Hyperledger Avalon Architecture Overview

Revision 0.3

Abstract: *This document provides a high-level overview of Hyperledger Avalon architecture. After reading this document, you will be able to understand the architectural design and rationale used to define the architecture.*

Table Of Contents

1 Introduction 3
1.1 Motivation 3
1.2 Acronyms 4
1.3 Glossary 4
1.4 Relevant Documents 4
2 Architecture 5
2.1 System Overview 5
2.2 System Description 6
2.2.1 Trusted Compute APIs 6
2.2.2 Workloads 7
2.2.3 KV Storage 8
2.2.4 On-Chain Components 8
2.2.5 Front-end Application 8
2.3 High Level Execution Flow 9
2.4 Worker Attestation Info 10
2.5 Work Order Execution Protection 11
3 Trusted Compute Service 13
3.1 Front-End Components 13
3.2 Middleware 14
3.3 Worker Types and Orchestration 17
3.3.1 Singleton Worker 17
3.3.2 Worker Pool 18
Appendix A: Current Implementation 20
3.4 Requester App Examples and Connector Library 20
3.5 Direct Model Supported Blockchains 21
3.6 KV Storage Manager 21
3.7 Enclave Manager 21

Table of Figures

Figure 1 System Overview 5
Figure 2 High Level Architecture 6
Figure 3 High Level Execution Flow 10
Figure 4 Chain of Trust 11
Figure 5 TCS Architecture 13
Figure 6 Current Implementation Components 20

1 Introduction

This document outlines the structure and major components that make up the Hyperledger Avalon (referred as Avalon in the rest of the document) architecture at the high level.

Avalon implementation is a work in progress. This document describes a target architecture that is being implemented but has not been completed yet. “Appendix A: Current Implementation Limitations” provides implementation details for the currently implemented version.

This doc is focused on internal core functionality. For information on how to build confidential compute applications using Avalon refer to the Avalon tutorials and FAQ at https://github.com/hyperledger/avalon/tree/master/docs.

1.1 Motivation

Avalon enables privacy in blockchain transactions, moving intensive processing from a main blockchain to improve scalability and latency, and to support attested oracles.

It is designed to help developers gain the benefits of computational trust and mitigate its drawbacks. In case of the Avalon a blockchain is used to enforce execution policies and ensure transaction auditability, while associated off-chain trusted compute resources execute transactions. By using trusted off-chain compute resources, a developer can accelerate throughput and improve data privacy.

Preservation of the integrity of execution and the enforcement of confidentiality guarantees come through the use of a Trusted Compute (TC) option, e.g. ZKP (Zero Knowledge Proof), MPC (Multi Party Compute), or HW based TEE (Trusted Execution Environment). While the approach will work with any TC option that guarantees integrity for code and integrity and confidentiality for data, our initial implementation uses Intel$^{@}$ Software Guard Extensions (SGX). Avalon leverages the existence of a distributed ledger to:

- Maintain a registry of the trusted workers (including their attestation info)
- Provide a mechanism for submitting work orders from a client(s) to a worker
- Preserve a log of work order receipts and acknowledgments

Avalon uses Off-Chain Trusted Compute Specification defined by Enterprise Ethereum Alliance (EEA) Trusted Compute Task Force as a starting point to apply a consistent and compatible approach to all supported blockchains.

1.2 Acronyms

Abbreviation	Abbreviated Phrase
EEA	Ethereum Enterprise Alliance
K8S	Kubernetes
KV	Key-value (storage)
SC	Smart Contract
TC	Trusted Compute
TCS	Trusted Compute Service
WO	Work Order

1.3 Glossary

Refer to “Terminology” section in the EEA TC API spec.

Additional items defined in the table below.

Phrase	Definition

1.4 Relevant Documents

N	Document	Location
1	EEA-TC-SPEC	https://entethalliance.github.io/trusted-computing/spec.html
2	TC-BLOG	https://software.intel.com/en-us/blogs/2018/10/30/a-new-trusted-compute-api-for-ethereum-blockchain-solutions
3	SGX-SW-MANUAL	https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf
4	DCAP	https://download.01.org/intel-sgx/dcap-1.1/linux/docs/Intel_SGX_DCAP_ECDSA_Orientation.pdf

2 Architecture

2.1 System Overview

Blockchains deliver computational trust via massive replication but had limited throughput, and imperfect privacy and confidentiality. Adding trusted off-chain execution to a blockchain is proposed as way to improve blockchain performance in these areas. A main blockchain maintains a single authoritative instance of the objects, enforces execution policies, and ensures transaction and results auditability, while associated off-chain trusted computing allows greater throughput, increases Work Order integrity, and protects data confidentiality.

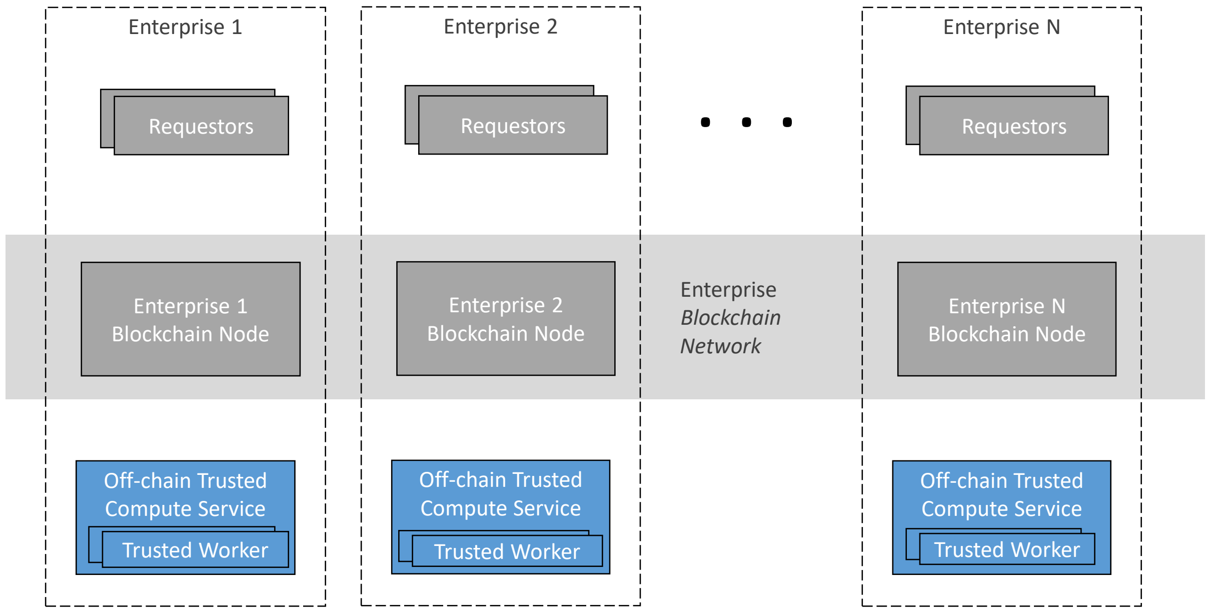

Enterprise 1
Enterprise 2
Enterprise N
Requestors
Requestors
Requestors
Enterprise 1 Blockchain Node
Enterprise 2 Blockchain Node
Enterprise Blockchain Network
Enterprise N Blockchain Node
Off-chain Trusted Compute Service
Trusted Worker
Off-chain Trusted Compute Service
Trusted Worker
Off-chain Trusted Compute Service
Trusted Worker

Figure 1 System Overview

The figure above depicts an example of blockchain with N member enterprises. Each enterprise has Requesters, a blockchain node and one or more Trusted Workers (hosted by a Trusted Compute Service). Requesters submit Work Orders, and Workers execute these Work Orders. Work Order receipts can be recorded on the blockchain. While each of the enterprises in the figure above contains all three major components, this is not necessary. For example, Requesters from Enterprise 1 may send Work Orders to a Worker at Enterprise 2 and vice versa. Ultimately, an enterprise is free to host any combination of the three elements depicted on the figure above (blockchain node, Requester, off-chain Trusted Compute Service). Accessing resources across multiple enterprises increases network resilience, allows more efficient use of resources, and provides access to greater total capacity than most individual enterprises can afford.

2.2 System Description

A diagram depicts Avalon high level architecture. The blue color depicts reusable components; the orange color depicts application specific components.

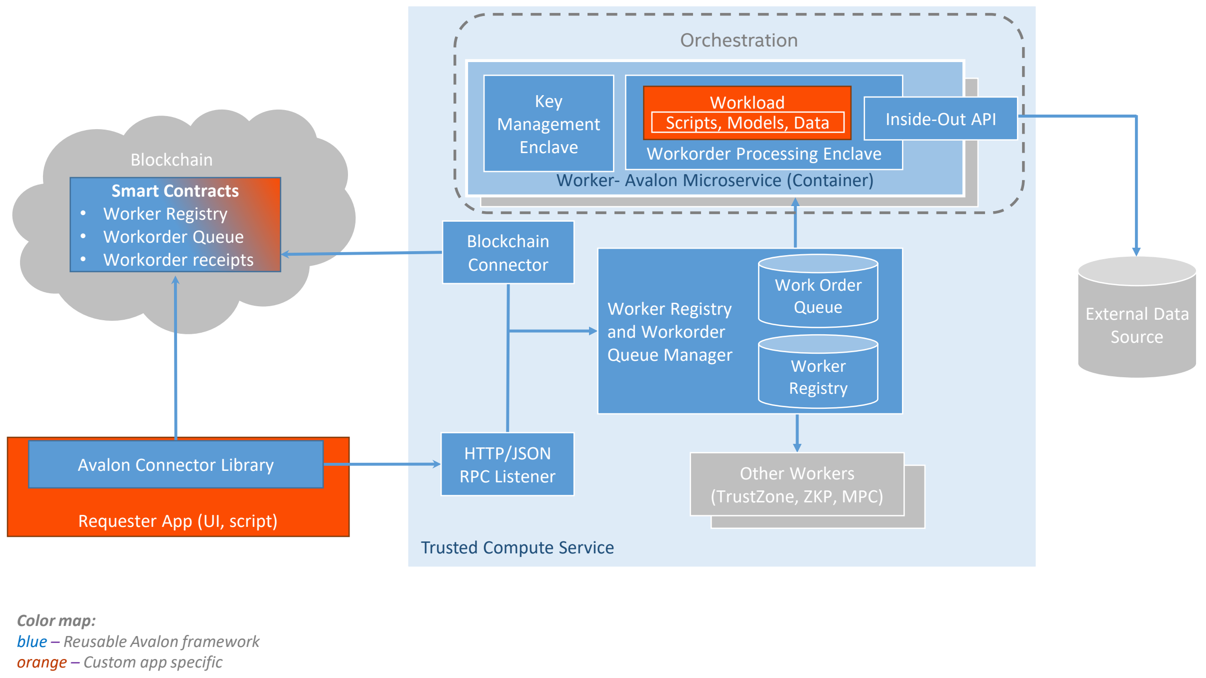


Figure 2 High Level Architecture

Trusted Compute Service (TCS) hosts trusted Workers and makes them available for execution of Work Orders (WO) submitted by Requesters via front end UI or command line tools. Work orders also can be submitted by (Enterprise app specific) smart contracts running on the DLT.

2.2.1 Trusted Compute APIs

There are four interfaces implemented according to [EEA-TC-SPEC]. Even though initially designed for Ethereum these interfaces are uniformly implemented for all supported DLTs.

- *TCS catalog lists* available TCS and provides blockchain address and/or URI where about their corresponding Workers can be discovered
- *Worker registry* that lists Trusted Compute Workers. It includes attestation verification report, public RSA encryption key, and public ECDSA SECP256K1 verification key
- *Work order processing API* that allows submit Work Order requests and receive corresponding responses with data encrypted end-to-end between the Requester the Worker. Both the request and response may include multiple data items independently encrypted by different parties using different keys. The Requester may optionally sign its requests. Even if the Requester doesn't sign the request (aka an anonymous request), there a mechanism enforcing Work Order request integrity. The enclave always signs its responses.
- *Work Order receipts API* that can be used for payment processing, auditing, and dispute resolution. The receipts may be submitted and signed by the worker, Requester, or other

participants (e.g. data owners). A work order may have zero, one, or more associated receipts

There are two models of operation:

- *Proxy model*, which relies on the smart contract (Ethereum or Sawtooth with Seth) or chaincode (Fabric) running on the DLT for managing all any subset APIs listed above
    - DLT connector on the diagram above depicts components implementing interactions between the DLT and TCS.
    - Pluggable connectors abstract DLT specific APIs from the rest of implementation. A single TCS may be connected to multiple DLTs concurrently (not shown on the diagram)
    - Requester utilizes an Avalon proxy (aka on-chain) APIs running on the blockchain and implemented using the blockchain specific programming model, e.g. Solidity smart contracts on Ethereum or chaincode on Fabric.
- *Direct model*, which provides a JSON RPC API for any of the APIs listed above except for the TCS catalog.
    - TCS listener on the diagram above depicts entry point for the direct model APIs.

Direct model was introduced as a complement to the proxy model to facilitate specific use cases that are hard to address relying on the proxy model alone, e.g. processing sensor high volume data streams (data filtering and pre-processing for IoT, supply chain), addressing regulation or enterprise policy requirements (e.g. even encrypted PII data cannot be shared outside of the organization managing PII), or aggregating (large volume of) worker receipts.

A hybrid model combining elements of both proxy and direct models can be also utilized.

2.2.2 Workloads

Trusted Worker executes applications specific workloads and implements following capabilities:

- Worker Attestation service, e.g. in case of Intel SGX TEE it can be an Intel IAS or 3rd party (DCAP) Service
- Work Order Invocation service that verifies integrity of the Work Order and the Requester signature, decrypts input and encrypts output data, signs the result, and creates the Work Order receipt update
- External IO Plug-in Interfaces that allows a workload running inside of TEE to read and write data from/to external data source, e.g. hosted locally by the TCS or remotely in the cloud. The Interface provide a basic infrastructure from crossing trust boundaries. Actual data access and format is application specific and depicted as Data Connector on the diagram
- Isolation of the work order execution from the worker key management to mitigate and limit impact from app or user errors resulting in a compromised a work order processing enclave. To isolate worker private key management only a fixed function Key Management Enclave (KME) is allowed an access to the worker's private signing and encryption keys. KME preprocesses work order requests and postprocesses work order responses therefore preventing enclaves processing work orders from accessing the

worker private keys. In addition, KME can enforce work order execution policies, e.g. allowing execution of only one work order before restarting the work order processing enclave.

Workloads can be either precompiled at the build time (e.g. written in Rust or C++) or provided at runtime scripts (e.g. Solidity or Python). In the latter case a corresponding script interpreter is precompiled into the worker at the build time and the scripts are loaded at runtime during the work order processing. The script can be chosen from a list provided by the TCS before the work order submission or dynamically provided by the Requester (unknown to the TCS). In the latter case the script can be included in the work order request directly or a reference to the script location can be provided in the work order request.

Workload logic is generally application specific. That is why they a depicted in orange. Consequently, the script execution policy enforcement is application dependent.

Initial Avalon implementation supports SGX based workers, but, as it is depicted on the diagram, additional worker types are expected to be supported in the future, e.g. ZKP, MPC, or different TEE types.

2.2.3 KV Storage

Interactions between TCS components is done via a KV (Key-Value) Storage that is implemented on the top of a memory-mapped database, LMDB. The KV Storage Connector allows utilizing the KV Storage on a single physical system or across multiple physical systems. The Adaptor abstracts LMDB from the rest of TCS so production implementation may choose to replace LMDB with another database of their choice.

2.2.4 On-Chain Components

On-chain components (Ethereum Solidity smart contracts or Fabric chaincode) implement proxy model APIs described in section "Trusted Compute APIs." Above.

Decentralized components (smart contracts or chaincode) running on DLT are depicted as a mix of both colors because in some cases Avalon's baseline implementation is sufficient, while in others decentralized application may have to change or extend baseline implementation.

2.2.5 Front-end Application

Front-end application can be a UI or a script that is used to initiate work order requests and consume work order responses. Front-end application sometimes can be called a client, but this document doesn't use this term because Ethereum uses this term for a blockchain node.

The diagram above depicts the front-end application completely outside of the blockchain for clarity. In real world scenarios the front-end application may include both on- and off-chain components (aka dApp).

Front-end application is depicted in orange because its functionality is use case specific. Avalon provides several front-end application examples. Avalon also provides "Avalon Connector Library" depicted in blue because it can be directly reused by the applications. The library can be used to:

- Abstract which model – proxy or direct – is used
- Abstract which DLT is used in the case of the proxy model
- Verify a worker attestation information

2.3 High Level Execution Flow

A diagram below depicts an overall execution flow.

Initially a TCS creates a new worker. During this process its attestation info is generated and published in the worker directory on the blockchain (or/and on an intermediately server). Details of the attestation info are defined in the following section "Worker Attestation Info."

Later a requester searches the worker directory and finds the worker, verifies its attestation info, and stores this information for further use

During worker order invocation the requester create a worker request and, optionally, a corresponding work order receipt. During the work order execution stage, the TCS retrieves a work order and processes. After the work order is processed – during the stage processing work order response – the TCS puts the work order response on the blockchain and, optionally, updates the work order receipt.

Additional details about last three stages can be found in section "Work Order Execution Protection" later in this chapter.

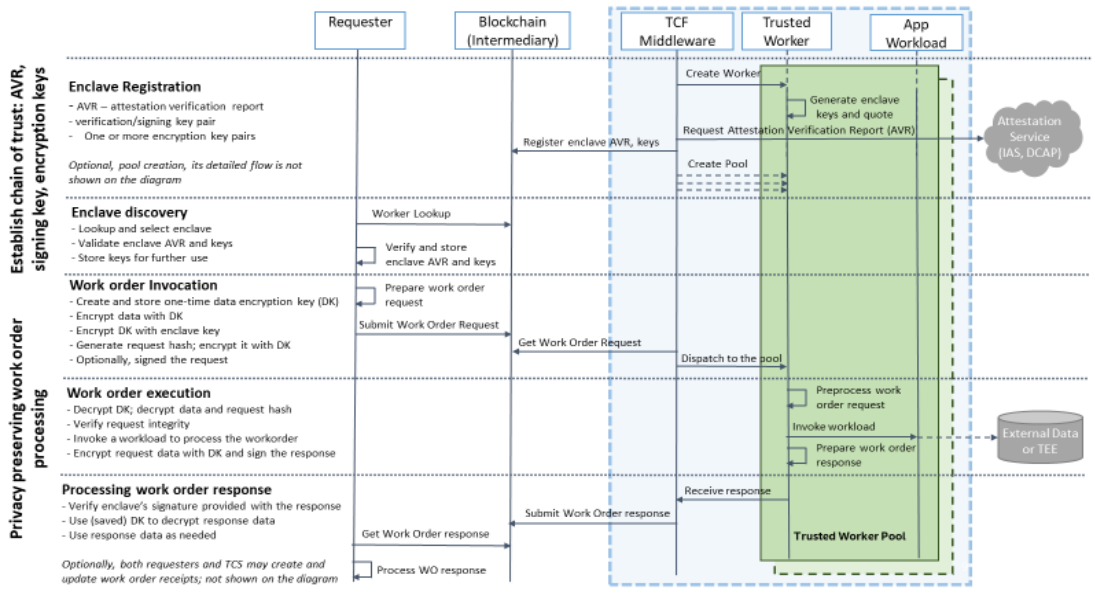

Requester
Blockchain (Intermediary)
TCF Middleware
Trusted Worker
App Workload
Establish chain of trust: AVR, signing key, encryption keys
Enclave Registration
- AVR – attestation verification report
- verification/signing key pair
- One or more encryption key pairs
Optional, pool creation, its detailed flow is not shown on the diagram
Create Worker
Generate enclave keys and quote
Request Attestation Verification Report (AVR)
Attestation Service (IAS, DCAP)
Register enclave AVR, keys
Create Pool
Enclave discovery
- Lookup and select enclave
- Validate enclave AVR and keys
- Store keys for further use
Worker Lookup
Verify and store enclave AVR and keys
Privacy preserving work order processing
Work order Invocation
- Create and store one-time data encryption key (DK)
- Encrypt data with DK
- Encrypt DK with enclave key
- Generate request hash; encrypt it with DK
- Optionally, signed the request
Prepare work order request
Submit Work Order Request
Get Work Order Request
Dispatch to the pool
Work order execution
- Decrypt DK; decrypt data and request hash
- Verify request integrity
- Invoke a workload to process the workorder
- Encrypt request data with DK and sign the response
Preprocess work order request
Invoke workload
External Data or TEE
Prepare work order response
Processing work order response
- Verify enclave's signature provided with the response
- Use (saved) DK to decrypt response data
- Use response data as needed
Receive response
Submit Work Order response
Get Work Order response
Trusted Worker Pool
Optionally, both requesters and TCS may create and update work order receipts; not shown on the diagram
Process WO response

Figure 3 High Level Execution Flow

2.4 Worker Attestation Info

A diagram below depicts the worker attestation chain of trust using Intel SGX based TEE as an example.

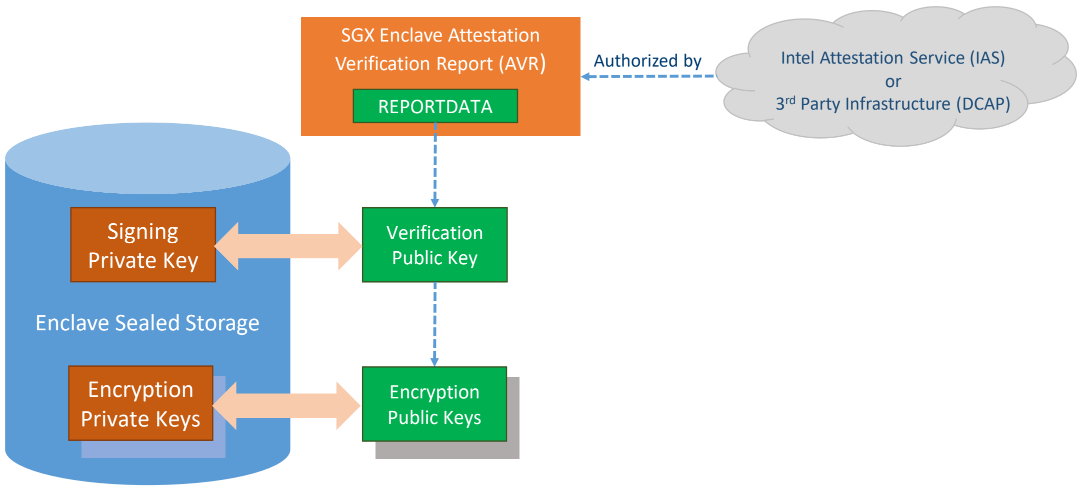


Figure 4 Chain of Trust

The worker creates an Attestation Verification Report (AVR) that is signed by Intel® IAS, if EPID attestation mechanism is used, or the worker creates a quote, if DCAP attestation mechanism is used.

- MRENCLAVE value can used to identify the worker's workload
- REPORTDATA data builds up the trust chain. It contains a hash of the verification key
- Details of the attestation mechanism can be found in the Intel® SGX SW development documentation
- Avalon framework abstracts away details of the AVR or quote creation so the app developer is not expected to learn how to generate them

Trusted code in SGX enclave generates asymmetric signing/verification key pair

- One signing key pair per the enclave lifetime
- Used to sign encryption keys and work order results
- Verification Public Key is published on blockchain
- Verification Public Key Hash in the AVR's REPORTDATA
- Private key is stored in the enclave sealed storage

Trusted code in SGX enclave generates asymmetric encryption key pairs

- One or more encryption keys pairs per enclave
- Used to encrypt one-time symmetric transaction key
- There can be reusable or one-time keys
- Encryption key lifetime is defined by app policy
- Public key is published on blockchain
- Public key is signed by the enclave signing key
- Reusable private key is stored in the enclave sealed storage
- One-time private key is stored in the enclave memory (not shown)

2.5 Work Order Execution Protection

Work order execution integrity and confidentiality are enforced by the following work order sequence of execution. This sequence is for a so-called worker singleton. Utilizing a worker pool involves additional steps described in section "Worker Pool"

1. Requester performs the following operations when it submits a new work order request:
   - Generates a one-time symmetric encryption AES-GCM-256 key (SEK)
   - Encrypts the work order request data with SEK
   - Calculates a SHA256 hash of the request and encrypts it with SEK
   - Encrypts the SEK with enclave's public (RSA-OAEP) encryption key
   - Optionally, signs the SHA256 hash using its verification key
2. Worker (e.g. trusted code inside of SGX enclave) performs the following operations:
   - Decrypts the one-time symmetric encryption AES-GCM-256 key (SEK)
   - Decrypts the work order request data
   - Calculates a SHA256 hash of the request

    - Decrypts the SHA256 hash provided in the request and compares it with the value calculated above
    - If the signature of request's SHA256 hash is provided, verifies it using the requester's key
    - Processes the work order
    - Encrypts the work order response data with SEK
    - Calculates a SHA256 hash of the response
    - Signs the hash value with the enclave's private signing key
3. Requester performs the following operations when it receives the work order response:
    - Retrieves (e.g. locally stored) the worker signing key
    - Verifies the work order response signature generated by the worker
    - Retrieves (e.g. locally stored) SEK used during the work order request submission
    - Decrypts output data items from the work order response

It should be noted that a work order may include multiple input and output items. Each of them can be encrypted by its own symmetric key, usually generated by a different party. In such case the flow above is still generally applicable, except that multiple SEKs are generated and used and step 3 is performed by all involved parties.

3 Trusted Compute Service

A diagram below depicts architecture of the Trusted Compute Service (TCS)

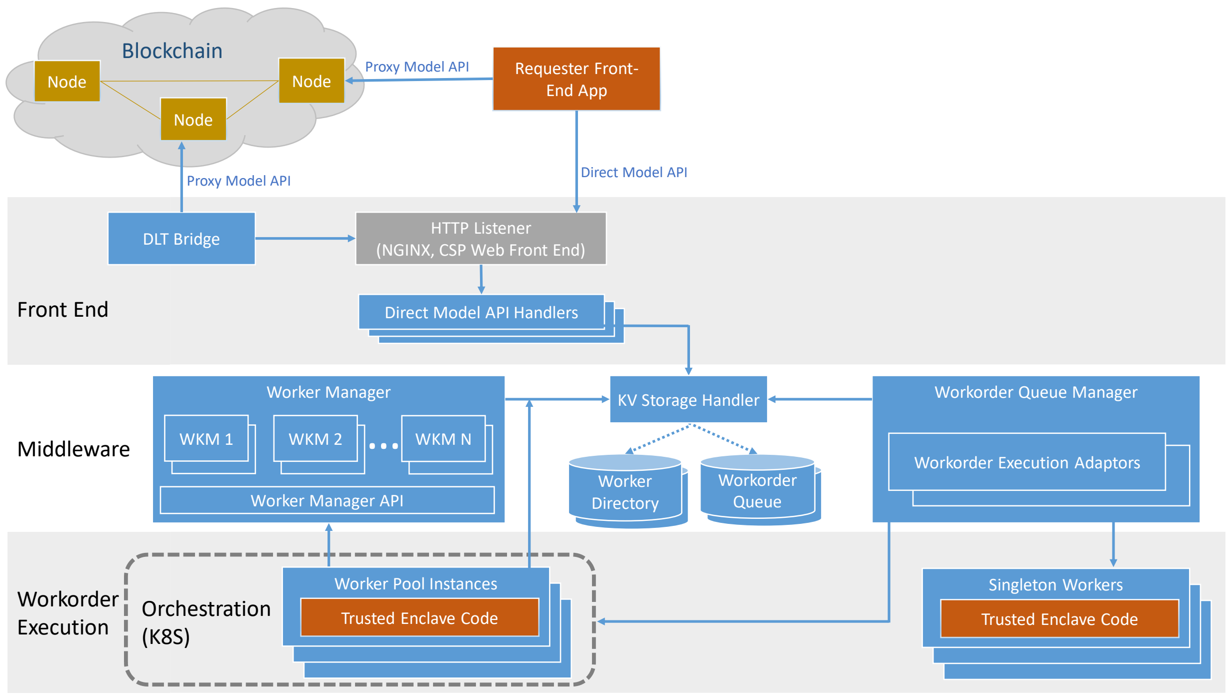


Figure 5 TCS Architecture

TCS can be divided into three tiers:

- Front-end that interacts with DLTs and processes JRPC requests
- Middleware manages the worker directory, the work order queue, and supports worker pools
- Workorder Execution tier processes scheduled work orders

Subsequent sections describe these tiers in more details.

3.1 Front-End Components

There are two types of Front-End modules:

- Direct Model API Handler, which processes JRPC requests from requesters
- DLT Bridge, which connects to the DLT and supports a proxy model interface

Direct Model API Handler can be utilized in two modes:

- Standalone mode for small scale installations. The handler receives incoming JRPC calls via its built-in HTTP listener. It also can be useful for an app development. It is not depicted on Figure 5.

- Plug-in model for large scale installations. The handler doesn't use its HTTP listener. Instead multiple handler instances are launched by a load balancer or a HTTP proxy, e.g. NGINX or CSP specific load balancer. Figure 5 above depicts this case.

DLT bridge can interact with a DLT in two ways:

- Submit DLT specific transactions, e.g. to update worker information, submit a workorder result, and submit a workorder receipt
- Optionally, it can also listen for DLT events, e.g. arrival of new workorders

DLT bridge interacts with the TCS Middleware tier indirectly via Direct Model API Handler. It allows to reuse existing functionality and handle all requests coming from different sources in a consistent manner.

3.2 Middleware

TCS middleware includes:

- KV Storage Manager
- Worker Manager
- Workorder Queue Manager

All communication between Front-End and Middleware components is done via the KV Storage Manager, which maintains:

- Worker Directory, which contains information about all available workers
  - Attestation info
  - Type (singleton or pool)
  - Execution parameters (e.g. K8S pod, max allowed number of concurrent instances)
- Workorder Queue, which contains work order requests, responses, and, optionally, receipts

The KV Storage Manager is a thin wrapper implemented on the top of LMDB with a sole goal of providing an abstraction JRPC API that allows to:

- Provide a uniformed mechanism for accessing KV Storage (LMDB) from other components running on different physical systems
- Provide an extensibility point for replacing underlying LMDB with another DB if needed

It is anticipated that some large-scale Avalon deployments may require to replace LMDB with a more scalable open source or commercial option. Porting the KV Storage Manager to support a different underlying DB should be a relatively simple task and, if its external facing API is preserved, overall Avalon functionality should stay intact.

The Worker Manager is responsible for:

- Worker encryption key creation and update policy enforcement
- Maintenance of Worker Key Managers (WKM) for the worker pools. There are multiple instances for WKMs for each pool for high availability and redundancy.
- Worker Manager API that supports operations of worker pools with isolated key management

Refer to section "Worker Types and Orchestration" for details about worker pools.

Initially, workers will be deployed manually (aka out-of-band) by the TCS personal and admin utility (not shown on Figure 5 above) will be used to create a corresponding entry in the Worker Directory.

In the future the Worker Manager will be extended to support worker deployment on-demand (in response to a programmatic requestor request). It will require a definition of new APIs in addition to a corresponding implementation.

Worker Manager and Worker Key Managers provide the following interfaces:

Redirection API. This interface is implemented by the Worker Manager only.

- *Redirect()* takes worker id as an input and returns a URI that should be used by the caller to invoke Key Management API

Key Management API is implemented by both Worker Manager and Worker Key Managers. The latter provides the actual implementation in the trusted code. The Worker Manager serves as a proxy and forwards requests to a corresponding Worker Key Manager.

- *GetRandomId()* returns a randomly generated number to ensure a freshness of the requesting worker instance. The requestor will include the random id in its attestation info, e.g. as a CONFIGID fin case of SGX2. This call is made by untrusted part of the worker instance and used to create a trusted SGX enclave
- *GetWorkerPrivateKeys()* takes as an input the requester's attestation info. It returns private signing and encryption keys of the called worker instance encrypted with the public encryption key of the caller (extracted from caller's attestation info). The input must satisfy one of the following conditions:
    - In the case where a new Worker Key Manager instance is being instantiated, attestation info input must match the info of the called worker instance with the exception of the random id and keys. The random id must be one of the ids returned by a previous *GetRandomId()* call that has not been used previously. The caller's encryption and verifications keys may have any values
    - In the case where an existing Worker Key Management instance needs to have its encryption key updated, the attestation info input must match the info of the called worker instance, including the signing key but excluding the random id and the encryption key.
- *SetWorkerPrivateKeys()* takes as input private keys returned by a corresponding call to *GetWorkerPrivateKeys()* and the attestation info of the worker that produced the private

keys. This function updates encryption and (on the first call) signing key of the called Worker Key Manager instance

- *GenerateNewEncryptionKey()* takes a nonce and tag as optional input parameters (if input parameters are not provided, they are generated). This function generates a new encryption key pair and returns nonce, tag, public key part, and a signature. The private key part is stored in the sealed storage. After calling this function, all other instances of the Worker Key Manager must be updated by calling *GetWorkerPrivateKeys()* and *SetWorkerPrivateKeys()*
- *PreprocessWorkorder()* takes as input a work order request JSON doc and the caller's attestation info. The attestation info must match one of the ids provided by *GetRandomId()* and not used by *GetPrivateKeys()*. It returns all the symmetric data encryption keys found in the request encrypted with the public encryption key of the caller. The work order hash and caller's public verification key are stored to be used during a corresponding *PosprocessWorkorder()* call
- *PostprocessWorkorder()* takes as input work order request doc and work order response JSON doc. This function returns an updated work order response JSON that is signed by the private signing key of the called Worker Key Manager

The use of the APIs above is described in section "Worker Pool".

The Workorder Queue Manager:

- Throttles the stream of work order requests, e.g. based on the available HW
- Serializes execution of work orders with dependencies, e.g. work order ordering or persistent storage access. The serialization model is multiple reads and one write.
    - Note that the manager can serialize only those work orders that they have their dependency explicitly specified in the request.
    - Some application may have their own serialization techniques utilized instead of relying on course grained TCS serialization, e.g. for fine grained access to an external DB
- Initiates work order execution via one or more Workorder Execution Adaptors
- Maintains work order queue size by purging old entries

The following Workorder Execution Adaptors will be supported:

- Basic, used for execution of singleton-workers. It also assumes a direct interaction with the workers without assistance from an orchestration engine. This option is for development support and small-scale installations
- Kubernetes (K8S). In this case workers are packaged as K8S pods and they launched via K8S API. Figure 5 shows that this mode is used for worker pools (and this is its primary intent), but it can be also used for singleton-workers

Figure 5 above depicts orchestration engine only around workers because in this case there is a programmatic interaction between Avalon code and K8S. K8S can be used to manage many other or all Avalon modules, but this would be DevOps choice that should not require any changes in the Avalon implementation.

If needed additional Workorder Execution Adaptors can be implemented, e.g. to support a CSP based orchestration mechanism.

3.3 Worker Types and Orchestration

There are two types of workers:

- Singleton workers. Only one instance of such worker exists on a fixed physical system
- Worker pools. There are multiple worker instances that can be deployed on dynamically allocated systems

Both worker types can be deployed:

- Statically. A singleton-worker or an instance of the worker pool is installed on a specific physical system. This is information is provided in Worker Directory and utilized by the Workorder Queue Manager to submit a work order to the worker. In this case a worker starts a HTTP listener and listens for work order requests from the Workorder Queue Manager.
- Dynamically. An orchestration engine (K8S) is utilized to manage work pool instances. In this case a worker is packaged into and launched as a K8S pod.

Refer to description of Workorder Queue Manager in section “Middleware” for more details.

Following sections describes execution of both worker types.

3.3.1 Singleton Worker

Singleton Worker has only one instance that can run on a specific physical system where the worker was initially instantiated. The attestation info generated during the worker instantiations is recorded in the Worker Directory. The worker encryption keys can be updated at the later time on the same physical system.

Multiple singleton workers can be created to execute the same time of workloads, but each of them will have its own attestation information and keys and will be presented as an independent record in the Worker Directory. If there are multiple singleton workers that can process the same workloads, it is responsibility of the requester(s) to load balance across of them, e.g. by using randomization or round-robin policy.

A singleton worker cannot be migrated to another physical system. It is a normal behavior a Trusted Execution Environment (TEE) that support HW based attestation (e.g. Intel® SGX enclave). Ability to migrate workers (aka SGX enclaves) to different physical systems is essential for scalable solutions, but it requires additional mechanism for secure migration worker private keys between physical systems. Avalon supports worker pools to handle this scenario.

Avalon singleton worker processes work order and manages its keys in the same TEE (aka SGX enclave) while in case of worker pools, a default behavior is to use different TTEs for key management and work order processing that allows mitigate and limit an impact of compromised workers, e.g. in case of user error or an exposed application vulnerability. Avalon allows to have a worker pool set up to have a single instance of key management TEE and a single instance of workload processing TEE, both bound to the same physical system. This effectively as having a singleton with key management isolation.

Refer to the following section for worker pool description.

Worker Pool

Worker pool includes:

3.3.2

- One or more Worker Key Management (WKM) instances
- One or more Workorder Processing (WP) instances

WKM has access to the worker pool private encryption and signing keys; WP does not. Instead WP relies on WKM to assist in decrypting symmetric data encryption keys in the work order request. Multiple WKM instances are needed for fault tolerance (if a physical system decommissioned), availability, and for work order processing scaling. Similarly, multiple WP instances provide high availability and scaling.

During build time the WKM is provision with these policies:

- Private key migration policy, which defines the worker's private keys that can be migrated to another WKM instance or must say within a single WKM instance
- WP association policy
    - Hardcoded at build time. In this case one or more MRENCLAVE values of the associated WPs are compiled-in in the WKM binary. This policy requires a coordinated build of corresponding WPs and WKM binaries. First one or more WP binaries are built and then their MRENCLAVE values are used to build the WKM
    - Dynamic at instantiation time. When a WKM instance is instantiated, it is provided with a list of the associated WPs. This list cannot be changed at the later time. All WKM instances must have the same list
- WP localization policy defines if WP(s) must run on the same physical system as WKM
- Maximum number of allowed concurrently running WP instances

When a new worker pool is created, additional WKM instances are created as needed (in the WKM's built-in policy allows replication):

- A WKM instance is created and its attestation info is used for the whole new worker pool. From the requester perspective it looks the same as a singleton worker. Number of actual WKM and WP instances is unknown outside of TCS with an exception when a built-in policy limits WKM or/and WP to a single instance only.
- Additional WKM instances are created by following steps
    - A new WKM instance call *GetRundomId()* AOPI at the existing instance
    - The new instance generates attestation info and calls *GetPrivateKeys()* API at the existing instance. The keys are updated in the trusted enclave by calling *SetPrivateKeys()*

The steps above performed by the Avalon personal manually. In the future Worker Manager can be extended to handle it programmatically. Attestation information of each WKM instance is recorded in the Worker Directory.

At runtime Worker Manager may refresh or create additional worker keys as follows:

- Call *GenerateNewEncryptionKey()* of one of the workers WKM instances
- Update the worker info in the Worker Directory
- Use *GetPrivateKeys()* and *SetPrivateKeys()* to update all other WKM instances

This is a high-level flow for a work order processing by the worker pool:

1. The Workorder Queue Managers dispatches a work order request for execution to the K8S Workorder Extension Adaptor
2. The adaptor checks the Worker Directory to see if there is an already running and available required Workorder Processor (WP) and, if there is no WP running it requests K8S to start it. The started WP marks itself as available in the Worker Registry
3. The adaptor submits a work order request to the WP by using a JRPC
4. The WP marks itself as busy in the Worker Registry
5. The WP calls Redirect() at the Worker Manager to get an end point for the Worker Key Management (WKM) interface
6. The WP calls *GetRandomId()* at the WKM
7. The WP starts the TEE and stores its attestation info that includes the id from step 6
8. The WP calls *PreporocessWorkorder()* at the WKM to receives data encryption keys
9. The WP processes a work order request
10. The WP calls *PostprocessWorkorder()* at the WKM to re-sign the work order response with the workers signing key instead of interim WP’s signing key
11. Depending on the policy WP may restart its TEE (enclave). If a restart is not required, steps 6 and 7 above are skipped during processing of subsequent work order requests
12. Depending on the policy WP may have to shut down after processing a specified number of work order requests. The WP marks itself as unavailable in the Worker Directory before shutting down. Otherwise, the WP keeps running and marks itself as available in the Worker Directory

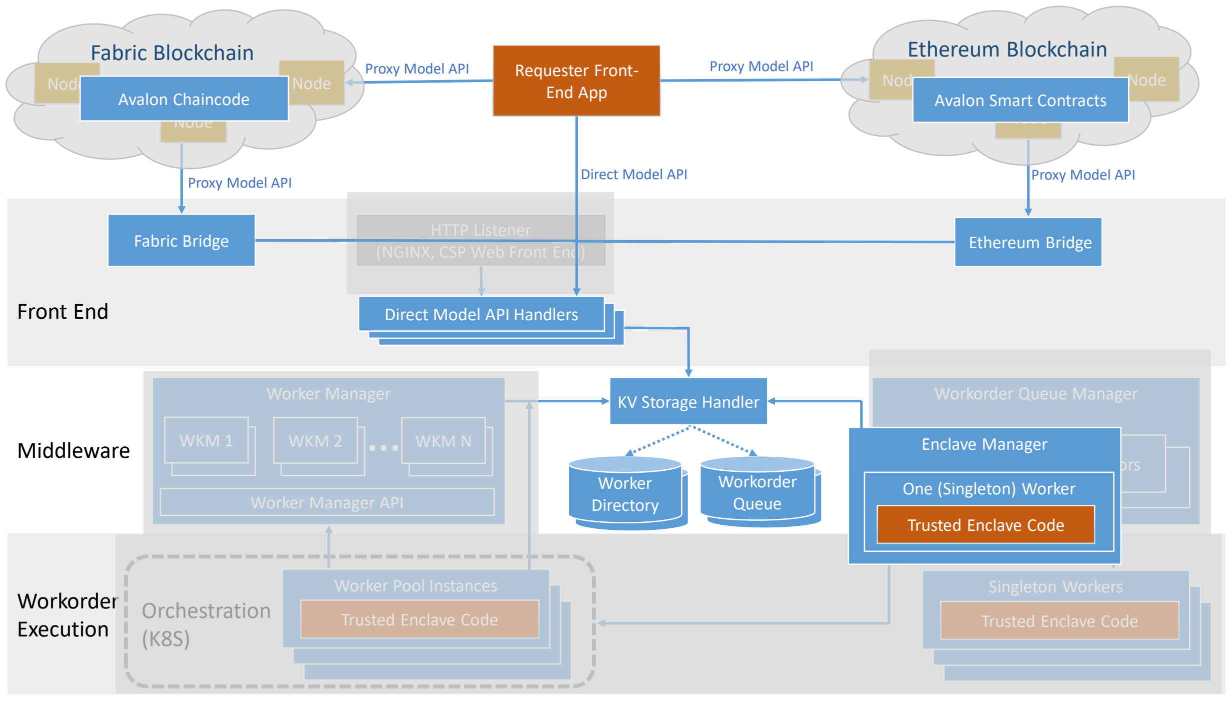
The flow above provides an abbreviated and simplified baseline logic for clarity. Performance optimization may require a modified and more complex flow that can maintain a small local work order request queue at WP and utilize K8S service facility.

Appendix A: Current Implementation

The existing Avalon implementation provides worker attestation and registration, enforcement of end-to-end workorder integrity and confidentiality, asynchronous workorder processing, direct model, and integration with Fabric and Ethereum blockchains.

Aggregation of different TEE runtimes, isolation of the key management from the workorder processing, support for the worker pools and multi-tenancy is underway.

From the design perspective, some of the components described in the document are not implemented or implemented only partially. Some of the implemented functionally is packaged differently. A diagram below provides an existing architecture mapped over the target architecture described in this document. Following sections provide additional details for specific components.



3.4 **Figure 6 Current Implementation Components**

Requester App Examples and Connector Library

The current Avalon version includes a number of examples of the Requester App https://github.com/hyperledger/avalon/tree/master/examples/apps.

The Connector Library used by (some of) these examples can be found at https://github.com/hyperledger/avalon/tree/master/blockchain_connector/avalon_blockchain_connector

There is ongoing Connector Library code optimization and refactoring, but it can be used in the direct model and in the proxy model with Ethereum and Fabric.

3.5 Direct Model Supported Blockchains

Currently supported blockchains:

- Hyperledger Fabric
  - Minifab is a convenient way to set up Fabric for development. The instructions are at https://github.com/hyperledger/avalon/tree/master/docs/dev-environments/fabric
  - Chaincode is at https://github.com/hyperledger/avalon/tree/master/sdk/avalon_sdk/fabric/chaincode
  - Fabric connector is at https://github.com/hyperledger/avalon/tree/master/blockchain_connector/avalon_blockchain_connector/fabric
  - Hyerledger Besu setup instructions are at https://github.com/hyperledger/avalon/tree/master/docs/dev-environments/besu
  - Ethereum smart contracts are at https://github.com/hyperledger/avalon/tree/master/sdk/avalon_sdk/ethereum/contracts
  - The Ethereum connector is at https://github.com/hyperledger/avalon/tree/master/blockchain_connector/avalon_blockchain_connector/ethereum

The Listener for the direct model is at https://github.com/hyperledger/avalon/tree/master/listener.

The Listener currently can work in standalone mode using its own built-in HTTP listener. There is no support for integration with a 3rd party proxy or load balancer.

3.6 KV Storage Manager

The current KV (Key/Value) Storage Manager is at https://github.com/hyperledger/avalon/tree/master/shared_kv_storage.

Its functionality will be extended to support worker pools and K8S integration.

3.7 Enclave Manager

The Enclave Manager includes partially implemented functionality of the Worker Manager, the Worker Queue Manager, and a singleton worker. This is the area where most of the new core development is going to happen. Currently, the Enclave Manager supports execution of only a single singleton worker.

The C++ source code part is at https://github.com/hyperledger/avalon/tree/master/tc/sgx/trusted_worker_manager.

The Python wrapper source code is at https://github.com/hyperledger/avalon/tree/master/examples/enclave_manager/tcf_enclave_manager