[AIG USA 2](#_Toc15419164)

[Inshoring SOX functions 2](#_Toc15419165)

[CLEARWATER 2](#_Toc15419166)

[IDR 2](#_Toc15419167)

[RPA 2](#_Toc15419168)

[SAP HANA 2](#_Toc15419169)

[TREASURY 2](#_Toc15419170)

[OL/OLE 2](#_Toc15419171)

[JAVAH 2](#_Toc15419172)

[SACM 2](#_Toc15419173)

[SONIC 2](#_Toc15419174)

[SAILPOINT 2](#_Toc15419175)

[SPLUNK 2](#_Toc15419176)

[SKILLS 2](#_Toc15419177)

[AGILE AUDIT 2](#_Toc15419178)

[CI/CD 2](#_Toc15419179)

[AZURE TFS 2](#_Toc15419180)

[JENKINS 2](#_Toc15419181)

[TRAVIS CI 2](#_Toc15419182)

[TDD 2](#_Toc15419183)

[DEVOPS 2](#_Toc15419184)

[HADOOPP 2](#_Toc15419185)

[MONGODB 2](#_Toc15419186)

[SPARK 2](#_Toc15419187)

[Core 2](#_Toc15419188)

[SQL 2](#_Toc15419189)

[Stream 2](#_Toc15419190)

[STREAM 2](#_Toc15419191)

[AWS 2](#_Toc15419192)

[DJANGO 2](#_Toc15419193)

[BIG DATA SQL 2](#_Toc15419194)

[API 2](#_Toc15419195)

[12 FACTORS 2](#_Toc15419196)

[REST 2](#_Toc15419197)

[DOCKER 2](#_Toc15419198)

[KUBERNETES 3](#_Toc15419199)

[PYTHON 3](#_Toc15419200)

[LUIGI 3](#_Toc15419201)

[DASK 3](#_Toc15419202)

[PANDAS 3](#_Toc15419203)

[SQL ALCHEMY 3](#_Toc15419204)

[R 3](#_Toc15419205)

[JAVASCRIPT 3](#_Toc15419206)

[SCIKIT-LEARN 3](#_Toc15419207)

[TENSORFLOW 3](#_Toc15419208)

[PYTORCH 3](#_Toc15419209)

[MACHINE LEARNING 3](#_Toc15419210)

[Regression 3](#_Toc15419211)

[Boost 3](#_Toc15419212)

[Deep Learning 3](#_Toc15419213)

[POWER BI 3](#_Toc15419214)

[ORACLE 3](#_Toc15419215)

[SQL SERVER 4](#_Toc15419216)

[EXHIBIT 5](#_Toc15419217)

[AGILE AUDIT 5](#_Toc15419218)

[NOVA Target State 7](#_Toc15419219)

[IDR Data Governance 8](#_Toc15419220)

[IDR Data Quality Lifecycle 8](#_Toc15419221)

[IDR State Architecture 10](#_Toc15419222)

# AIG USA

## Inshoring SOX functions

## CLEARWATER

## IDR

## 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

## SAP HANA

## TREASURY

#### OL/OLE

#### JAVAH

#### SACM

## SONIC

## SAILPOINT

## SPLUNK

# SKILLS

## AGILE AUDIT

## CI/CD

#### AZURE TFS

#### JENKINS

#### TRAVIS CI

## TDD

## DEVOPS

## 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

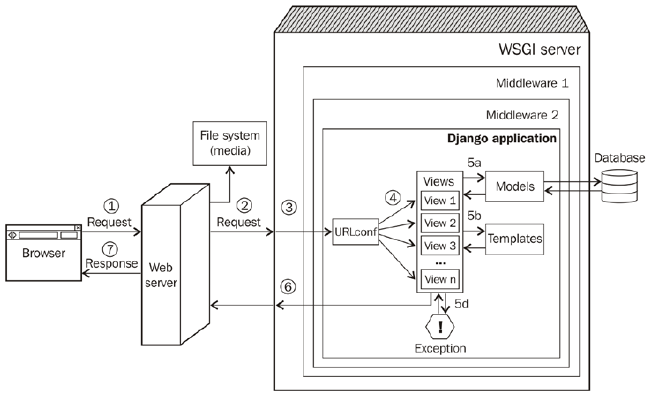
#### Core

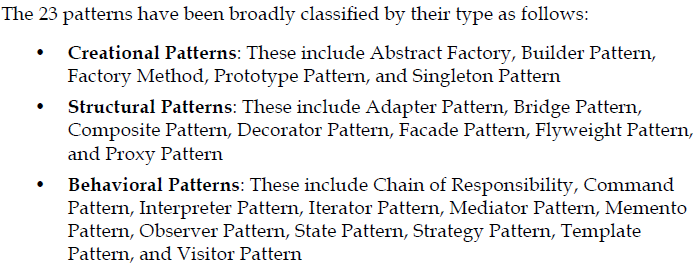
#### SQL

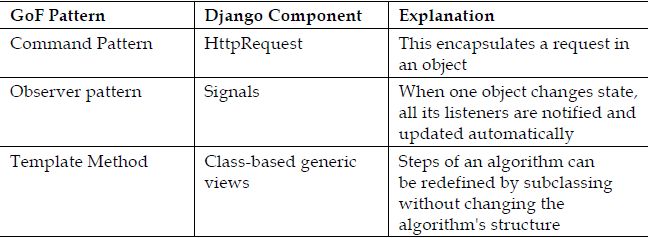
#### Stream

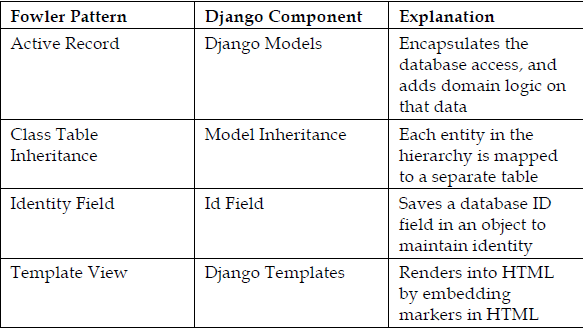
## STREAM

## DJANGO







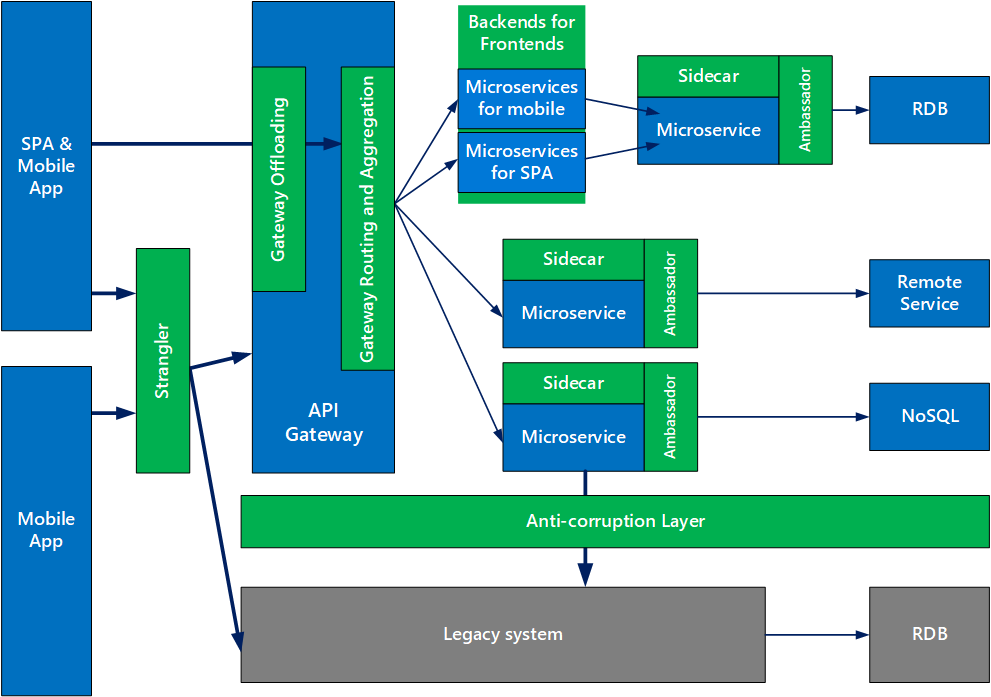


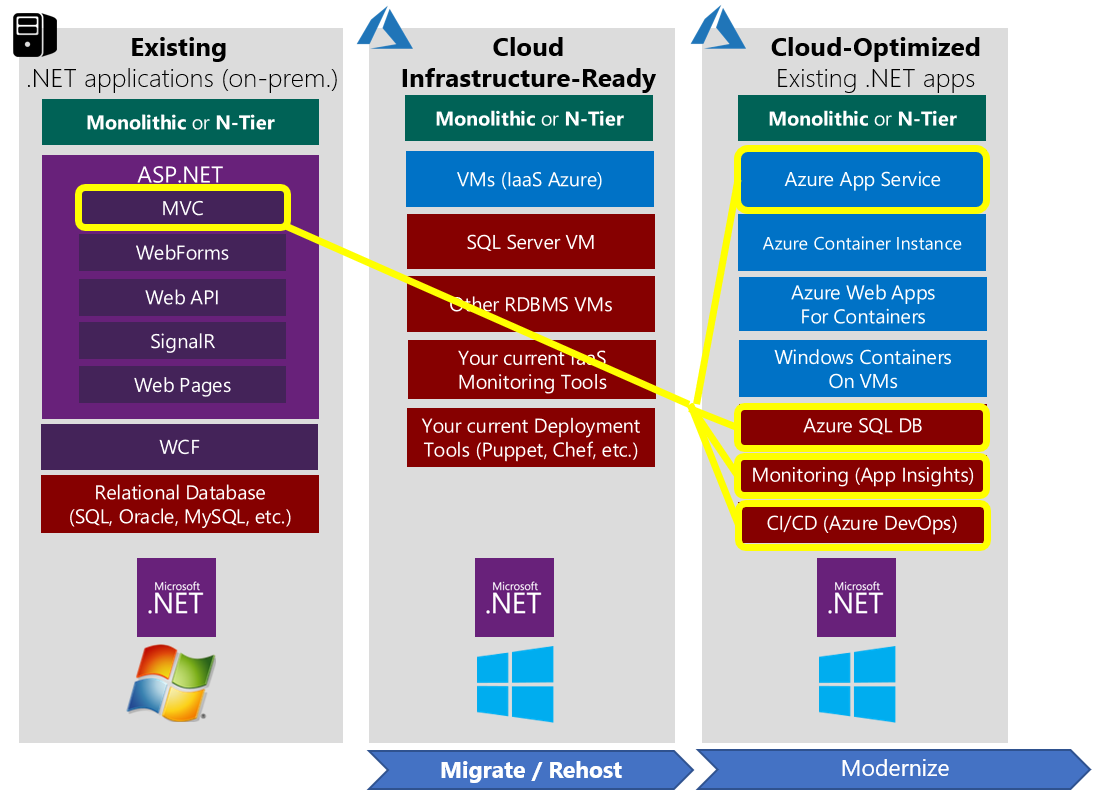
**Structural patterns**: ⬩normalized models ⬩model mixins ⬩user profiles ⬩service objects

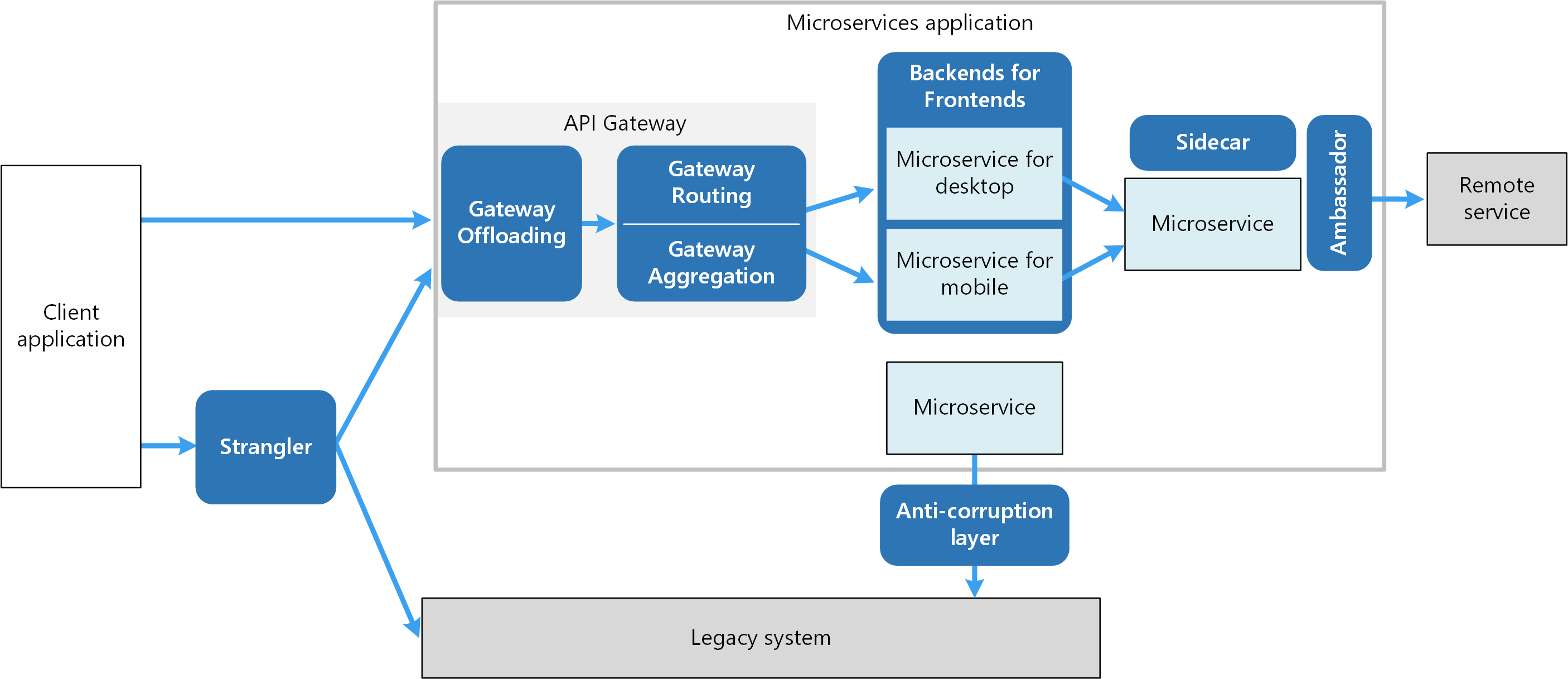
**Retrieval patterns**: ⬩property field ⬩custom model managers **View patterns**: ⬩acces controlled ⬩context enhancers ⬩services **Template patterns**: ⬩inheritance ⬩active link

## BIG DATA 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

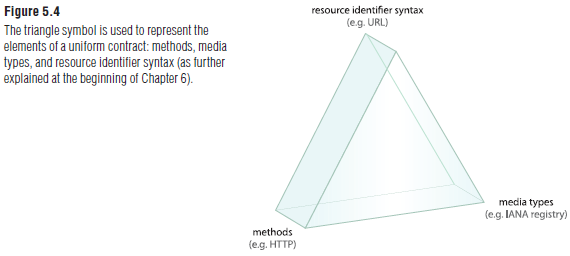
should contain all the necessary information for the service to understand the meaning of the request, and all session state data should then be returned to the service consumer at the end of each request. Statelessness is one of the primary influences over service contract design in REST-style architecture. 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.

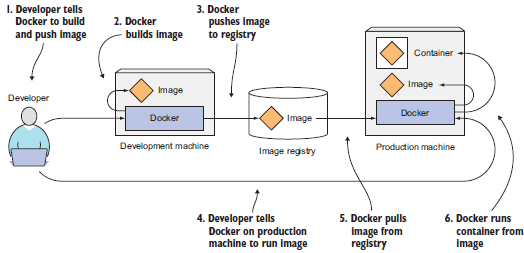
• **Code-On-Demand** {407} This optional constraint is primarily intended to allow logic within clients (such as Web browsers) to be updated independently from server-side logic. Code-On-Demand {407} typically relies on the use of Web-based technologies, such as Web browser plug-ins, applets, or client-side 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.

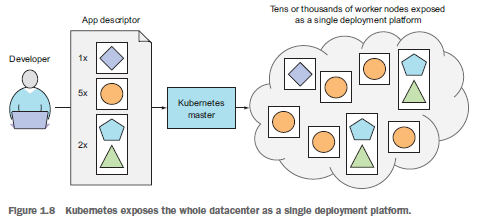


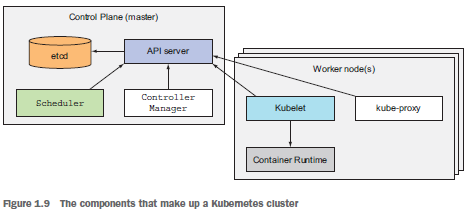
#### KUBERNETES

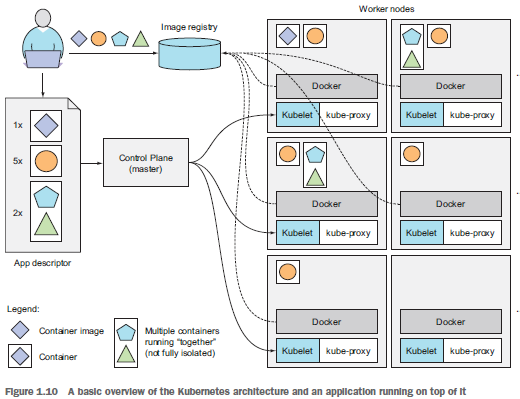
Kubernetes is a software system that allows you to easily deploy and manage containerized

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

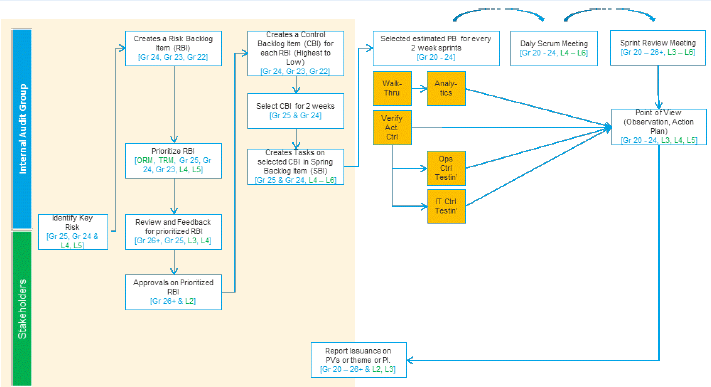
## ORACLE

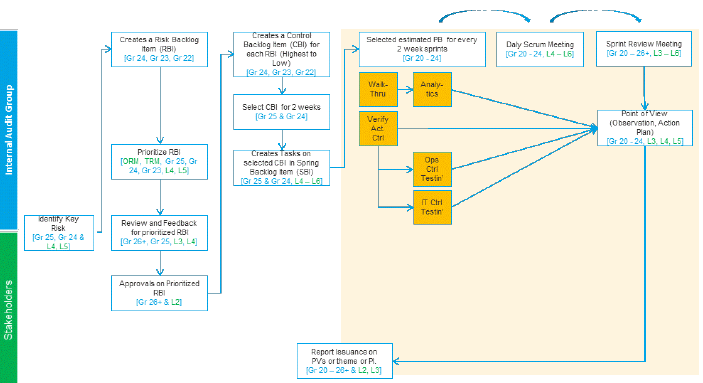
## SQL SERVER

# EXHIBIT

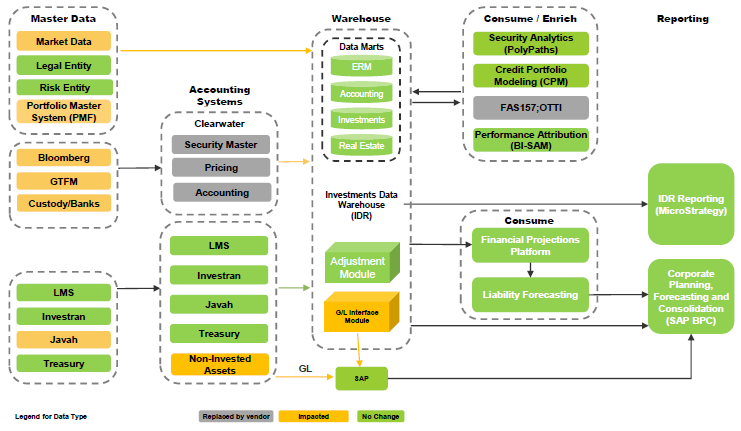
## AGILE AUDIT

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





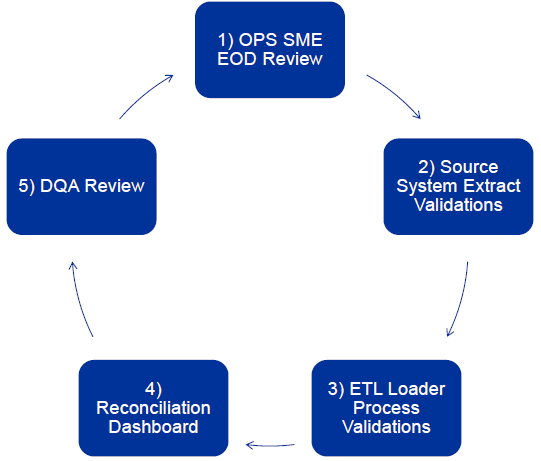
## NOVA Target State

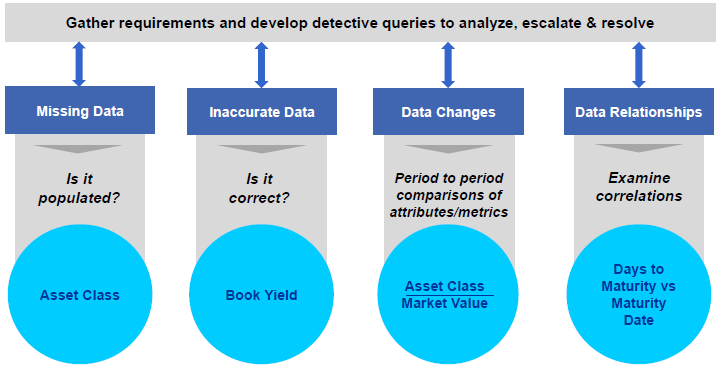


## IDR Data Governance



## IDR Data Quality Lifecycle





## IDR State Architecture

