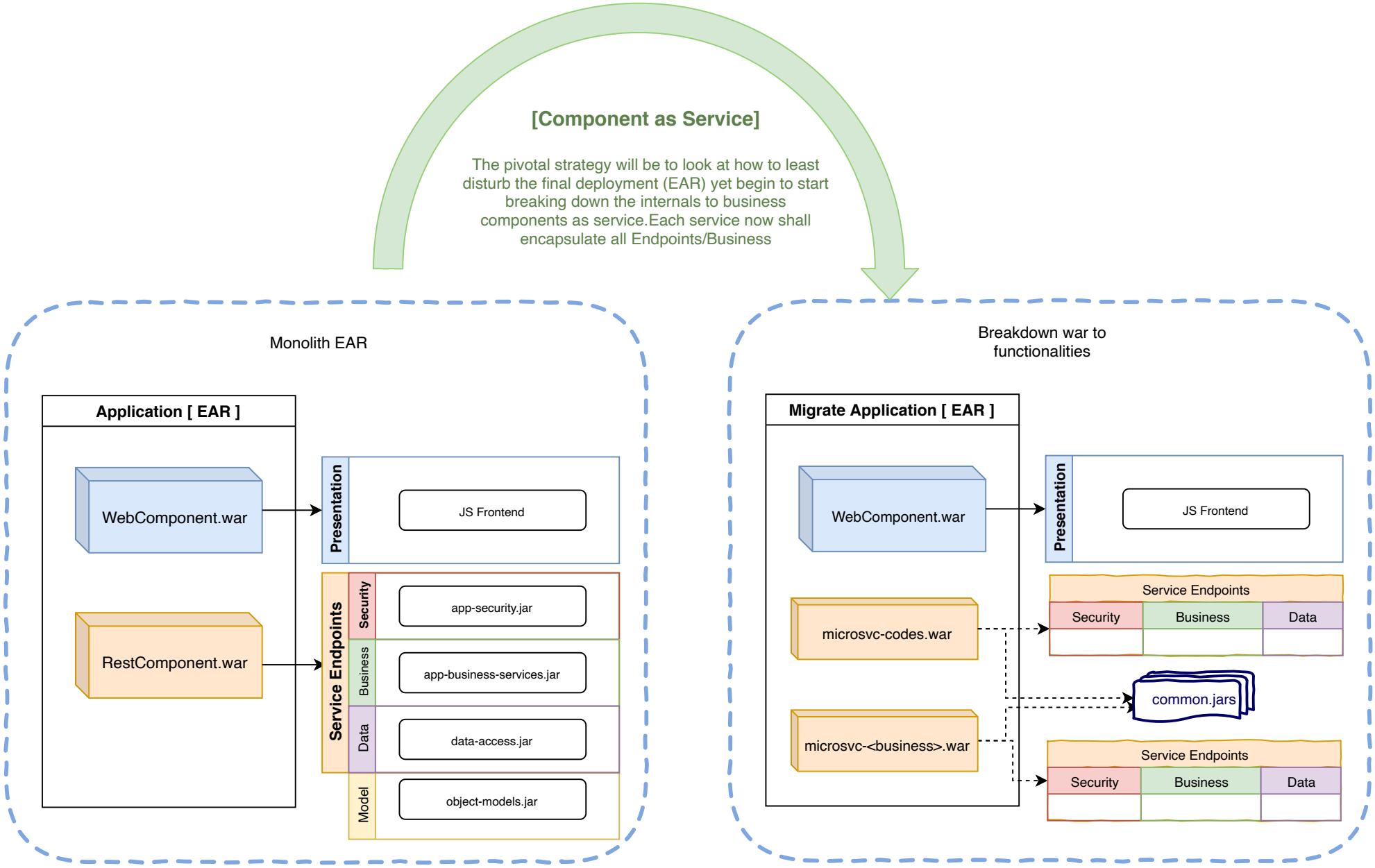
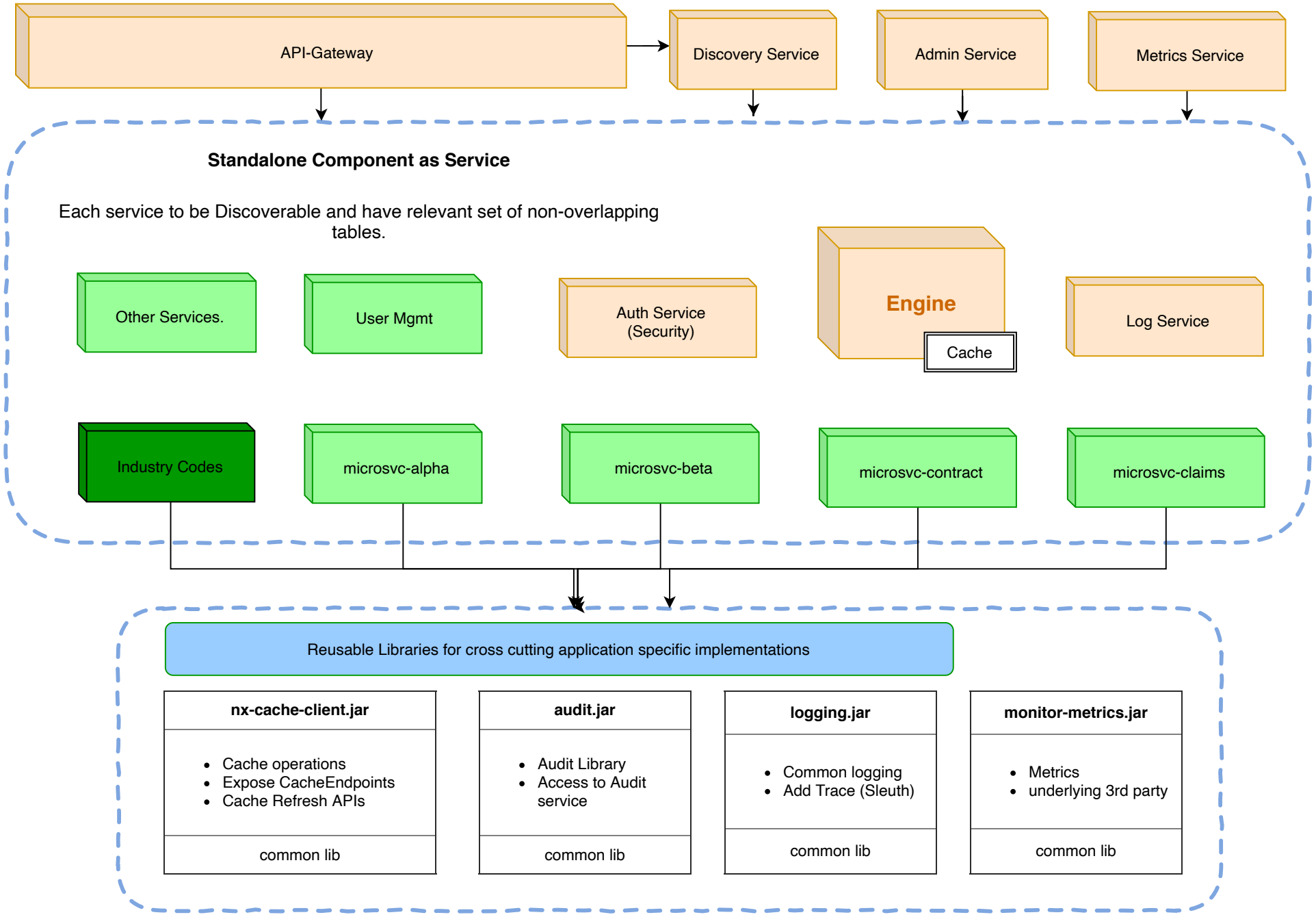

[Component as Service]
The pivotal strategy will be to look at how to least disturb the final deployment (EAR) yet begin to start breaking down the internals to business components as service.Each service now shall encapsulate all Endpoints/Business
Monolith EAR
Application [EAR]
WebComponent.war
RestComponent.war
Presentation
JS Frontend
Service Endpoints
Security
app-security.jar
Business
app-business-services.jar
Data
data-access.jar
Model
object-models.jar
Breakdown war to functionalities
Migrate Application [EAR]
WebComponent.war
microsvc-codes.war
microsvc-<business>.war
Presentation
JS Frontend
Service Endpoints
Security
Business
Data
common.jars
Service Endpoints
Security
Business
Data


API-Gateway
Discovery Service
Admin Service
Metrics Service
Standalone Component as Service
Each service to be Discoverable and have relevant set of non-overlapping tables.
Other Services.
User Mgmt
Auth Service (Security)
Engine
Cache
Log Service
Industry Codes
microsvc-alpha
microsvc-beta
microsvc-contract
microsvc-claims
Reusable Libraries for cross cutting application specific implementations
nx-cache-client.jar
Cache operations
Expose CacheEndpoints
Cache Refresh APIs
common lib
audit.jar
Audit Library
Access to Audit service
common lib
logging.jar
Common logging
Add Trace (Sleuth)
common lib
monitor-metrics.jar
Metrics
underlying 3rd party
common lib

Extract Business Component

Carve out all relevant code into a project which encapsulates all services offered by this module.Define the business boundary into interfaces , hence package classes as per.All API to Data layer implementations.

1 Assimilate Functionality

Identify APIs and assimilate functionalities into a single deploy-able project (JAR/WAR) and define Business boundary.

1.Rightful API interfaces defined , agreed with business boundary.
2. Not to address security layers for now.(Later integration).
3. Clear Pom.xml (Build) and keep only relevant external libraries.
(To be later migrated to latest stack from Spring boot Bill of materials)

2 Integrataion to existing UI

Validate with Postman project / Current UI integration test with the new component as a WAR (No security**)

1.mute security for APIs extracted (No Spring Auth Filters)
2. Consider Cross-Origin handling as new war is external.

3 Stack Migrate to latest

Re Code Framework to use Spring boot 2.1.3 Bill Of Materials.Create reference for all other modules.

Stack Upgrade (2 Major version Jumps)

1. **Hibernate** : 3.6.8.[Oct-2011] --> 5.3.7 [Oct-2018]
2. **Rest** : Jersey 1.9 [Sep-2011] --> Spring Rest 5 [Feb-2019]
3. **Spring** : 3.2.9 [May-2014] --> 5.1.5 [Feb-2019]
4. **Jackson** :Codehaus 1.8.3 [Jul-2011] --> FasterXml 2.9.8 [Dec-2018]
5. **WAR -> JAR :** Spring Boot 2.1.3 executable JAR

4 Java Annotation Config (Boot)

No more XML bean configs.Move to Java Configs.

5 POC Endpoints

As POC have certain endpoints ready and tested with all Nature of CRUD operation with new Stack.Postman Project checked in.

Spring Boot microservice plug and play with existing monolith and work through application UI

Enter Microservice Pattern (Netflix Stack)

Go through the journey of applying netflix inspired stack and microservice integration pattern and expose as a service.

6 Eureka Discovery Server

Spring Boot Discovery Server.Script assembly to generate deployable jar artifact + Docker Image

7 Zuul Gateway Server

Boot API Gateway server.Script assembly to generate standalone jar + docker image

1. Stress Test the Gateway to get numbers on throughput.
2. Add pre filter for auth and error filters.
(Have custom library)

8 Monitor Server

Boot Admin Monitor server which administers the microservices & have a discovery pattern to identify other services.

1. All microservices must expose actuator and other management endpoints explicity. Boot 2.1.3
2. Customize the theme per product template vue.js

9 First Microservice

Create this service as a deployable artifact with autodiscovery , which integrates with the above infra servers.

Maven assembly descriptor to have external application.properties and a executable jar with external log config file.(Correct Folder Structure)

10 Central Logging

Create Central Logging server system (elk) and integrate with microservices.

Hibernate: (Work Effort : 1 time framework bean configs + recode all DAO)

i) No HibernateTemplate --> EntityManagerFactory/LocalSessionFactoryBean
ii) Entity Classes marked Sequence Generator. (Identity vs Table Sequence). MSSQL : id generation Use Table Identity.
ORACLE : Table Sequence.

Jersey to Spring Rest: (Work Effort : 1 time framework bean configs + Rest Endpoints annotations migrations)

i) Jersey fetched from latest boot starters moved to glassfish.(Jersey from 1.9 to 2.27) com.sun.jersey -----> org.glassfish.jersey
Jackson to 2.9.x jersey-media-json-jackson 2.27)
ii) InjectionResolver and Provider classes no longer usable.
For Spring ObjectMapping use MappingJackson2HttpMessageConverter.
Custom Mapping providers
objectMapper DeserializationFeature/Serialization Feature.

iii) Jax-RS annotations to Spring Rest annotation in controller classes.

iv) Jersey Exception mapper --> Spring Controller Advice.

iv) Model objects library re-annotate with JacksonJson and not old JAXB.

v) Spring Integration Test cases with IN-Memory H2 DB.
schema.sql / data.sql. Use least mocking.

11 Cache Strategy

Identify cache strategy for SYSTEM to reduce network calls.

12 Runtime Metrics

Integrate with micrometer -Prometheus libraries and External Prometheus/Grafana servers

Prometheus and Grafana server Instance Setup.
Create relevant Prom Queries and pre defined dashboard of business / administrative importance.

Continue Extending.....

1. Security Layer
2. Microservice network call Tracing service
3. Establish Autoscale behaviour
4. Microservice to Microservice secure communication strategy
5. Establish some performance reports
6. Deployment strategy
7. Docker image for POC and extend to Kubernatis
8. Component Test and Integration test strategy
9. Now with Owners defined refactor component further
10. Build Pipelines for project and blackduck/Sonar integrations

Feature / Defect Delivery (Time to Market)

Monolith

Wait till End - Quarter Release

- Even if defect or feature is ready , wait till entire package is tested and shipped with all other release work.
- If any release work causes breakage , confirmed fixes also suffer delivery to client.

Component

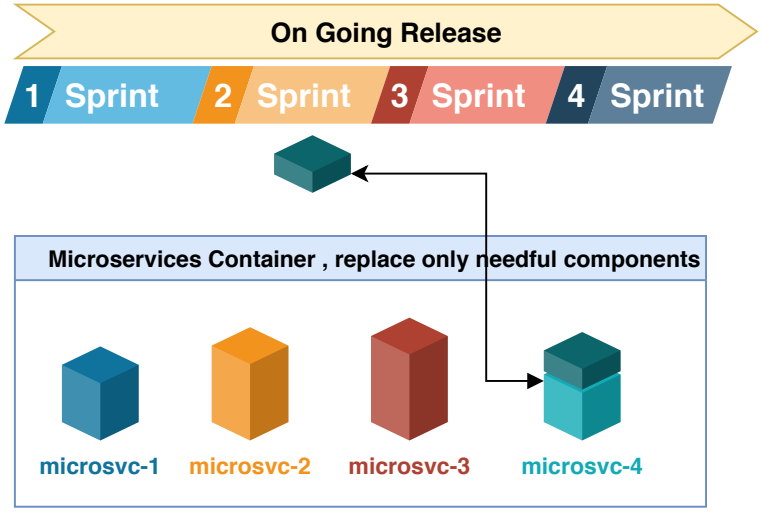
On Demand - Sprint Drop (Component Upgrade)

- With Component having its cleaner tests and integration tests , easier to sanitize and run regression .
- Clients can update only the specific component and not worry about the entire package change bringing in all nature of changes not asked for.
- Easy component deployment and rollback deliver when ready.

Will Customers want this CAPABILITY

1. **Eagerly Yes** : When can we shift to such a model.
2. **Yes** : Good To have
3. **No** : we are happy waiting a release

Compare delivering a small Calc specific change monolith way.



Scalable during peak processing (Dynamic Scaling)

Static Configurations of clustered monoliths

Monolith

- Current and future solution's in monolith design pattern.
- Static configurations for multiple nodes
- If Clients scale volume suddenly , there is a cap to performance per nodes configurations.

Scale / Descale on Demand

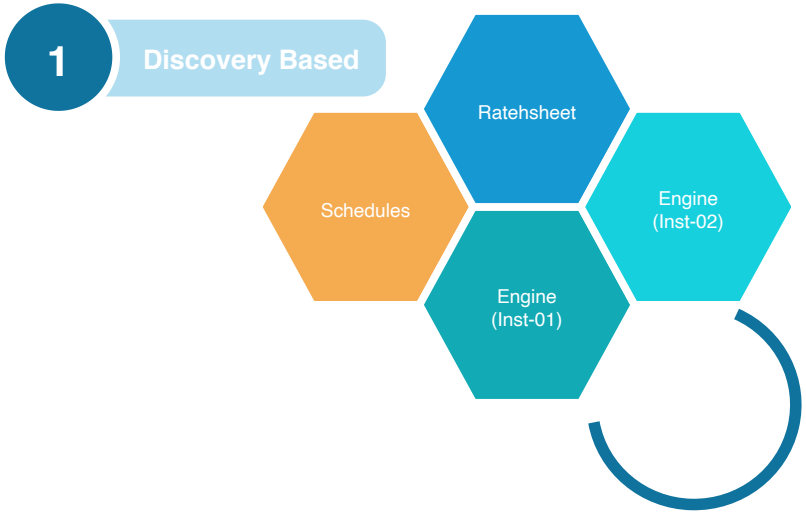
Component

- Solutions will start on microservices pattern , proven scalability.
- Dynamic Auto-discovery , 0 downtime node additions.
- If Clients scale volume suddenly they can just spawn more processing-engine nodes at runtime and close after use.

Will Customers want this CAPABILITY

1. **Eagerly Yes** : When can we shift to such a model.
2. **Yes** : Good To have
3. **No** : Don't forsee need of such features

Customers will not need re-configurations and restarts , as microservice can add/remove node by service discovery. (0 downtime)



Production Ready Metrics and monitoring dashboard

Monolith

Needs new development for such features

- Custom framework and development time cycle.
- Such Features will be developed on top of monolith and existing App server dependency.
- No Existing common framework.

Component

Multiple Boilerplate functionalities ready

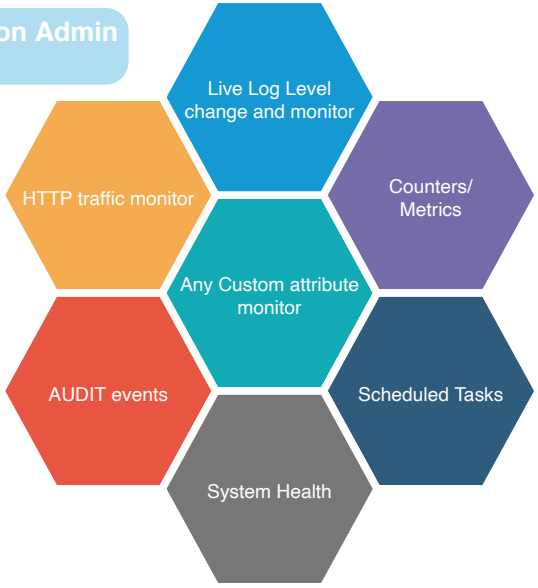
- Industry Standard Production Ready features already available.
- Pluggable features are provided just by configurations
- Monitor Health / Traffic / Time and Counter Metrics templates can be used to provide Custom metrics and counters.
  - Dynamic Log Change
  - Counter / Stats on any attribute of application.

Will Customers want this CAPABILITY

1. **Eagerly Yes** : When can we shift to such a model.
2. **Yes** : Good To have
3. **No** : Don't forsee need of such features

Customers will have dashboard view and can pour new requirements from custom attribute monitors to be added

1 Production Admin Features



Failure Isolation (only component upgrade / rollback)

Monolith

Anything Fails - Rollback Monolith

- A monolith rollback is taking back entire release features and fixes for one identified issue.
- Next Fixed version is again a monolith needing its own space for quality assurance before released to client.

Component

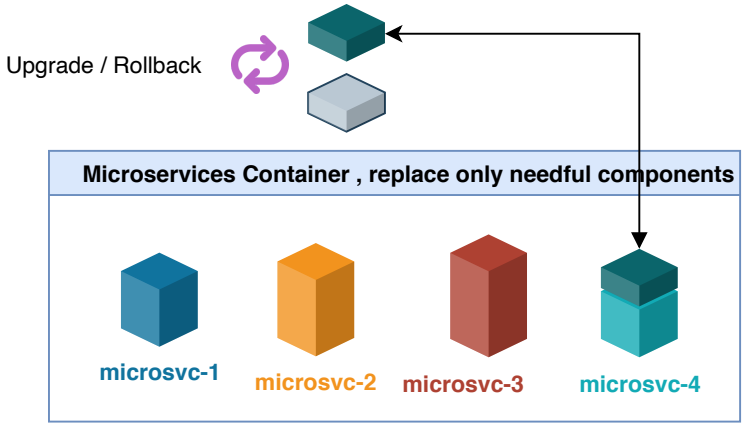
Something Fails - isolated component rollback

- Only selective component rollback.
- Next Fixed version component deployed and not change other non-impacted components.
- Component only deployment - deliver when ready.

Will Customers want this CAPABILITY

1. **Eagerly Yes** : When can we shift to such a model.
2. **Yes** : Good To have
3. **No** : we are happy waiting a release

Compare delivering a small Calc specific change monolith way.



Development Patterns

Monolith

Monolith code package structure-still wired

- JUNITs :aren't meaningful as lot of mocking across packages.Adding further junits only adding coverage not functionality.
- Workspace Setup : All Monolith setup.
- Defect Fix lifecycle - More research and setup times spanning across different project code bases.
- Code Packaging : Still creating redundant code versions every release, even if unchanged.
- Implementation uniformity : Any developer might choose own implementations to achieve a feature enhancement.

Component

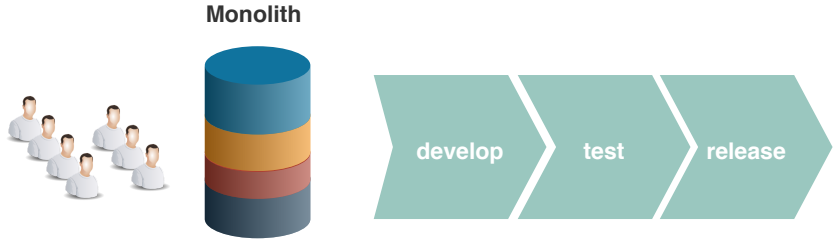
True end-end component (Wireless)

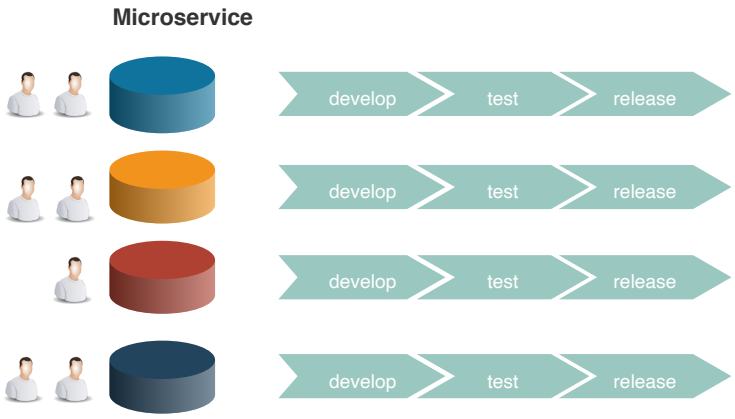
- JUNITs : component JUNIT will cover Endpoint to Data layer unit and integrations tests which are more meaningful.
- Workspace Setup : Only needful component setup.
- Defect Fix Lifecycle : Clearcut issue isolation to component alongside reducing setup and analysis time.Reliability on component tests.
- Code Packaging : Components upgrade version for change only.Easier Maintenance strategy for future.
- Implementation uniformity : Component owners own development pattern and stays uniform , common components enforced.
- Upgrade old Libraries: Replacing decade old jars and more cloud aware libraries can be incorporated component by component.

Desirable developer practice CAPABILITY

1. **Eagerly Yes** : When can we shift to such a model.
2. **Yes** : Good To have
3. **No** : Ok with criss-cross library codes

Developers can easily identify the impacted component.
Also increases component ownership in true sense.





Quality Assurance and DevOps

Monolith

Roadblocked for true Devops pipeline

- Sanity and Regression :aren't meaningful as lot of mocking across packages.Adding further junits only adding coverage not functionality.
- Pipeline Tangle : With current code structure , tangled dependencies are blocker to independent pipeline definitions.
- Test Plans : Test Plans are ever growing trying to deal with monolith and business behaviors spanning across.
- Automation: Existing Automation still usable as is.
- Deployment: Dependency of Application Servers setup.

Component

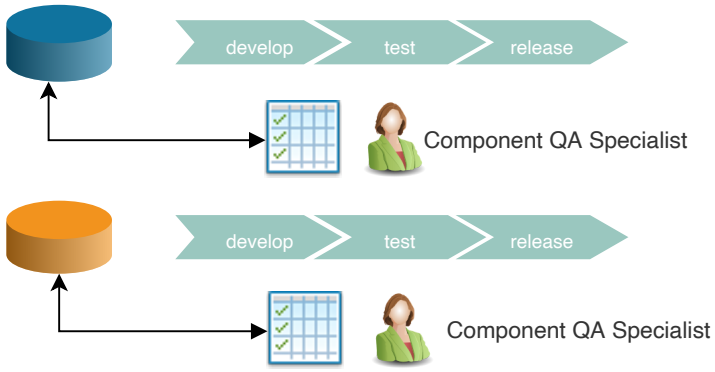
Enabling true Devops Pipeline with Stop the line

- Sanity and Regression: Component test and pipelineJUNIT will cover Endpoint to Data layer unit and integrations tests which are more meaningful.
- Pipeline Clean: With clearer code component restructuring , we can get dependable pipelines and tests integrated.
- Test Plans: Component Test Plans can now be more robust and focused knowledge growth for teams owning.
- Automation: Existing Selenium Automation suite still completely reusable for sanity / regression. (NO UI Migration yet).
- Deployment: Self Executing microservices do away with external Containers.

Desirable QA practice CAPABILITY

1. **Eagerly Yes** : When can we shift to such a model.
2. **Yes** : Good To have
3. **No** : Don't see value add proposition

QA can enrich each component for robustness over period of time with focus.Form stable test suites for each component. Form a reliable base of components an investment for tomorrows sanity/regression pipelines.



Cloud Ready Product Solution ?

Not Ready - Only Deploy On Cloud as Infra

Monolith

- Application Library :Some Core frameworks date ~ 2011.With old libraries lot of functionalities have to be coded as against new libraries which have cloud enabled offerings.(No Code)
- Cloud Deploy : Deploy the whole Monolith still using the same underlying Application Servers.

Enabling Microservices pattern for cloud

Component

- Application Library: Use cloud enabled libraries (industry standard). New Age libraries open world for lot of functionalities already available as against custom development.
- Cloud Deploy: With Microservices pattern , we have a base for true cloud deployment patterns which can be customized per client.Leverage certain cloud services instead of inhouse development.
- Database: With rightful component definitions , database per microservice can be acheived and better solutions for tenancy can be developed.

Desirable VISION ?

1. **Eagerly Yes**: When can we shift to such a model.
2. **Yes** : Good To have , begin with POCs
3. **No** : Dont want to think and act on this path

This is not a transition but working POCs and proving integrations and feasibility for further incremental demos