# Legato: A Component Model to Decouple Application Assembly from Nested and Heterogeneous Deployments

## STACK Team Seminar

Quentin GUILLOTEAU[1], Hélène COULLON[2], Christian PEREZ[1]

2026-07-16

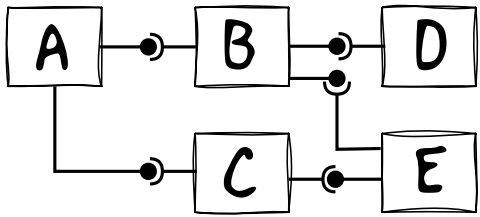
[1]Inria
[2]IMT Atlantique

RÉPUBLIQUE FRANÇAISE
*Liberté*
*Égalité*
*Fraternité*

FRANCE 2030
PROGRAMME DE RECHERCHE
CLOUD

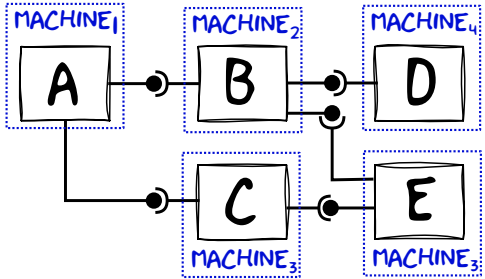
# Context – Deployment of Cloud Applications

- deployment of cloud applications
- application = set of (micro)services with interfaces, interconnected
- can be modeled as black box components with ports (e.g., `USE`/`PROVIDE`) for better composability
- connected components = assembly



# Context – Deployment of Cloud Applications

- deployment of cloud applications
- application = set of (micro)services with interfaces, interconnected
- can be modeled as black box components with ports (e.g., `USE`/`PROVIDE`) for better composability
- connected components = assembly




↪ Software components need to be deployed onto "**machines**"

# Resources, Provisioning, and Config. Management

**What even are "machines"?**

Does not have to be machines $\rightsquigarrow$ VMs, containers, baremetal, etc.

$\hookrightarrow$ "Resources" $\rightsquigarrow$ Execution units

Usually:

1. Provisioning (e.g., via Terraform):
   - Descriptions of the resources
   - Manual, depends on the tool used
2. Configuration Management (e.g., with Ansible):
   - Installation/configuration/etc.
   - Might depend on the resources

# Resources, Provisioning, and Config. Management

**What even are "machines"?**

Does not have to be machines $\rightsquigarrow$ VMs, containers, baremetal, etc.

$\hookrightarrow$ "Resources" $\rightsquigarrow$ Execution units

Usually:

1. Provisioning (e.g., via Terraform):
   - Descriptions of the resources
   - Manual, depends on the tool used
2. Configuration Management (e.g., with Ansible):
   - Installation/configuration/etc.
   - Might depend on the resources

$\hookrightarrow$ Mapping between the resources and the components?

# Motivations and Research Statement

## Research Statement

How to model component assemblies to support their deployment in **various execution contexts**?

## Portability

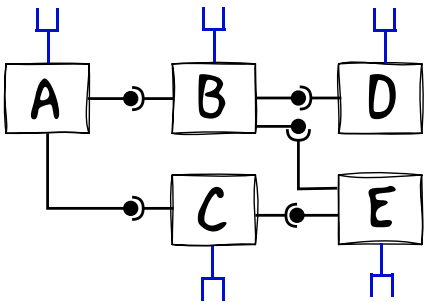
- Application developer not responsible to execution context
- Avoiding vendor-locking
- Multi-cloud deployments
- Resource heterogeneity

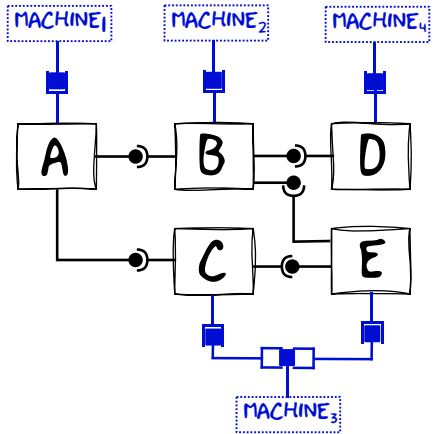
## Unavailable Services

- Resource provider has some issue
- Some resources or services are not available
- but need to deploy the application right now, with whatever resources available

# Legato – Modelisation

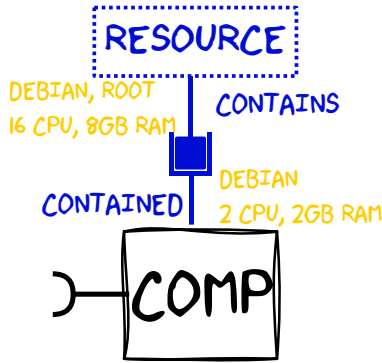
## ⇝ Model resources as components



# Legato – Model – Components, ResourcePorts, and Attributes

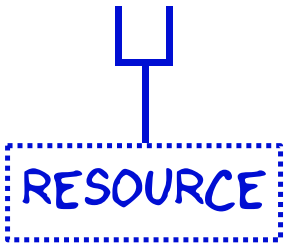
- Components have the classical `USE/PROVIDE` ports
- but also ResourcePorts
  - port types (eq. to use/provide): `CONTAINED/CONTAINS`
  - models where a component is deployed
  - One port = one resource
  - One component can model several resources
- Set of **attributes** linked to the ResourcePorts (e.g., `CPU`, `MEMORY`, CO2/W, etc.)
- Variants: same ports `USE/PROVIDE`, but various ports `CONTAINED/CONTAINS`



# Legato – Categories of Resource Components

## Applicative Components

- Uses one, or more, resources
- Does not provide any resource

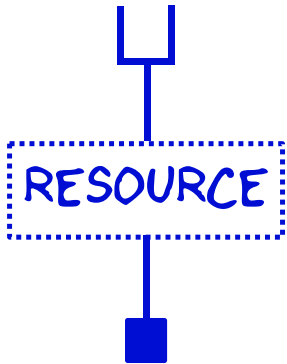


# Legato – Categories of Resource Components

## Applicative Components

## Transformers/Converters

- "Transforms" one resource into another one
- Use one resource, provide one resource
- e.g., Virtual Machines, Containers,



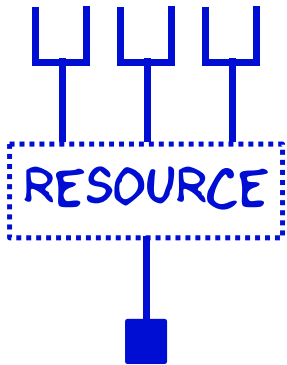
# Legato – Categories of Resource Components

## Applicative Components

## Transformers/Converters

## Resources Groups

- Gather several resources into a single interface
- e.g., Kubernetes



# Legato – Categories of Resource Components

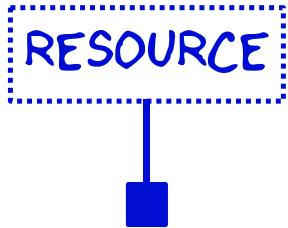
**Applicative Components**

**Transformers/Converters**

**Resources Groups**

**Allocations**

- Already allocated resources
- Resources acquired via allocation
- Only provides resource(s)
- e.g., local machine, Grid'5000, AWS



# Legato – Model – Instances and Assemblies

## Instances

Instance = Component Type + Identifier

(For both applicative and resource components!)

Component Type = Ports + ResourcePorts + Attributes

# Legato – Model – Instances and Assemblies

## Instances – Component Type + ID

## Assembly

- Set of component instances
- Port Connections
  - between application ports
  - but also between ResourcePorts

# Legato – Model – Instances and Assemblies

**Instances – Component Type + ID**

**Assembly – Set of Component Instances + Port Connections**

**Incomplete Assembly**

Assembly where at least one ResourcePort of type `CONTAINED` is not connected

**Complete Assembly**

Assembly where all the ResourcePorts of type `CONTAINED` are connected

$\hookrightarrow$ A complete assembly is "deployable"

# Legato – Model – Instances and Assemblies

**Instances – Component Type + ID**

**Assembly – Set of Component Instances + Port Connections**

**Incomplete Assembly**

Assembly where at least one ResourcePort of type `CONTAINED` is not connected

How to go from an incomplete to a complete assembly?

**Complete Assembly**

Assembly where all the ResourcePorts of type `CONTAINED` are connected

$\hookrightarrow$ A complete assembly is "deployable"

# Assigner – Definition

## Assigner – Entity assigning components to resources

$$\text{Assigner} : \mathit{Assembly} \rightarrow \mathit{Assembly}$$

- Only manipulates ResourcePorts
- Does not have to return a complete assembly (i.e., can solve a sub-problem)

# Assigner – Definition

## Assigner – Entity assigning components to resources

$$\text{Assigner} : \textit{Assembly} \times \textbf{Universe} \rightarrow \textit{Assembly}$$

- Only manipulates ResourcePorts
- Does not have to return a complete assembly (i.e., can solve a sub-problem)

## Universe

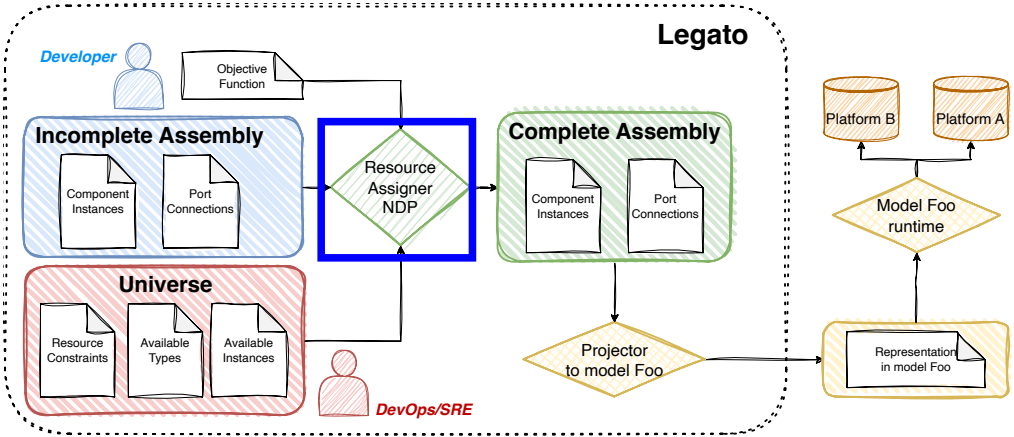
- Set of resource component instances: resources already available
- Set of resource component types: resources that can be instantiated
- A relation to model if a resource can be deployed onto another

$\hookrightarrow$ Can be generated based on the state of the platform(s)

# Assigner – Definition

**Assigner – Entity assigning components to resources**

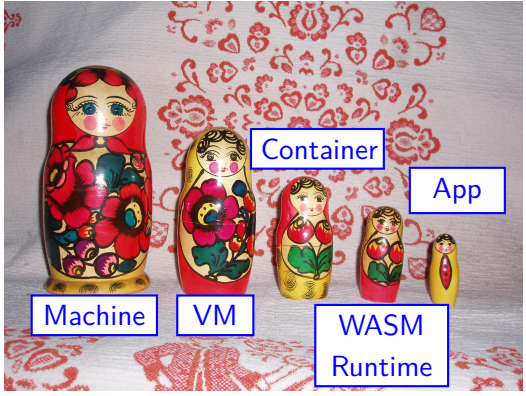
**Universe – Set of resource instances and types to complete assemblies**



# Nested Dolls Problem

## Nested Dolls Problem

"Bins become objects" $\rightarrow$ not trivial...




# Formulation of the Nested Dolls Problem

- $P_q^p$: connection between `CONTAINED` port $p \in P_\circ$ and `CONTAINS` port $q \in P_\circledcirc$
- $D_c$: 1 if component $c$ has to be fully connected to the assembly, 0 otherwise

**Capacities constraints**

$$\forall q \in P_\circledcirc(c), \forall k \in K, \quad \sum_{p \in P_\circ} P_q^p \times getAttVal(p,k) \leq getAttVal(q,k) \tag{1}$$

# Formulation of the Nested Dolls Problem

- $P_q^p$: connection between `CONTAINED` port $p \in P_\circ$ and `CONTAINS` port $q \in P_\circledcirc$
- $D_c$: 1 if component $c$ has to be fully connected to the assembly, 0 otherwise

**Capacities constraints**

**Cannot deploy onto itself**

$$\forall p \in P_\circ \text{ and } \forall q \in P_\circledcirc, P_q^p = 1 \implies Comp(p) \neq Comp(q) \tag{1}$$

# Formulation of the Nested Dolls Problem

- $P_q^p$: connection between `CONTAINED` port $p \in P_\circ$ and `CONTAINS` port $q \in P_\circledcirc$
- $D_c$: 1 if component $c$ has to be fully connected to the assembly, 0 otherwise

**Capacities constraints**

**Cannot deploy onto itself**

**Deploying all the needed components**

$$\forall c \in C \text{ where } D_c = 1, \forall p \in P_\circ(c), \exists q \in P_\circledcirc, P_q^p = 1 \tag{1}$$

and

$$\forall p \in P_\circ, q \in P_\circledcirc, P_q^p = 1 \implies D_{Comp(p)} = 1 \wedge D_{Comp(q)} = 1 \tag{2}$$

# Formulation of the Nested Dolls Problem

- $P_q^p$: connection between `CONTAINED` port $p \in P_\circ$ and `CONTAINS` port $q \in P_\circledcirc$
- $D_c$: 1 if component $c$ has to be fully connected to the assembly, 0 otherwise

**Capacities constraints**

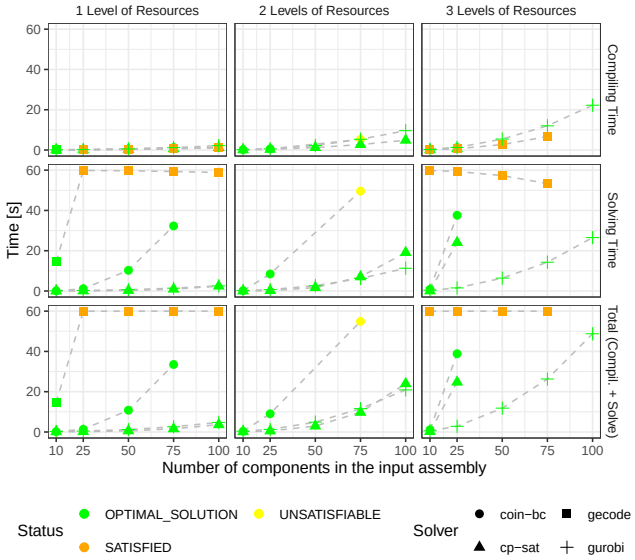
**Cannot deploy onto itself**

**Deploying all the needed components**

**Deploying the entire Assembly ($\mathcal{A}$)**

$$\forall c \in \mathcal{A}, D_c = 1 \tag{1}$$

# Difficulty of the Nested Dolls Problem




Assigners: Greedy, CP Solvers

## Experimental Setup

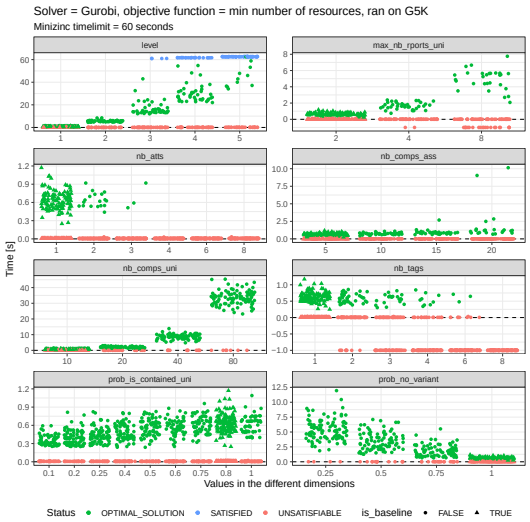
- Grid'5000 (`dahu`)
- MiniZinc (Coin-BC, Gecode, CP-SAT, Gurobi) capped 60s
- Factors: levels, comps, solvers
- Measure: solve time, status
- `SAT` and `MIN` nb components

## Results

- `SAT`: CP-SAT performs best
- `MIN`: Gurobi performs best

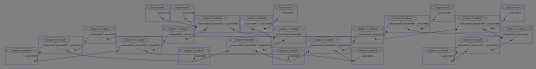
# Evaluation – Probabilist approach

- !!! WIP++ !!!
- Generate random assemblies and universes
- Vary the distribution of components (nb ports, attributes, etc.)
- Check how the solver performs, which dimensions have more influence

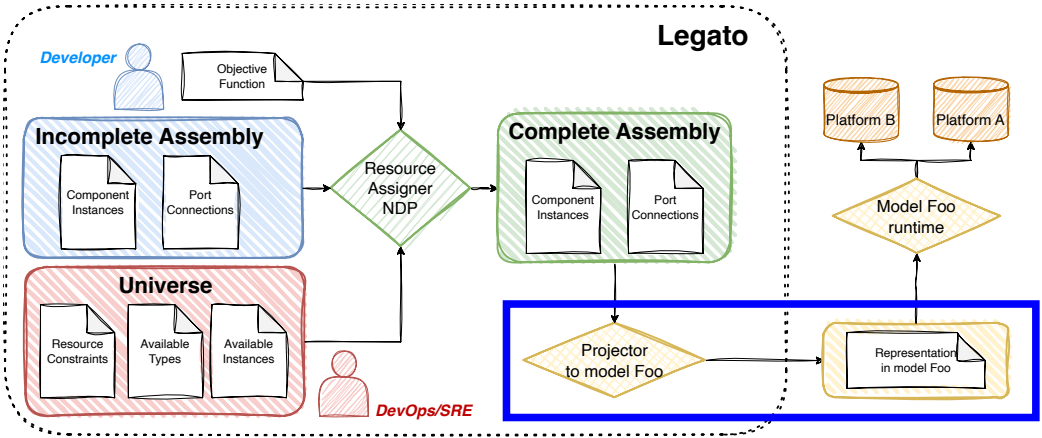


# Evaluation – Probabilist approach

- !!! WIP++ !!!
- Generate random assemblies and universes
- ↗ levels and ↗ nb components ⟹ ↗↗ complexity
  (Ongoing: characterizing the complexity)
- Check how the solver performs, which dimensions have more influence



# Where are we at?



# Projectors

## Projectors – Model (WIP)

- Projector to deployment model $m \in \{Concerto, Ansible\}$
- $S$ a set of key-value parameters
- $\mathcal{P}_{roj,m} : \mathcal{A}^* \times S \rightarrow \mathcal{D}_m$
- All components have a *templated* implementation in model $m$
  - $\forall c \in \mathcal{A}^*, \mathcal{I}_{c,m}(S)$ instantiates $c$ in $m$, with config $S$

**Projector Steps**

1. **Create** : instantiate the templated implem. (maybe reorder too)
2. **Connect** : represents the port connections in the deployment model (map ports)
3. **Deploy** : execute the representation in the deployment model

# Projectors – Projections into Execution Models

## Transformation into Ansible

- We transform the Assembly in our model into a Ansible master playbook
- All of our components have a Ansible implementation (parametrized playbook)
- Legato ports are represented via variables in Ansible, passed to the playbooks

## Transformation into Concerto

- We transform the Assembly in our model into a Concerto assembly
- All of our components have a Concerto implementation in Python
- Relies on the Concerto ports for representing the Legato ports
- (behavior support = "only `pushB(start)`")

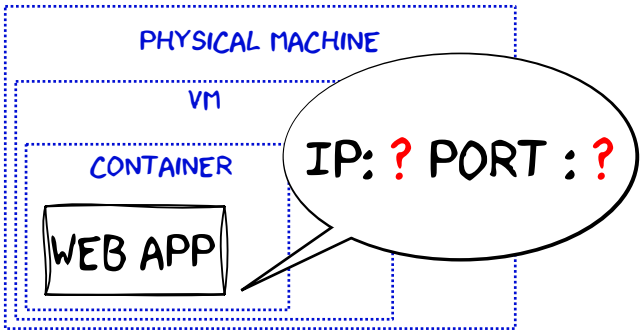
# Projectors – Details : Network Ports

## Network Ports

- Hard to access when nested
- Rely on forwarding ports

Components need to :

- ☛ announce their net ports
- ☛ have a way to fwd ports



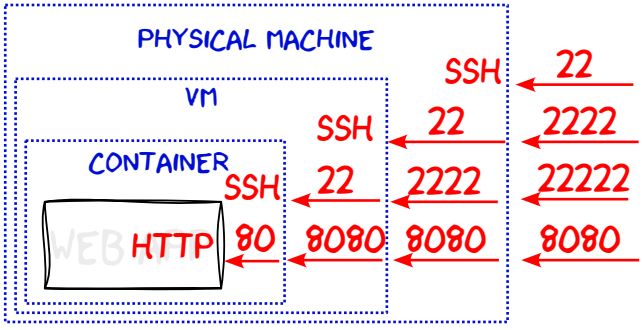
# Projectors – Details : Network Ports

## Network Ports

- Hard to access when nested
- Rely on forwarding ports

Components need to :

- ☛ announce their net ports
- ☛ have a way to fwd ports



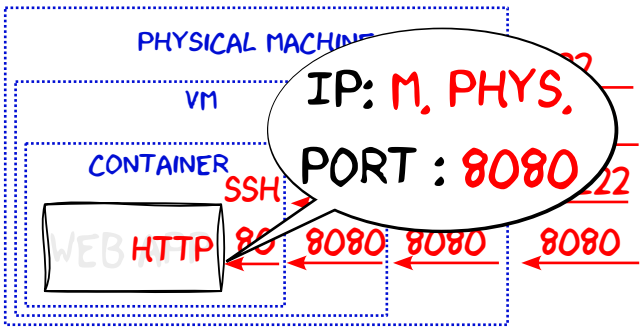
# Projectors – Details : Network Ports

## Network Ports

- Hard to access when nested
- Rely on forwarding ports

Components need to :

- ☛ announce their net ports
- ☛ have a way to fwd ports



# Implementation / Evaluation

# Implementation / Evaluation

- Realization of our Model in Nickel (see next slides)
- Implemented several resource components (in our model, in Concerto, in Ansible)
  - Grid'5000 (basic env.), Kadeploy,
  - QEMU (`x86`, `arm64`)
  - Docker,
  - K3S cluster (control node + agents)
  - preallocated resource (e.g., laptop)

# Nickel Propaganda

## Nickel

- "Universal configuration language"
- Can import/export JSON, YAML, XML, TOML ...
- Lang = JSON + types + contracts + $\lambda$
- ($\simeq$ similar as CUE, Pkl)

```
let
    x = 3
in
{
    foo = x,
    bar = "bar-%{x+1}"
}
```

```
{
"foo": 3,
"bar": "bar-4"
}
```

# Nickel Propaganda

## Nickel

- "Universal configuration language"
- Can import/export JSON, YAML, XML, TOML ...
- Lang = JSON + types + contracts + $\lambda$
- ($\simeq$ similar as CUE, Pkl)

```
let
  x = 3
in
{
  foo : Number = x,
}
```

```
{
"foo": 3
}
```

# Nickel Propaganda

## Nickel

- "Universal configuration language"
- Can import/export JSON, YAML, XML, TOML ...
- Lang = JSON + types + contracts + $\lambda$
- ($\simeq$ similar as CUE, Pkl)

```
let                                  1
  x = "foo"                          2
in                                   3
{                                    4
  foo : Number = x,                  5
}                                    6
```

Error!

# Nickel Propaganda

## Nickel

- "Universal configuration language"
- Can import/export JSON, YAML, XML, TOML ...
- Lang = JSON + types + contracts + $\lambda$
- ($\simeq$ similar as CUE, Pkl)

```
let
  n = 3
in
n |> std.array.generate
    (fun x ⇒ {
      field = "f%{x}",
      value = x})
  |> std.record.
     from_array
```

```
{
"f0": 0,
"f1": 1,
"f2": 2
}
```

# Nickel Propaganda

## Nickel

- "Universal configuration language"
- Can import/export JSON, YAML, XML, TOML ...
- Lang = JSON + types + contracts + $\lambda$
- ($\simeq$ similar as CUE, Pkl)

```
let
 Month =
  std.contract.
     from_predicate
   (fun value ⇒
     std.is_number value
     && std.number.
        is_integer value
     && value >= 1
     && value <= 12
   )
in
{
okMonth | Month = 5,
koMonth | Month = 31
}
```

# Nickel Propaganda – Example of Components and Assembly in Nickel

```
# components.ncl
{
  client = {
    componentTypeID = "Client",
    ports = {
      "server_service" = { type = 'USE }
    },
    resourcePorts = {
      "container" = {
        type = 'CONTAINED,
        attributes = { "CPU" = 12, "MEMORY" = 8 }
      }
    }
  },
  server = ...
}
```

# Nickel Propaganda – Example of Components and Assembly in Nickel

```
let                                                                    1
  comps = (import "components.ncl"),                                   2
  nbClients = 4,                                                       3
in {                                                                   4
componentInstances = [                                                 5
  { id = "Server", componentType = comps.server },                     6
] @ ( nbClients |> std.array.generate (fun x ⇒                         7
        { id = "Client%{x}", componentType = comps.client })),         8
portConnections = nbClients |>                                         9
    std.array.generate (fun x ⇒ {                                      10
      use = { instanceID = "Client%{x}", port = "srv_service"},        11
      provide = { instanceID = "Server", port = "service" }})          12
} | Assembly                                                           13
```

# Scenario – Gitea and CI Runners

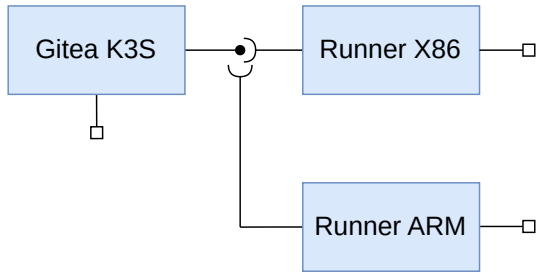
## Assembly

- Gitea: Git hosting service
  - K3S (http, DB, storage)
- CI runners
  - Docker container
  - `x86` and `arm64`

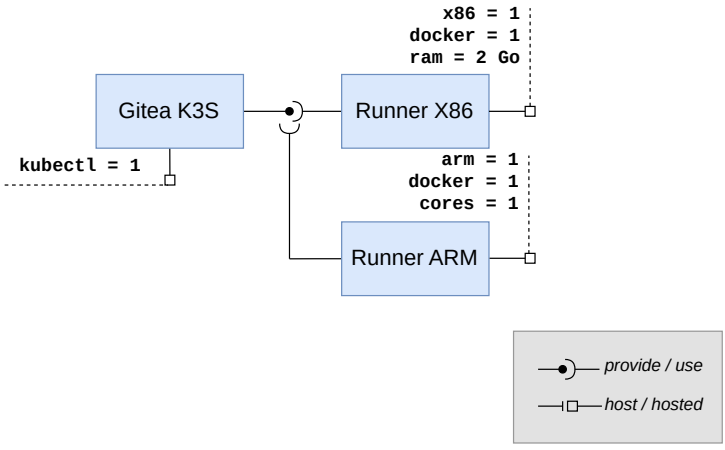
## Scenario

Deploy the same assembly ...

- ... in production (Grid'5000)
- ... in "incomplete" prod.

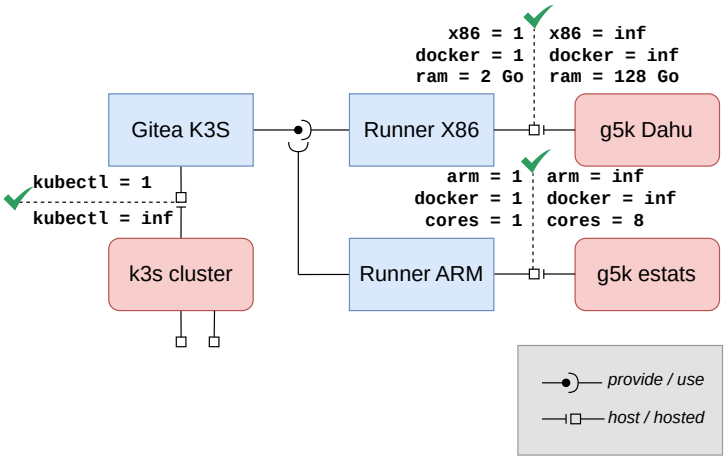


# Scenario – Gitea and CI Runners – Production



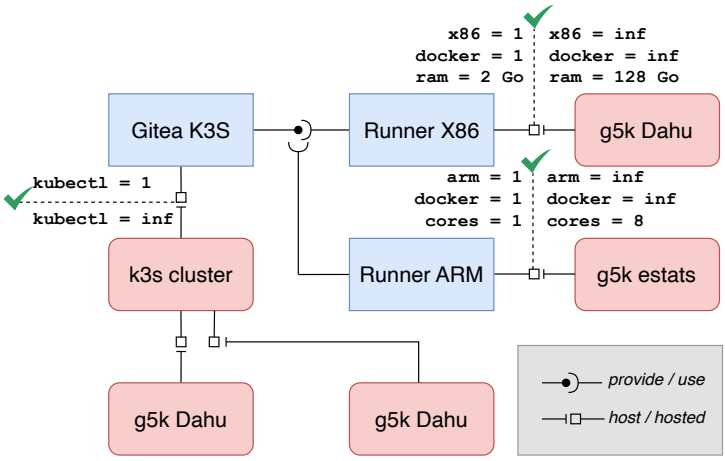

- Had to deploy own K3S cluster
- using one `x86` node (`dahu`) for runner
- using one `arm64` node (`estats`) for runner

# Scenario – Gitea and CI Runners – Production



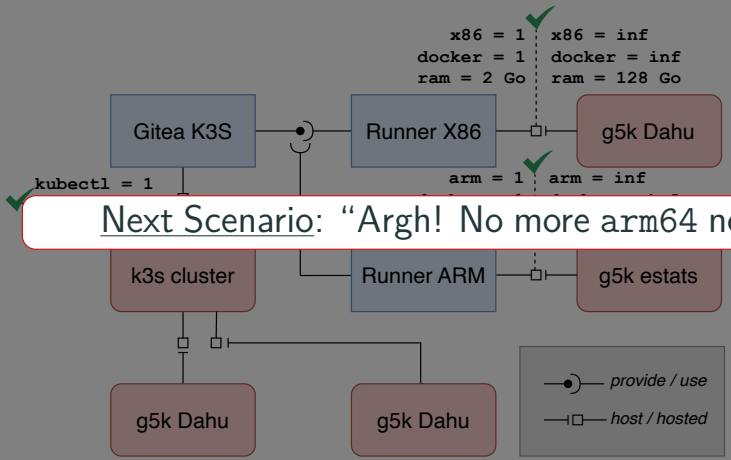

- Had to deploy own K3S cluster
- using one `x86` node (`dahu`) for runner
- using one `arm64` node (`estats`) for runner

# Scenario – Gitea and CI Runners – Production




- Had to deploy own K3S cluster
- using one `x86` node (`dahu`) for runner
- using one `arm64` node (`estats`) for runner

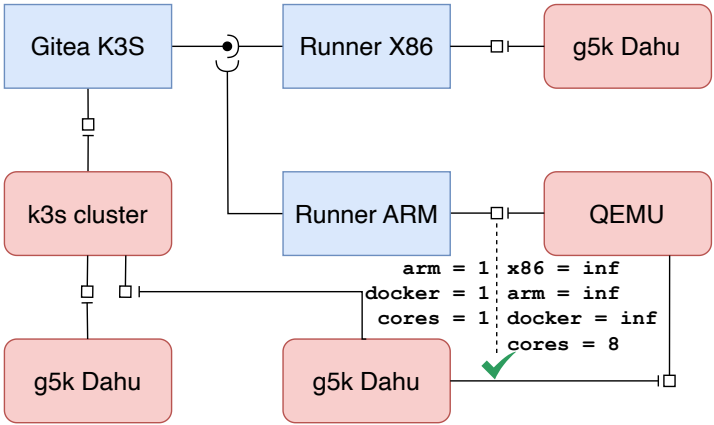
# Scenario – Gitea and CI Runners – Production




Next Scenario: "Argh! No more `arm64` node available..."

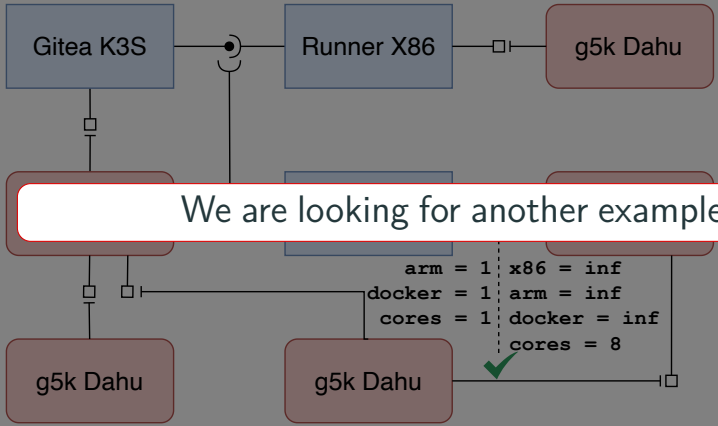
- Had to deploy own K3S cluster
- using one `x86` for runner
- using one `arm64` node (`estats`) for runner

# Scenario – Gitea and CI Runners – Incomplete Production




- no `arm64` node available!
- use a `x86` node + **QEMU**
- "deep encapsulation"!

# Scenario – Gitea and CI Runners – Incomplete Production




We are looking for another example/scenario!

- no `arm64` node available!
- “deep encapsulation”!

# Conclusion & Perspectives

# Conclusion & Perspectives

Pb: "How to model assemblies to support their execution in various contexts?"

**Conclusion and Analysis**

- ✓ Model to express application assemblies independently from the resources
- ✓ Allow for portability of assemblies, and heterogeneous deployments
- ✓ Able to model complex deep "encapsulation levels"
- ✓ Can generate problem for a solver, and transform into Concerto/Ansible
- ≈ Resource components not (yet) too difficult to implement in Concerto/Ansible

# Conclusion & Perspectives

Pb: "How to model assemblies to support their execution in various contexts?"

## Conclusion and Analysis

It kinda works!

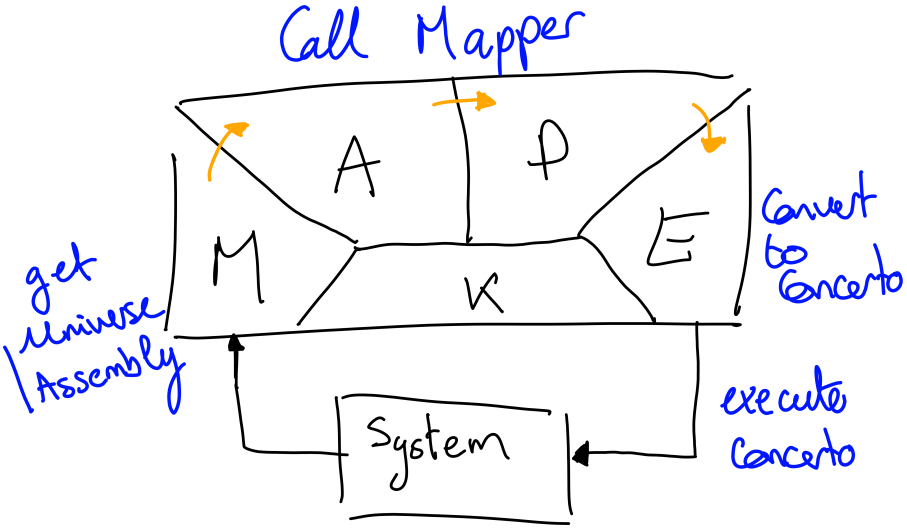
## Perspectives

- Reconfiguration
- Software environments
- Coordinate several execution models (Ansible + Concerto)
- **Network, storage elements as resources ?**

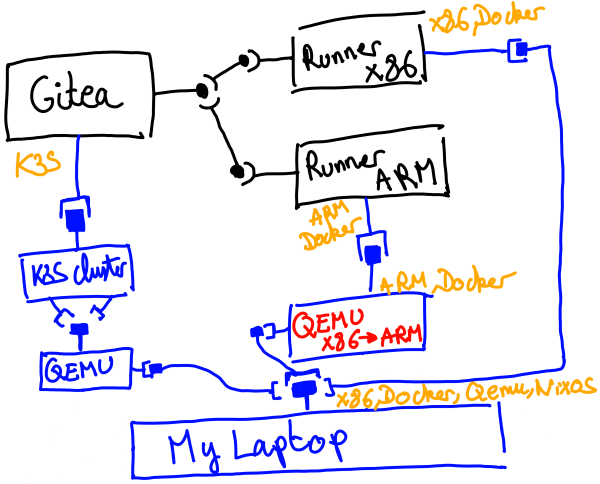
By making deployments easier/optimized, we also make **more** deployments possible...

$\hookrightarrow$ Are we participating in the acceleration of the Cloud?

# The Global Picture



# Scenario – Gitea and CI Runners – Development




- need to reduce the resource requirements (functional attributes) to fit on laptop
- K3S needs Debian
- VM for K3S cluster and VM for arm worker

# NDP – Examples of Assigners

**Warning!**: We are **not** proposing new placement algorithms!

## Simple Greedy Mapper

- for each unconnected `CONTAINED` ports
  - go through the set of resources instances and see if one can connect
  - if not, go through the set of resources types, and see if one can connect, if so instantiate this resource
- continue until no `CONTAINED` port is left unconnected (might need to backtrack)

## Integration with Constraint Solvers (via Minizinc)

- Difficult to express our problem as a simple variant of knapsack/bin-packing ...
- A "flat" representation of the problem
- Have to "instantiate" the resource types to feed the solver: how many?