
KubeCon
CloudNativeCon
North America 2019


# Cloud Native Smart Contract with Knative

*GUO Jiannan (Jay), GUO Yingchun (Daisy)*

*IBM China*

# Agenda



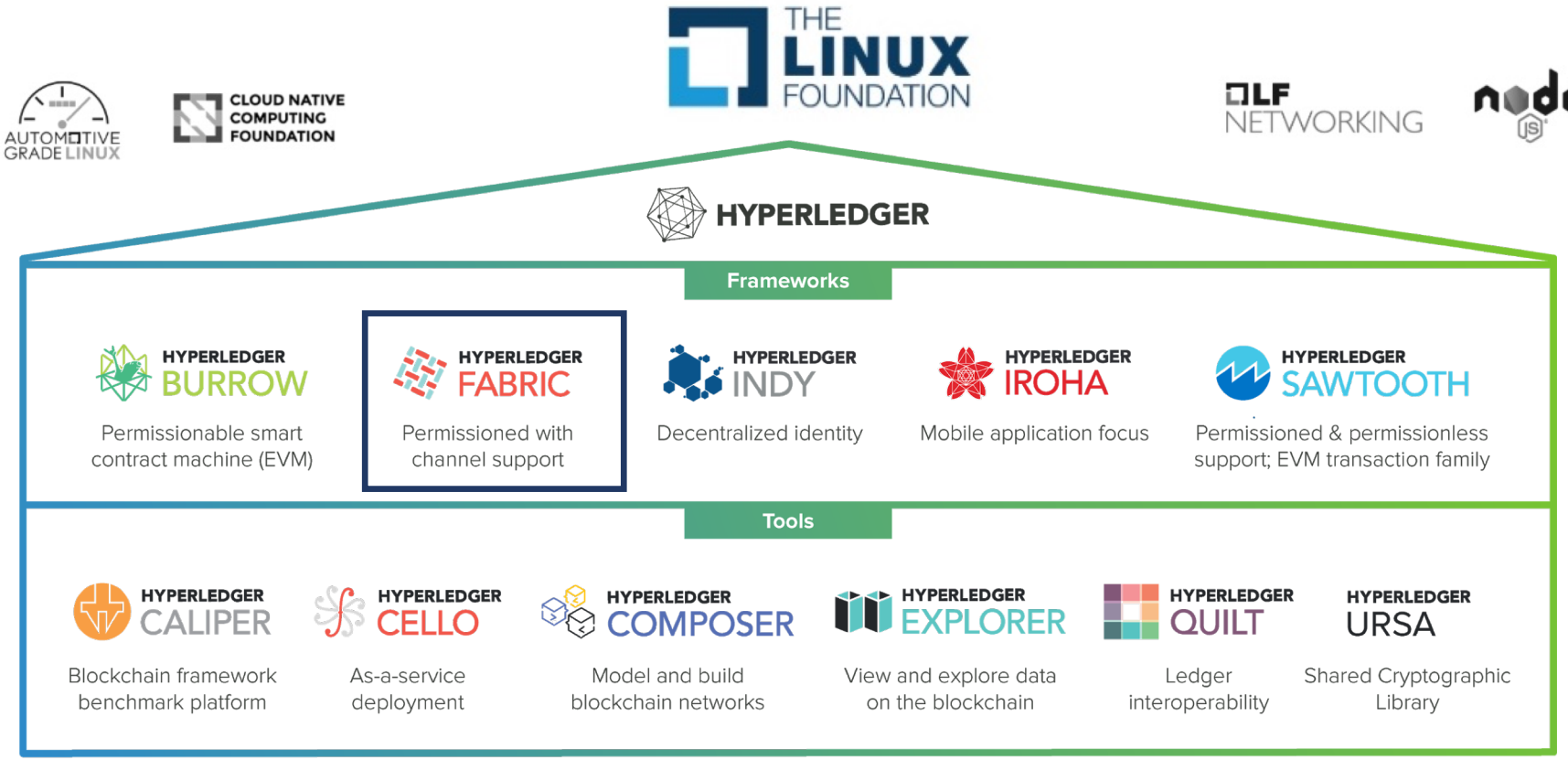
- Blockchain and Hyperledger Fabric
- Fabric Transaction Flow
- Current Smart Contract Lifecycle
- Improve with Tekton + Knative
- Demo
- Conclusion and Future Work

# A Very Short Introduction to Blockchain



From technical perspective:

- Distributed System that tolerates Byzantine Faults (*consensus*)
- Verifiable Temper-proof Append Only Logs (*chained blocks*)
- Public or Consortium (*permissionless or permissioned*)

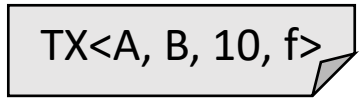



- Hyperledger is part of Linux Foundation with 270+ members
- Greenhouse for **Enterprise Blockchain Technologies**
- Fabric
  - The first graduated project
  - Apache 2.0
  - Core written in Go
  - SDK & **Smart Contract** in Go, Java, Node


*Premium Members*
*Screenshot from hyperledger.org 2019-11*

# Fabric Transaction Flow

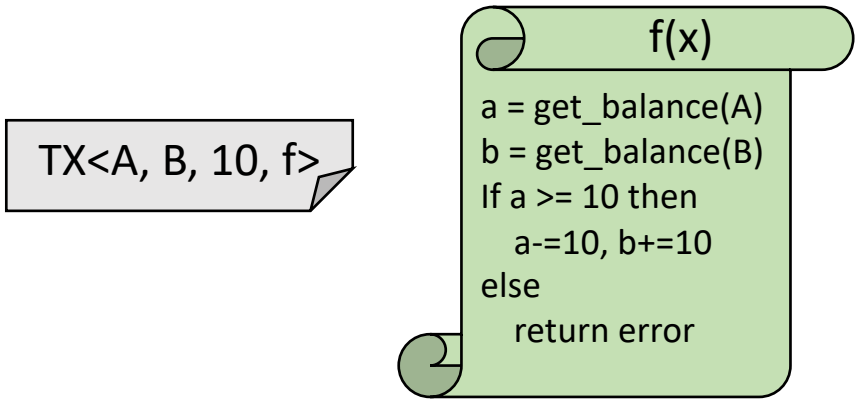





- Tx <args, targeting contract>

# Fabric Transaction Flow

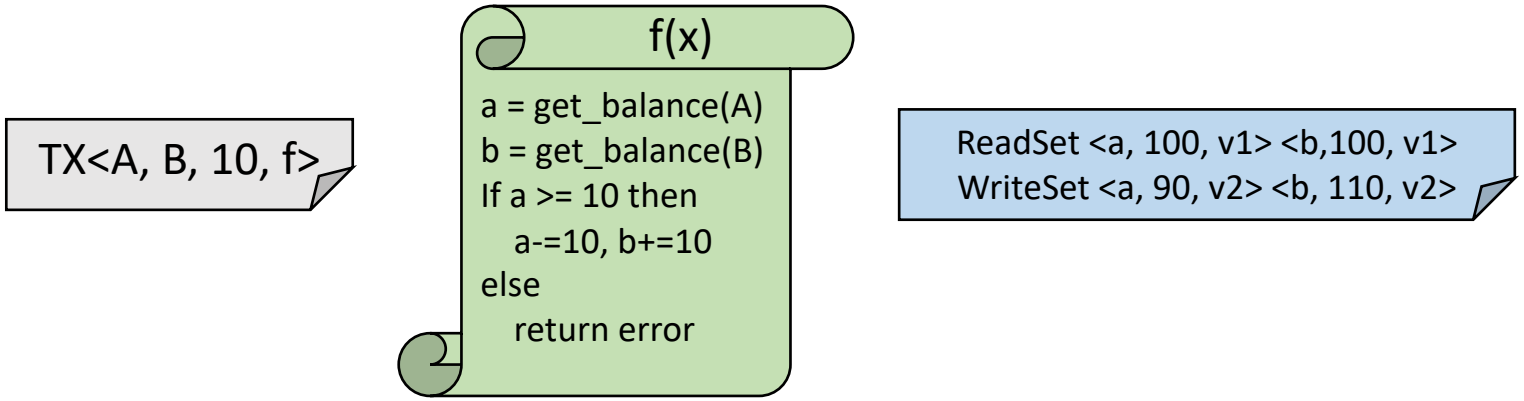





- Tx <args, targeting contract>
- fn(inputs)

# Fabric Transaction Flow

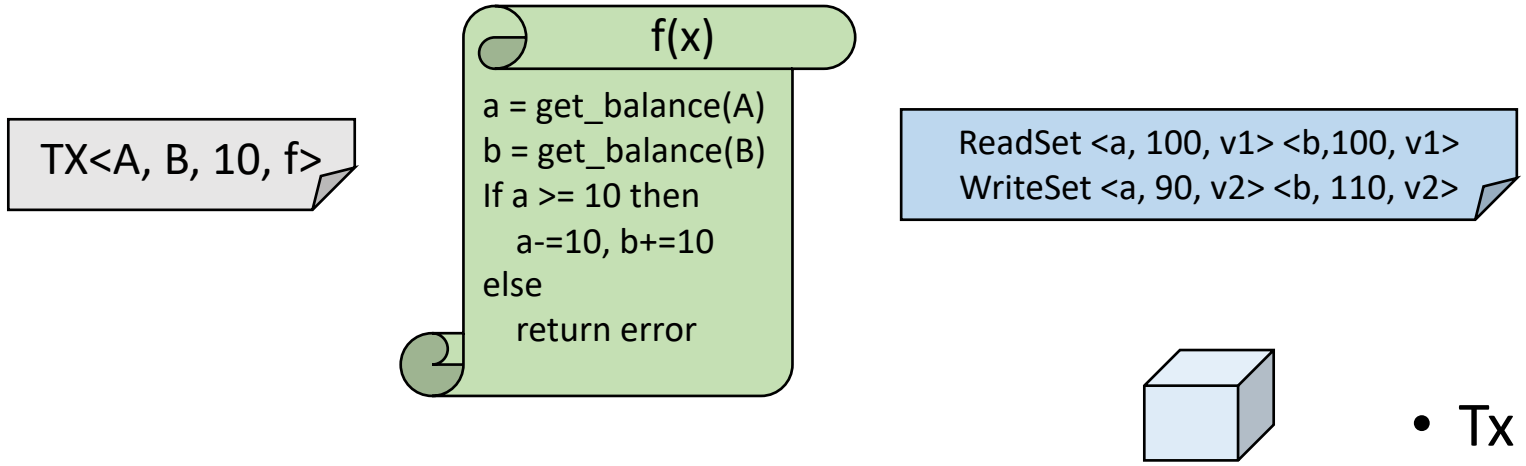





- Tx <args, targeting contract>
- fn(inputs) produces ReadWriteSet

# Fabric Transaction Flow

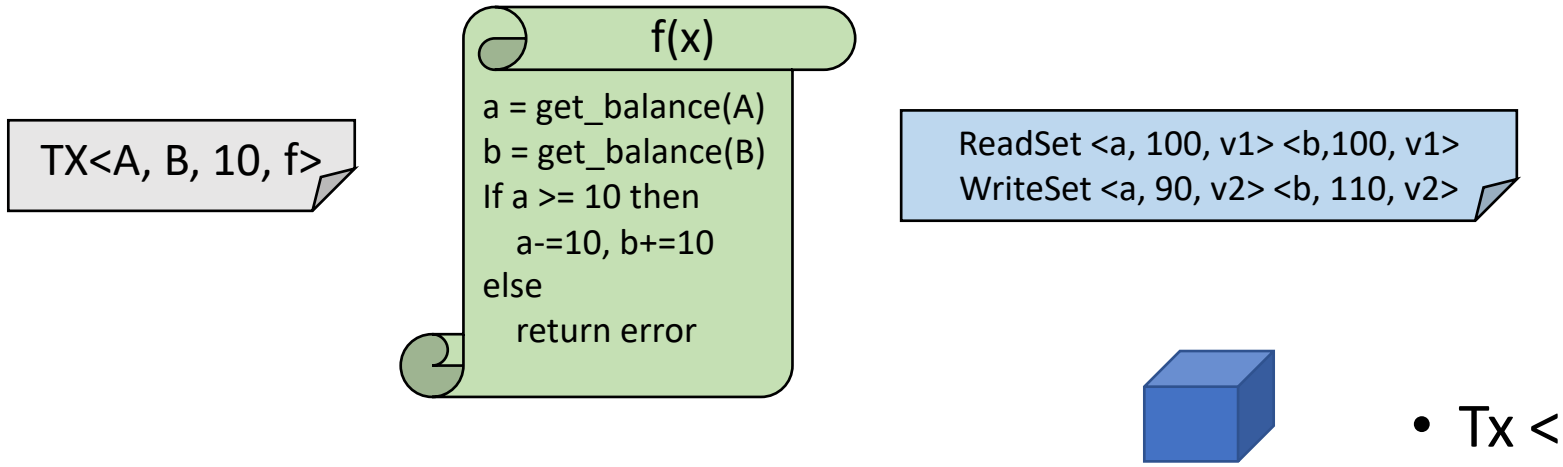





- Tx <args, targeting contract>
- fn(inputs) produces ReadWriteSet
- RWSets are accumated into blocks

# Fabric Transaction Flow

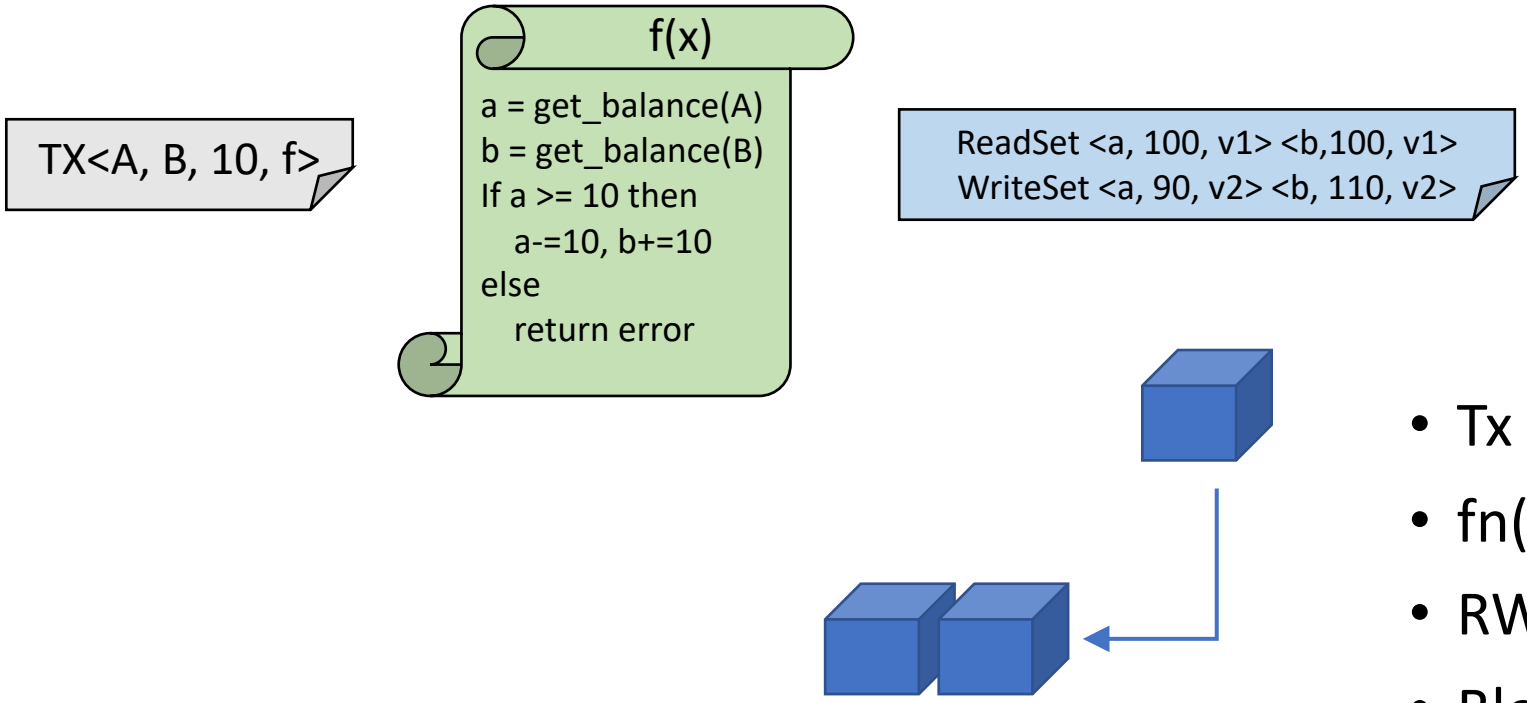





- Tx <args, targeting contract>
- fn(inputs) produces ReadWriteSet
- RWSets are accumated into blocks
- Blocks are disseminated and agreed upon via consensus protocol

# Fabric Transaction Flow

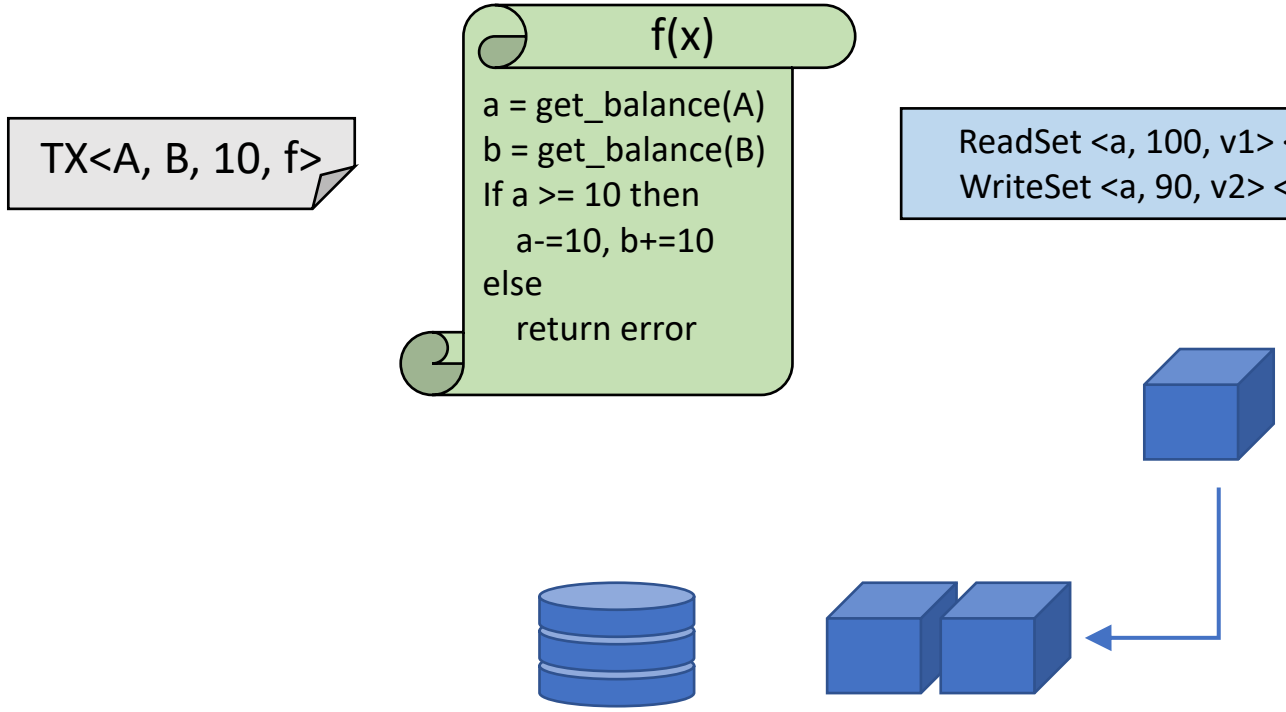





- Tx <args, targeting contract>
- fn(inputs) produces ReadWriteSet
- RWSets are accumated into blocks
- Blocks are disseminated and agreed upon via consensus protocol
- Block appended to the tail of chain

# Fabric Transaction Flow

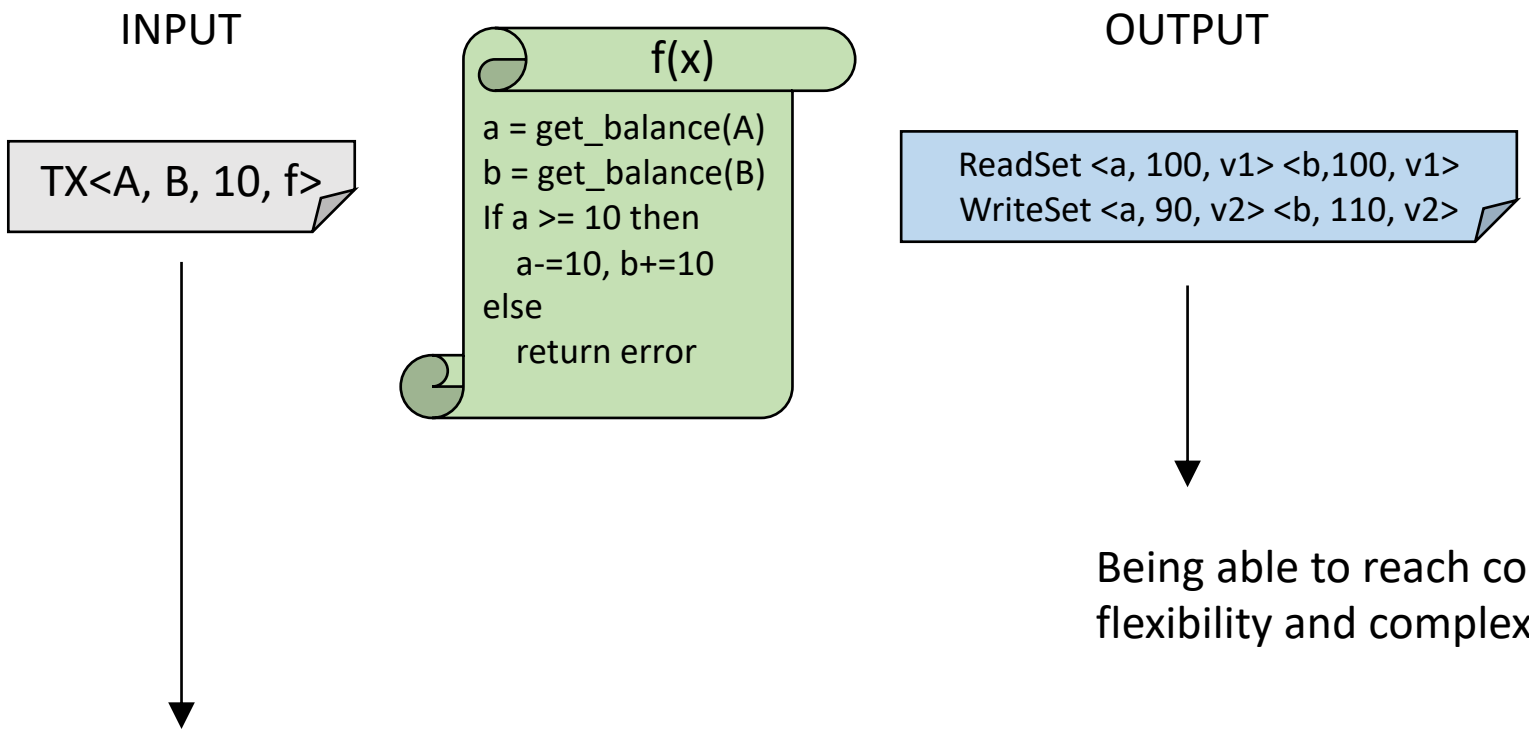





- Tx <args, targeting contract>
- fn(inputs) produces ReadWriteSet
- RWSets are accumated into blocks
- Blocks are disseminated and agreed upon via consensus protocol
- Block appended to the tail of chain
- RWSets are validated and applied against state of ledger

# Fabric Transaction Flow





# Smart Contract in Blockchain



- Programs embedding business logics
- Deployed on blockchain nodes
- Executed on-demand with external inputs
- Side effects (modifying ledger data) or return value (transaction output)
- Stateless, light-weight, ephemeral, deterministic, composable

```
pragma solidity >=0.4.0 <0.7.0;

contract SimpleStorage {
    uint storedData;

    function set(uint x) public {
        storedData = x;
    }

    function get() public view returns (uint) {
        return storedData;
    }
}
```

https://solidity.readthedocs.io/en/v0.5.12/introduction-to-smart-contracts.html

Solidity - Ethereum

```
// Get value by key
func (s *Storage) Get(stub shim.ChaincodeStubInterface, args []string) pb.Response {
    key := args[0]

    value, err := stub.GetState(key)
    if err != nil {
        return shim.Error("Failed to get")
    }

    return shim.Success(value)
}

// Put key value pair
func (s *Storage) Put(stub shim.ChaincodeStubInterface, args []string) pb.Response {
    key, value := args[0], args[1]

    err := stub.PutState(key, []byte(value))
    if err != nil {
        return shim.Error("Failed to put")
    }

    return shim.Success(nil)
}
```

Golang - Hyperledger Fabric

# Smart Contract in Fabric



- Runs as a separate process
- Communicates with Fabric Peer via gRPC stream
- CRUD API

```
// GetState returns the value of the specified `key` from the
// ledger. Note that GetState doesn't read data from the writeset, which
// has not been committed to the ledger. In other words, GetState doesn't
// consider data modified by PutState that has not been committed.
// If the key does not exist in the state database, (nil, nil) is returned.
GetState(key string) ([]byte, error)

// PutState puts the specified `key` and `value` into the transaction's
// writeset as a data-write proposal. PutState doesn't effect the ledger
// until the transaction is validated and successfully committed.
// Simple keys must not be an empty string and must not start with a
// null character (0x00) in order to avoid range query collisions with
// composite keys, which internally get prefixed with 0x00 as composite
// key namespace. In addition, if using CouchDB, keys can only contain
// valid UTF-8 strings and cannot begin with an underscore ("_").
PutState(key string, value []byte) error

// DelState records the specified `key` to be deleted in the writeset of
// the transaction proposal. The `key` and its value will be deleted from
// the ledger when the transaction is validated and successfully committed.
DelState(key string) error
```

```
// Get value by key
func (s *Storage) Get(stub shim.ChaincodeStubInterface, args []string) pb.Response {
    key := args[0]

    value, err := stub.GetState(key)
    if err != nil {
        return shim.Error("Failed to get")
    }

    return shim.Success(value)
}

// Put key value pair
func (s *Storage) Put(stub shim.ChaincodeStubInterface, args []string) pb.Response {
    key, value := args[0], args[1]

    err := stub.PutState(key, []byte(value))
    if err != nil {
        return shim.Error("Failed to put")
    }

    return shim.Success(nil)
}
```

# Fabric Smart Contract Lifecycle

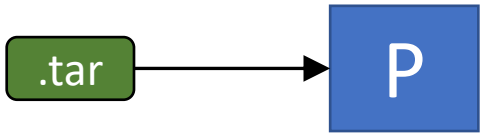


- Install smart contract on a peer (*class*)

# Fabric Smart Contract Lifecycle



- Install smart contract on a peer (*class*)
  - Upload tar



# Fabric Smart Contract Lifecycle



- Install smart contract on a peer (*class*)
  - Upload tar
  - Compile
  - Build Docker image

# Fabric Smart Contract Lifecycle



- Install smart contract on a peer (*class*)
  - Upload tar
  - Compile
  - Build Docker image
- Instantiate smart contract on a channel (*instance*)


Supply chain

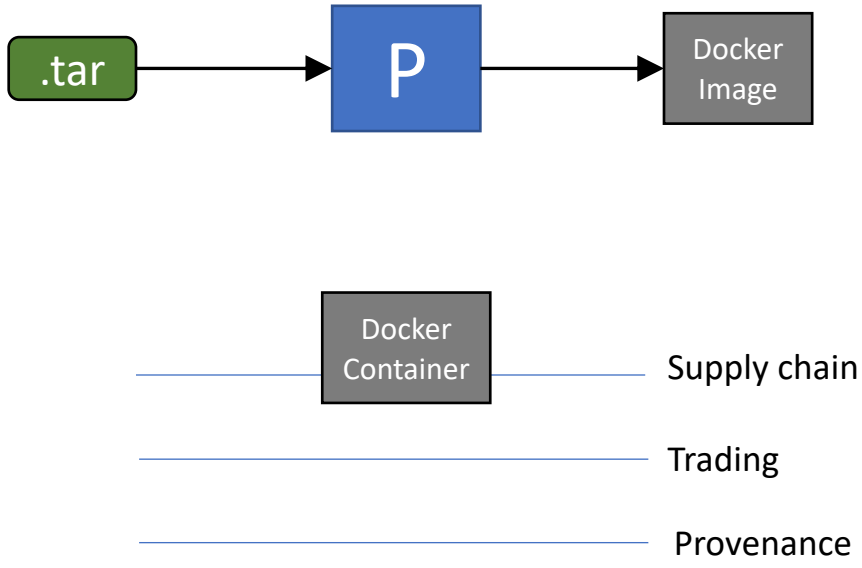
Trading

Provenance

# Fabric Smart Contract Lifecycle



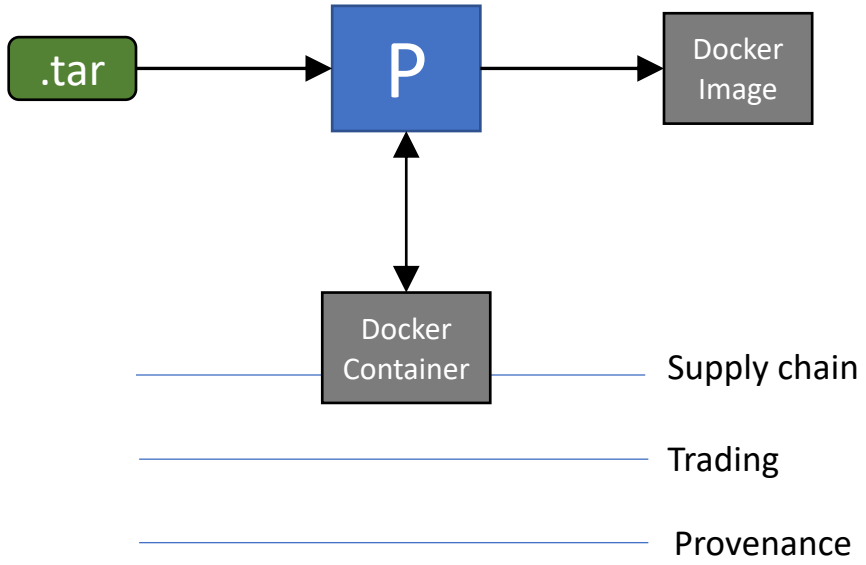
- Install smart contract on a peer (*class*)
  - Upload tar
  - Compile
  - Build Docker image
- Instantiate smart contract on a channel (*instance*)
  - Start Docker container



# Fabric Smart Contract Lifecycle



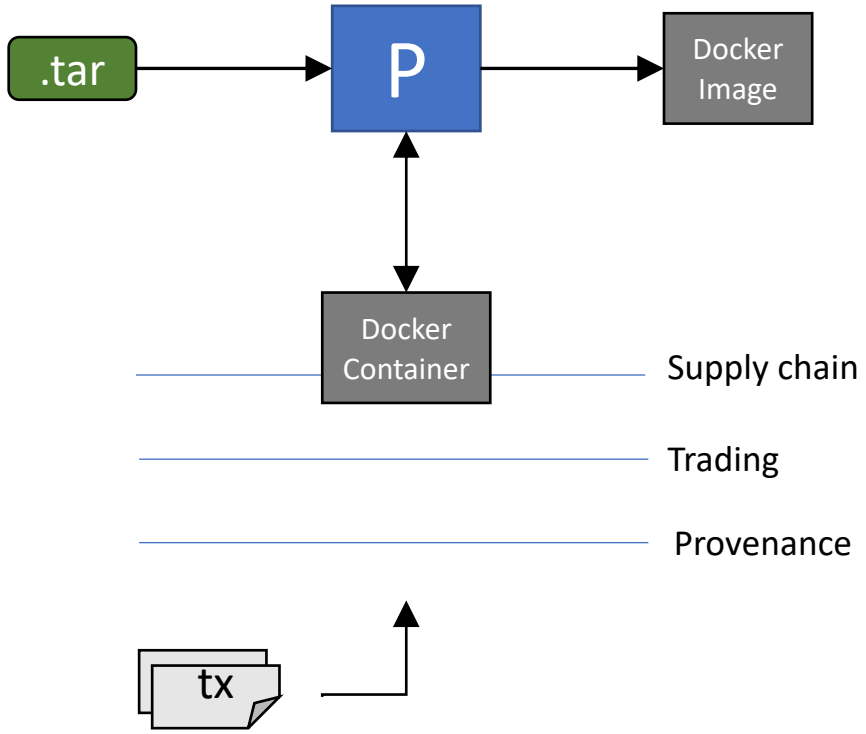
- Install smart contract on a peer (*class*)
  - Upload tar
  - Compile
  - Build Docker image
- Instantiate smart contract on a channel (*instance*)
  - Start Docker container
  - Smart contract initiates gRPC connection with peer



# Fabric Smart Contract Lifecycle



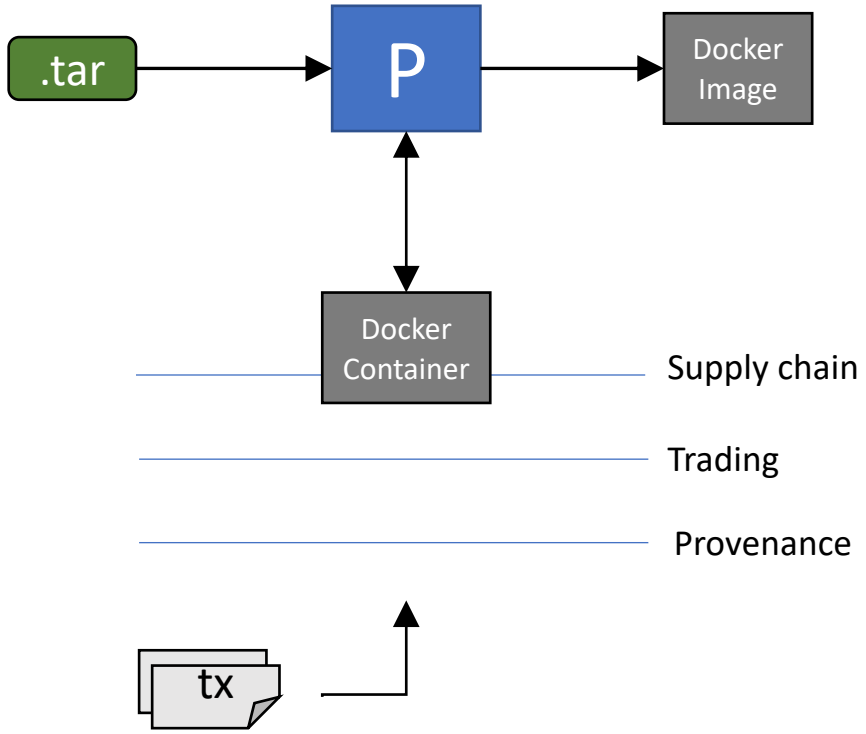
- Install smart contract on a peer (*class*)
  - Upload tar
  - Compile
  - Build Docker image
- Instantiate smart contract on a channel (*instance*)
  - Start Docker container
  - Smart contract initiates gRPC connection with peer
- Invoke smart contract (*call*)
  - Tx is sent via gRPC stream



# Problem with current model



- Violate security practice by requiring a Docker daemon exposed to Fabric Peer process
- Complexity in managing containers
- Hard to develop and test contracts
- Waste of resources by running idle contracts
- Co-location of Peer and contracts

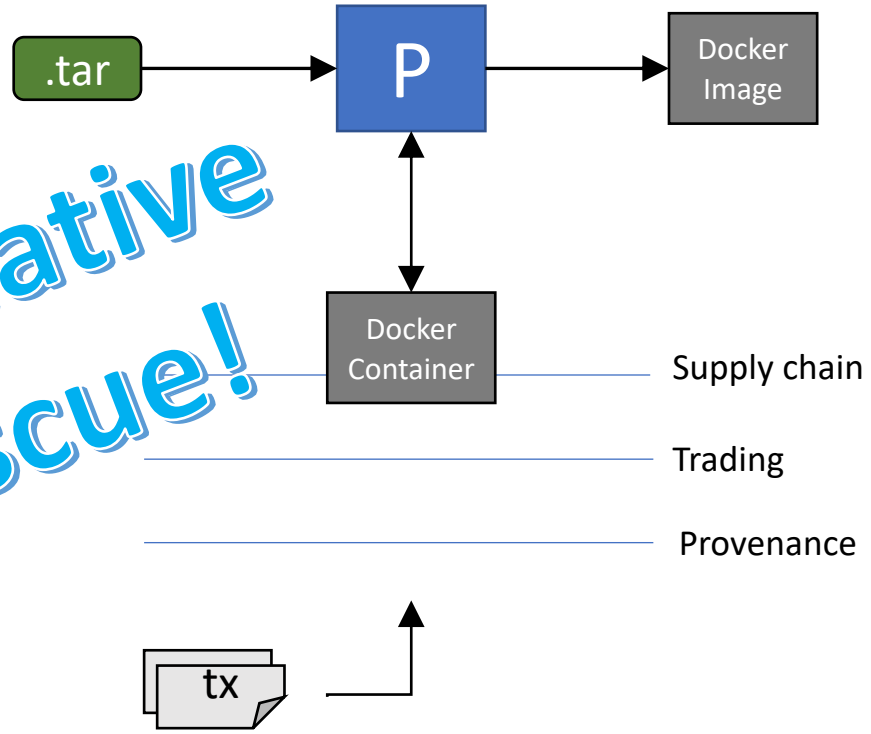


# Problem with current model



- Violate security practice by requiring a Docker daemon exposed to Fabric Peer process
- Complexity in managing containers
- Hard to develop and test contracts
- Waste of resources by running idle contracts
- Co-location of Peer and contracts

Tekton + Knative to the rescue!



## Standard Kubernetes-style pipelines

Declarative pipelines with standard Kubernetes custom resources (CRDs) based on Tekton*

## Build images with Kubernetes tools

Use tools of your choice (source-to-image, buildah, kaniko, jib, etc) for building container images

## Run pipelines in containers

Scale pipeline executions on-demand with containers on Kubernetes

## Deploy to multiple platforms

Deploy applications to multiple platforms like serverless, virtual machines and Kubernetes

## Powerful command-line tool

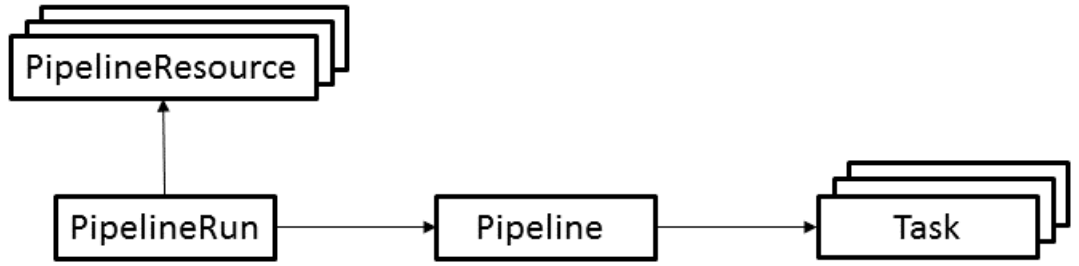
Run and manage pipelines with an interactive command-line tool

# Trigger, Build and Deploy with Tekton



**Tekton Pipeline**

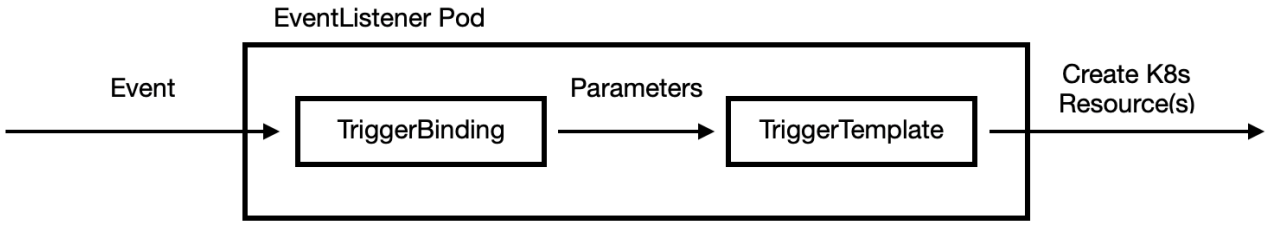
The Tekton Pipelines project provides k8s-style resources for declaring CI/CD-style pipelines. It lets you build, test, and deploy across multiple cloud providers or on-premises systems by abstracting away the underlying implementation details.




Tekton Pipeline Resources

**Tekton Trigger**

Trigger is a Kubernetes Custom Resource Defintion (CRD) controller that allows you to extract information from events payloads (a "trigger") to create Kubernetes resources.




Tekton Trigger Resources

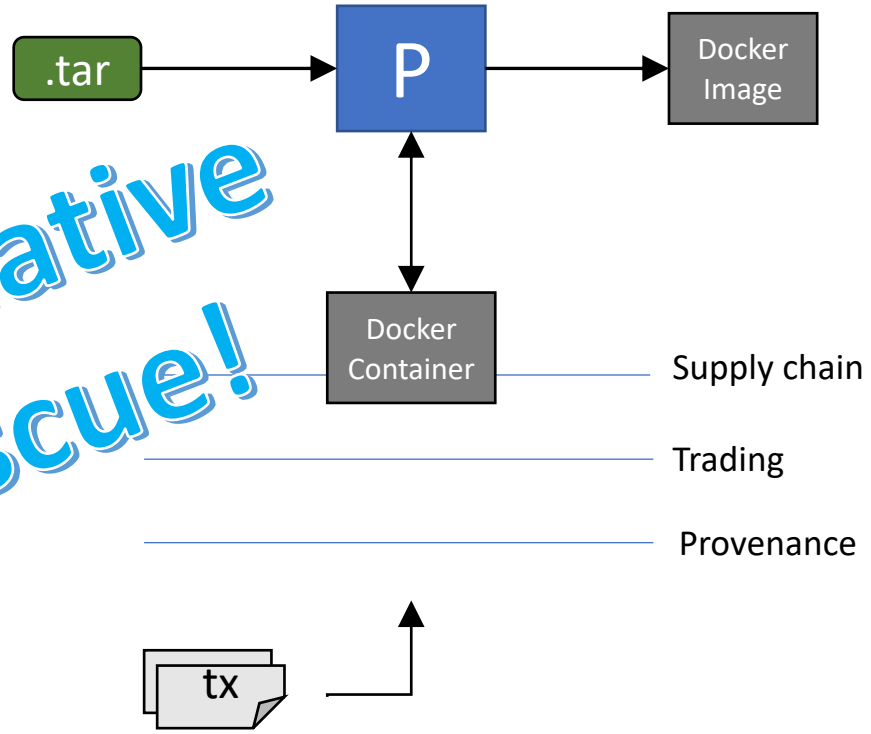
Using triggers in conjunction with tekton pipeline enables you to easily create full-fledged CI/CD systems!

# Problem with current model



- ✗ Violate security practice by requiring a Docker daemon exposed to Fabric Peer process
- Complexity in managing containers
- ✗ Hard to develop and test contracts
- Waste of resources by running idle contracts
- Co-location of Peer and contracts

Tekton + Knative to the rescue!



# Knative



**Define a set of primitives**

to build modern, source-centric, and container-based applications

**Based on Kubernetes**

a set of Kubernetes Custom Resource Definitions (CRDs)

scale automatically from zero and size workloads based on demand

**Leverage service mesh**

defines routing and network programming

observability and security

Knative Serving

**Enable Serverless**

define and control serverless workload behaves on the cluster

Scale up from zero

Knative Eventing

**Eventing ecosystem**

enable late-binding event sources and event consumers

# Knative Services



- Micro services / light weight
- Ephemeral
- Containerized
- Stateless
- Endpoint + load balance
- Scale to zero
- Auto scaling
- Pay by usage / resources saving
- Observable
- Secured access through mTLS

Knative Service

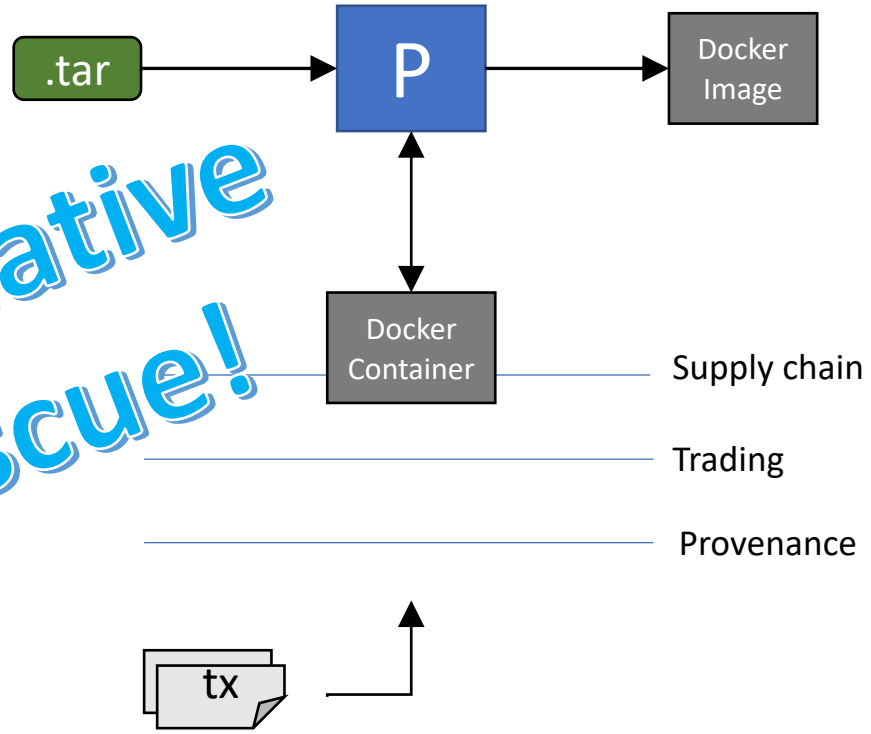
```
apiVersion: serving.knative.dev/v1alpha1
kind: Service
metadata:
  name: contract
spec:
  template:
    spec:
      containers:
        - image: docker.io/daisyycguo/contract:0.1.7
          ports:
            - name: h2c
              containerPort: 8080
          env:
            - name: CORE_CHAINCODE_ID_NAME
              value: "my_prebuilt_chaincode"
```

# Problem with current model



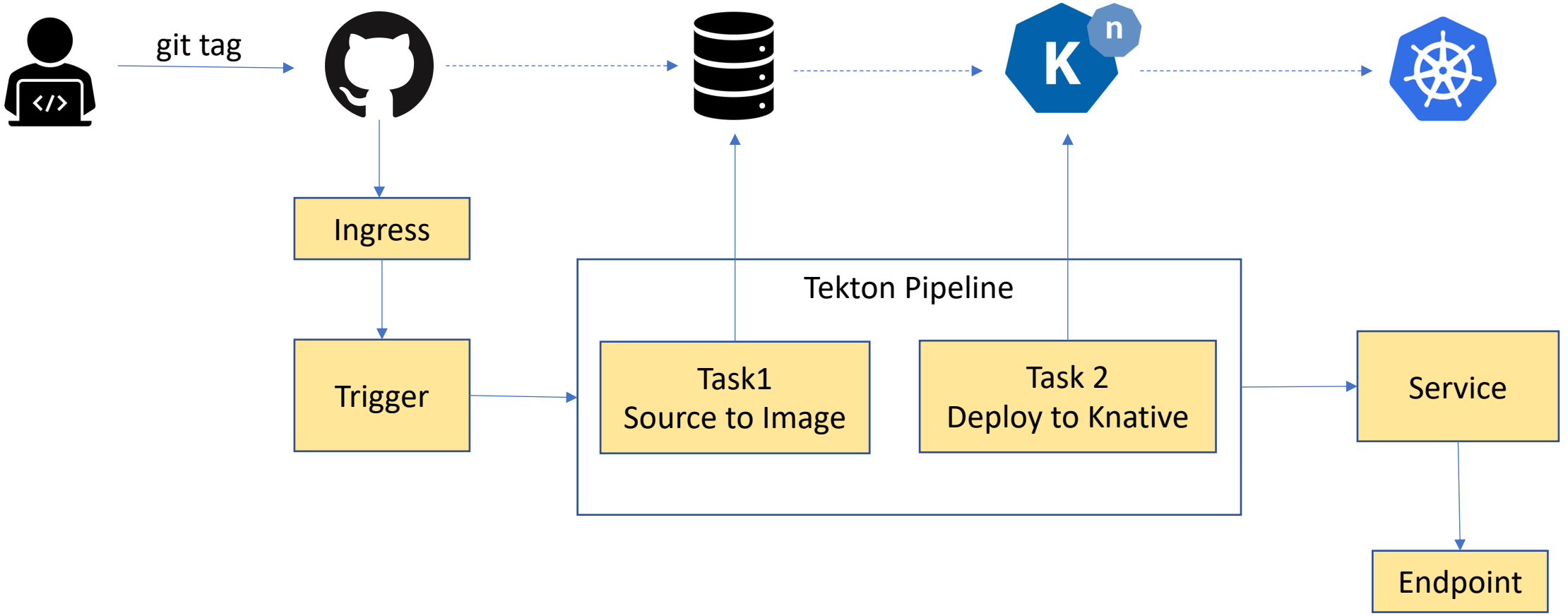
- ✗ Violate security practice by requiring a Docker daemon exposed to Fabric Peer process
- ✗ Complexity in managing containers
- ✗ Hard to develop and test contracts
- ✗ Waste of resources by running idle contracts
- ✗ Co-location of Peer and contracts

Tekton + Knative to the rescue!



# What did we do



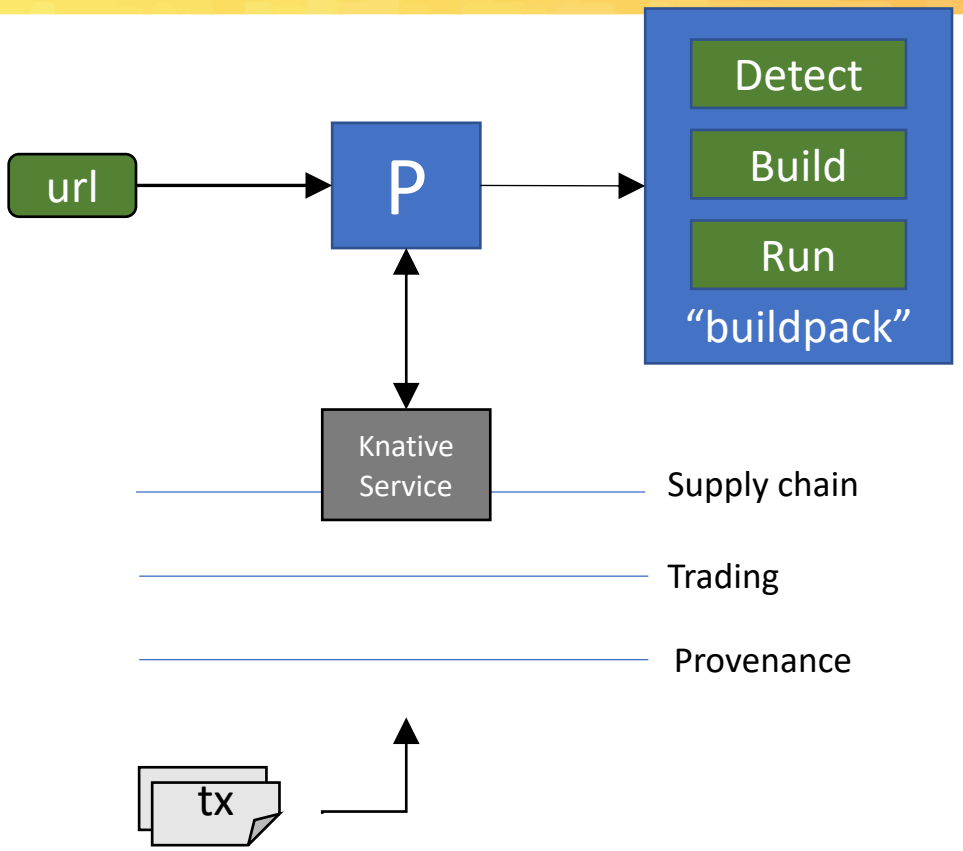


# New Fabric Contract Lifecycle



- Detect/Build/Run

- Install smart contract on a peer (*class*)
  - Upload ~~tar~~ url
  - ~~Compile~~
  - ~~Build Docker image~~
- Instantiate smart contract on a channel (*instance*)
  - ~~Start Docker container~~
  - ~~Smart contract initiates gRPC connection with peer~~
- Invoke smart contract (*call*)
  - Tx is sent via gRPC stream



# Demo

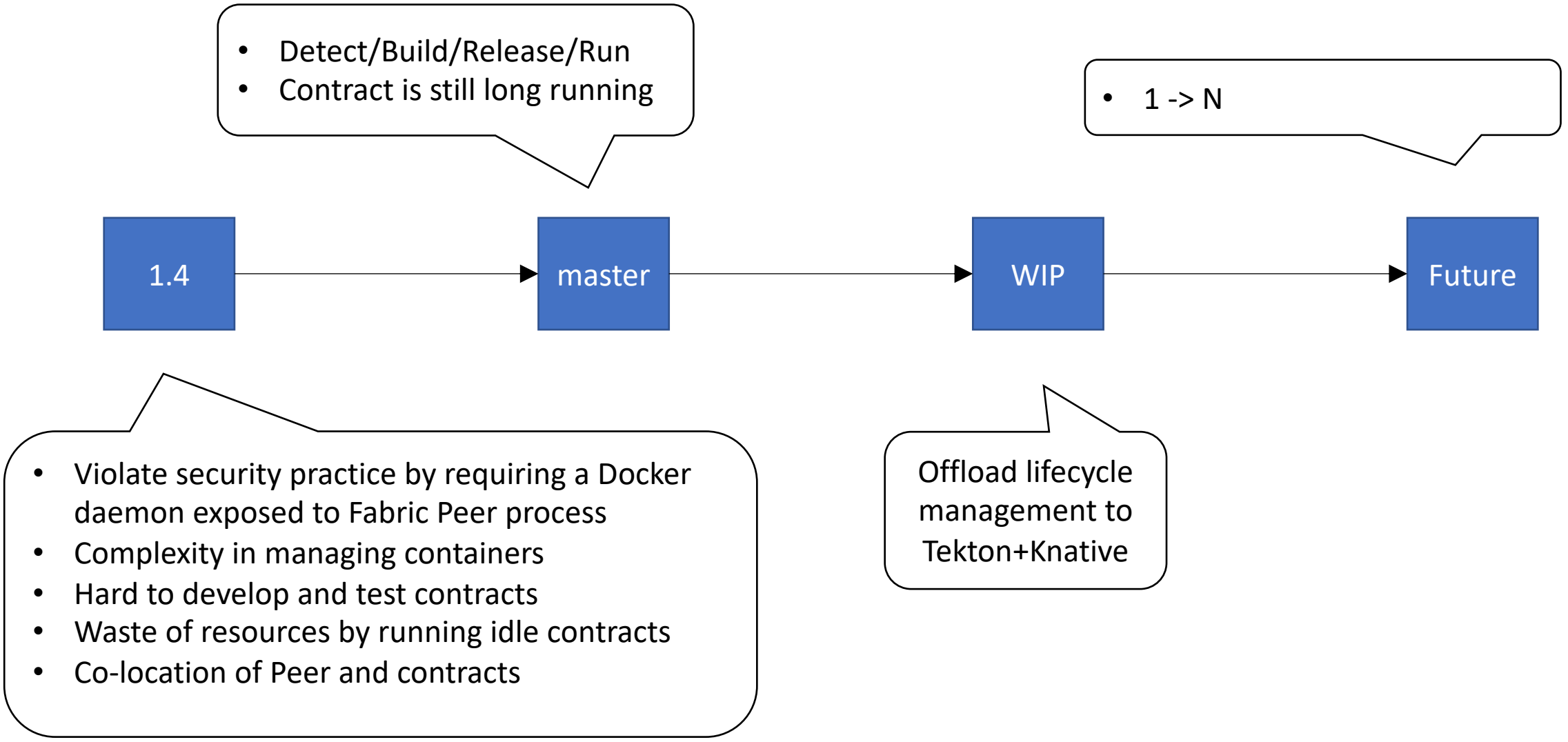


**Steps:**

1. Install smart contract as service url
2. Tag repo
3. Invoke smart contract
4. Modify code and re-tag repo
5. Invoke smart contract again

# Conclusion and Future Work






KubeCon
CloudNativeCon
North America 2019
Question?