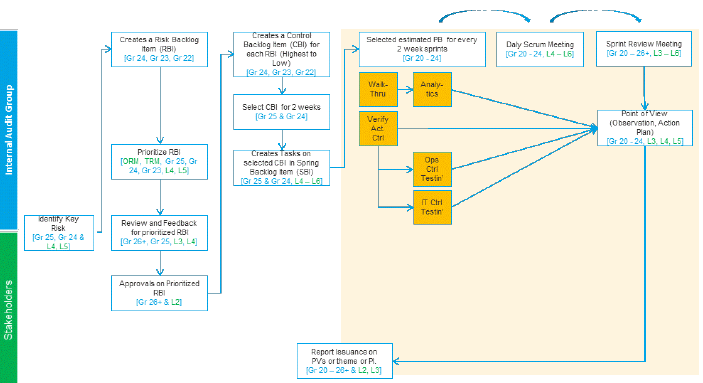
## Audit Lifecycle



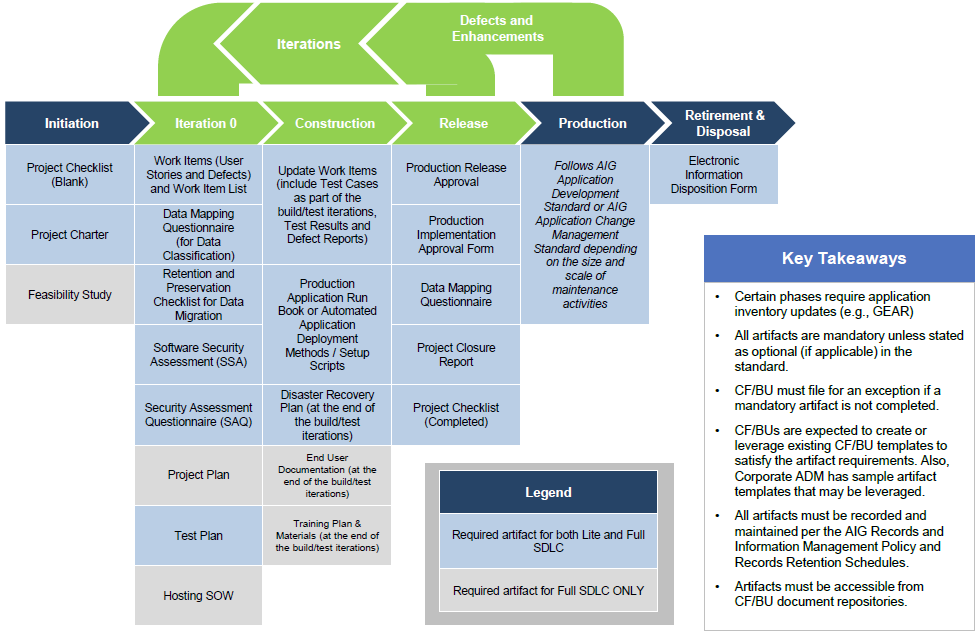
## AGILE AUDIT

Agile concepts: •Audit Increment planning” build a backlog of key risks and controls •Execute each sprint (2 week intervals) •After each sprint have a sprint review meeting with L4 to discuss results and initiate. After each sprint have tollgate to discuss stopping or continuing with audit •After each sprint and before next Sprint have Lessons learned session to discuss went well in sprint and what needs enhancements from next sprint •Holding daily scrum meetings (10 minutes) to discuss progress from yesterday, plan for current day and if any escalation is required

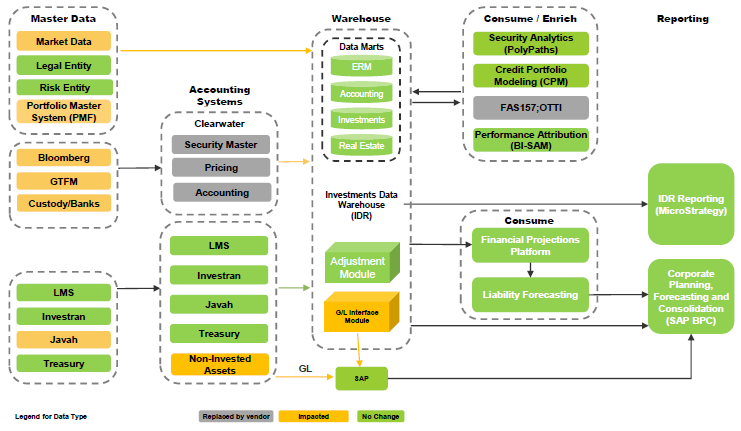




## APPLICATION DEVELOPMENT AGILE



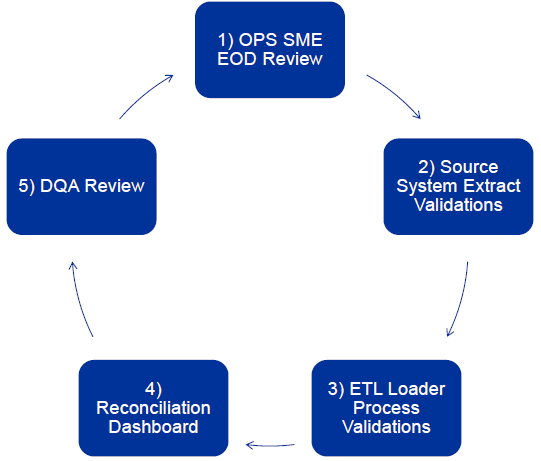
## NOVA Target State

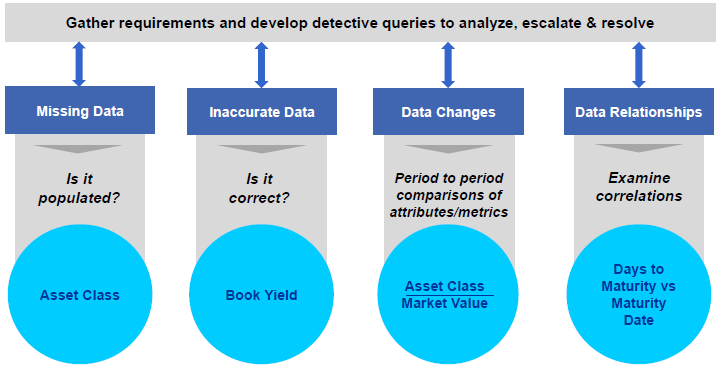


## IDR Data Governance



## IDR Data Quality Lifecycle





## IDR State Architecture

