Cloud Orchestration

(SummerSoC 2014: June 30 – July 5, 2014 – Hersonissos, Crete, Greece)

University of Stuttgart
Universitätsstr. 38
70569 Stuttgart
Germany

Prof. Dr. Frank Leymann
Institute of Architecture of Application Systems
Leymann@iaas.uni-stuttgart.de

Phone +49-711-685 88470
Fax +49-711-685 88472

Agenda

The Need for Topologies

TOSCA Quick Overview

Declarative vs Imperative Processing

TOSCA Simple Profile

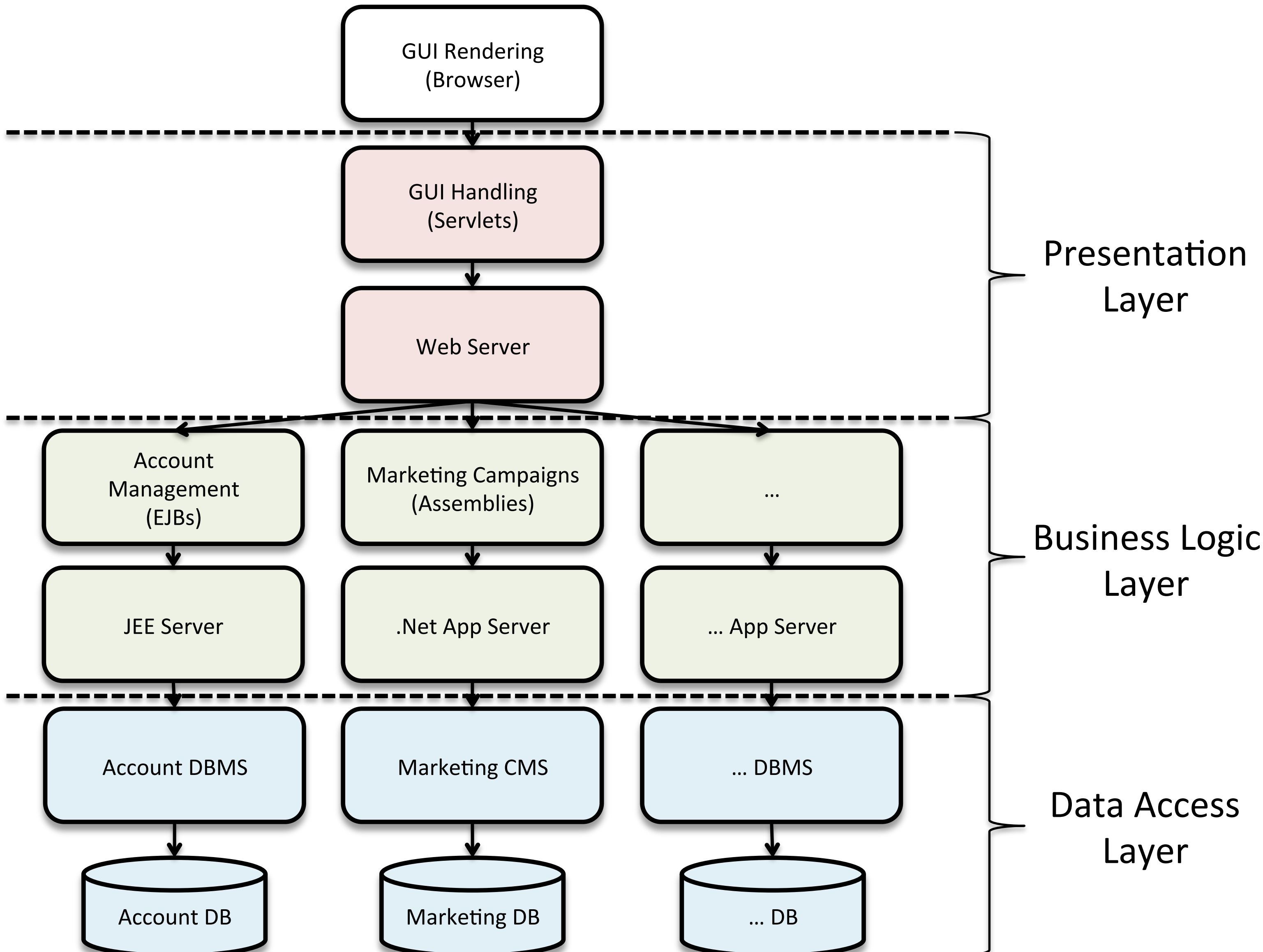
Orchestration Engines Architecture

Summary

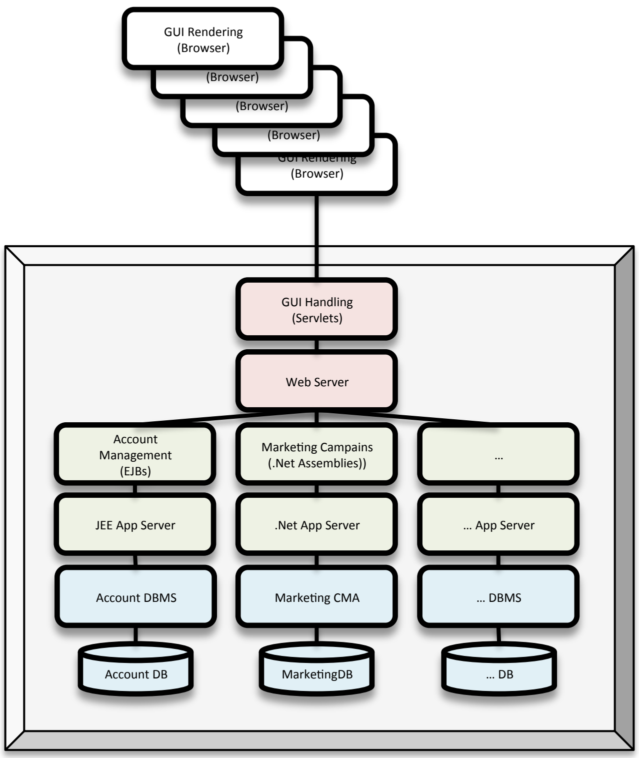
Agenda

- The Need for Topologies
- TOSCA Quick Overview
- Declarative vs Imperative Processing
- TOSCA Simple Profile
- Orchestration Engines Architecture
- Summary

Sample Application

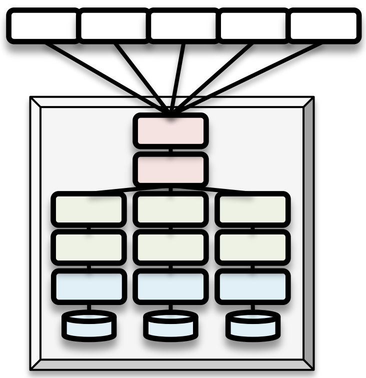


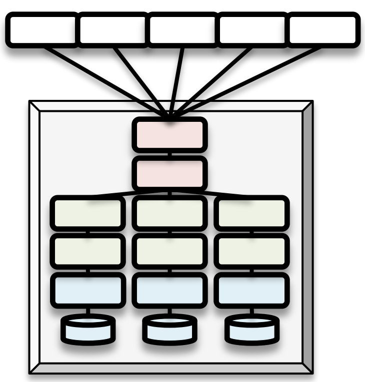
Packaging in a Virtual Machine



- First and naïve approach: you package the whole application into a single virtual machine and move it to the cloud
- Customers start using it from their browsers
- They like it, and more and more are using it ☺
- Thus, you need to scale!

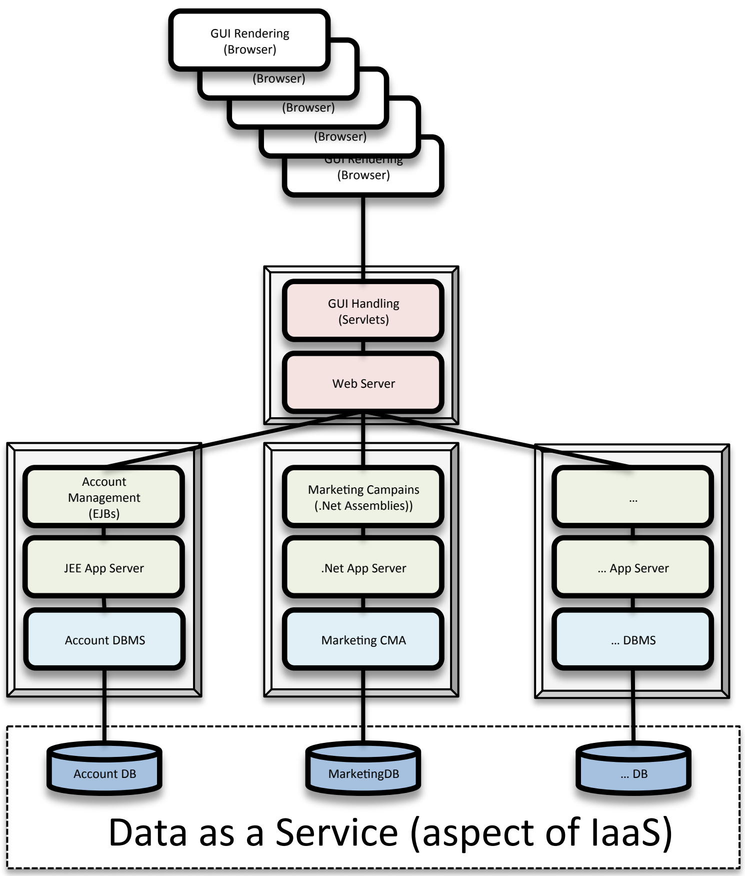
Scaling Based on VMs





- You instantiate a second VM containing your application in the cloud
- Thus, your customers are happy!
- But, what about you?
  - How many licenses of App Servers, DBMS, CMS,… do you have to pay?
    - For example, if the customers use the Account features mostly, why do you replicate the Marketing stack and pay for the corresponding licenses?
  - What about your Account DB getting out of sync?
    - Storage is associated with single VM, but updates need to be synchronized across VMs to result in consistent data

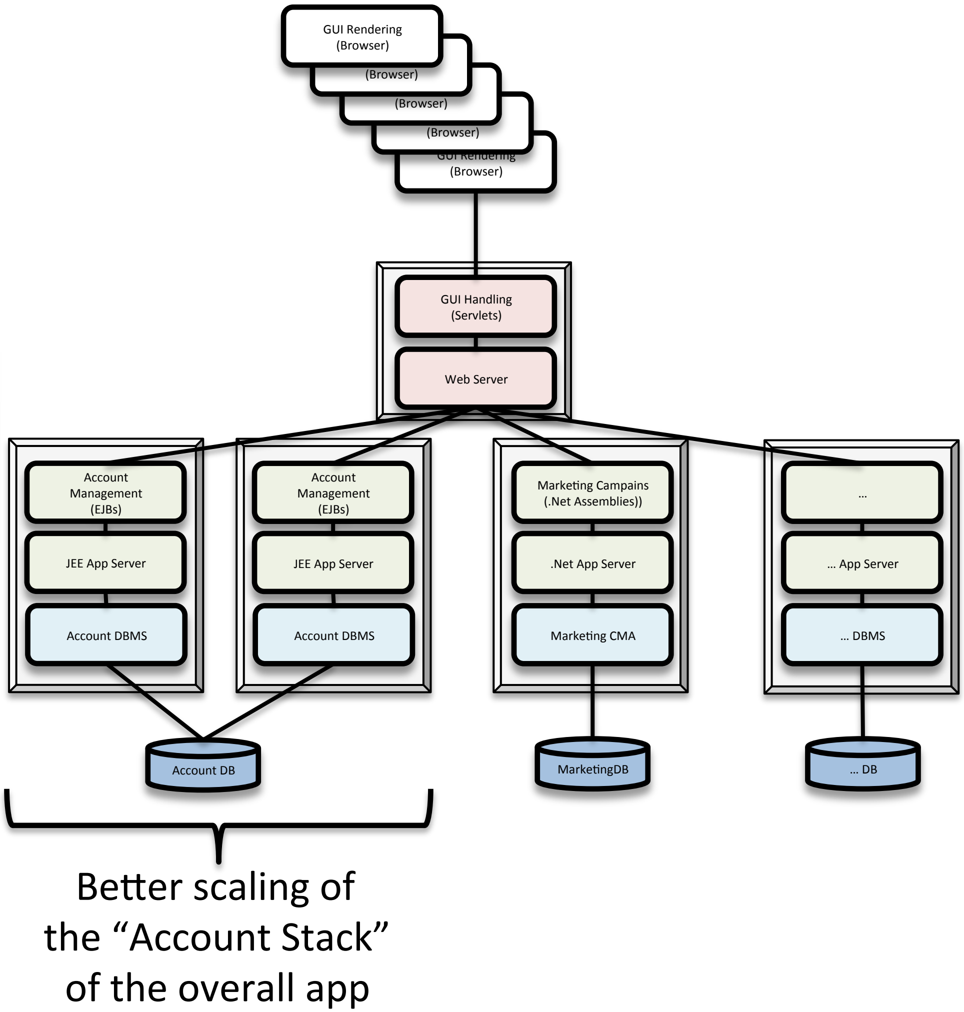
Solving Scaling Related Problems: First Step



- You package the different stacks of your applications into separate VMs
- You persist your data in storage features of the cloud ("Data as a Service")
  - Data can then be shared when scaling out
- This enables replication of individual stacks for scaling
  - Avoiding the problems indicated before (licensing, data consistency,...)

© Frank Leymann

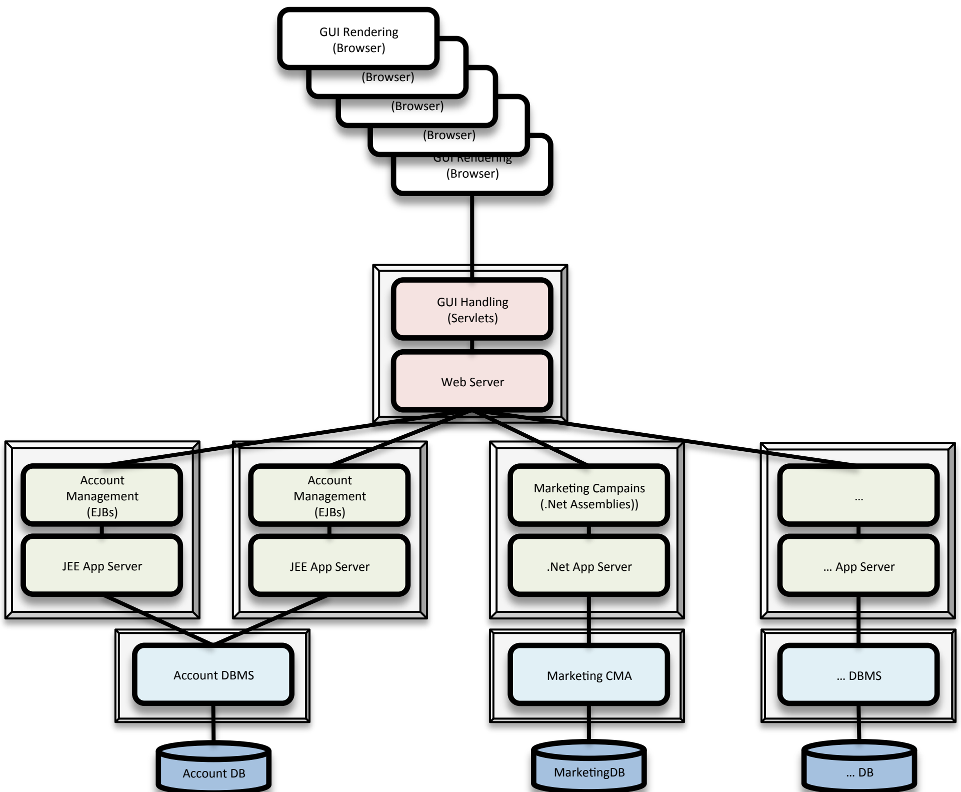
Scaling Related Problems: Further Granularity Issues



- When a particular stack is under high request load, it can be scaled by starting multiple instances of the corresponding VMs
- Data is shared between these VMs because database content is stored in storage features of IaaS
- But maybe the underlying DBMS can sustain the load generated by many App Servers?
  - I.e. license cost can be reduced, etc

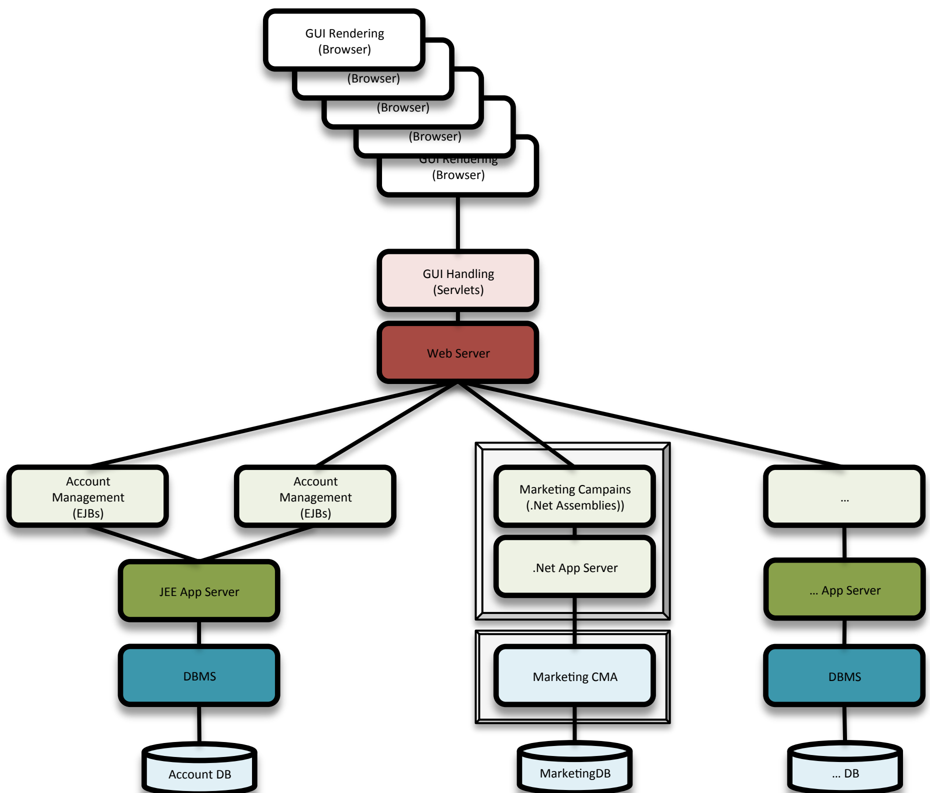
© Frank Leymann

Proper Granularity for Scaling



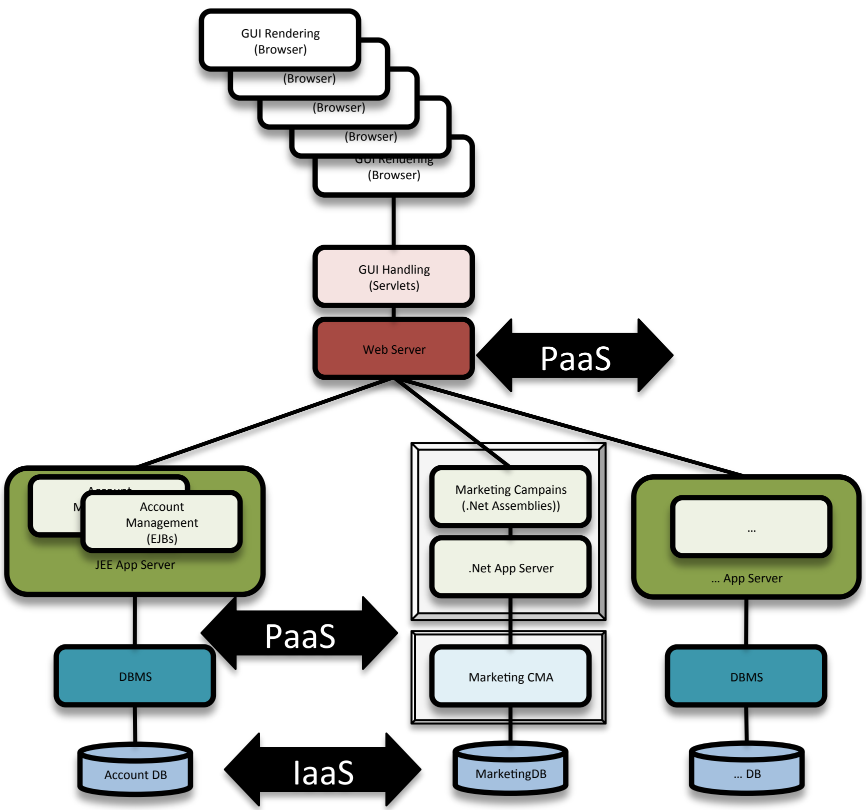
- You package "appropriate" components of your application in separate VMs so that they can scale independently
- Now multiple VMs containing the App Server can use the same DBMS
- But the DBMS in the separate machine needs maintenance
- Do you want to do it by yourself?

Consequences of Proper Granularity



- Next step is to consider features provided by the cloud environment that may substitute components of your VMs
  - For example, DBMS, App Server
    - E.g. Amazon SimpleDB, Google AppEngine,…

Towards "Cloud Native"



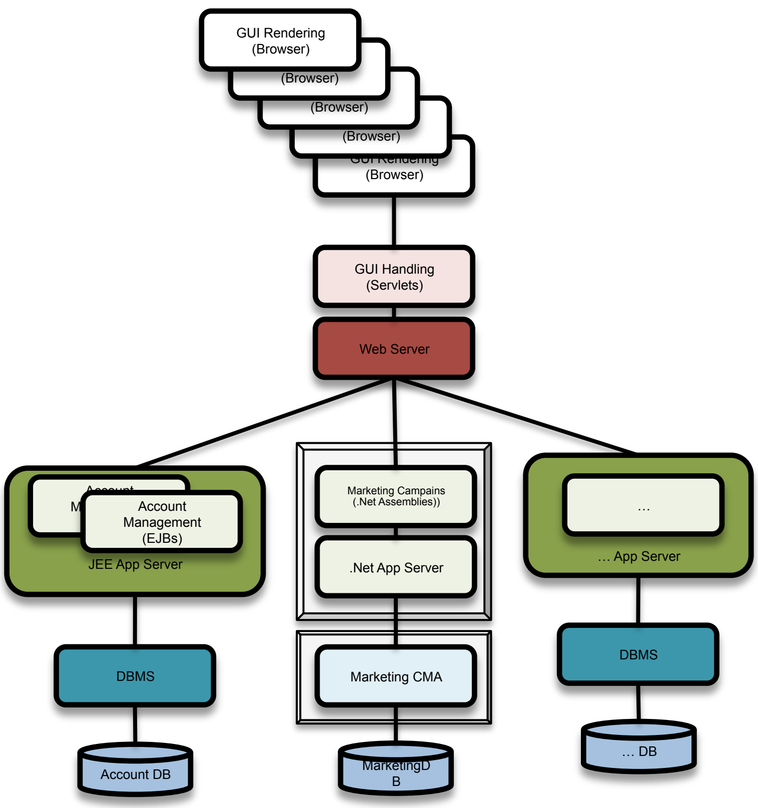
- Next, elasticity (i.e. on-demand scale-in & scale-out) requires…
  - Loose coupling of components
  - Automatic start/stop of instances of components
  - Stateless components
  - …

© Frank Leymann

Agenda

- The Need for Topologies
- TOSCA Quick Overview
- Declarative vs Imperative Processing
- TOSCA Simple Profile
- Orchestration Engines Architecture
- Summary

What We Understood So Far



- So, your application is componentized
- You specify all middleware and infrastructure the application needs
- You specify all relations between these pieces and what the nature of that relations are

- You specified the ***topology*** of the application

© Frank Leymann

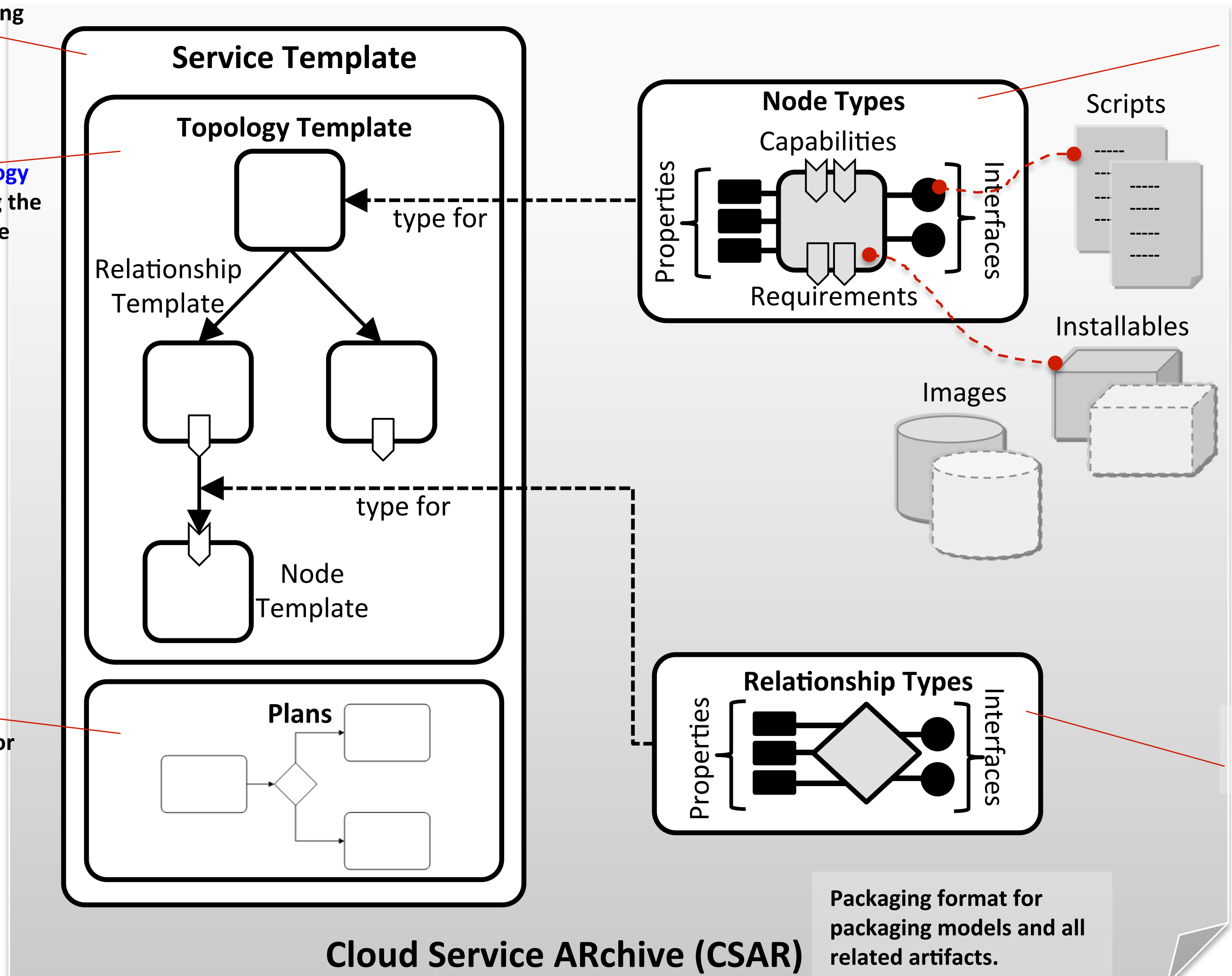
TOSCA

OASIS Topology and Orchestration Specification for Cloud Applications

A language for defining Service Templates …

… including a Topology Template describing the structure of a service

… including the definition of Plans for orchestrating the application



Definition of building blocks for services

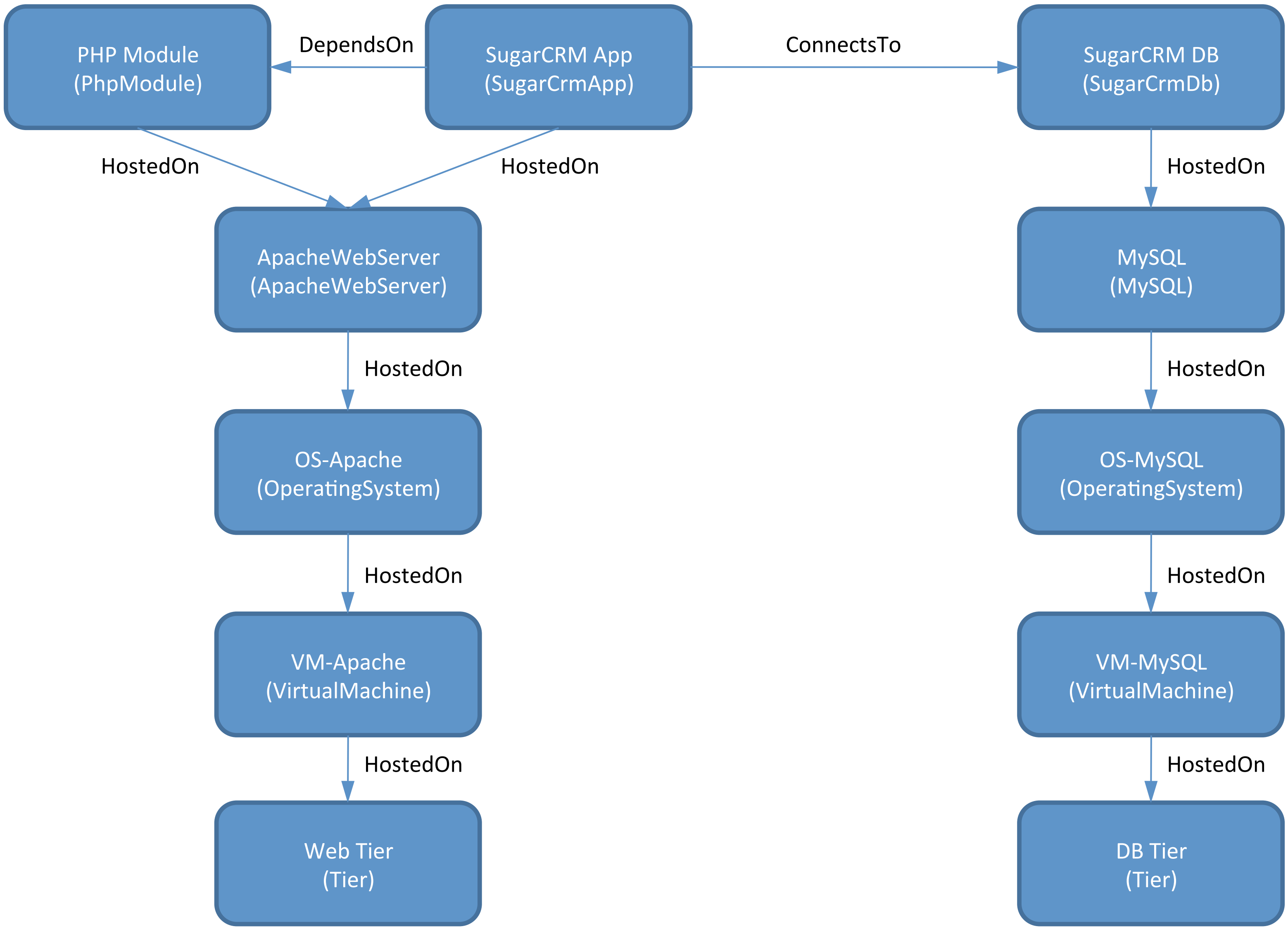
… along with the implementation artifacts for manageability operations

… and the definition of deployment artifacts for components

Definition of possible links between components

© Frank Leymann

Sample Topology: SugarCRM



© Frank Leymann

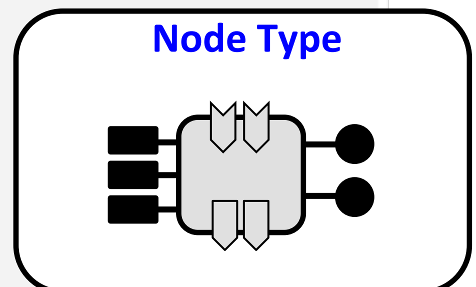
Definitions File: Overall Structure

```
<Definitions id="xs:ID" name="xs:string"? targetNamespace="xs:anyURI">

  <Extensions/>?
  <Import />*
  <Types/>?

 (  <ServiceTemplate/>
  | <NodeType/>
  | <NodeTypeImplementation/>
  | <RelationshipType/>
  | <RelationshipTypeImplementation/>
  | <RequirementType/>
  | <CapabilityType/
  | <ArtifactType/>
  | <ArtifactTemplate/>
  | <PolicyType/>
  | <PolicyTemplate/>   ) +

</Definitions>
```

© Frank Leymann

Node Type: Overall Structure

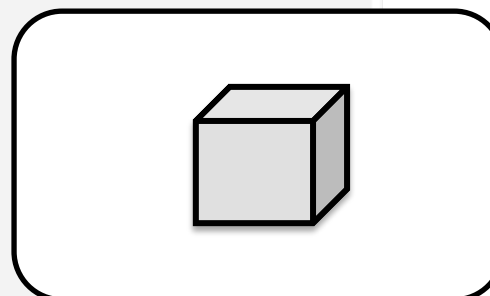
```
<NodeType name="xs:NCName" targetNamespace="xs:anyURI"?
               abstract="yes|no"? final="yes|no"?>+

  <Tags/>?
  <DerivedFrom nodeTypeRef="QName"/>?
  <PropertiesDefinition element="Qname"?
    type="QName"?/>?
  <RequirementDefinitions/>?
  <CapabilityDefinitions/>?
  <InstanceStates/>?
  <Interfaces/>?
</NodeType>
```



© Frank Leymann

Artifact Types

```
<ArtifactType name="xs:NCName"
          targetNamespace="xs:anyURI"?
          abstract="yes|no"?
          final="yes|no"?>

  <Tags>
    <Tag name="xs:string" value="xs:string"/> +
  </Tags> ?

  <DerivedFrom typeRef="xs:QName"/> ?

  <PropertiesDefinition element="xs:QName"? type="xs:QName"?/> ?

</ArtifactType>
```



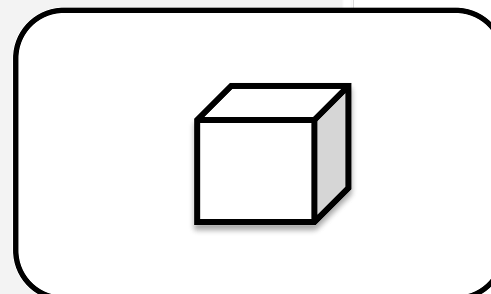
Invariant properties;
e.g. hash of the artifact

© Frank Leymann

Artifact Templates

```
<ArtifactTemplate id="xs:ID" name="xs:string"? type="xs:QName">

  <Properties>
   XML fragment
  </Properties> ?

  <PropertyConstraints>
   <PropertyConstraint property="xs:string"
                  constraintType="xs:anyURI"> +
    constraint ?
   </PropertyConstraint>
  </PropertyConstraints> ?

  <ArtifactReferences>
   <ArtifactReference reference="xs:anyURI">
    (  <Include pattern="xs:string"/>
     | <Exclude pattern="xs:string"/>  )*
   </ArtifactReference> +
  </ArtifactReferences> ?

</ArtifactTemplate>
```

Variant properties;
e.g. directory where to store the artifact



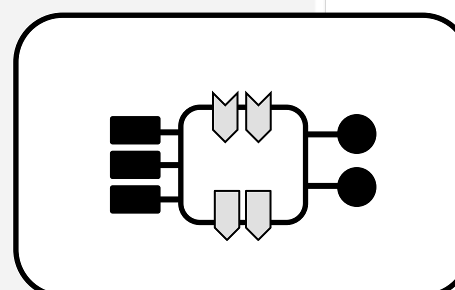
Relative URI is interpreted as pointer into CSAR;
Absolute URI specifies where to get the artifact

Can be used to define which files are collected in case the attribute “references” points to a complete directory (e.g. in the CSAR)

© Frank Leymann

Node Type Implementations

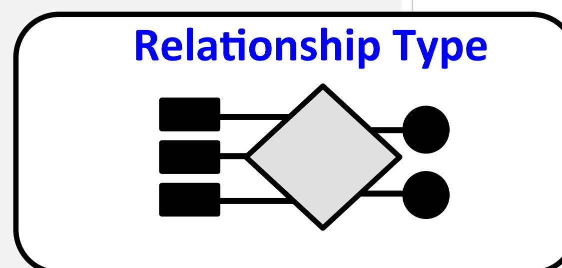
```
<NodeTypeImplementation name="xs:NCName"
        targetNamespace="xs:anyURI"?
        nodeType="xs:QName" abstract="yes|no"?  final="yes|no"?>

  <Tags/> ?

  <DerivedFrom nodeTypeImplementationRef="xs:QName"/> ?

  <RequiredContainerFeatures>
    <RequiredContainerFeature feature="xs:anyURI"/> +
  </RequiredContainerFeatures> ?

  <ImplementationArtifacts/> ?

  <DeploymentArtifacts/> ?

</NodeTypeImplementation>
```



© Frank Leymann

Relationship Types

```
<RelationshipType name="xs:NCName"
          targetNamespace="xs:anyURI"?
          abstract="yes|no"?
          final="yes|no"?> +

  <DerivedFrom typeRef="xs:QName"/> ?

  <PropertiesDefinition element="xs:QName"? type="xs:QName"?/> ?

  <InstanceStates>
    <InstanceState state="xs:anyURI"> +
  </InstanceStates> ?

  <SourceInterfaces.../>?

  <TargetInterfaces.../>?

  <ValidSource typeRef="xs:QName"/> ?

  <ValidTarget typeRef="xs:QName"/> ?

</RelationshipType>
```



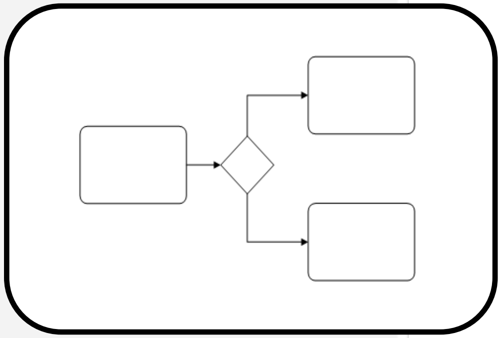
NodeType or Requirement Type

NodeType or Capability Type

© Frank Leymann

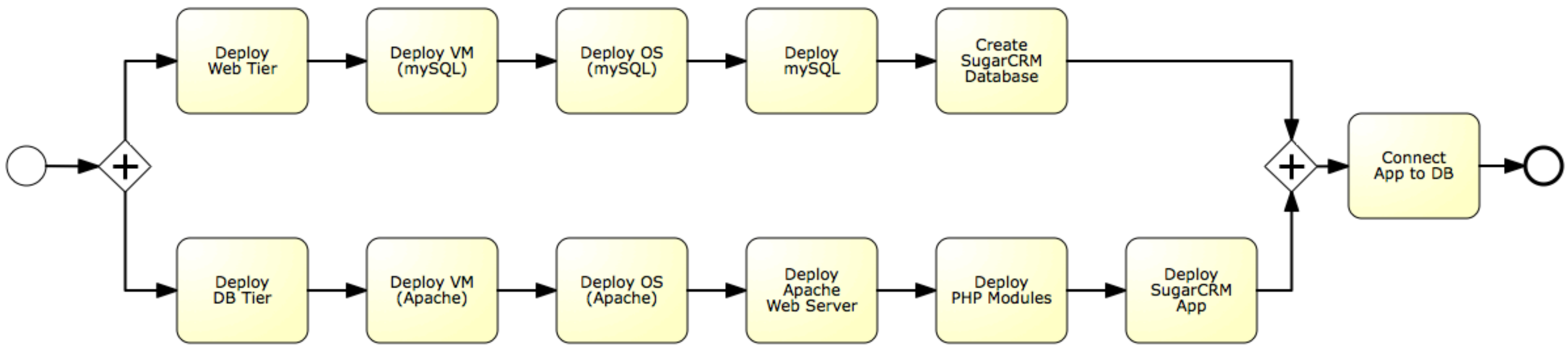
Plans

```
<Plans>
 <Plan id="ID"
      name="string"?
      planType="anyURI"
      languageUsed="anyURI">
   <PreCondition expressionLanguage="anyURI">?
      condition
   </PreCondition>
   <InputParameters>
      <InputParameter name="xs:string" type="xs:string"
                      required="yes|no"?/> +
   </InputParameters> ?
   <OutputParameters>
      <OutputParameter name="xs:string" type="xs:string"
                       required="yes|no"?/> +
   </OutputParameters> ?
   ( <PlanModel>  actual plan </PlanModel>
   |
   <PlanModelReference reference="anyURI"/>  )
 </Plan>+
</Plans>
```

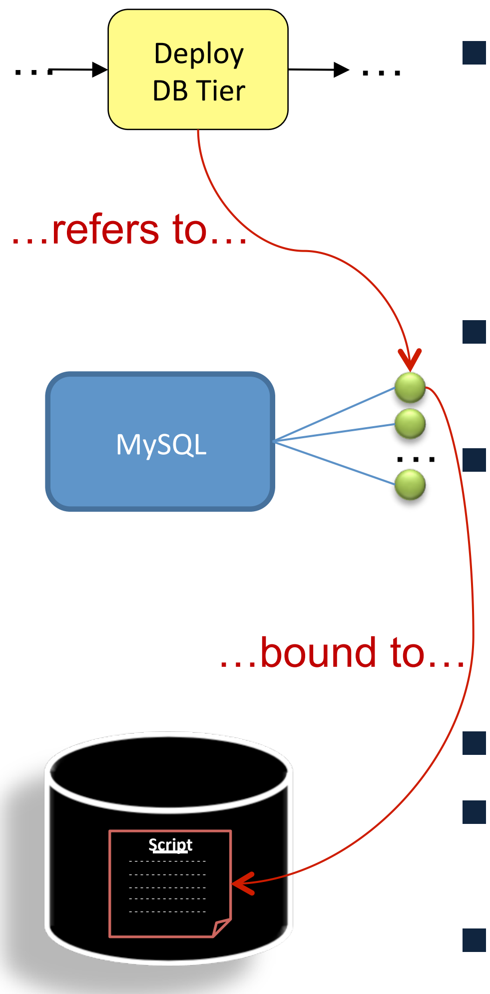


Plans

Sample: SugarCRM Build Plan

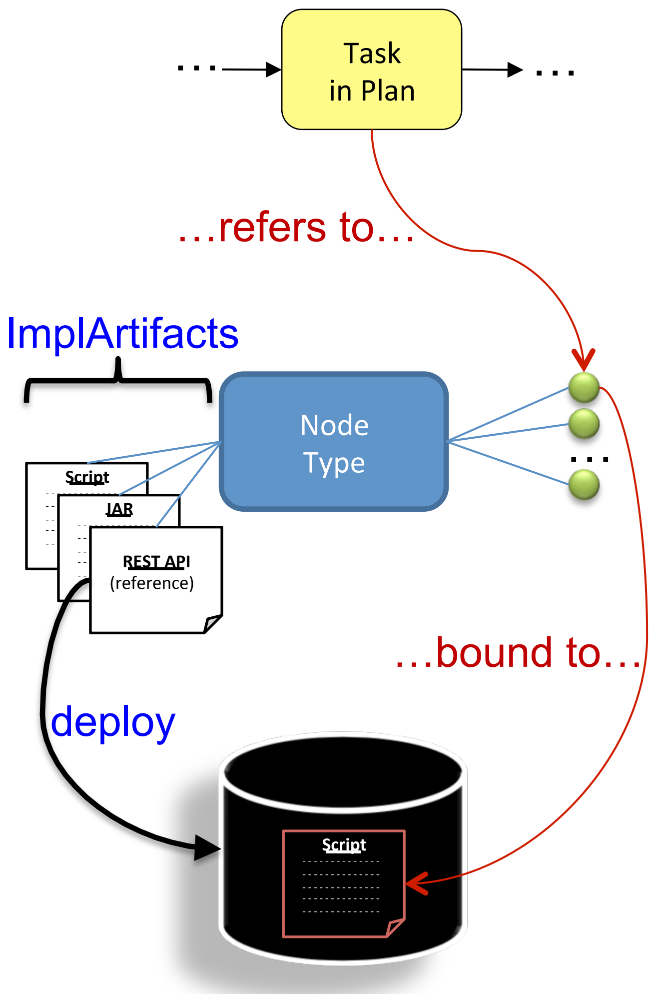


How Plans and Nodes Fit Together



- Task of a plan refers to interface of a topology node
- Topology node specifies all interfaces offered to manage it
- Interface is bound to a concrete implementation
- Implementation already available at providers side, or
- Implementation is copied from CSAR (Cloud Service ARchive), or
- A standardized Cloud Interface (Iaas, PaaS, SaaS) is used, or ...

Implementation Artifacts

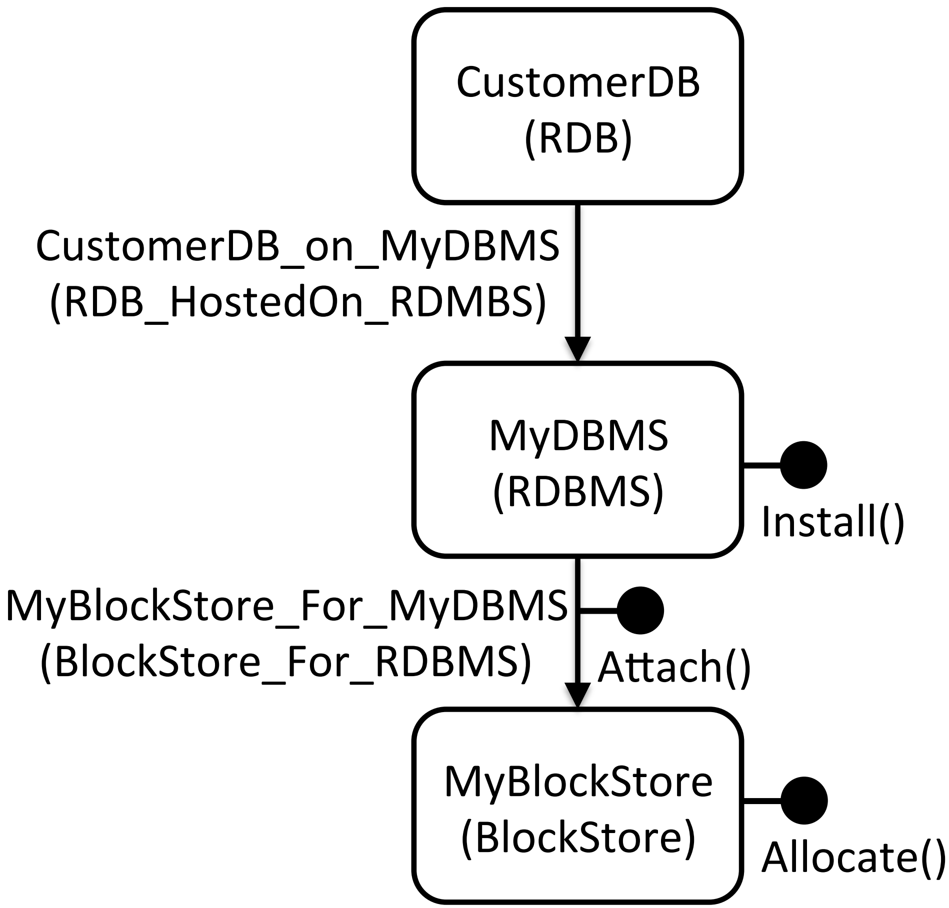


- When a node type implementation is imported, its implementation artifacts are deployed
  - From that time on, the operations of the node types can be used in the particular environment
- Now, tasks of the plans can be bound to the implementation of the operations in this environment
  - I.e. plans are bound to the environment (as usual) in which they are executing

Agenda

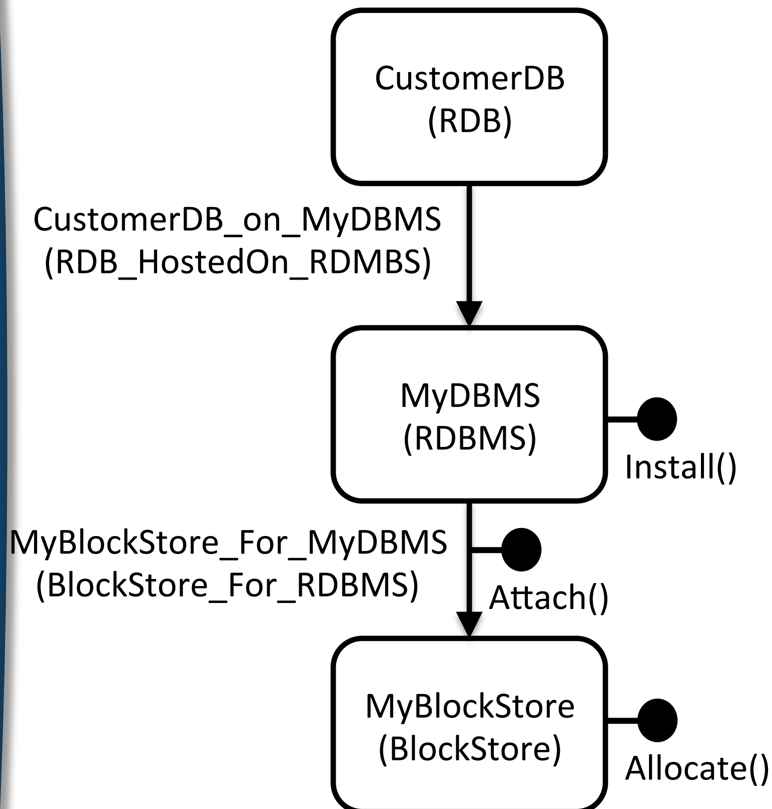
- The Need for Topologies
- TOSCA Quick Overview
- Declarative vs Imperative Processing
- TOSCA Simple Profile
- Orchestration Engines Architecture
- Summary

A Sample Topology



© Frank Leymann

...And Its *Declarative* Processing

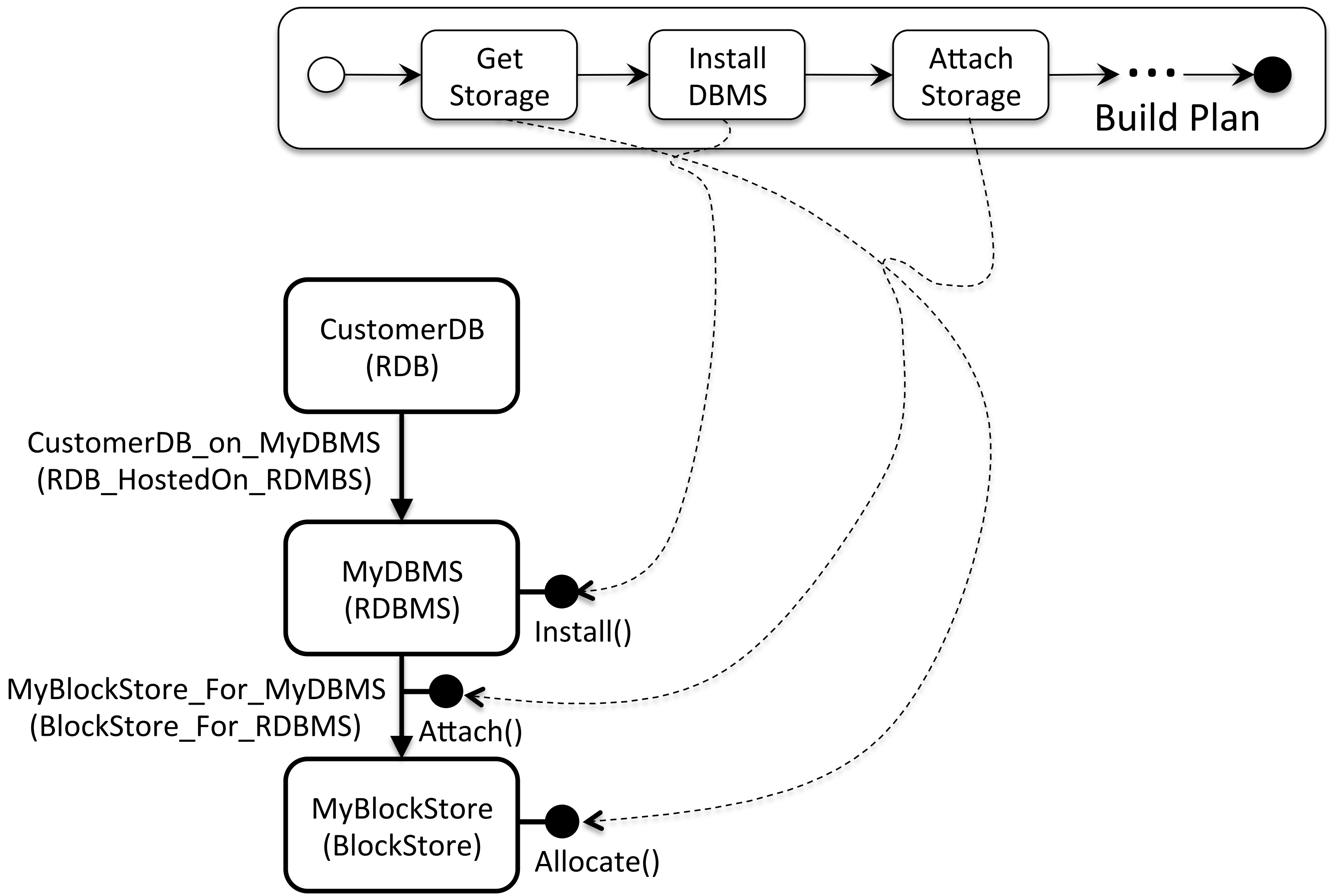


- In a declarative mode of processing, the environment does understand the specific processing requirements of all types
  - Node types
  - Relationship types
  - ...
- It further understands the dependencies of all these types
  - E.g. that hosted_on relationships must be processed before connected_to relationships

PRO: For provisioning and decommissioning, no plans need to be specified
CON: Very precise definition of all types and their dependencies must be specified

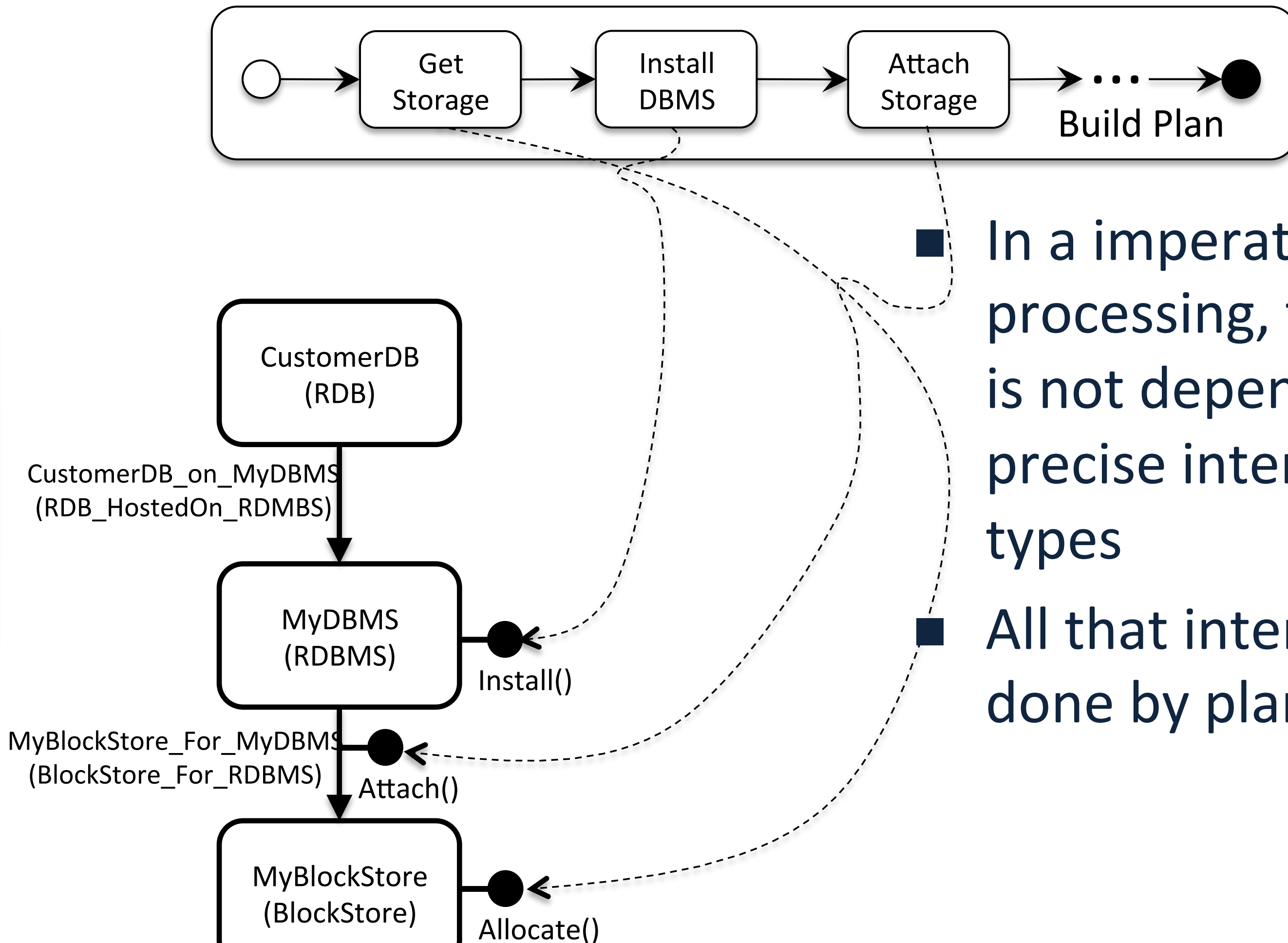
© Frank Leymann

A Sample Topology With Plans



© Frank Leymann

...And Its *Imperative* Processing



- In a imperative mode of processing, the environment is not dependent on a precise interpretation of the types
- All that interpretation is done by plans

PRO: No precise definition of all types, their processing, their behavior,... needed
CON: Plans must be specified even for "simple" provisioning and decommissioning needed

Declarative vs Imperative: Some Pros and Cons

Declarative

+ Simplicity
  - No plans modeling
+ No requirement for additional middleware
  - WfMS,...
– Restricted coverage of orchestrations
  - Deployment & Decommissioning only
– Limited support of complex topologies
  – „Interpreting“ cycles, multiple links between two nodes...
– Clear definition of semantics required

Imperative

+ Full coverage of orchestrations
  - Licensing, monitoring,...
+ All workflow features
  - Compensation, Humans,...
– Additional skills required
– Additional middleware required
– Increased maintenance effort
  - Plans must be maintained

Agenda

- The Need for Topologies
- TOSCA Quick Overview
- Declarative vs Imperative Processing
- TOSCA Simple Profile
- Orchestration Engines Architecture
- Summary

Goals of the Simple Profile

Make TOSCA consumable by a broader community

This implies:

- Allow to omit language elements that are not needed in "simple cases"
  - E.g. don't use Relationship Types, Plans

TOSCA Simple Profile becomes fully declarative

- Extend TOSCA with language elements that make simple cases simpler
  - E.g. Template Inputs and Outputs
- Don't enforce XML
  - Instead, provide a YAML rendering of TOSCA Simple

...and here, latest, we get very religious!

...people can really fight about this rendering issue! ☹

Node Templates

```
tosca_definitions_version: tosca_simple_yaml_1_0

description: Template for deploying a single server with predefined properties.

node_templates:
  my_server:
    type: tosca.nodes.Compute
    properties:
      # compute properties
      disk_size: 10
      num_cpus: 2
      mem_size: 4
      # host image properties
      os_arch: x86_64
          os_type: linux
          os_distribution: rhel
          os_version: 6.5
```

© Frank Leymann

Inputs and Outputs of a Template

```
inputs:
  cpus:
    type: integer
    description: Number of CPUs for the server.
    constraints:
      - valid_values: [ 1, 2, 4, 8 ]

node_templates:
  my_server:
    type: tosca.nodes.Compute
    properties:
      # Compute properties
      num_cpus: { get_input: cpus }
      mem_size: 4
      disk_size: 10
      # host image properties
      os_arch: x86_32
             os_type: linux
             os_distribution: ubuntu
             os_version: 12.04

outputs:
  server_ip:
    description: The IP address of the provisioned server.
    value: { get_property: [ my_server, ip_address ] }
```

© Fra

Associating Node Templates

```
node_templates:
  mysql:
    type: tosca.nodes.DBMS.MySQL
    properties:
      dbms_root_password: { get_input: my_mysql_rootpw }
      dbms_port: { get_input: my_mysql_port }
    requirements:
      - host: db_server

  db_server:
    type: tosca.nodes.Compute
    properties:
      # omitted here for sake of brevity
```

© Frank Leymann

Requirements

```yaml
node_templates:
  my_app:
    type: my.types.MyApplication
    properties:
      admin_user: { get_input: admin_username }
      admin_password: { get_input: admin_password }
      db_endpoint_url: { get_ref_property: [ database, db_endpoint_url ] }
    requirements:
      - database: tosca.nodes.DBMS.MySQL
        constraints:
          - mysql_version: { greater_or_equal: 5.5 }
```

Lifecylce Interface

```yaml
tosca_definitions_version: tosca_simple_yaml_1_0

description: Template for deploying a single server with MySQL software on top.

inputs:
  # omitted here for sake of brevity

node_templates:
  mysql:
    type: tosca.nodes.DBMS.MySQL
    properties:
      dbms_root_password: { get_input: my_mysql_rootpw }
      dbms_port: { get_input: my_mysql_port }
    requirements:
      - host: db_server
    interfaces:
      Lifecycle:
        configure: scripts/my_own_configure.sh

  db_server:
    type: tosca.nodes.Compute
    properties:
      # omitted here for sake of brevity
```

Artifacts

```
node_templates:
  my_db:
    type: tosca.nodes.Database.MySQLDatabase
    properties:
      db_name: { get_input: database_name }
      db_user: { get_input: database_user }
      db_password: { get_input: database_password }
      db_port: { get_input: database_port }
    artifacts:
      - db_content: files/my_db_content.txt
        type: tosca.artifacts.File
    requirements:
      - host: mysql
```

Relationship Types

```
node_templates:
 wordpress:
  type: tosca.nodes.WebApplication.WordPress
  properties:
   # omitted here for sake of brevity
  requirements:
   - host: apache
   - database: wordpress_db
    relationship_type: my.types.WordpressDbConnection

relationship_types:
 my.types.WordpressDbConnection:
  derived_from: tosca.relations.ConnectsTo
  interfaces:
   Configure:
    pre_configure_source: scripts/wp_db_configure.sh
```

Standardized Types

- To help declarative processing succeed very (very!!!) detailed descriptions of standardized types must be provided
  - Especially the operational semantics of these types must be very precisely defined, e.g.
    - The effects of operations
    - The order in which relationship types are to be processed
    - How to match requirements
    - …
  - And this makes defining your own corresponding types really hard
    - How to define how your custom types are to be processed, i.e. what the effects of operations are; in which order your relationship types have to be considered

Again: another source of significant fights! ☹

Standardized Capabilities - Samples

```
tosca.capabilities.Endpoint:
  properties:
    protocol:
      type: string
      default: http
    port:
      type: integer
      constraints:
        - greater_or_equal: 1
        - less_or_equal: 65535

tosca.capabilities.DatabaseEndpoint:
  derived_from: tosca.capabilities.Endpoint
```

© Frank Leymann

Standardized Relationship Types - Samples

```
tosca.relationships.Root:
  # The TOSCA root relationship type has no property mappings
  interfaces: [ tosca.interfaces.relationship.Configure ]

tosca.relationships.DependsOn:
  derived_from: tosca.relationships.Root
  valid_targets: [ tosca.capabilities.Feature ]

 tosca.relationships.HostedOn:
   derived_from: tosca.relationships.DependsOn
   valid_targets: [ tosca.capabilities.Container ]

tosca.relations.ConnectsTo:
  derived_from: tosca.relationships.DependsOn
  valid_targets: [ tosca.capabilities.Endpoint ]
```

© Frank Leymann

Standardized Interfaces - Samples

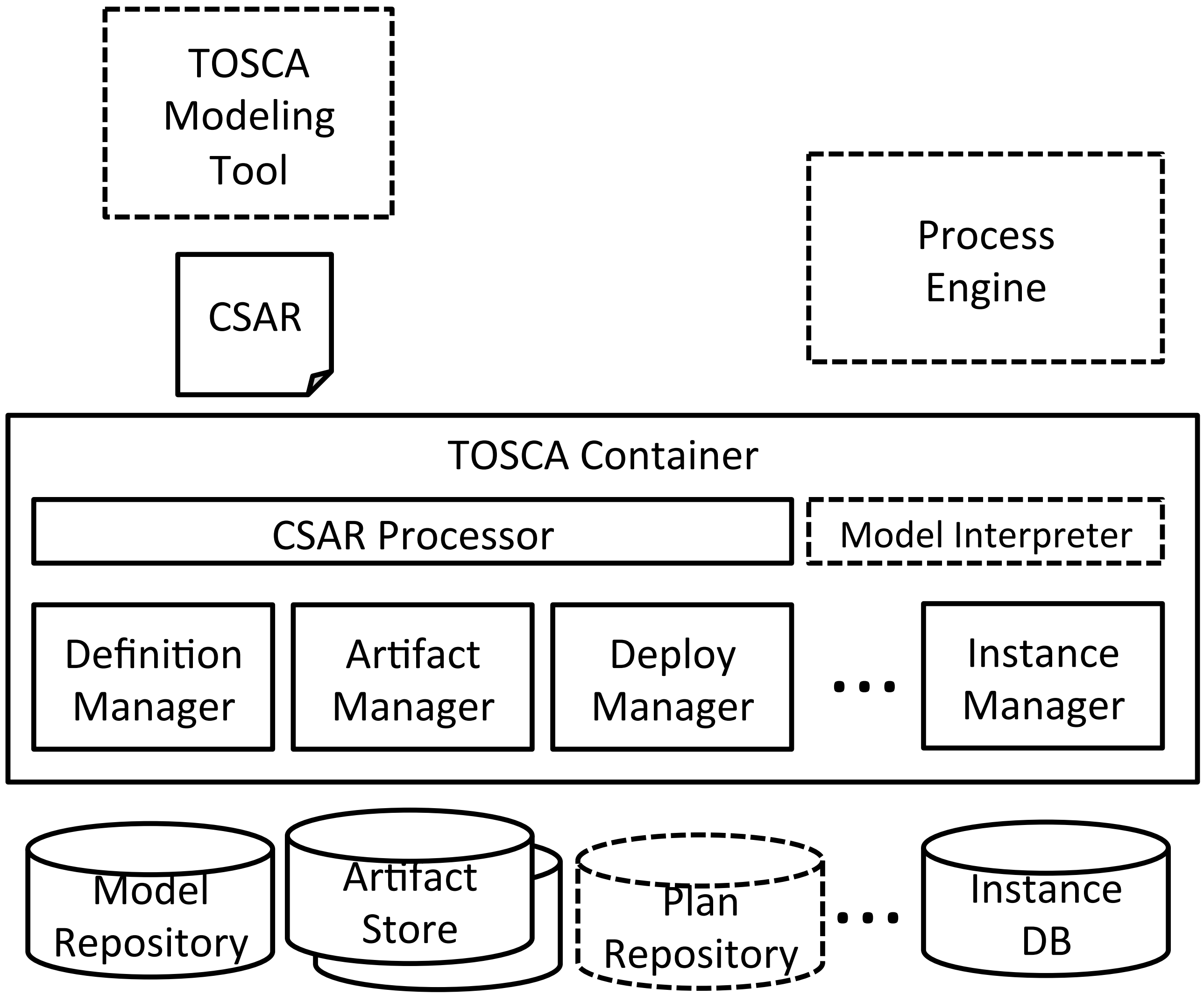
```
tosca.interfaces.node.Lifecycle:
 create:
  description: Basic lifecycle create operation.
 configure:
  description: Basic lifecycle configure operation.
 start:
  description: Basic lifecycle start operation.
 stop:
  description: Basic lifecycle stop operation.
 delete:
  description: Basic lifecycle delete operation.
```

© Frank Leymann

Agenda

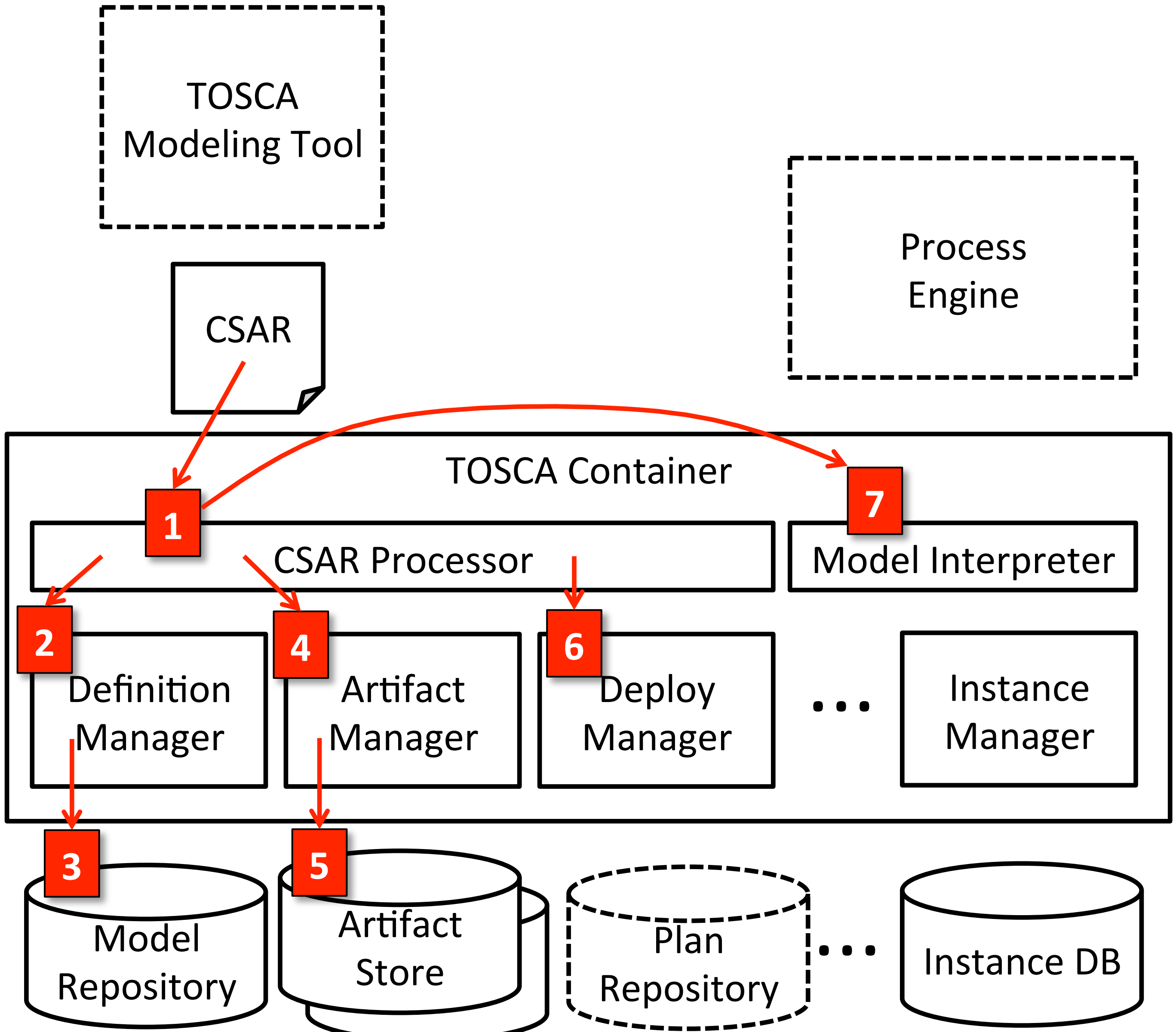
- The Need for Topologies
- TOSCA Quick Overview
- Declarative vs Imperative Processing
- TOSCA Simple Profile
- Orchestration Engines Architecture
- Summary

TOSCA Environment: Sample High-Level Architecture



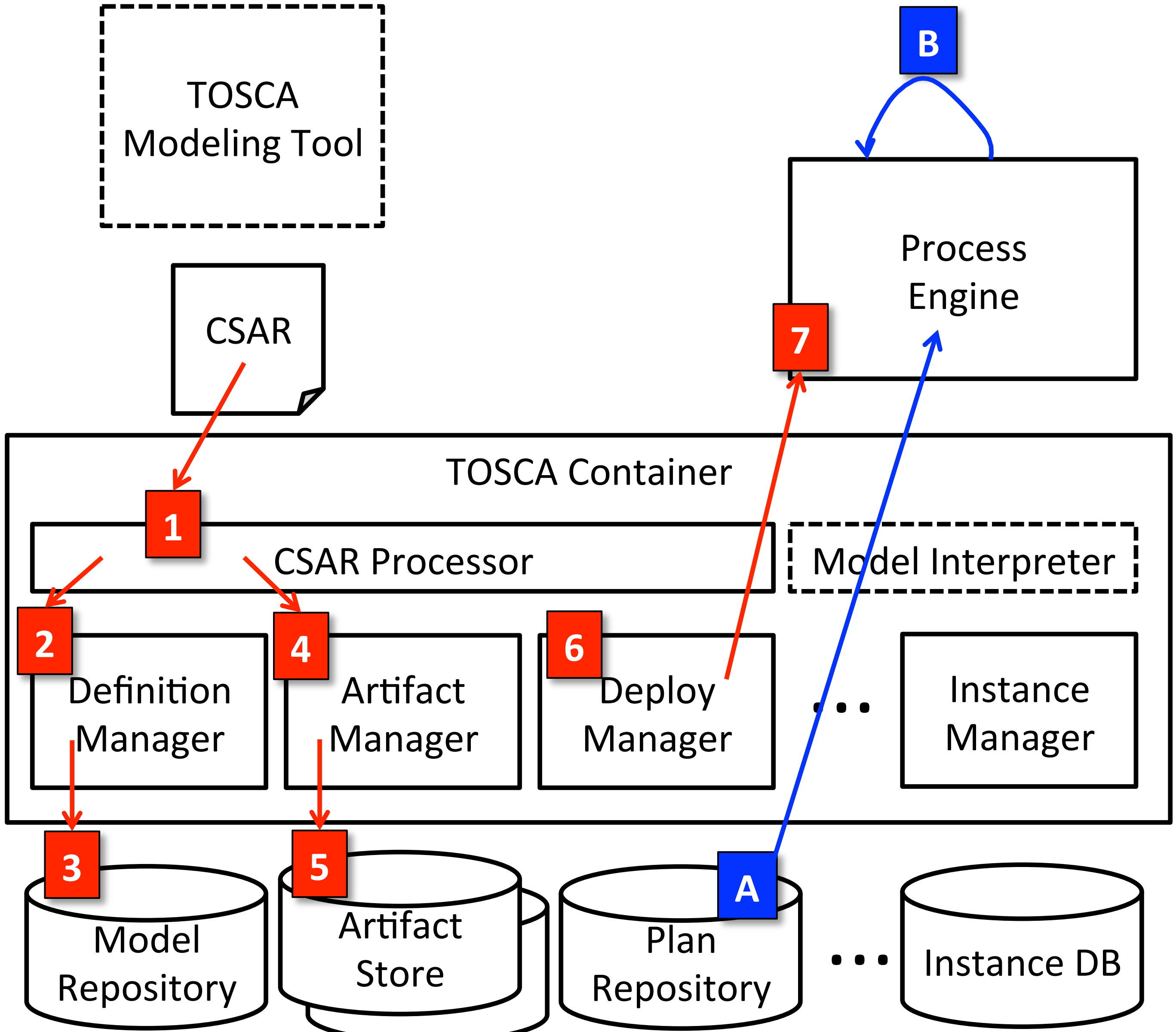
© Frank Leymann

Declarative Approach: Component Flow

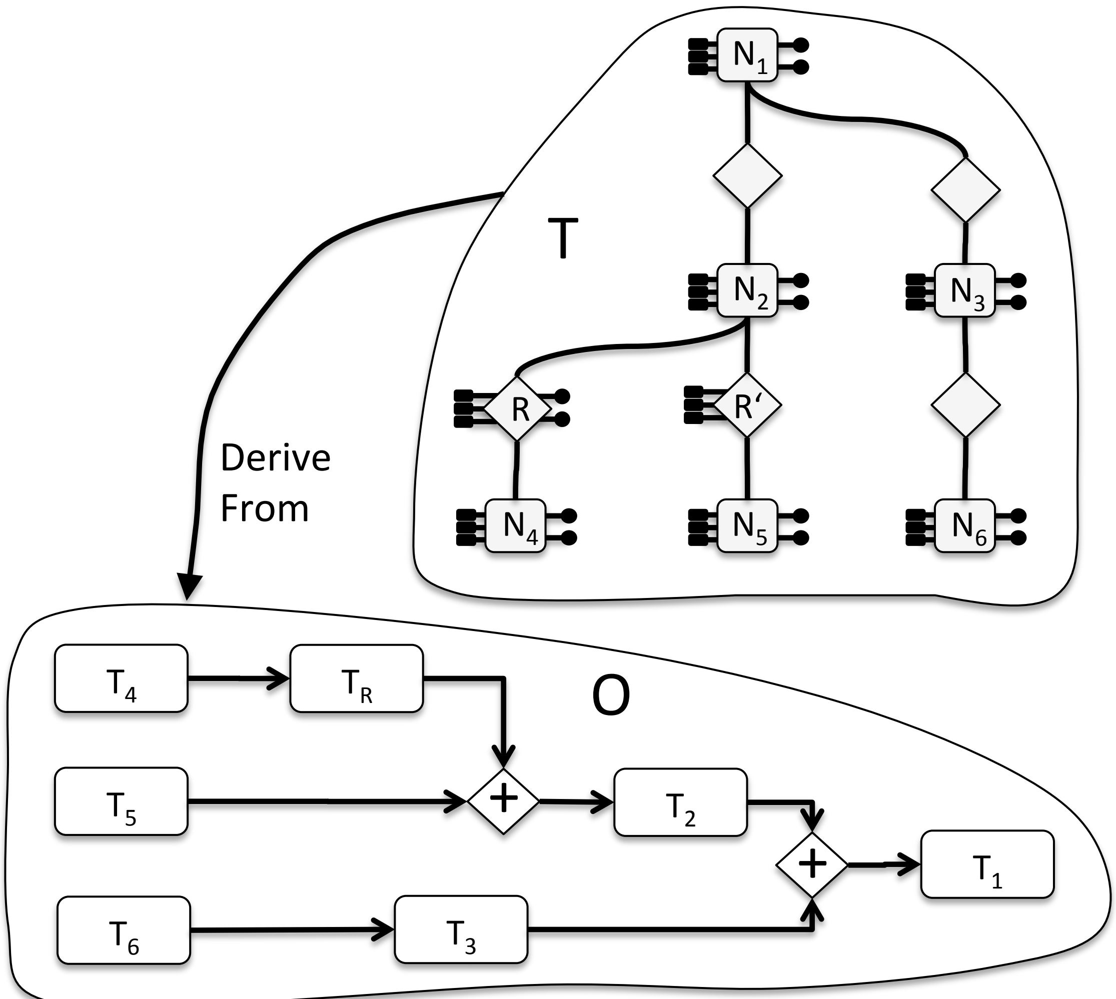


© Frank Leymann

Imperative Approach: Component Flow

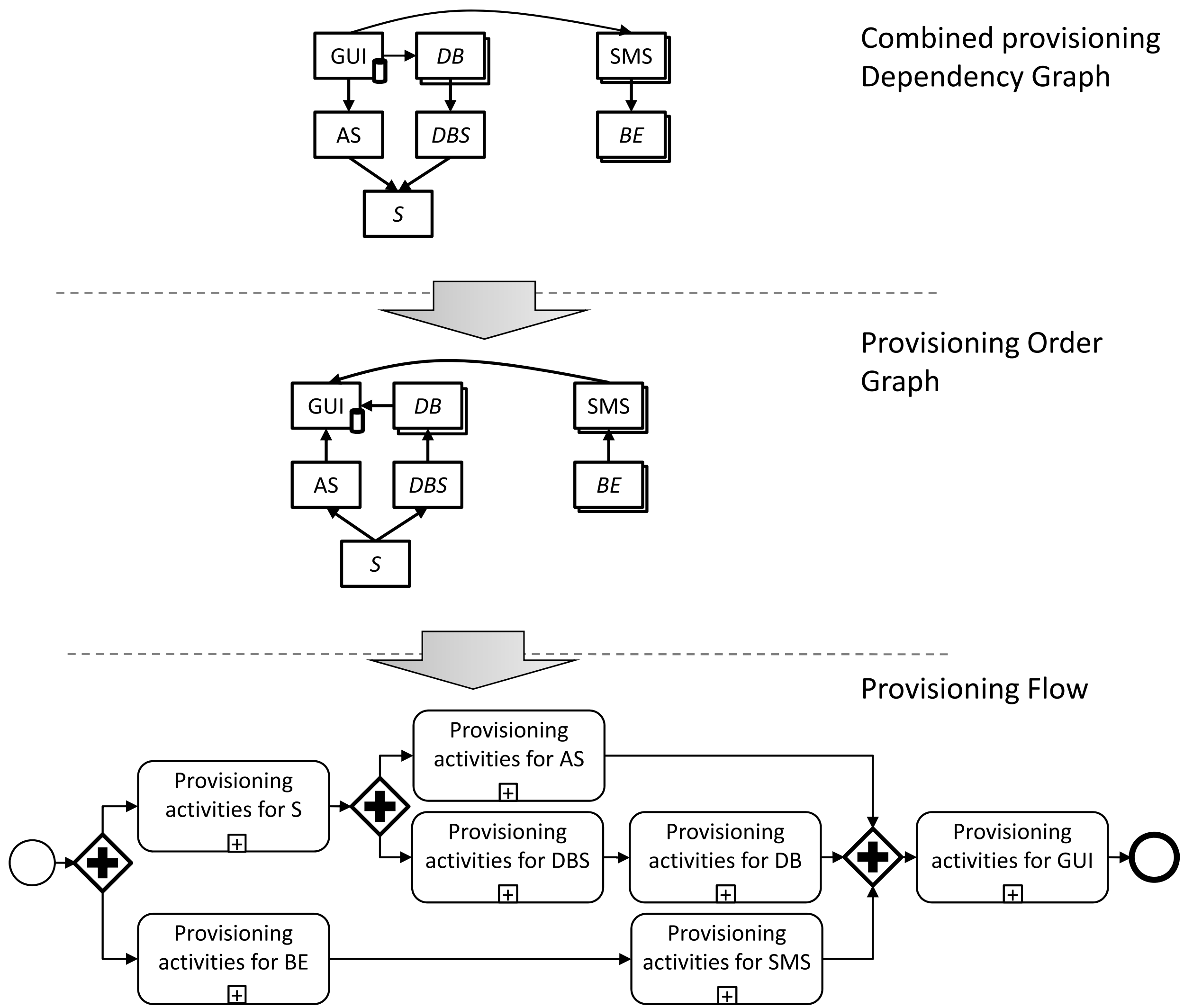


Deriving Plans from Topologies: The Basic Principle



© Frank Leymann

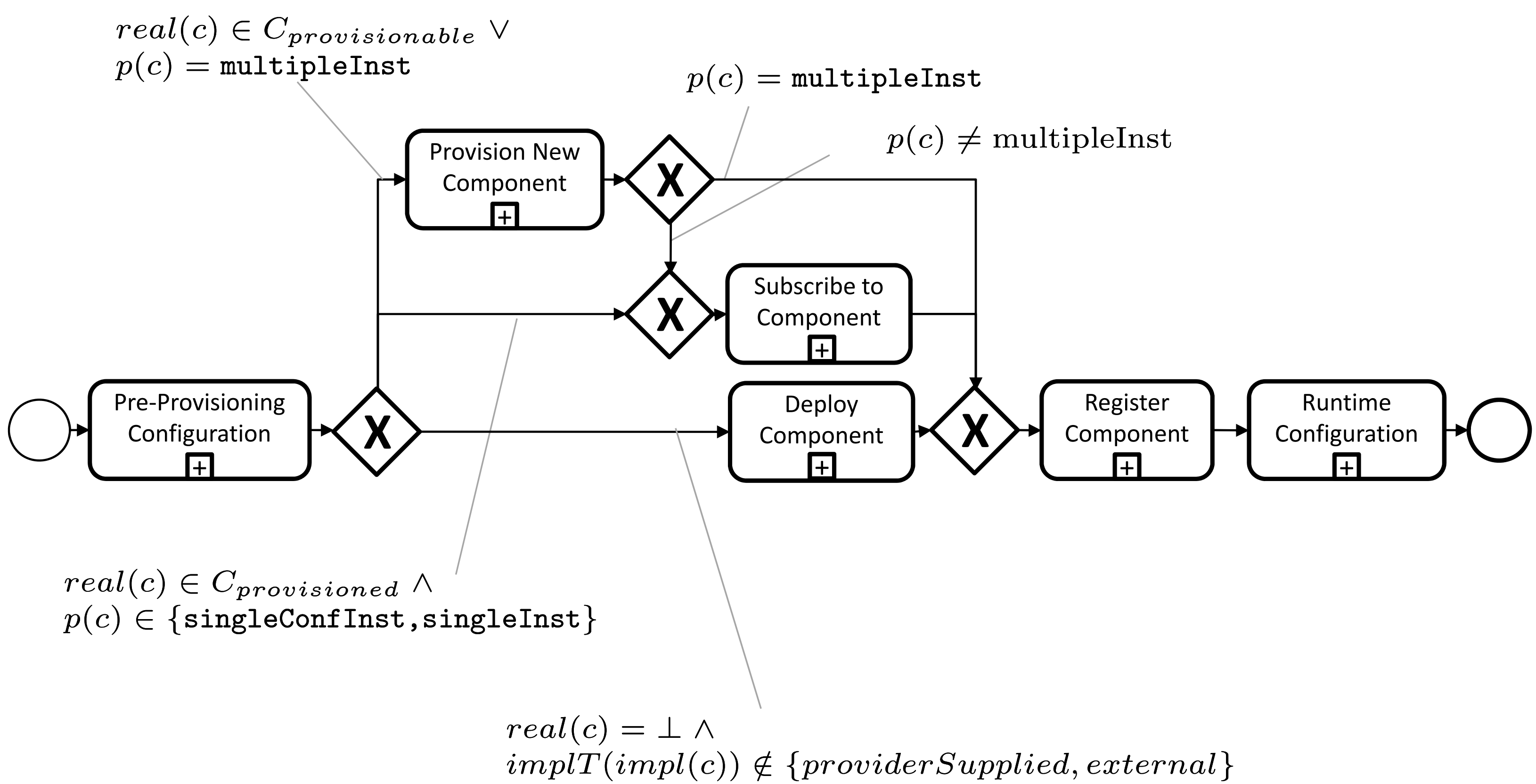
Some More Details – At a Glimpse



From Ralph Mietzner's PhD Thesis, 2010

© Frank Leymann

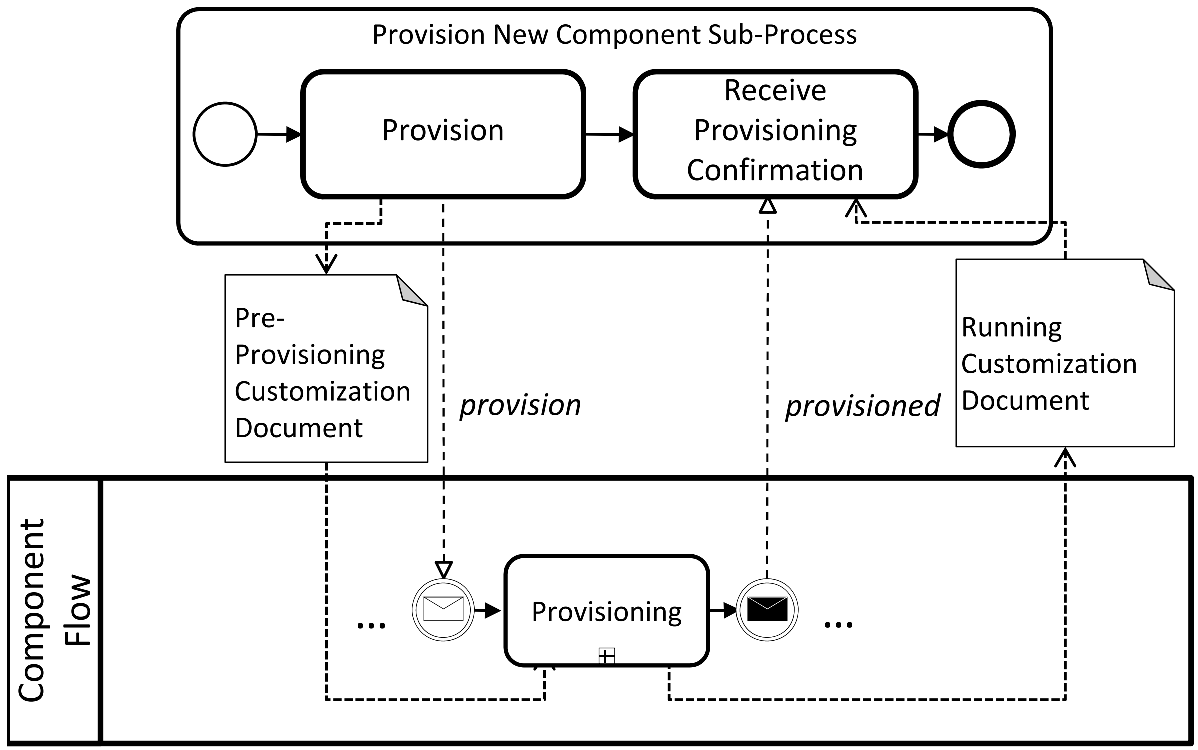
At a Glimpse: The Provisioning Subflows



From Ralph Mietzner's PhD Thesis, 2010

© Frank Leymann

At a Glimpse: Provision New Component Subflow



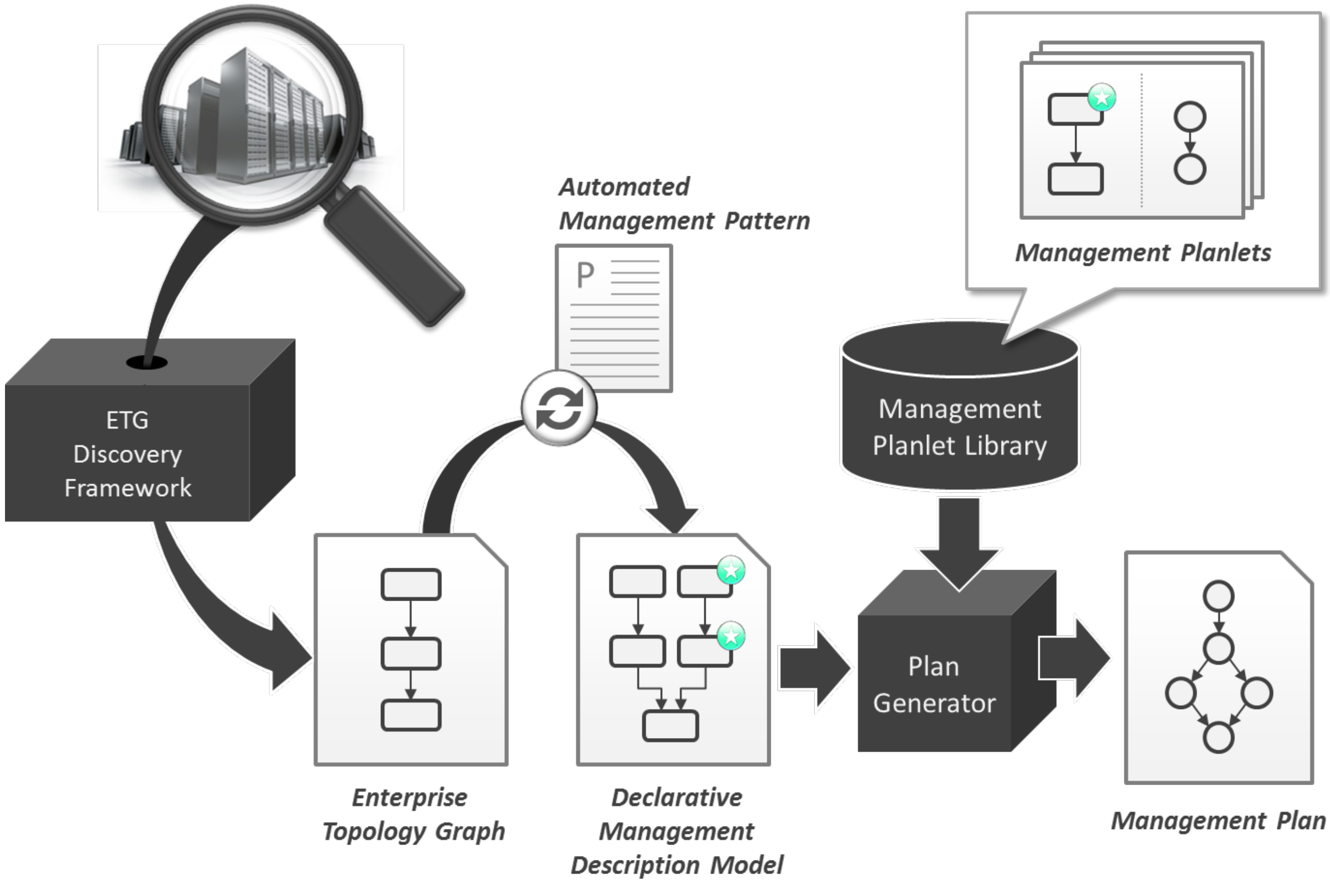
...and so on: the whole generation of "build plans" can be read in Ralph's PhD thesis ☺

From Ralph Mietzner's PhD Thesis, 2010

© Frank Leymann

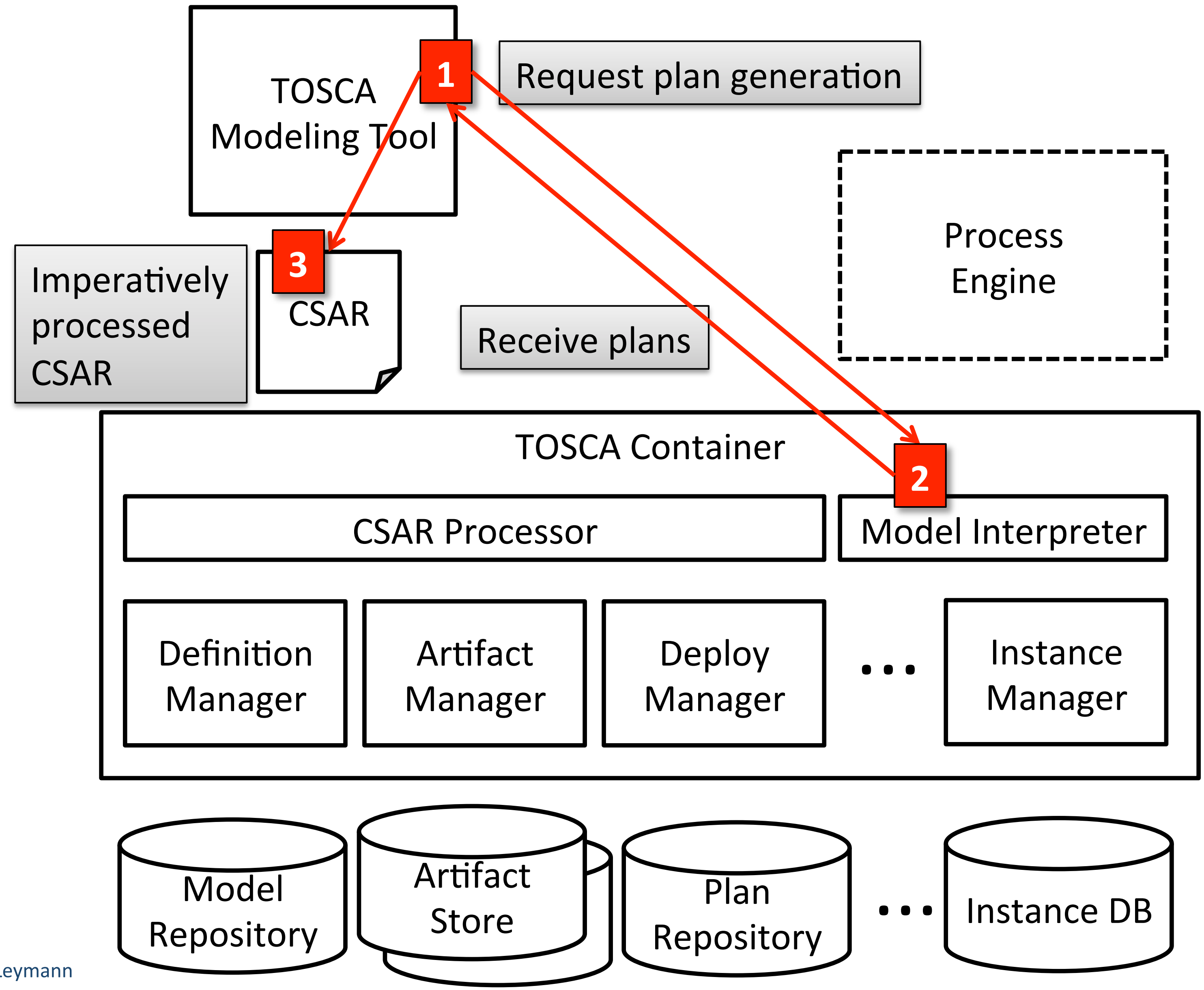
Generating Management Plans

- This is more complicated!



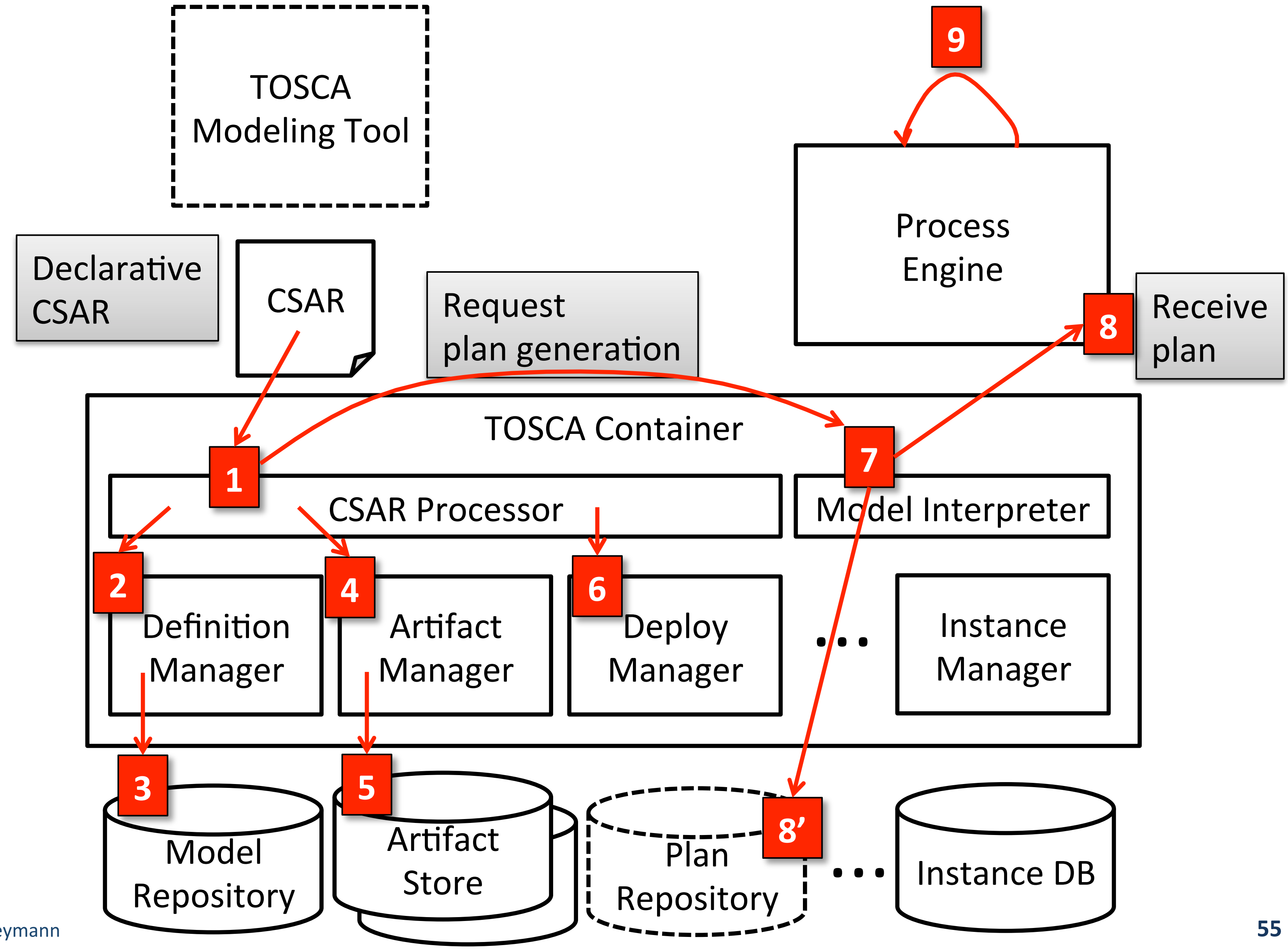
…see Uwe Breitenbücher's Poster on his PhD thesis ☺

Turning Declarative into Imperative: Buildtime



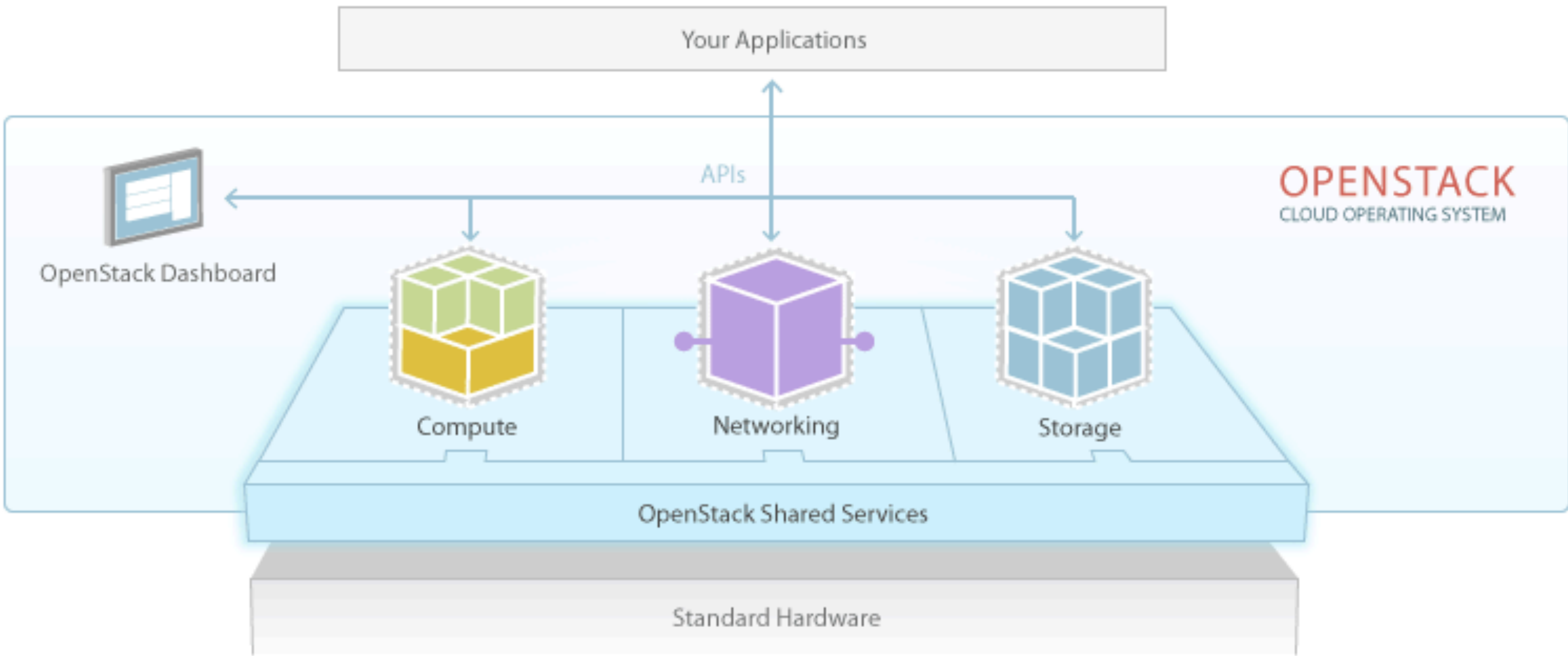
© Frank Leymann

Turning Declarative into Imperative: Runtime



© Frank Leymann

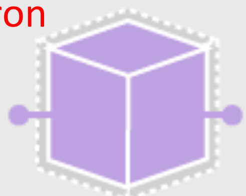
OpenStack – High Level Architecture Components



© By OpenStack

Openstack Components

Neutron



OpenStack Networking: Pluggable, scalable, API-driven network and IP management

Nova

OpenStack Compute: Provision and manage large networks of virtual machines

Cinder

Swift

OpenStack Storage: Object and Block storage for use with servers and applications

Image Management
Glance

Orchestration Service
Heat

Relational Database
Trove

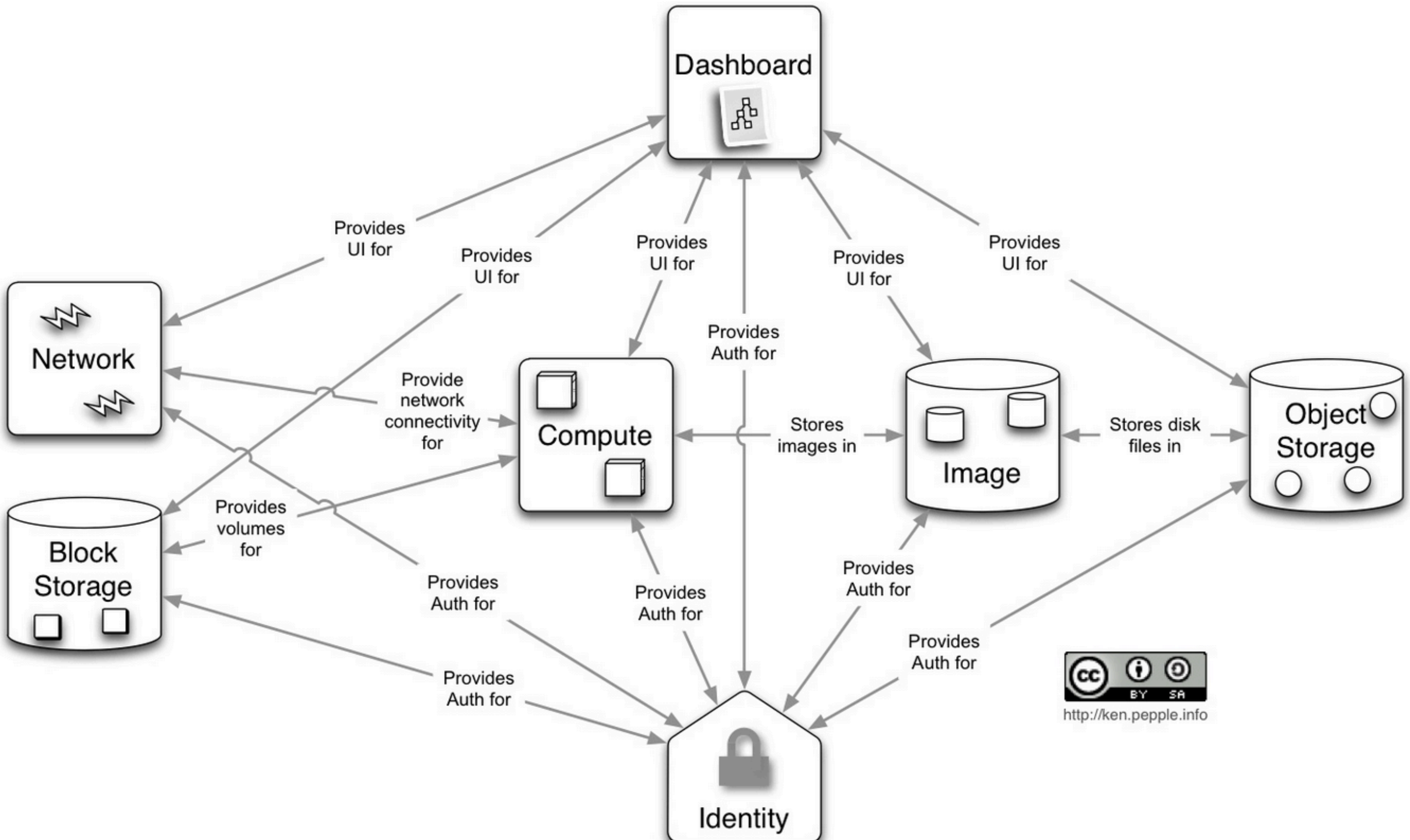
Telemetry
Ceilometer

Identity Management
Keystone

OpenStack Shared Services

© By OpenStack

OpenStack: Associations Between Components



© By Wikipedia

Agenda

- The Need for Topologies
- TOSCA Quick Overview
- Declarative vs Imperative Processing
- TOSCA Simple Profile
- Orchestration Engines Architecture
- Summary

Summary

- Capturing images of an application and bursting it to the cloud is the wrong way
  - You loose most benefits of the cloud
- To enable applications to benefit from cloud properties their topology and management behavior must be defined
- Standards and (open source) implementations exist for such orchestration of cloud applications
- There are two approaches for realizing cloud orchestration: declarative and imperative
- Lot’s of research opportunities in this space

© Frank Leymann

Next parts of the Tutorial:

Provisioning Techniques - Johannes

OpenTOSCA Deep Dive - Uwe

The End