

# GoWood Proof of Concept - Technical description

## Abstract

The document to gives (1) an overview of the Proof of Concept implementation, it's limitations, and architectural vision and reasoning, and it (2) explains how the solution works on such a detailed level that it's possible to plan how to continue development work or to use it in another research project.

Ari-Pekka Lappi  
ari-pekka.lappi@flowa.fi

## Table of Content

Purpose and Audience of this document.....	2
Background and Scope of Proof of Concept .....	2
On source code, tooling and licences .....	3
Hardware requirements (recommendation) .....	3
Software requirements .....	4
Architectural overview .....	4
User interface.....	5
Use cases.....	8
Frontend technical stack.....	8
Query service .....	9
Data query and aggregation .....	10
Query Service's Technical Stack.....	12
Data model.....	12
Test data description .....	12
Critical points when linking data.....	13
On exposing and querying a distributed graph data model .....	15
On approaches to do computation over distributed data sources .....	15
Holochain data storage .....	18
How token is generated? .....	18
How row level access is granted? .....	19
Limitations and further development ideas .....	20
Deployment considerations.....	21
AWS architecture example .....	22
Architecture in global scale.....	23

## Purpose and Audience of this document

The purpose of this document is:

- 1) to give a good overview of the Proof of Concept implementation, its limitations, and architectural vision and reasoning.
- 2) to explain how the solution works on such a detailed level that it's possible to plan how to continue development work or to use it in another research project.

This document is written for a person who does not have significant experience on software development. Document presumes that the reader understands the basics of programming and information networks on high level.

## Background and Scope of Proof of Concept

The main technical problem of Go Wood project was: How to track back the origin of raw materials used in a building so that it is possible to calculate carbon footprint for the building? In the Go Wood project, the focus was in a particular type of wood components: tracking plywood used in a building to its origin in forest.

The problem can be split into five broad subproblems.

- 1) **Asset identification.** How might we identify a trackable entity? For instance, how might we identify a tree trunk delivered to a sawmill, or how might we identify a sheet of plywood delivered to a building yard.
- 2) **Asset linking.** How might we link different kind of entities as a part of the production process? For instance, when a tree trunk is delivered to sawmill, how to link it to plywood sheets produced from it?
- 3) **Cross-asset query.** How to make it possible to query data from diverse data sources in a secure way? For instance, how to link a tree trunk record that is produced by a tree harvester and the plywood sheet produced from the tree in such way that a stakeholder gets only the information that they are authorized to get. For instance, it should not be possible for the harvesting company to estimate the inventory size or order base of the sawmill by using shared data and utilize this information in pricing and competition.
- 4) **Data delivery.** What kind of data do the stakeholders need to share in order to make asset linking and cross-asset query possible? How might this data be delivered in the system?
- 5) **Data visualization and exploration tooling.** In addition, it should be possible to visualize this all so that a non-technical person can explore the data.

The following table summarizes how these subproblems are studied and experimented in the PoC:

PROBLEM	IN POC?	NOTES
1. ASSET IDENTIFICATION.	Not included	
2. ASSET LINKING	Partially	Demo contains mostly mocked, simplistic asset linking scheme. Data is mocked simply because we got very limited amount of data from the stakeholders.
3. CROSS-ASSET QUERY	Yes	Demo contains a query service, that has 3 simplistic data sources.

4. DATA DELIVERY	Not included	
5. DATA VISUALIZATION AND EXPLORATION TOOLING	Yes	Demo contains a UI to explore a simplistic data model. UI is designed to be relatively easy to extend to more complex models.

Main constraints and impediments thinking of the scope of Proof of Concept were:

- **Short time frame.** We only had 6 weeks of time to implement the proof of concept. In addition to this we had to reserve some time for planning and documenting. Given the short timeframe a more extensive solution was delivered than we expected at first.
- **We got limited harvester dataset.** For instance, timestamps were not included in the dataset.
- **We did not get any data from sawmill** – not even database schemes. This made it practically impossible to evaluate feasibility of asset linking, and hence it's mostly scoped out from the demo. We also didn't try to get real data from a construction company or wood retailer company, as we didn't get it from sawmill.
- **We did not have real 3D models of the building**, that could be applied within that time frame of demo. The 3D model used is taken from a 3D model catalogue.

## On source code, tooling and licences

Source code is available in <https://github.com/AaltoGoWood/gowood-poc>. PoC is open source (MIT licence) and all used dependencies are open source (multiple licences). Used 3D model is used with Creative Commons licence (CC BY 4.0). Model is created by lowlypoly (<https://sketchfab.com/3d-models/building-apartment-2-305a61a885b24540af1c408b78a01b45>).

Used tree trunk data is provided by Arbonaut Oy and used with their permission. Building shown in map including Geojson model is OpenData provided by Helsinki City.

Instructions on how to run or develop the PoC are available in the Git repository. To run the demo, Docker Community Edition (free and open source) or Docker Enterprise (proprietary & not free) is needed.

## Hardware requirements (recommendation)

To run the PoC solution you need to start 4 virtualized servers using Docker.

- RAM: 16GB
- Processor: Duo core 64bit (Tested with 1.90GHz duo core processor (Intel® Core™ i7-8650U))
- HDD: approximately 20 GB (+ 3GB for Docker)
- Network connection

### Notice:

- Docker for Windows requires that hardware virtualization (VT-X) is activated in BIOS.

## Software requirements

APPLICATION	FOR RUNNING DEMO	FOR DEVELOPING
<b>DOCKER</b>	Yes	Yes
<b>MODERN BROWSER</b>	Yes	Yes
<b>GIT</b>	Recommended	Yes
<b>NODE.JS AND NPM</b>	No	Yes
- SEE README IN GITHUB FOR DETAILS		
<b>LEININGER AND JAVA DEVELOPMENT ENVIRONMENT</b>	No	Yes
- SEE README IN GITHUB FOR DETAILS		

### Notice:

- Running Docker in Windows requires Hyper-V
- Application is tested with Chrome 78. Application does not work in Internet Explore.
- See instructions for setting up development environment and demo environment in GitHub repository: <https://github.com/AaltoGoWood/gowood-poc>

## Architectural overview

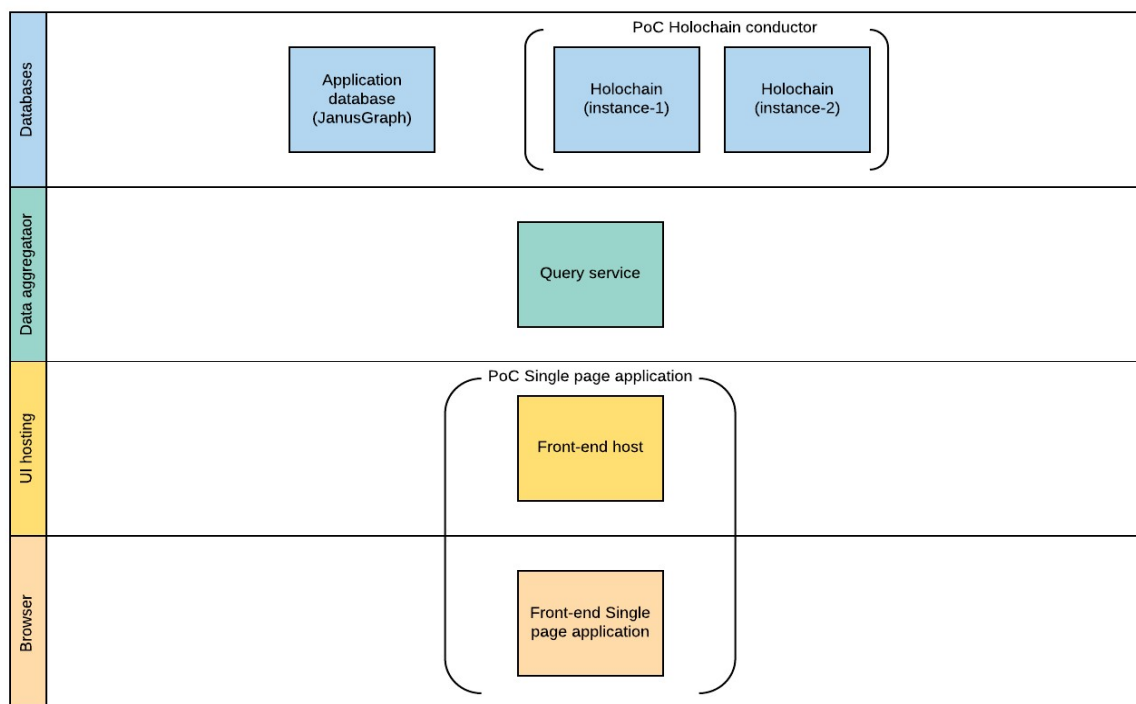


Figure 1 Architectural overview

Component	Description
<b>Frontend Single Page Application (SPA)</b> <ul style="list-style-type: none"> <li>- <b>Frontend service</b></li> <li>- <b>Browser application</b></li> </ul>	<p>User interface logic.</p> <p>A frontend service is needed to host static files required by the frontend (html, JavaScript files, and styles) and proxy requests to the query service. In the PoC environment frontend service contains some development tooling (Webpack).</p> <p>When developing, the frontend service and the frontend SPA are run with a single command and the code related to them both is in the same folder /frontend in the GitHub repository.</p>
<b>Query service</b>	<p>Query service orchestrates and combines data from multiple database.</p> <p>Current version of Query server does not support CORS and hence, queries from frontend are proxied through frontend host service.</p>
<b>Application database</b>	<p>In this PoC, the application database represents a building information model database (BIM). It could as well be a Holochain instance in some future version. In the PoC, it is needed for just one thing – to map a building identifier to plywood sheets. See <b>matcher</b> in data model chapter for more details.</p> <p>In the PoC JanusGraph is used for main two reasons:</p> <ul style="list-style-type: none"> <li>- JanusGraph uses Apache Tinkerpop's Gremlin as a query language. Gremlin is not only a convenient way to query a graph data model, but also extensible and m</li> <li>- Apache Tinkerpop can be extended to use Holochain as a storage layer for queries. Currently Apache Tinkerpop is supported in 22 different databases including AWS Neptune and Azure CosmosDB.</li> </ul>
<b>Holochain instances 1 and 2</b>	<p>There are two separate Holochain instances running in the same DNA (smart contract). Plywood data is stored to instance-1 and Tree trunk data is stored to instance-2.</p> <p>Both instances are run on the same Holochain container for the sake of convenience. It should be possible to start Holochain instances in different containers located in different networks.</p>

## User interface

User interface contains two areas:

- Details panel is on the left
- Visualization panel is on the right

From the end user perspective, the Proof of Concept contains 4 main features:

1. **Building browser.** Building browser view works as a landing page to the application. Click the building on the map to navigate to the detail view. In the PoC, there is only one building included (see Figure 2). The building view uses the map visualization component (i.e. trees and buildings are in the same map).

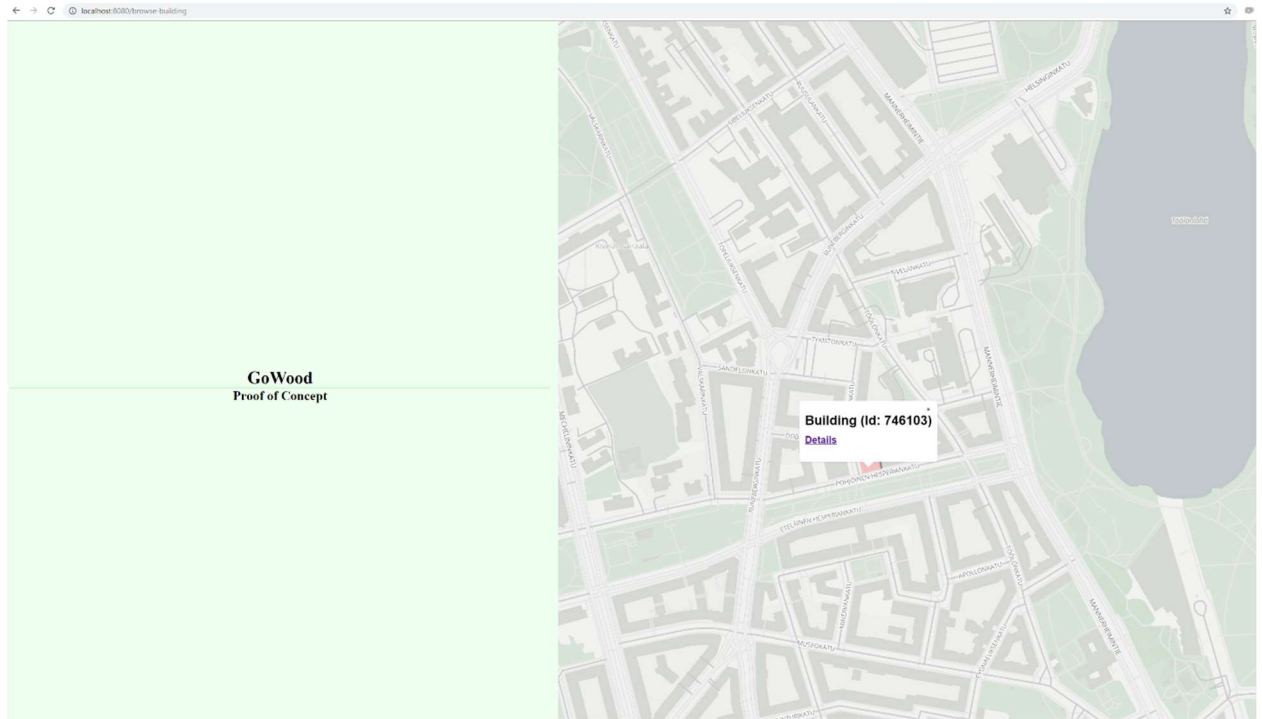


Figure 2 Building browser

2. **Building 3D visualization.** 3D visualization is the only visualization for building, and the default visualization for plywood sheets. In the PoC there is only one 3D model available and its not a real model of the building on a map. (see Figure 3)

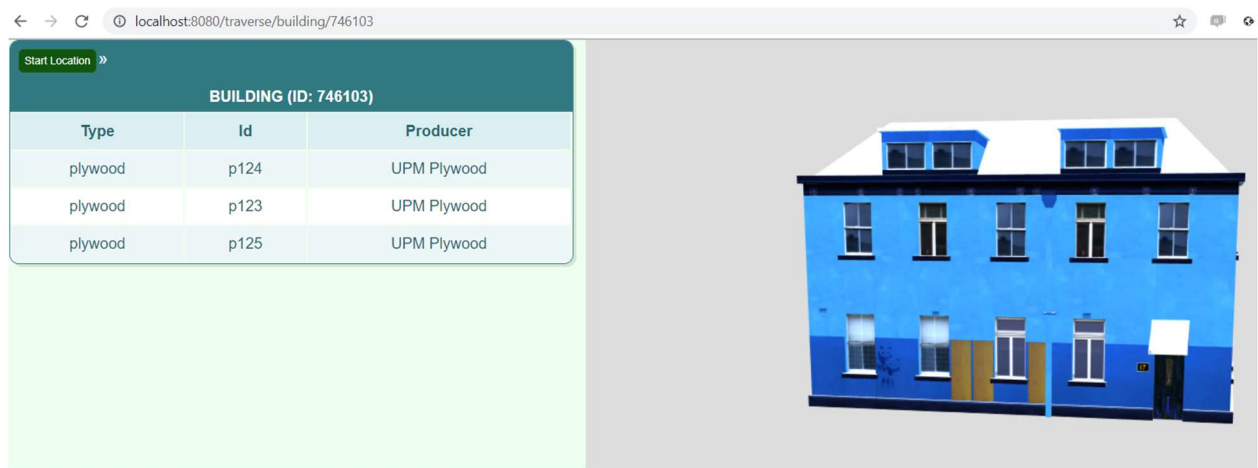


Figure 3 Building details and 3D model visualization

3. **Asset details.** Asset details component is currently used to show building, plywood and tree trunk details. Asset details component is dynamic and it's possible to add new asset types to the UI easily. Currently you need to specify a layout for an asset (see Data model): layout specifies what fields are shown, field labels if not the same as the ones in the data source and possible value formatting (e.g. how coordinates are shown). (see Figure 4)

← → ↻ ⓘ localhost:8080/traverse/building/746103

Start Location >> Building (Id: 746103) >>

**PLYWOOD (ID: P123)** [Show origins in map](#)

**Entity properties**

Property	Value
Producer	UPM Plywood

**Parts and components**

Type	Id	Coordinates
tree-trunk	455829	{"lng":30.15397461,"lat":62.4168632}
tree-trunk	455874	{"lng":30.15477914,"lat":62.41683654}

Figure 4 Plywood asset details



4. **Map visualization.** Map visualization is used as a default visualization for tree trunks and when showing plywood origins in map (see Figure 5).

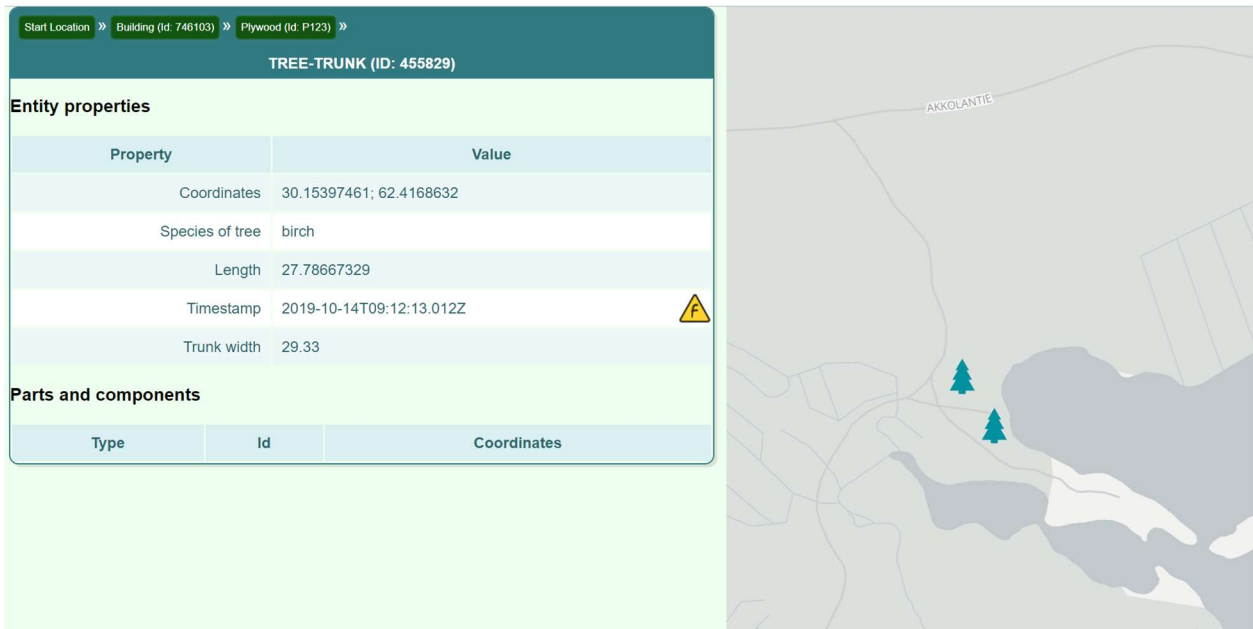


Figure 5 Tree trunk details and map visualization. It's possible to show all tree trunks related to a sheet of plywood in map by clicking Show origin in map from plywood details.

Use cases

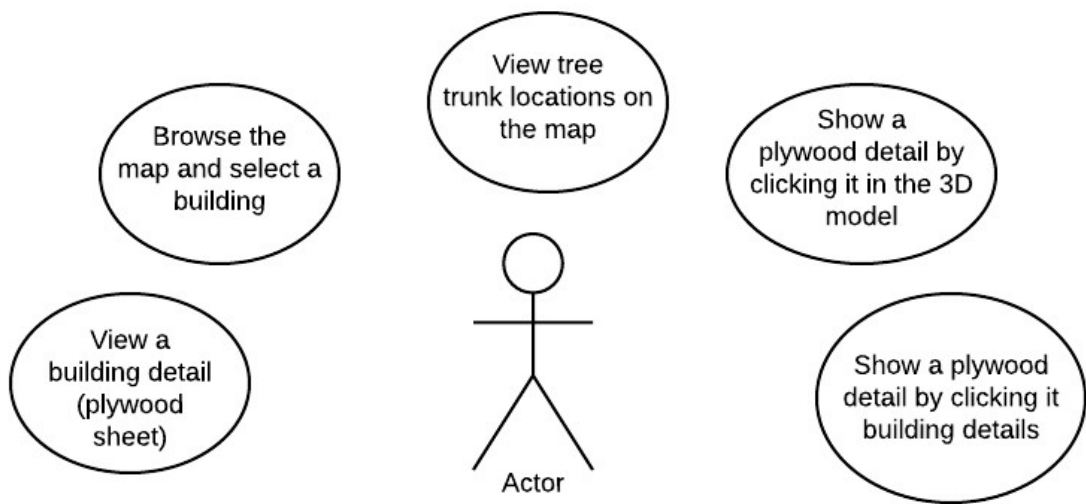


Figure 6 Use cases implemented in the PoC

Frontend technical stack

Following table summarize key technologies

TECHNOLOGY	DESCRIPTION
<b>NODE.JS</b>	Node.js is used to host the application and as a way to bundle the SPA solution.
<b>TYPESCRIPT</b>	Frontend code is written in TypeScript and compiled into JavaScript
<b>WEBPACK</b>	Webpack is used as a tool to manage build and for hosting during development.
<b>CYCLE.JS</b>	Cycle.js is light-weight UI framework. It is similar to React.
<b>MAPBOX.JS</b>	Map visualization library
<b>THREE.JS</b>	3D visualization library


See frontend/readme.md in git repository for further details.

## Query service

Query service has currently two purposes

- 1) It aggregates data from multiple data sources. (See Data query and aggregation below).
- 2) It provides an administrative interface for initialization of the test dataset.

In addition, query service provides OpenApi 2 (Swagger) API documentation for the available interfaces and generates Swagger client that can be used to test interfaces from browser. For more detailed description see generated OpenApi 2 API documentation available in path <query-servier-baseurl>/api/api-docs/. For running the query service locally, documentation is available in <http://localhost:3000/api/api-docs/>


**swagger**

<http://localhost:3000/api/swagger.json>

**Explore**

## GoWood backend APIs

**default**
[Show/Hide](#)
[List Operations](#)
[Expand Operations](#)

**GET** /api/ping

**query-api**
[Show/Hide](#)
[List Operations](#)
[Expand Operations](#)

**POST** /api/query/{operation}
 [Run named query starting from a node](#)

**Parameters**

Parameter	Value	Description	Parameter Type	Data Type
operation	info-with-first-level-components		path	string
query-service.routes.services/operations-body	(required) <div></div> Parameter content type: application/json		body	Model Example Value <pre>{   "from": {     "type": "string",     "id": "string"   } }</pre>

Try it out!

**data**
[Show/Hide](#)
[List Operations](#)
[Expand Operations](#)

**POC admin**
[Show/Hide](#)
[List Operations](#)
[Expand Operations](#)

**DELETE** /api/db/janus-graph
 [Delete all data from data base](#)

**POST** /api/db/janus-graph
 [Create new database](#)

[ BASE URL: ]

Figure 7 OpenAPI 2 (Swagger) API documentation

### Data query and aggregation

The only API request that is done by the client from browser is `/api/query/info-with-first-level-components`

Info-with-first-level-component gets root node id and type as parameters and then request info from Application database or Holochain depending on the type. If there are assets of type *holochain-link*, the query service fetches data for each link from Holochain.

The following describes querying *info-with-first-level-components* in a building query

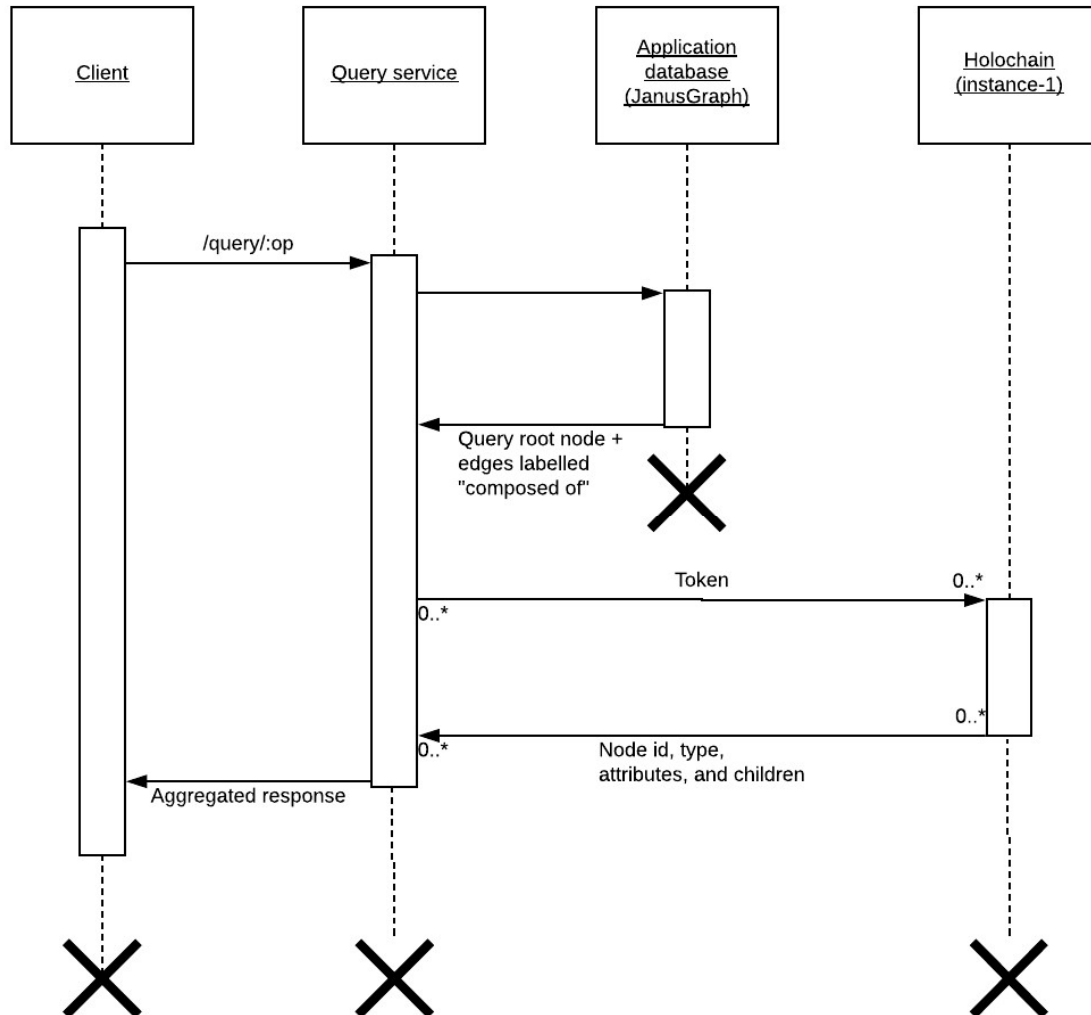


Figure 8 Data query aggregation logic

If the root asset of the query is a link to Holochain, also the root node and it's first level components are fetched from Holochain, but otherwise the logic is the same. See Data model for more details on this.

Current version of query logic is incomplete in two ways:

- 1) Query aggregation logic is incomplete: If a new data source is added, changes are needed in both the query aggregator and possibly the other data source. This is not a problem as long as there are only two data sources and the existing mock data source is intended to make UI development and testing easier.
- 2) Types should have a namespace, but namespaces are not implemented. Hence Asset types are used in a slightly inconsistent way. This is not a problem as there are only 4 types in the data model.

Query service is designed work in such a way that there can be arbitrary number of diverse data sources as long as each data source data can be mapped to the consistent format. Current version works nicely for two different type data source types, but once a third data source is added, the query aggregation logic should be completed and generalized properly.

### Query Service's Technical Stack

Following table summarize key technologies

TECHNOLOGY	DESCRIPTION
<b>JETTY</b>	Application server used to host the application.
<b>MOUNT</b>	Mount is used for configuration management, and as a tooling to make development more convenient.
<b>CLOJURE</b>	Query service is written in Clojure.
<b>LEININGEN</b>	The de-facto build tool for Clojure. Builds and runs the service.
<b>REITIT (+ MALLI)</b>	Rest API is implemented using Reitit library. Reitit autogenerates API documentation. REST API Request and request schemas are described using Malli.
<b>OGRE</b>	Ogre is a Clojure API to Apache Tinkerpop compatible databases; in this case JanusGraph.

See query-service/readme.md in git repository for further details.

### Data model

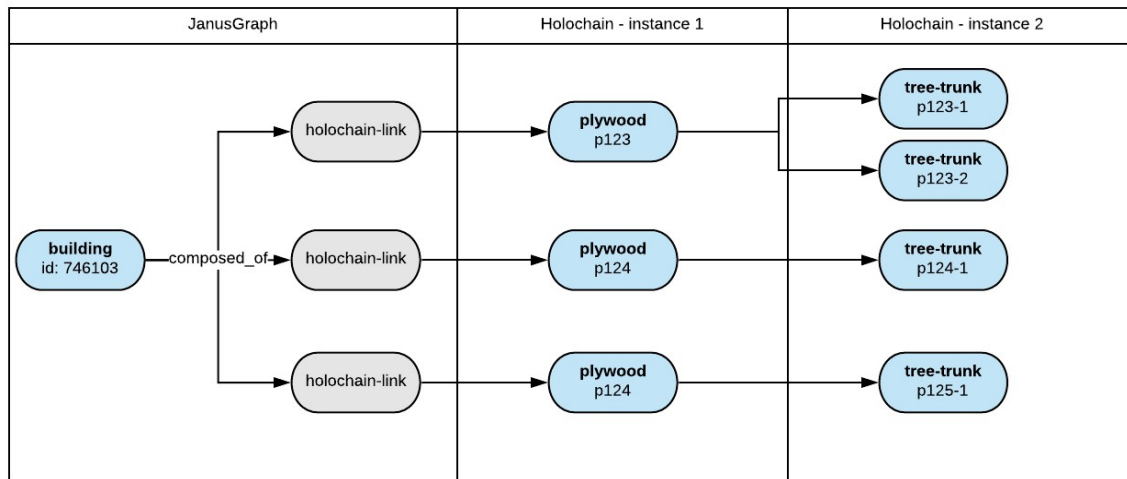
When using test data, data is distributed and queried from three data sources (of two different types).

#### Test data description

Test data contains 7 records and links between them. Technically the links between JanusGraph and Holochain are records (nodes) in JanusGraph, but these records as never exposed to the user.

Data in JanusGraph is persisted to BerkleyDB SQL database that can be changed e.g. to Casandra. Data in Holochain is persisted as text files in the file system. Bridge between Holochain and a legacy database was not included into PoC.

The following diagram illustrates where the data is stored:



### Used data sample and data mocking

Data related to tree trunks resembles real data from a tree harvester. Also the building id is a real building id from public building data. Plywood records and links between entities are mocked.

Following table summarizes what data was available for PoC:

Data source type	Schemas	Data
<b>Wood harvester data</b>	Provided – two standards available	Limited sample provided
<b>Sawmill data</b>	Not available	Not available
<b>Building information model &amp; map data</b>	Limited to map data	Limited; Used Open data map data form <a href="https://kartta.hel.fi/ws/geoserver/avoidata/wfs">https://kartta.hel.fi/ws/geoserver/avoidata/wfs</a>

Data from tree harvester was limited. Timestamps were removed because they could be used to calculate how productive the harvester company have been and this might have negative business impact if available for the other stakeholders in the supply chain.

Data from a sawmill was not gotten probably because it was hard to estimate risks related to delivering data for this kind of purpose. If too much data is exposed to other stakeholders, they could – at least in theory – use it to calculate inventory size, changes in the order base and problems in production. Ironically, a goal of the PoC was to explore ways to ensure that the shared data can be exposed in such a way that it's other stakeholders cannot exploit the data and utilize it to their competitive advantage.

As the data sample from sawmill was not available, building information model data was not even requested: it would not have delivered much additional value within the scope of the PoC.

### Critical points when linking data

The following table summarizes expected challenges and possible solutions to those data linking problems we are aware of:

Link	Description
------	-------------

<b>Tree trunk -&gt; sawmill tree trunk inventor</b>	Currently, tree harvester produces detailed data for each cut of a tree trunk. In modern sawmills also each tree trunk is imaged in order to optimize the use of the material.
	However, currently the record from harvester and sawmill optimization system are not connected. A possible solution that is already experimented by a commercial party is to use image cutting surface as a fingerprint that identifies the tree.
<b>Tree trunk inventory -&gt; wood block inventory</b>	There is a work phase in modern sawmills for optimizing the utilization of each tree trunk. This is done by scanning the tree trunk and using the resulting 3D model and other data provided by sensors.
	After wood utilization optimization, the tree trunk is cut into blocks (or slices) that can be used in the final product. It should be possible to map each tree to a produced block by using optimization data. As far as we know, the data about the unique wood blocks (including slices) is not saved.
<b>Wood block inventory -&gt; wood product inventory</b>	A sheet of plywood (wood product) is composed of multiple slices (wood block) that are glued together. Currently these slices do not have unique identity (see the previous row in the table). Therefore, the wood blocks do not have unique identity, and information chain is broken here. Even if we knew which wood block came from which tree trunk, we wouldn't know which wood block ended in which final product (eg. Plywood sheet)
	<p>There are two obvious ways, how the missing linking could be added to the process:</p> <ol style="list-style-type: none"> <li>1) Add an identifier to each block of wood (e.g. using a QR code) and map it to optimization data so that optimization data can be used to map each produced wood block to a tree trunk.</li> <li>2) Deduce statistically which tree blocks are used in which product (I.e. accept errors that do not have significant impact to carbon accounting)</li> </ol>
	There might be other ways, but they are not obvious without knowing many other details, e.g. how the wood utilization optimization is done and how the data is stored in data storages, and how the sawmill ERP system is implemented.
<b>Wood product inventory -&gt; building database</b>	Each wood product probably has unique identity in sawmill inventory system, and if they don't have it, it should be possible to add it. It's also possible to add a QR code to each plywood sheet. For building company it's possible to link data to building information model e.g. by using QR code. We don't know if this is done right now.

The links mentioned in the table above are needed so that it's possible even in theory to track a sheet of plywood used in a building back to the tree trunk that was cut in a forest. However, there are a number of other links and nodes that should be added to the model so that it would be possible to calculate accurate carbon footprint values, for instance: transportation, and storage in retailer's warehouse. The goal of this PoC was to study what kind of data is needed to track raw material and to do carbon accounting. Adding more nodes and edges to the graph data model does not significantly change the

way to traverse through it. Even the current, limited PoC implementation supports breadth-first graph traversal in an arbitrarily deep, non-cycle graph.

### On exposing and querying a distributed graph data model

While the current solution supports already graph traversal to multiple distributed data sources, it provides row level access control without need to authenticate and manage authorization in data source. Row level access control is a rather common feature in many database systems. However row level access control requires authentication against an internal identity and access management (IAM) system. Configuring access for a client is properly set in multiple private IAMs which is hard and error prone and not feasible in practice.

In this PoC, access to records plywood and tree trunk records is granted using “permissionless cryptographically authorized” approach. If a user has Hololink link token to a record, he has access to it and all related tokens in other data sources, but not any other records (see Figure 9).

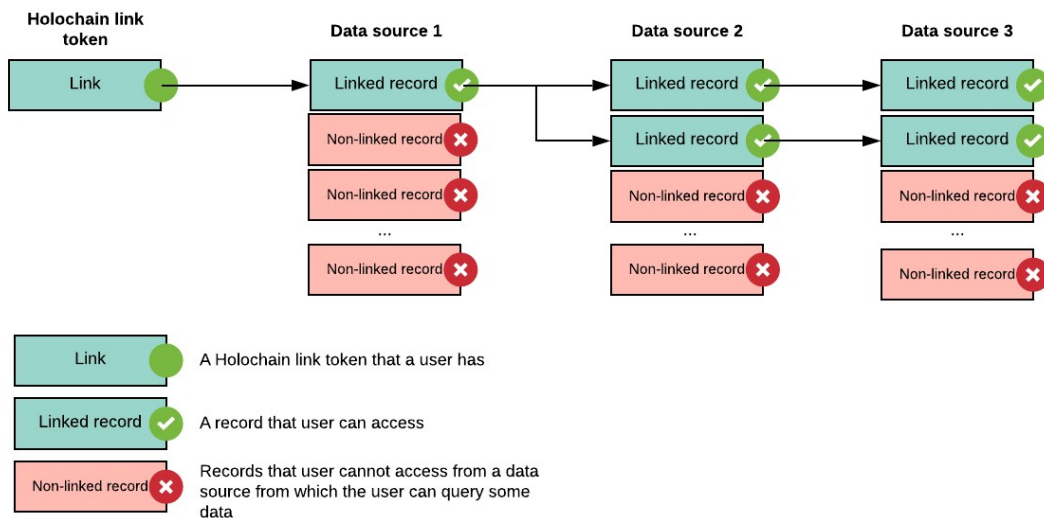


Figure 9 Holochain link token grants row level access to a record it refers to and to all the linked records in other data sources

See chapter Holochain data storage for more details description how this is implemented in PoC.

### On approaches to do computation over distributed data sources

Another important aspect that differentiates this PoC from commonly used approaches relates to where data is located for computations and what data a stockholder needs to share before it's possible to do computations.

The easiest way to understand the difference is to compare how computation over multi-stakeholder dataset can be done.



To summarize, there are three substantially different approaches:

- 1) **Share all data or an extensive enough subset of the data to the party that does the computation.** Data from all stockholders collected to one place and linked before actual computation.

This approach is problematic, if stakeholders are unwilling to deliver data and they are not obligated to do so. It's also challenging and laborious to link records in diverse data sources. This is the traditional approach, and a goal of this PoC is to experiment a better approach. If this is the simplest and most convenient way for computations. In ideal case the data is stored in a graph database that can be easily traversed.

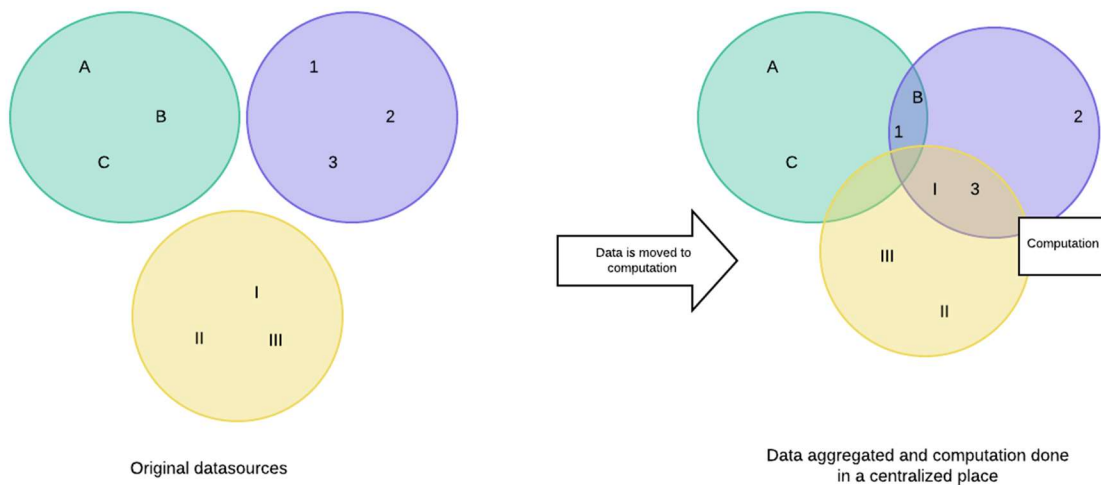


Figure 10 Traditional approach is to collect needed data to one place, and then run computation over it

- 2) **Share identifiers to the data. Share actual data if and only if the computation has an identifier stored in stakeholder's own private DLT database (i.e. permissionless row level access control).** Collect unique identifiers of records that are relevant for a computation in one place and then fetch only the data when doing calculation and only if a record is needed for it.

This is possible if each stakeholder in the system has a DLT -database that contains all identifiers that are relevant to a computation (e.g. carbon accounting) and the party who is running the computation has access to the root node where the computation starts. Only thing that is shared is a string identifier (in PoC "Holochain link token") that does not contain any useful data about the entity it is related to, and that can be used to fetch record and the links to the other record.

The PoC explores this approach.

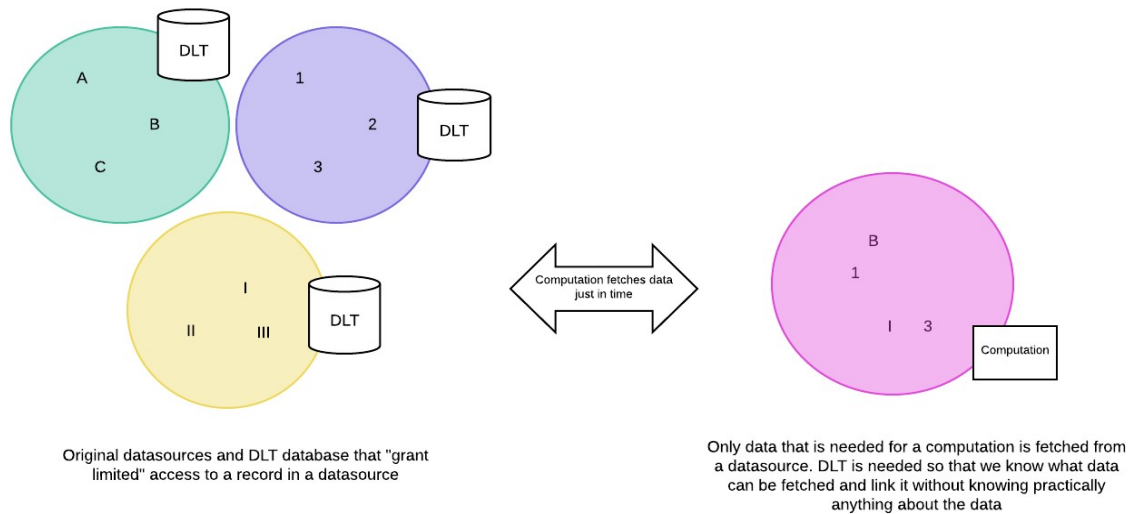


Figure 11 Only identifier to the data is shared. DLT is used for permissionless row level access control.

- 3) **have only results of computations and link to entities that the stakeholder doesn't have access.** Instead of trying to get data to computation move computation to data.

While this allows companies to share partially no data at all, the solution is also complex, and we are not aware if anyone has implemented anything like this. This option is mentioned, because Holochain probably makes this approach possible. This option extends the option 2, and if that option had not worked technically, then this option also would not work: In this solution also, it's necessary to share some identities so that records located in different data sources can be linked. As in option 2 the links alone do not contain any useful data in addition to the fact that some data exists and some of these identities are linked to some other identity.

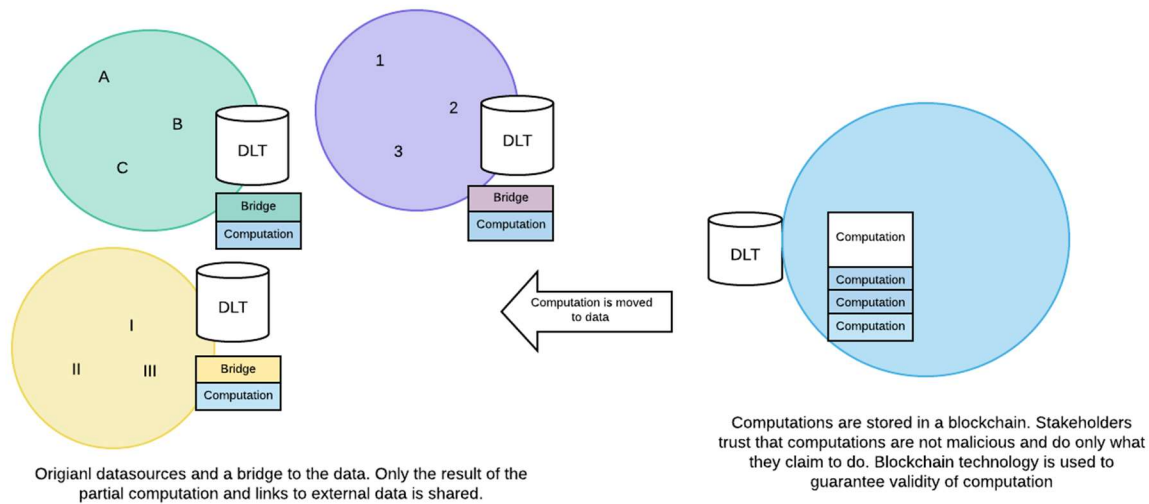


Figure 12 If it's not possible to get data to the place where computation is done, then we might use DLT and blockchain technology to move computation to the places where data is and then aggregate the results of multiple nodes in an orchestrating node outside data

### Holochain data storage

Within the scope of the PoC, plywood and tree trunk records are stored in Holochain instances as private records. This means that they can be directly accessed only by the Holochain instance itself. After a record is added to a Holochain instance, GoWood Holochain function generates a link token, that can be used to access the record.

After this, the data provider distributes the token to other stakeholders. There are many possible ways to distribute a token, for instance:

- 1) Attach it as a QR to a plywood sheet as part of a packaging process in a plywood factory
- 2) Attach a token to a tree trunk cutting surface fingerprint. Distribute the token to any actor who has a tree trunk that matches the cutting surface fingerprint.
- 3) Associate any other QR code or barcode to the token. Distribute the token to any actor who has a code associated to the token. E.g. in transport industry barcodes and QR codes are used widely to track deliveries.

Token distribution is not included in the Proof of Concept.

### How token is generated?

Token generation is done by the function ***create\_signed\_token\_from\_value***. This function takes the following parameters:

- Internal id
- The type of the record
- The record itself
- Links to other external or internal records upstream.

The record is committed to private DLT storage of the Holochain instance. If data is committed successfully, Holochain returns an address to it. After this a Holochain link token is created. Token format is:

<Holochain agent address>.<encrypted address of data>.<signature of first two parts>

Data is encrypted using agent's private key and a symmetric encryption algorithm. Agent address and encrypted key is signed using agent's private key as well. The token format validation logic is similar but shorter than the one in JSON Web Tokens (JWT) but shorter.

#### How row level access is granted?

Token validation and data retrieval is done by the function ***get\_value\_from\_signed\_token***. If a token is in a correct format (i.e. token string has three parts separated by '.'), the called (=target or callee) agent validates the signature using the public key of the agent that generated the token.

It is possible that agent that receives ***get\_value\_from\_signed\_token*** call is the same as the agent that generated the token in the first place. If this was the case, the agent decrypts the address using agent's own private key, fetches data from the private DLT store and returns it to the caller. If not, agent sends token as a message to the agent that generated the token and proxies the response the caller. If an agent calls another agent with token, the token validation data retrieval occurs in similar way: i.e. first token is validated, then the key is decrypted and finally data is fetched from the private DLT storage.

#### Short introduction to blockchain technology, smart contracts and Holochain for a non-technical reader

The fundamental breakthrough of blockchain, is how to authorize a transaction without a centralized authority. For instance, how to authorize that some money is transferred from A to B without a bank in between? The solution is two-fold: First, you need to have a way to *guarantee identity* of A and B. This is done by using a cryptographic key pair (= a shared *public key* and a personal *private key*) and cryptographic signing. Second you need to *validate that a transaction can be done*. The solution to this, is that you "know" all previous transactions (but not necessarily the content of them) that A and B have done since they were added to the system, i.e. since the cryptographic key pair that identify them was created.

In case of transferring money, you essentially can check that there are only valid financial transactions in the transaction history and after them all A can or cannot transfer money to B. The logic needed to validate history of financial transaction is just *a (smart) contract*. Within the scope of this document it's not relevant how the integrity of the chain of transaction (i.e. blockchain) is guaranteed. The transaction history is stored in a **distributed ledger** (DLT database). What is essential is to understand that it's almost impossible to add a fake transaction in the transaction history, or to change the state of A after the last transaction, if the smart contract has been properly implemented and validates a transaction properly before committing it to the chain. A smart contract can be any set of computations that may mutate its state and should validate all calls.

There are many blockchain frameworks, and blockchain as a technical term refers to a family of similar but mutually incompatible technologies. For instance, some early blockchain solutions (such as

Bitcoin) implemented just one contract and usually such a hard-coded contract is not called “smart”. In platforms such as Ethereum and Holochain, they are fully programmable.

In case of Holochain, the implementation of a *smart contract* is called **DNA** and a running instance of it is an **agent**. In Holochain documentation, agent refers sometimes also to a cryptographic identity of an actor. It is usually clear which agent is referred, but for the sake of clarity, in this document agent always refers to *running instance of a DNA*, and *identity of agent* (cryptographic key pair) is called **agent address**. A same DNA can be used in many agents but they each have a different state – this is similar to the fact that you can have same kind of currency (~DNA) on many wallets (~Agent). Agents can interact with each other – even with agents that use instances of different DNA.

In addition to the technical differences among blockchain technologies, there are even bigger non-technical ones related to *how trust is created in the social sphere*. In pre-blockchain technologies, trust based on centralized authorities - such as banks - that are supervised by the governmental legal authorities. In blockchain frameworks there is no such central authority. The oldest and currently the most popular method for creating trust is **Proof of Work (PoW)**. Proof of Work mean essentially that you need to perform some resource-intensive computations in order to add a transaction to a blockchain and this costs money and time. If you want to try to counterfeit, it costs a lot more money and time, and hence cheating is not a feasible option. The drawback of this approach is that PoW requires a lot of resources in the form of electricity and machines to do the computations.

Other well-known approaches are **Proof of Stake (PoS)** and **Proof of Authority (PoA)**. In the former, there is substantial financial bid that you can lose if another node suspects that you try to cheat. In PoA, you put your reputation in stake. Unlike in case of pre-blockchain technologies, less manual work is required to guarantee accountability of the actors of the system: if you try cheat in PoA, other nodes in the system notice most likely that something is wrong, and you are caught by an automatic mechanism in the system. In pre-blockchain approach laborious manual audits are required in many cases.

Blockchain technologies do not completely remove the need for audits and other manual legal activities. In all blockchain systems, there are social incentives (e.g. money) to play nicely or social penalties (e.g. loss of money) if you cheat. Rather, the increased transparency and automation makes *the legal activities that are required to counter unacceptable (harmful) behaviour* less error prone and less laborious, and in many cases unnecessary. For instance, in case of cryptocurrencies you don't need an army of lawyers and governmental authorities to guarantee that a bank won't default your bank account and claim that you never had one – for a bank that's technically trivial and hence, in the end, a reliable governmental legal system is the only way to guarantee that the banks won't exploit the power they have. Yet, it's possible that the cryptocurrency loses its value or that the whole currency was not a cryptocurrency, and someone just fooled you to believe so.

In systems implemented with Holochain, one can create trust by PoS or PoA mechanisms or with some other not-yet known mechanism. Holochain is a meta-blockchain framework: it gives you the freedom to choose how trust is created.

### Limitations and further development ideas

In this PoC records are stored in the DLT as a file. While this might be sufficient, it's probably not an optimal solution in many situations. For instance, current file system solutions do not scale nicely if

there are hundreds of gigabytes of data or when there are a large number of concurrent requests to a Holochain node. Also, when data is replicated, data privacy and security might require quite a bit of additional work in some cases.

An option to solve data storage issues is to just store data needed to fetch record from a database and possible external links from the record to the private DLT store. When data is requested, the request is bridged from Holochain instance to an external data query adapter that queries data from the database where the record is located.

An external adapter component is needed for two reasons:

- 1) Holochain DNA is compiled into WebAssembly. The only way to access data outside WebAssembly is to create an external function and an extern in WebAssembly source code that calls the external function.
- 2) Holochain DNA is shared by all instances that use it and hence dependencies to private datastore must be inverted in some way. Use of command pattern and with adapter pattern is a common way to solve this kind of design problems.

## Deployment considerations

PoC is designed so that it can be easily deployed to on-premises data centre or to a cloud platform. PoC contain scripts needed to start a fully virtualized environment on one physical machine using Docker Compose.

## AWS architecture example

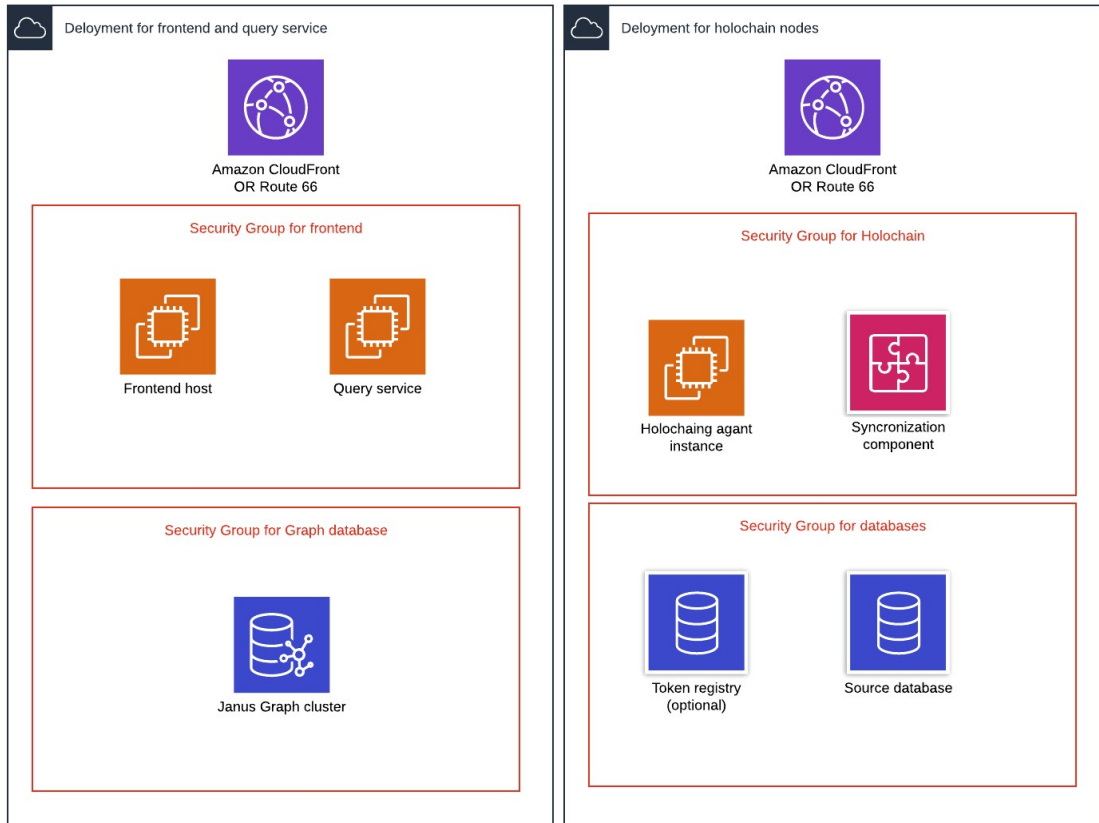


Figure 13 Possible architecture (simplified version)

Notes:

- Holochain nodes should be deployed separately from other logic. A Holochain node should be deployed into a location where it can access the source database. In addition to Holochain node and source database there should be a synchronization component that creates Holochain links for each record in the source database and distributes them to those stakeholders that should have access to the data in an appropriate manner.
- Probably, in real life use cases it's necessary to do some book keeping on created and external Holochain link tokens, but in the simplest use cases this is not necessary.
- AWS is used as reference platform for the service, but similar configuration should work in on-premises solution.
- Frontend and query service could be deployed into a separate security group and possibly even completely different physical location. In PoC implementation frontend proxy forwards requests to Query service, but in production version this should be refactored so that CloudFront takes care of this. In overall, security configurations and networks segmentation vary by case, and the figure above should be taken just as an example.

## Architecture in global scale

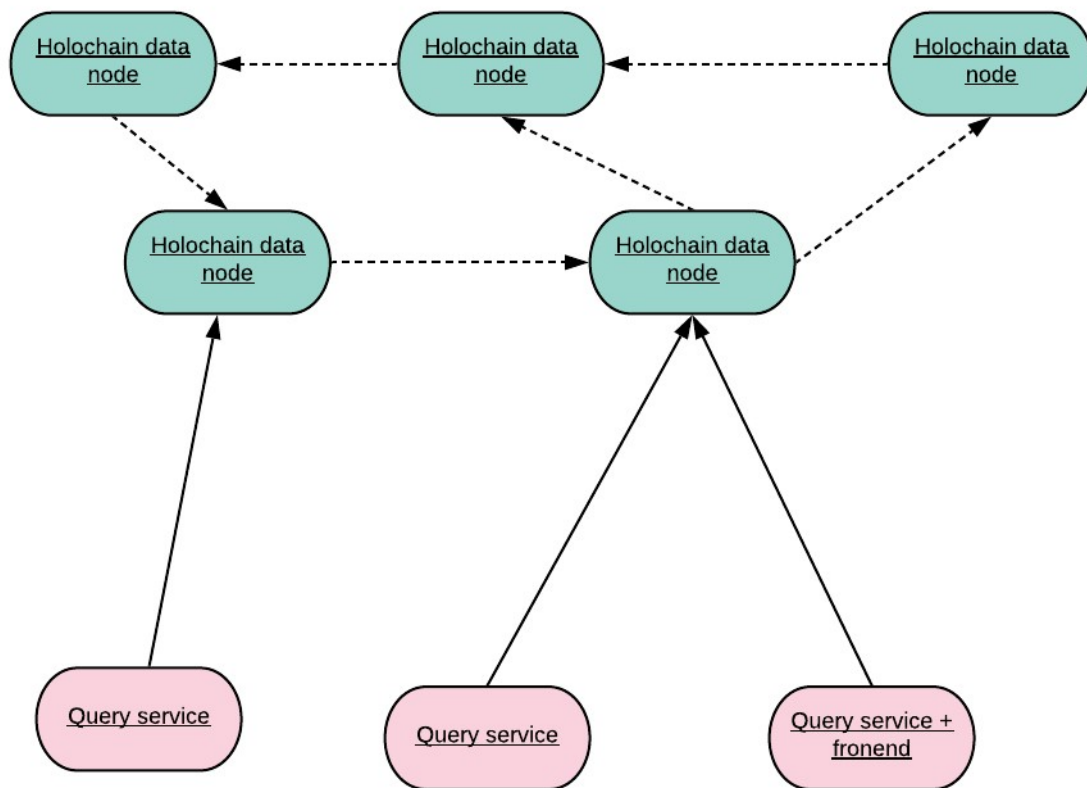


Figure 14 Overall architecture when there are multiple Holochain nodes and multiple client applications or public query interfaces.

When a Holochain node is connected to the network you need to reference at least one other node if it was not the first node. Holochain uses Gossip protocol to connect nodes running the same DNA into a network. Query nodes can try to query data from any Holochain node. Holochain routes requests to a correct node.