Technical White Paper

Public version (short version)

Simone Orso
Karat Lab ltd.
May 14, 2018

Contents

Background 2
EOS technology 2
Design Philosophy 2
Consensus Mechanism & Governance 3
Scalability 3
Denial-of-Service Attacks 3
Economics of the Network 4
Delegated proof of stake (DPoS) 4
The Bancor Integration 5
Code Snippets 5
EOS Blockchain Token 5
EOS Blockchain Token ABI 7
Identity 9
Proxy 12
Test API 14
Exchange 15
Bancor 18
Bancor ABI 19
The DApp 20
ERC-20 Token 20
Methods 21
Events 22
ICO (Initial Coin Offering) 23
Code Snippet 23
Key Processes 26
Gold Registration 28
Gold Minting (KCG creation) 29
Gold Asset Audit 30

Certificate Registration 31
Certificate Registration with KCG 32
Certificate Recast 33
Withdrawal 34
Redemption 35
Smart Contracts Stack Map 36

Background

EOS technology

The blockchain of Karatcoin will be developed on top of EOS technology, a software developed in C++ compiled with Clang that introduces a blockchain architecture designed to enable vertical and horizontal scaling of decentralized applications. This is achieved through an operating system-like construct upon which applications can be built. The software provides accounts, authentication, databases, asynchronous communication and the scheduling of applications across multiple CPU cores and/or clusters. The resulting technology is a blockchain architecture that has the potential to scale to millions of transactions per second, eliminates user fees and allows for quick and easy deployment of decentralized applications

Design Philosophy

- Provide all of the cryptography and app/blockchain communication functions to allow developers to focus on their business-specific logic functions
- Generalized role-based permissions
- Web toolkit for interface development
- Self-describing interfaces
- Self-describing database schemes
- Declarative permission scheme

Consensus Mechanism & Governance

- Delegated Proof of Stake
- Mechanism to freeze and fix broken or frozen applications
- A legally binding constitution establishes a common jurisdiction for disputes
- EOS will also include self-funded community benefit apps selected by vote
- No risk of fork spawning multiple chains

Scalability

- Single-threaded performance of 10,000-100,000 transactions per second
- Parallelization will scale the network to millions of transactions per second
- Supports thousands of commercial scale decentralized applications
- Asynchronous communications
- Separates authentication from execution
- Does not require counting operations

Denial-of-Service Attacks

- The ownership of EOS tokens gives users a proportional stake in the network bandwidth, storage, and computing power
- Spammers can only consume the proportion of the network that their EOS tokens entitle them to
- Denial-of-service attacks on a given app cannot disrupt the entire network
- Startups with a very small stake in the network will have guaranteed, reliable bandwidth and computational power

Economics of the Network

- Ownership model
- Owning EOS tokens gives a proportional share in network bandwidth, storage, and processing power
- Reliable, predictable network bandwidth and computing power for small businesses
- Relatively small investment for minimum bandwidth and computing power
- Zero transaction fees, no cost for developers except the initial EOS tokens

Delegated proof of stake (DPoS)

A variation of proof of stake in which the responsibility for creating blocks is delegated to third party nodes, known as "witnesses."

Delegated Proof of Stake (DPOS) is the fastest, most efficient, most decentralized, and most flexible consensus model available.

- DPOS leverages the power of stakeholder approval voting to resolve consensus issues in a fair and democratic way.
- All network parameters, from fee schedules to block intervals and transaction sizes, can be tuned via elected delegates.
- Deterministic selection of block producers allows transactions to be confirmed in an average of just 1 second.
- The protocol is designed to protect all participants against unwanted regulatory interference.
- DPoS also allows for reliably confirmed transactions every 10 seconds. With Bitcoin, this takes six confirmations at average time of 10 minutes each.
- It works by using reputation systems and frictionless, real-time voting to create a panel of limited trusted parties. These parties then have the right to create

blocks to add to the Blockchain and prohibit un-trusted parties from participating.

- The panel of trusted parties take turns creating blocks in a randomly assigned order that changes with each iteration.

The Bancor Integration

On our blockchain, we will integrate the Bancor Protocol that will help make our ecosystem even stronger, as it will bring us one step closer to our vision of a truly global and inclusive P2P lending platform.

- Bancor Protocol is standard for the creation of Smart Tokens, cryptocurrencies with built-in convertibility linked directly to their smart contracts.
- Bancor utilizes an innovative token “connector” method to enable formulaic price calculation and continuous liquidity for all compliant tokens, without needing to match two parties in an exchange.

Smart Tokens interconnect to form token liquidity networks, allowing user-generated cryptocurrencies to thrive.

Code Snippets

EOS Blockchain Token

Creation

...

```
void token::create( account_name issuer,
                    asset        maximum_supply,
                    uint8_t      issuer_can_freeze,
                    uint8_t      issuer_can_recall,
                    uint8_t      issuer_can_whitelist )
{
    require_auth( _self );


    auto sym = maximum_supply.symbol;
    eosio_assert( sym.is_valid(), "invalid symbol name" );
```

```
    eosio_assert( maximum_supply.is_valid(), "invalid supply");
    eosio_assert( maximum_supply.amount > 0, "max-supply must be positive");


    stats statstable( _self, sym.name() );
    auto existing = statstable.find( sym.name() );
    eosio_assert( existing == statstable.end(), "token with symbol already exists" );


    statstable.emplace( _self, [&]( auto& s ) {
       s.supply.symbol = maximum_supply.symbol;
       s.max_supply    = maximum_supply;
       s.issuer        = issuer;
       s.can_freeze    = issuer_can_freeze;
       s.can_recall    = issuer_can_recall;
       s.can_whitelist = issuer_can_whitelist;
    });
 }
...
```

Token Transfer

```
...

 void token::transfer( account_name from,
                       account_name to,
                       asset        quantity,
                       string       /*memo*/ )
 {
    print( "transfer from ", eosio::name{from}, " to ", eosio::name{to}, " ", quantity,
 "\n" );
    eosio_assert( from != to, "cannot transfer to self" );
    require_auth( from );
    eosio_assert( is_account( to ), "to account does not exist");
    auto sym = quantity.symbol.name();
    stats statstable( _self, sym );
    const auto& st = statstable.get( sym );


    require_recipient( from );
    require_recipient( to );


    eosio_assert( quantity.is_valid(), "invalid quantity" );
    eosio_assert( quantity.amount > 0, "must transfer positive quantity" );
    eosio_assert( quantity.symbol == st.supply.symbol, "symbol precision mismatch" );
```

```
      sub_balance( from, quantity, st );
      add_balance( to, quantity, st, from );
  }
...
```

EOS Blockchain Token ABI

Use of JSON ABI data transfer for EOS Token.

```
{
          "types": [],
          "structs": [{
              "name": "transfer",
              "base": "",
              "fields": [
                {"name":"from", "type":"account_name"},
                {"name":"to", "type":"account_name"},
                {"name":"quantity", "type":"asset"},
                {"name":"memo", "type":"string"}
              ]
            },{
             "name": "create",
             "base": "",
             "fields": [
                {"name":"issuer", "type":"account_name"},
                {"name":"maximum_supply", "type":"asset"},
                {"name":"can_freeze", "type":"uint8"},
                {"name":"can_recall", "type":"uint8"},
                {"name":"can_whitelist", "type":"uint8"}
             ]
          },{
             "name": "issue",
             "base": "",
             "fields": [
                {"name":"to", "type":"account_name"},
                {"name":"quantity", "type":"asset"},
                {"name":"memo", "type":"string"}
             ]
          },{
              "name": "account",
```

```
      "base": "",
      "fields": [
        {"name":"balance", "type":"asset"},
        {"name":"frozen", "type":"uint8"},
        {"name":"whitelist", "type":"uint8"}
      ]
    },{
      "name": "currency_stats",
      "base": "",
      "fields": [
        {"name":"supply", "type":"asset"},
        {"name":"max_supply", "type":"asset"},
        {"name":"issuer", "type":"account_name"},
        {"name":"can_freeze", "type":"uint8"},
        {"name":"can_recall", "type":"uint8"},
        {"name":"can_whitelist", "type":"uint8"},
        {"name":"is_frozen", "type":"uint8"},
        {"name":"enforce_whitelist", "type":"uint8"}
      ]
    }
  ],
  "actions": [{
      "name": "transfer",
      "type": "transfer",
      "ricardian_contract": ""
    },{
      "name": "issue",
      "type": "issue",
      "ricardian_contract": ""
    }, {
      "name": "create",
      "type": "create",
      "ricardian_contract": ""
    }


  ],
  "tables": [{
      "name": "accounts",
      "type": "account",
      "index_type": "i64",
      "key_names" : ["currency"],
      "key_types" : ["uint64"]
    },{
      "name": "stat",
```

```
      "type": "currency_stats",
      "index_type": "i64",
      "key_names" : ["currency"],
      "key_types" : ["uint64"]
    }
  ],
  "ricardian_clauses": []
}
```

Identity

This contract maintains a graph database of certified statements about an identity. An identity is separated from the concept of an account because the mapping of identity to accounts is subject to community consensus.

Some use cases need a global source of trust, this trust rooted in the voter who selects block producers. A block producer's opinion is "trusted" and so is the opinion of anyone the block producer marks as "trusted".

When a block producer is voted out the implicit trust in every certification they made or those they trusted made is removed. All users are liable for making false certifications.

An account needs to claim the identity and a trusted account must certify the claim.

Data for an identity is stored:

DeployToAccount / identity / certs / [property, trusted, certifier] => value

Questions database is designed to answer:

1. has $identity.$unique been certified a "trusted" certifier

2. has $identity.$property been certified by $account

3. has $identity.$trusted been certified by a "trusted" certifier

4. what account has authority to speak on behalf of identity?

   - for each trusted owner certification check to see if the account has claimed it

5. what identity does account have authority to speak on behalf?

   - check what identity the account has self-certified owner

- verify that a trusted certifier has confirmed owner

This database structure enables parallel opeartions on independent identities.

...

```
   public:
          using identity_base::identity_base;


           void settrust( const account_name trustor, ///< the account authorizing the
    trust
                          const account_name trusting, ///< the account receiving the
    trust
                          const uint8_t      trust = 0 )/// 0 to remove, -1 to mark
    untrusted, 1 to mark trusted
           {
              require_auth( trustor );
              require_recipient( trusting );


              trust_table table( _self, trustor );
              auto itr = table.find(trusting);
              if( itr == table.end() && trust > 0 ) {
                 table.emplace( trustor, [&](trustrow& row) {
                       row.account = trusting;
                   });
              } else if( itr != table.end() && trust == 0 ) {
                 table.erase(itr);
              }
           }
```

...

This action creates a new globally unique 64-bit identifier, to minimize collisions each account is automatically assigned a 32-bit identity prefix based upon hash(account_name) ^ hash(tapos).

With this method no two accounts are likely to be assigned the same 32-bit prefix consistently due to the constantly changing tapos. This prevents abuse of 'creator' selection to generate intentional conflicts with other users.

The creator can determine the last 32-bits using an algorithm of their choice. We presume the creator's algorithm can avoid collisions with itself.

Even if two accounts get a collision in first 32-bits, a proper creator algorithm should generate randomness in last 32-bits that will minimize collisions. In event of collision transaction will fail and creator can try again.

A 64-bit identity is used because the key is used frequently and it makes for more efficient tables/scopes/etc.

…

```
void create( const account_name creator, const uint64_t identity ) {
            require_auth( creator );
            idents_table t( _self, _self );
            auto itr = t.find( identity );
            eosio_assert( itr == t.end(), "identity already exists" );
            eosio_assert( identity != 0, "identity=0 is not allowed" );
            t.emplace(creator, [&](identrow& i) {
                  i.identity = identity;
                  i.creator = creator;
               });
         }


         void certprop( const account_name bill_storage_to, ///< account which is paying
for storage
                              const account_name       certifier,
                              const identity_name      identity,
                              const vector<certvalue>& values )
         {
            require_auth( certifier );
            if( bill_storage_to != certifier )
               require_auth( bill_storage_to );


            idents_table t( _self, _self );
            eosio_assert( t.find( identity ) != t.end(), "identity does not exist" );


            /// the table exists in the scope of the identity
            certs_table certs( _self, identity );
            bool trusted = is_trusted( certifier );
…

if (itr != idx.end() && itr->property == value.property &&
itr->trusted == trusted && itr->certifier == certifier) {
                     idx.modify(itr, 0, [&](certrow& row) {
```

```
                row.confidence = value.confidence;
                row.type       = value.type;
                row.data       = value.data;
            });
        } else {
            auto pk = certs.available_primary_key();
            certs.emplace(_self, [&](certrow& row) {
                row.id = pk;
                row.property   = value.property;
                row.trusted    = trusted;
                row.certifier  = certifier;
                row.confidence = value.confidence;
                row.type       = value.type;
                row.data       = value.data;
            });
        }
```

...

Proxy

Proxy contract get and store configurations.

...

```
bool get(config &out, const account_name &self) {
          auto it = db_find_i64(self, self, N(config), config::key);
          if (it != -1) {
             auto size = db_get_i64(it, (char*)&out, sizeof(config));
             eosio_assert(size == sizeof(config), "Wrong record size");
             return true;
          } else {
             return false;
          }
       }


       void store(const config &in, const account_name &self) {
          auto it = db_find_i64(self, self, N(config), config::key);
          if (it != -1) {
             db_update_i64(it, self, (const char *)&in, sizeof(config));
          } else {
             db_store_i64(self, N(config), self, config::key, (const char *)&in,
sizeof(config));
          }
       }
```

...

Apply transfer and set owner functions.

...

```
  void apply_transfer(uint64_t receiver, account_name code, const T& transfer) {
        config code_config;
        const auto self = receiver;
        auto get_res = configs::get(code_config, self);
        eosio_assert(get_res, "Attempting to use unconfigured proxy");
        if (transfer.from == self) {
           eosio_assert(transfer.to == code_config.owner,  "proxy may only pay its owner"
  );
        } else {
           eosio_assert(transfer.to == self, "proxy is not involved in this transfer");
           T new_transfer = T(transfer);
           new_transfer.from = self;
           new_transfer.to = code_config.owner;


           auto id = code_config.next_id++;
           configs::store(code_config, self);


           transaction out;
           out.actions.emplace_back(permission_level{self, N(active)}, N(eosio.token),
  N(transfer), new_transfer);
           out.delay_sec = code_config.delay;
           out.send(id, self);
        }
     }


     void apply_setowner(uint64_t receiver, set_owner params) {
        const auto self = receiver;
        require_auth(params.owner);
        config code_config;
        configs::get(code_config, self);
        code_config.owner = params.owner;
        code_config.delay = params.delay;
        eosio::print("Setting owner to: ", name{params.owner}, " with delay: ",
  params.delay, "\n");
        configs::store(code_config, self);
     }
```

...

The apply method implements the dispatch of events to this contract

...

```
  void apply( uint64_t receiver, uint64_t code, uint64_t action ) {
        if( code == N(eosio) && action == N(onerror) ) {
           apply_onerror( receiver, onerror::from_current_action() );
        } else if( code == N(eosio.token) ) {
           if( action == N(transfer) ) {
              apply_transfer(receiver, code,
  unpack_action_data<eosio::token::transfer_args>());
           }
        } else if( code == receiver ) {
           if( action == N(setowner) ) {
              apply_setowner(receiver, unpack_action_data<set_owner>());
           }
        }
     }
```

...

Test API

Interaction with the EOS API, test different classes and functions.

...

```
  #include "test_action.cpp"
  #include "test_print.cpp"
  #include "test_types.cpp"
  #include "test_fixedpoint.cpp"
  #include "test_compiler_builtins.cpp"
  #include "test_crypto.cpp"
  #include "test_chain.cpp"
  #include "test_transaction.cpp"
  #include "test_checktime.cpp"
  #include "test_permission.cpp"
  #include "test_datastream.cpp"
```

...

Start handler tests.

...

```
  void apply( uint64_t receiver, uint64_t code, uint64_t action ) {
```

```
      if( code == N(eosio) && action == N(onerror) ) {
         auto error = eosio::onerror::from_current_action();
         eosio::print("onerror called\n");
         auto error_trx = error.unpack_sent_trx();
         auto error_action = error_trx.actions.at(0).name;

         // Error handlers for deferred transactions in these tests currently only
support the first action


         WASM_TEST_ERROR_HANDLER("test_action", "assert_false", test_transaction,
assert_false_error_handler );

         return;
      }
      if ( action == N(cf_action) ) {
         test_action::test_cf_action();
         return;
      }
      WASM_TEST_HANDLER(test_action, assert_true_cf);

      if (action != WASM_TEST_ACTION("test_transaction", "stateful_api") && action !=
WASM_TEST_ACTION("test_transaction", "context_free_api"))
         require_auth(code);
```

...

Exchange

Verification and initial validation of the withdrawal transaction, check quantity and sell amount.

...

```
void exchange::deposit( account_name from, extended_asset quantity ) {
      eosio_assert( quantity.is_valid(), "invalid quantity" );
      currency::inline_transfer( from, _this_contract, quantity, "deposit" );
      _accounts.adjust_balance( from, quantity, "deposit" );
   }

   void exchange::withdraw( account_name from, extended_asset quantity ) {
      require_auth( from );
      eosio_assert( quantity.is_valid(), "invalid quantity" );
```

```
      eosio_assert( quantity.amount >= 0, "cannot withdraw negative balance" ); //
Redundant? inline_transfer will fail if quantity is not positive.
      _accounts.adjust_balance( from, -quantity );
      currency::inline_transfer( _this_contract, from, quantity, "withdraw" );
   }

   void exchange::on( const trade& t ) {
      require_auth( t.seller );
      eosio_assert( t.sell.is_valid(), "invalid sell amount" );
      eosio_assert( t.sell.amount > 0, "sell amount must be positive" );
      eosio_assert( t.min_receive.is_valid(), "invalid min receive amount" );
      eosio_assert( t.min_receive.amount >= 0, "min receive amount cannot be negative"
);

      auto receive_symbol = t.min_receive.get_extended_symbol();
      eosio_assert( t.sell.get_extended_symbol() != receive_symbol, "invalid conversion"
);

      market_state market( _this_contract, t.market, _accounts );

      auto temp   = market.exstate;
      auto output = temp.convert( t.sell, receive_symbol );

      while( temp.requires_margin_call() ) {
         market.margin_call( receive_symbol );
         temp = market.exstate;
         output = temp.convert( t.sell, receive_symbol );
      }
      market.exstate = temp;

      print( name{t.seller}, "   ", t.sell, "  =>  ", output, "\n" );

      if( t.min_receive.amount != 0 ) {
         eosio_assert( t.min_receive.amount <= output.amount, "unable to fill" );
      }

      _accounts.adjust_balance( t.seller, -t.sell, "sold" );
      _accounts.adjust_balance( t.seller, output, "received" );

      if( market.exstate.supply.amount != market.initial_state().supply.amount ) {
         auto delta = market.exstate.supply - market.initial_state().supply;

         _excurrencies.issue_currency( { .to = _this_contract,
                                         .quantity = delta,
                                         .memo = string("") } );
```

```
        }
        /// TODO: if pending order start deferred trx to fill it
        market.save();
```

...

Create new exchange deposit, check initial supply, deposit and currencies.

...

```
 void exchange::createx( account_name    creator,
                  asset           initial_supply,
                  uint32_t        fee,
                  extended_asset  base_deposit,
                  extended_asset  quote_deposit
                ) {
        require_auth( creator );
        eosio_assert( initial_supply.is_valid(), "invalid initial supply" );
        eosio_assert( initial_supply.amount > 0, "initial supply must be positive" );
        eosio_assert( base_deposit.is_valid(), "invalid base deposit" );
        eosio_assert( base_deposit.amount > 0, "base deposit must be positive" );
        eosio_assert( quote_deposit.is_valid(), "invalid quote deposit" );
        eosio_assert( quote_deposit.amount > 0, "quote deposit must be positive" );
        eosio_assert( base_deposit.get_extended_symbol() !=
 quote_deposit.get_extended_symbol(),
                     "must exchange between two different currencies" );


        print( "base: ", base_deposit.get_extended_symbol() );
        print( "quote: ",quote_deposit.get_extended_symbol() );


        auto exchange_symbol = initial_supply.symbol.name();
        print( "marketid: ", exchange_symbol, " \n " );


        markets exstates( _this_contract, exchange_symbol );
        auto existing = exstates.find( exchange_symbol );


        eosio_assert( existing == exstates.end(), "market already exists" );
        exstates.emplace( creator, [&]( auto& s ) {
            s.manager = creator;
            s.supply  = extended_asset(initial_supply, _this_contract);
            s.base.balance = base_deposit;
            s.quote.balance = quote_deposit;
```

```
            s.base.peer_margin.total_lent.symbol          = base_deposit.symbol;
            s.base.peer_margin.total_lent.contract        = base_deposit.contract;
            s.base.peer_margin.total_lendable.symbol      = base_deposit.symbol;
            s.base.peer_margin.total_lendable.contract    = base_deposit.contract;


            s.quote.peer_margin.total_lent.symbol         = quote_deposit.symbol;
            s.quote.peer_margin.total_lent.contract       = quote_deposit.contract;
            s.quote.peer_margin.total_lendable.symbol     = quote_deposit.symbol;
            s.quote.peer_margin.total_lendable.contract   = quote_deposit.contract;
        });


        _excurrencies.create_currency( { .issuer = _this_contract,
                                    // TODO: After currency contract respects maximum
supply limits, the maximum supply here needs to be set appropriately.
                                   .maximum_supply = asset( 0, initial_supply.symbol ),
                                   .issuer_can_freeze = false,
                                   .issuer_can_whitelist = false,
                                   .issuer_can_recall = false } );


        _excurrencies.issue_currency( { .to = _this_contract,
                                        .quantity = initial_supply,
                                        .memo = string("initial exchange tokens") } );


        _accounts.adjust_balance( creator, extended_asset( initial_supply, _this_contract
), "new exchange issue" );
        _accounts.adjust_balance( creator, -base_deposit, "new exchange deposit" );
        _accounts.adjust_balance( creator, -quote_deposit, "new exchange deposit" );
    }
```

...

Bancor

Integration of the Bancor module within the blockchain

...

```
namespace bancor {
extern "C" {
```

```
    /// The apply method implements the dispatch of events to this contract
    void apply( uint64_t r, uint64_t c, uint64_t a ) {
       bancor::example_converter::apply( c, a );
    }
 }
 }
```

...

Bancor ABI

This is the structure in JSON format for ABI (application binary interface) of Bancor

```
 {
          "types": [{
              "new_type_name": "account_name",
              "type": "name"
            }
          ],
          "structs": [{
              "name": "transfer",
              "base": "",
              "fields": [
                {"name":"from", "type":"account_name"},
                {"name":"to", "type":"account_name"},
                {"name":"quantity", "type":"uint64"}
              ]
            },{
              "name": "account",
              "base": "",
              "fields": [
                {"name":"key", "type":"name"},
                {"name":"balance", "type":"uint64"}
              ]
            }
          ],
          "actions": [{
              "name": "transfer",
              "type": "transfer"
            }
          ],
          "tables": [{
              "name": "account",
              "type": "account",
```

```
      "index_type": "i64",
      "key_names" : ["key"],
      "key_types" : ["name"]
    }
  ]
}
```

The DApp

Karatcoin will create a decentralized gold market application by enabling established gold traders and providers to tokenize the same metal under one crypto-asset token, generating a highly liquid and stable market.

We will create a large, independent, decentralized and easily-scalable gold market not beholden to any central government organization, allowing multiple gold traders and providers to use a single protocol to tokenize gold for all their needs.

The Karatcoin ecosystem codebase will be released under the MIT License to ensure any network participant can extend and adapt the platform for its specific use cases.

The DApp consists of:

- Blockchain
- Wallets
- Smart-contracts
- Marketplace

There will be versions for desktop, Windows and Mac, and mobile, iOS and Android, written in ReactJS and React Native respectively.

The DApp will communicate directly with the Karat Blockchain and the Karatcoin server (for the interface) run by Dreammy on Amazon.

The server will have a 64-bit architecture on Linux and the database (which will be used for secondary data and cold storage) will be hosted on Amazon RDS, both will always guarantee maximum security and best performance using the autoscaling function.

ERC-20 Token

The following standard allows for the implementation of a standard API for tokens within smart contracts. This standard provides basic functionality to transfer tokens, as

well as allow tokens to be approved so they can be spent by another on-chain third party.

A standard interface allows any tokens on Ethereum to be re-used by other applications: from wallets to decentralized exchanges.

An ERC-20 token contract is defined by the contract's address and the total supply of tokens available to it but has a number of optional items that are usually provided as well to provide more detail to users. Methods and events detail below.

Methods

name

Returns the name of the token.

```
function name() view returns (string name)
```

symbol

Returns the symbol of the token.

```
function symbol() view returns (string symbol)
```

decimals

Returns the number of decimals the token.

```
function decimals() view returns (uint8 decimals)
```

totalSupply

Returns the total token supply.

```
function totalSupply() view returns (uint256 totalSupply)
```

balanceOf

Returns the account balance of another account with address `_owner`.

```
function balanceOf(address _owner) view returns (uint256 balance)
```

transfer

Transfers `_value` amount of tokens to address `_to`, and MUST fire the `Transfer` event. The function SHOULD `throw` if the `_from` account balance does not have enough tokens to spend.

```
function transfer(address _to, uint256 _value) returns (bool success)
```

transferFrom

Transfers `_value` amount of tokens from address `_from` to address `_to`, and fire the `Transfer` event.

The `transferFrom` method is used for a withdraw workflow, allowing contracts to transfer tokens.

```
function transferFrom(address _from, address _to, uint256 _value) returns (bool
success)
```

approve

Allows `_spender` to withdraw from your account multiple times, up to the `_value` amount. If this function is called again it overwrites the current allowance with `_value`.

allowance

Returns the amount which `_spender` is still allowed to withdraw from `_owner`.

```
function allowance(address _owner, address _spender) view returns (uint256
remaining)
```

Events

Transfer

Trigger when tokens are transferred, including zero value transfers.

```
event Transfer(address indexed _from, address indexed _to, uint256 _value)
```

Approval

Trigger on any successful call to `approve(address _spender, uint256 _value)`.

```
event Approval(address indexed _owner, address indexed _spender, uint256 _value)
```

ICO (Initial Coin Offering)

ERC20 standard, main Karatcoin Crowdsale contract, written in Solidity and deployed on Ethereum, based on references from OpenZeppelin.

Code Snippet

```
pragma solidity ^0.4.11;

import "../node_modules/zeppelin-solidity/contracts/ownership/Ownable.sol";
import "../node_modules/zeppelin-
solidity/contracts/crowdsale/FinalizableCrowdsale.sol";
import "./TokensCappedCrowdsale.sol";
import "./PausableCrowdsale.sol";
import "./BonusCrowdsale.sol";
import "./PreKCDToken.sol";


contract KCDCrowdsale is FinalizableCrowdsale,
TokensCappedCrowdsale(KCDCrowdsale.CAP), PausableCrowdsale(true),
BonusCrowdsale(KCDCrowdsale.TOKEN_USD_PRICE) {

    // Constants
    uint256 public constant DECIMALS = 18;
    uint256 public constant CAP = 1 * (10**9) * (10**DECIMALS);                //
1B KCD
    uint256 public constant KARATCOIN_AMOUNT = 1 * (10**9) * (10**DECIMALS);  //
1B KCD
    uint256 public constant TOKEN_USD_PRICE = 10;                              //
$0.10

    // Variables
    address public remainingTokensWallet;
    address public presaleWallet;

    /**
    * @dev Sets KCD to Ether rate. Will be called multiple times durign the
crowdsale to adjsut the rate
    * since KCD cost is fixed in USD, but USD/ETH rate is changing
    * @param _rate defines KCD/ETH rate: 1 ETH = _rate KCDs
    */
    function setRate(uint256 _rate) external onlyOwner {
        require(_rate != 0x0);
        rate = _rate;
        RateChange(_rate);
    }

    /**
    * @dev Allows to adjust the crowdsale end time
    */
```

```
    function setEndTime(uint256 _endTime) external onlyOwner {
        require(!isFinalized);
        require(_endTime >= startTime);
        require(_endTime >= now);
        endTime = _endTime;
    }

    /**
    * @dev Sets the wallet to forward ETH collected funds
    */
    function setWallet(address _wallet) external onlyOwner {
        require(_wallet != 0x0);
        wallet = _wallet;
    }

    /**
    * @dev Sets the wallet to hold unsold tokens at the end of ICO
    */
    function setRemainingTokensWallet(address _remainingTokensWallet) external
onlyOwner {
        require(_remainingTokensWallet != 0x0);
        remainingTokensWallet = _remainingTokensWallet;
    }

    // Events
    event RateChange(uint256 rate);

    /**
    * @dev Contructor
    * @param _startTime startTime of crowdsale
    * @param _endTime endTime of crowdsale
    * @param _rate KCD / ETH rate
    * @param _wallet wallet to forward the collected funds
    * @param _remainingTokensWallet wallet to hold the unsold tokens
    * @param _karatCoinWallet wallet to hold the initial 1B tokens of Karatcoin
    */
    function KCDCrowdsale(
        uint256 _startTime,
        uint256 _endTime,
        uint256 _rate,
        address _wallet,
        address _remainingTokensWallet,
        address _karatCoinWallet
    ) public
        Crowdsale(_startTime, _endTime, _rate, _wallet)
    {
        remainingTokensWallet = _remainingTokensWallet;
        presaleWallet = this;

        // allocate tokens to Karatcoin
        mintTokens(_karatCoinWallet, KARATCOIN_AMOUNT);
    }

    // Overrided methods

    /**
    * @dev Creates token contract for ICO
```

```
    * @return ERC20 contract associated with the crowdsale
    */
    function createTokenContract() internal returns(MintableToken) {
        PreKCDToken token = new PreKCDToken();
        token.pause();
        return token;
    }

    /**
    * @dev Finalizes the crowdsale
    */
    function finalization() internal {
        super.finalization();

        // Mint tokens up to CAP
        if (token.totalSupply() < tokensCap) {
            uint tokens = tokensCap.sub(token.totalSupply());
            token.mint(remainingTokensWallet, tokens);
        }

        // disable minting of KCDs
        token.finishMinting();

        // take onwership over KCDToken contract
        token.transferOwnership(owner);
    }

    // Owner methods

    /**
    * @dev Helper to Pause KCDToken
    */
    function pauseTokens() public onlyOwner {
        PreKCDToken(token).pause();
    }

    /**
    * @dev Helper to UnPause KCDToken
    */
    function unpauseTokens() public onlyOwner {
        PreKCDToken(token).unpause();
    }

    /**
    * @dev Allocates tokens from preSale to a special wallet. Called once as part
of crowdsale setup
    */
    function mintPresaleTokens(uint256 tokens) public onlyOwner {
        mintTokens(presaleWallet, tokens);
        presaleWallet = 0;
    }

    /**
    * @dev Transfer presaled tokens even on paused token contract
    */
    function transferPresaleTokens(address destination, uint256 amount) public
onlyOwner {
```

```
        unpauseTokens();
        token.transfer(destination, amount);
        pauseTokens();
    }

    //
    /**
    * @dev Allocates tokens for investors that contributed from website. These
include
    * whitelisted investors and investors paying with BTC
    */
    function mintTokens(address beneficiary, uint256 tokens) public onlyOwner {
        require(beneficiary != 0x0);
        require(tokens > 0);
        require(now <= endTime);                              // Crowdsale
(without startTime check)
        require(!isFinalized);                                //
FinalizableCrowdsale
        require(token.totalSupply().add(tokens) <= tokensCap); //
TokensCappedCrowdsale

        token.mint(beneficiary, tokens);
    }}
```

Key Processes

All processes have been designed with extreme attention in order to give excellent results, especially providing validation for asset documentation, its correct redemption, and developing ĐApps for future applications.

Gold Registration

Verification process which records and provides an audit trail of an asset on the Karat Blockchain using sequential digital signatures from the entities in the chain of custody, namely Vendor, Custodian, Auditor, which are further validated with proof of depository receipts provided and uploaded onto IPFS for permanent record. The PoA Verification also contains a sub process for regular audits called Auditor Process, which are executed each quarter.

Gold Minting (KCG creation)

This process allows the creation of KCG gold tokens to be used as crypto-assets inside the Karat Blockchain.

Gold Asset Audit

In this process Auditor checks total gold asset holdings against physical contents vault and all registered assets.

Certificate Registration

This process allows the registration of Karat Certificates inside the Karat Blockchain.

Certificate Registration with KCG

This process allows the registration of Karat Certificates using KCG gold tokens.

Certificate Recast

This process allows the recast of a Karat Certificates into KCG gold tokens.

Withdrawal

This process allows the withdrawal of KCG gold tokens into fiat.

Redemption

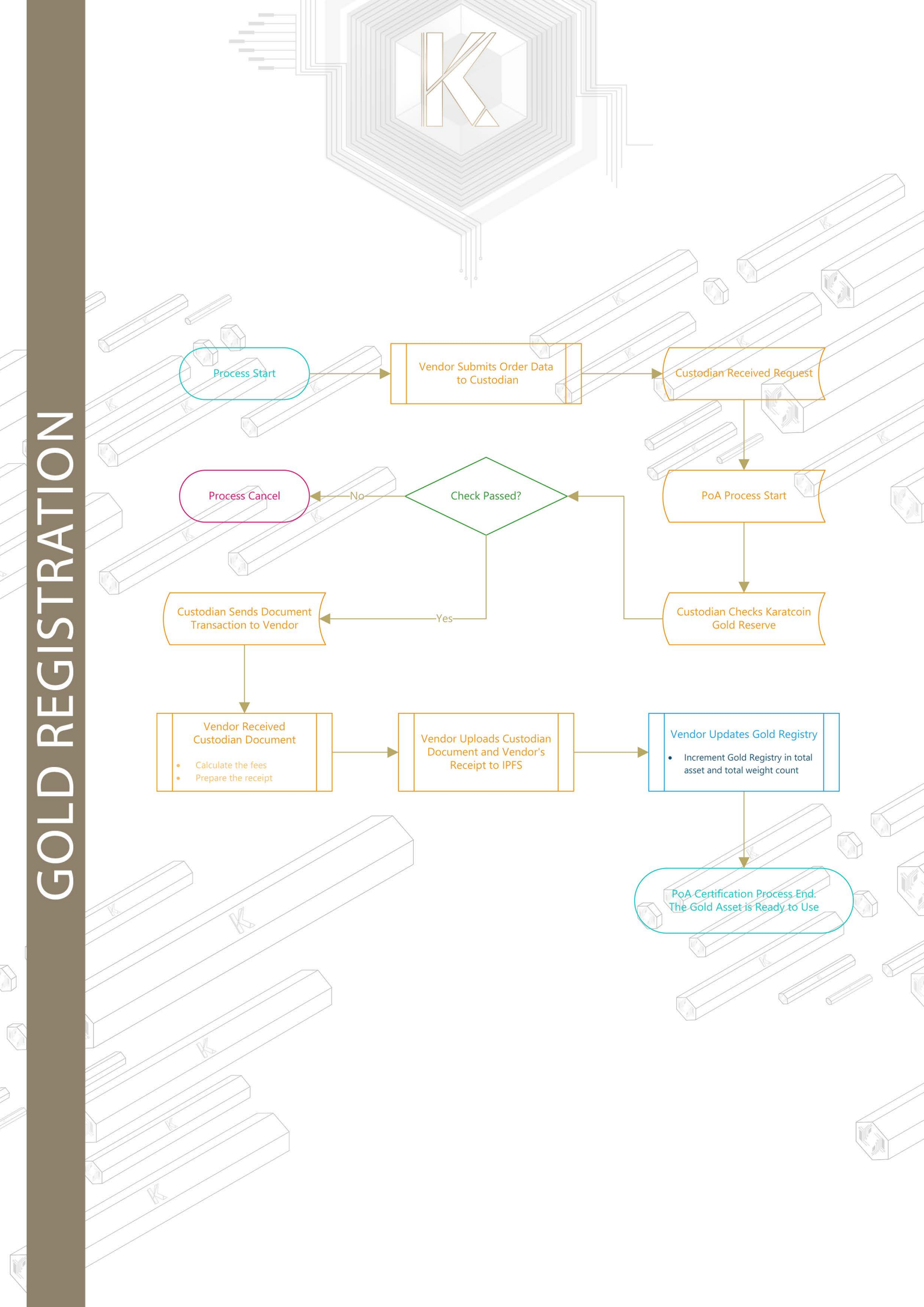
Redeeming physical gold bars. The user can choose between withdrawal agreed on the spot or by courier.

Smart Contracts Stack

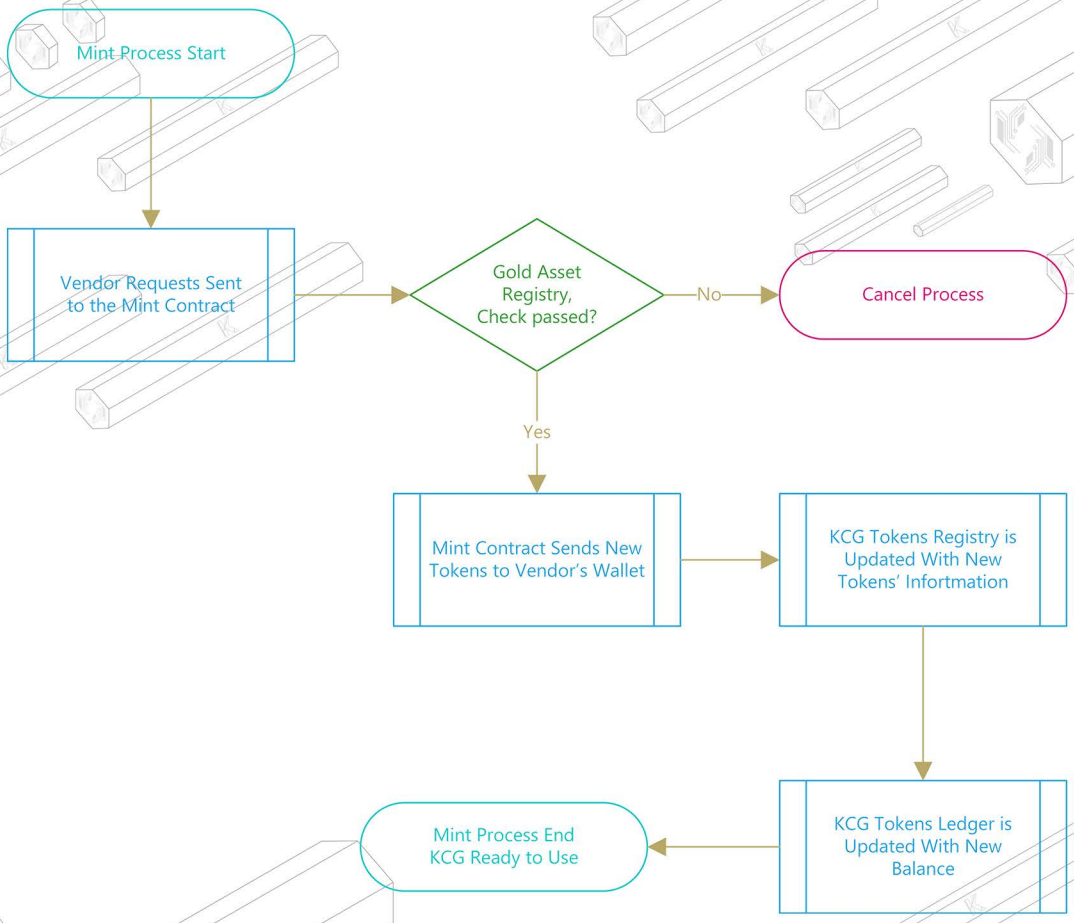
The process related to the creation of individual Smart Contracts distributed in the Blockchain; this is beneficial because it allows developers to utilize Karatcoin tokens for ĐApp development.

Here below all graphs for these processes:

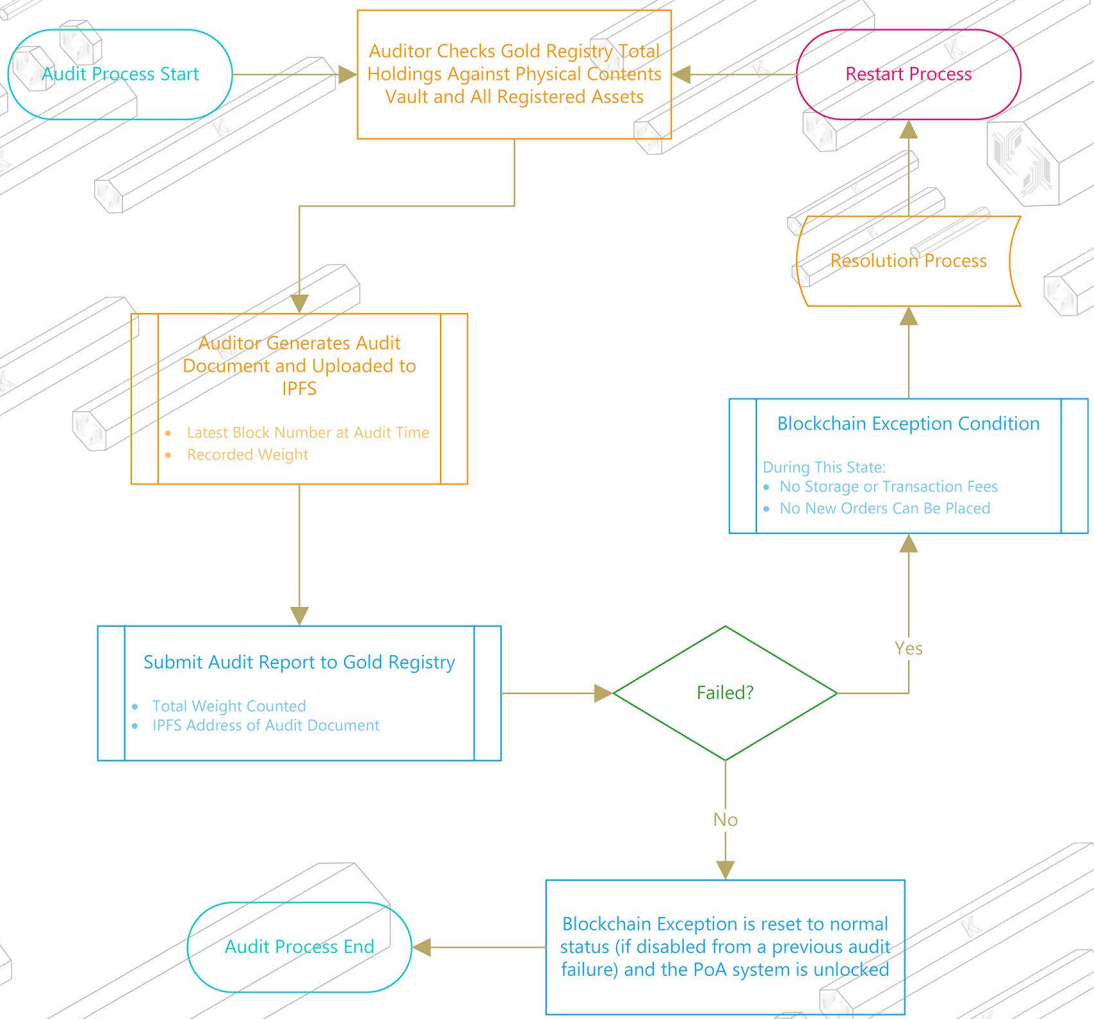
GOLD REGISTRATION



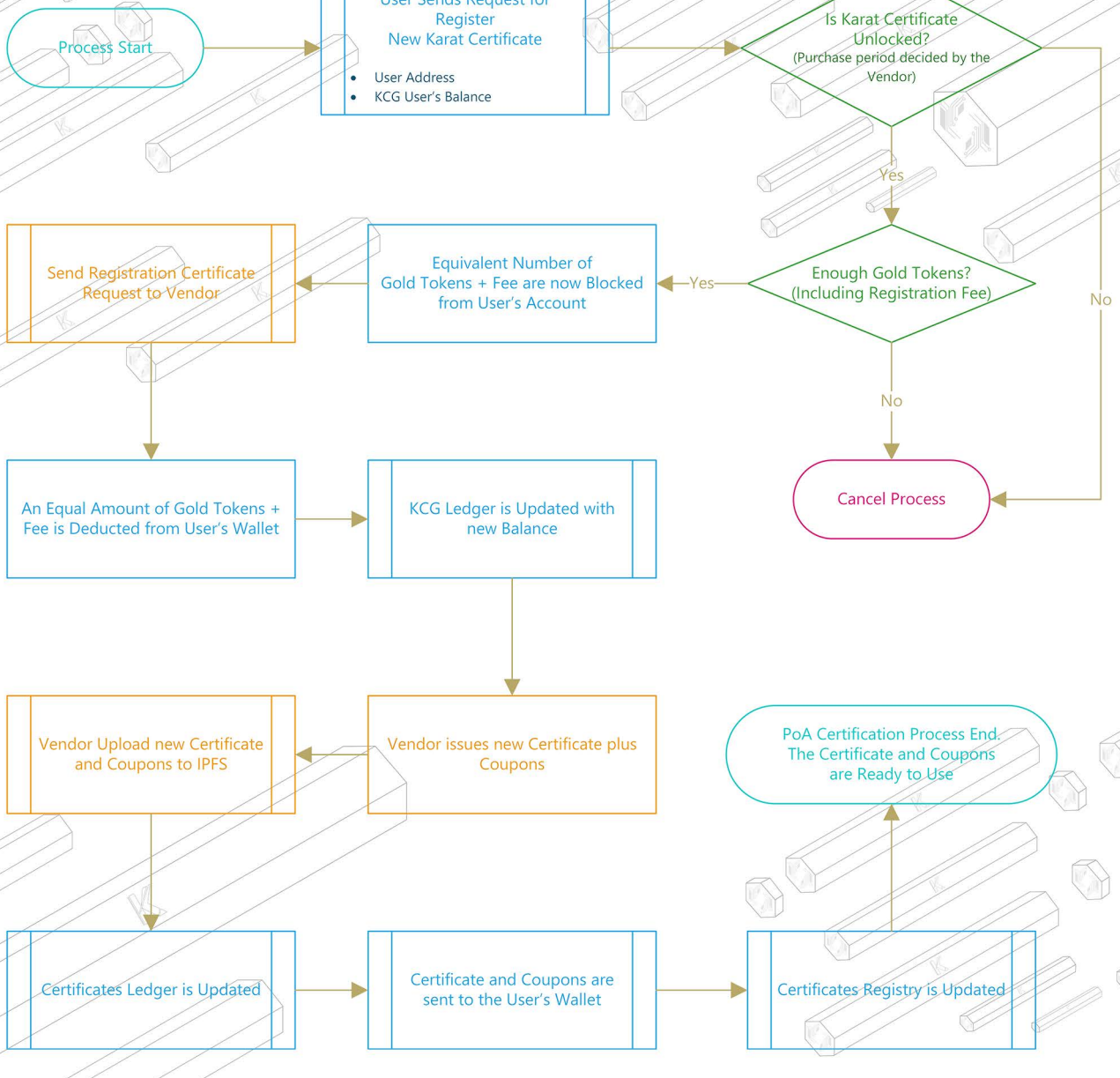
GOLD MINTING



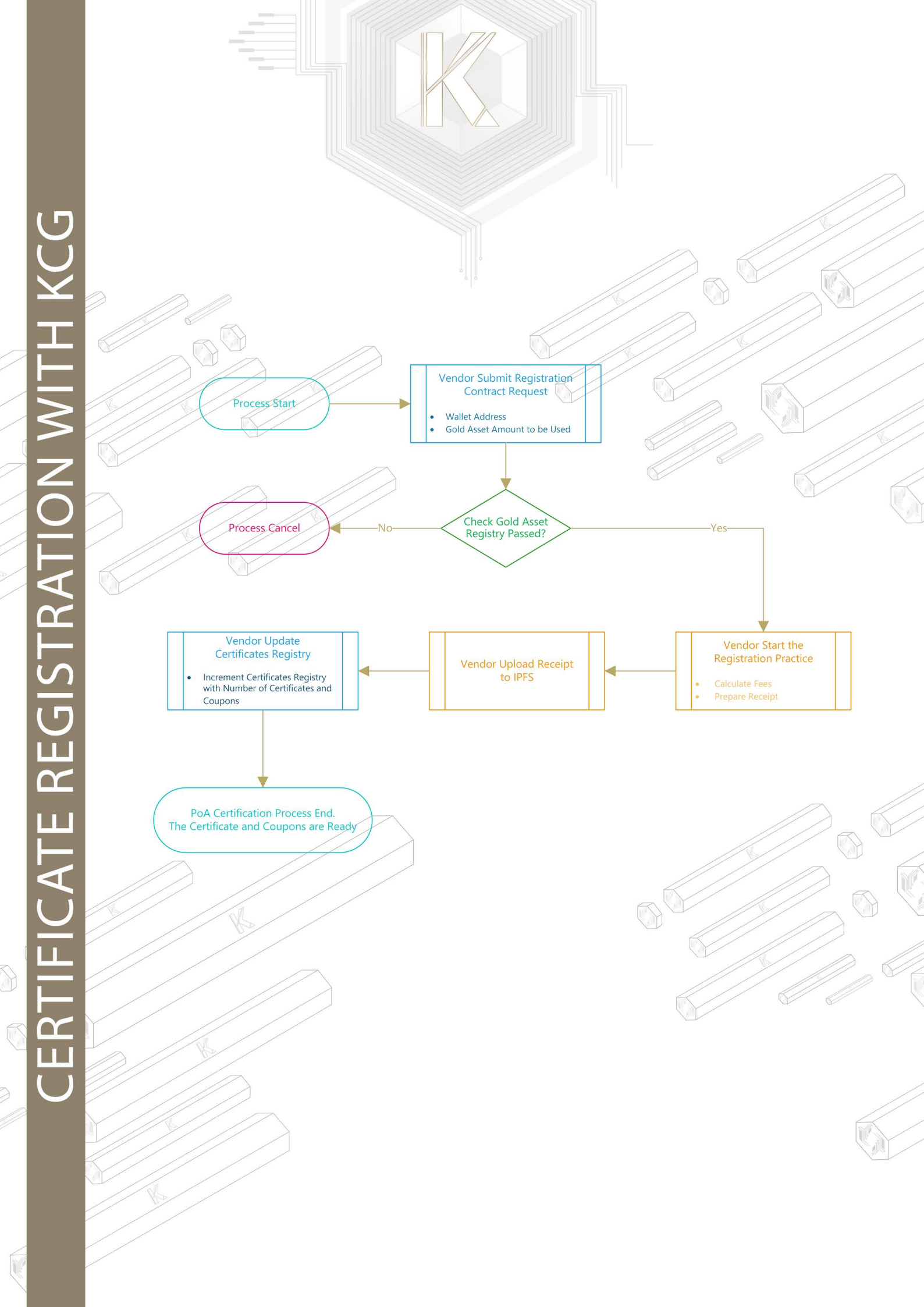
GOLD ASSET AUDIT



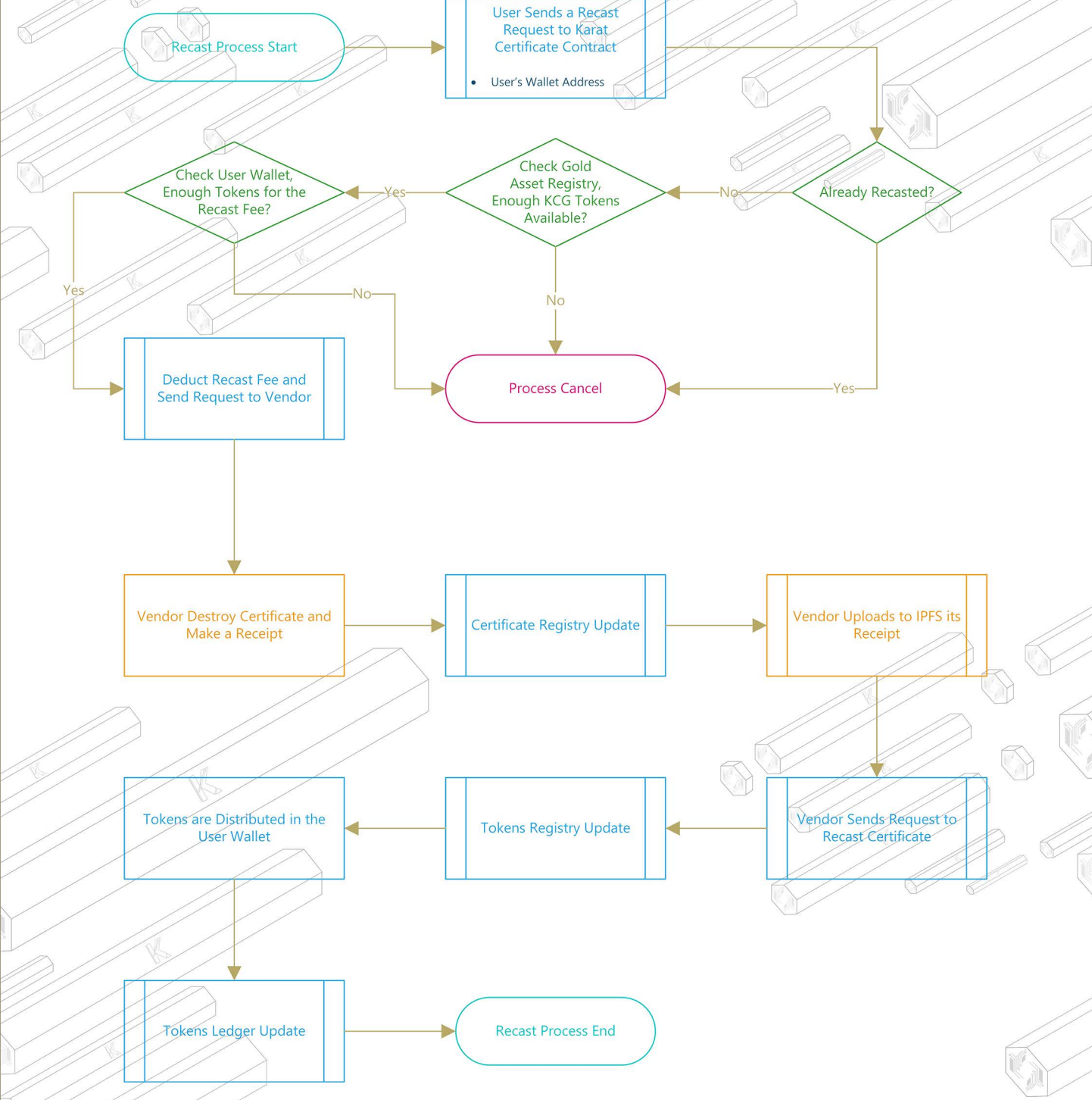
CERTIFICATE REGISTRATION



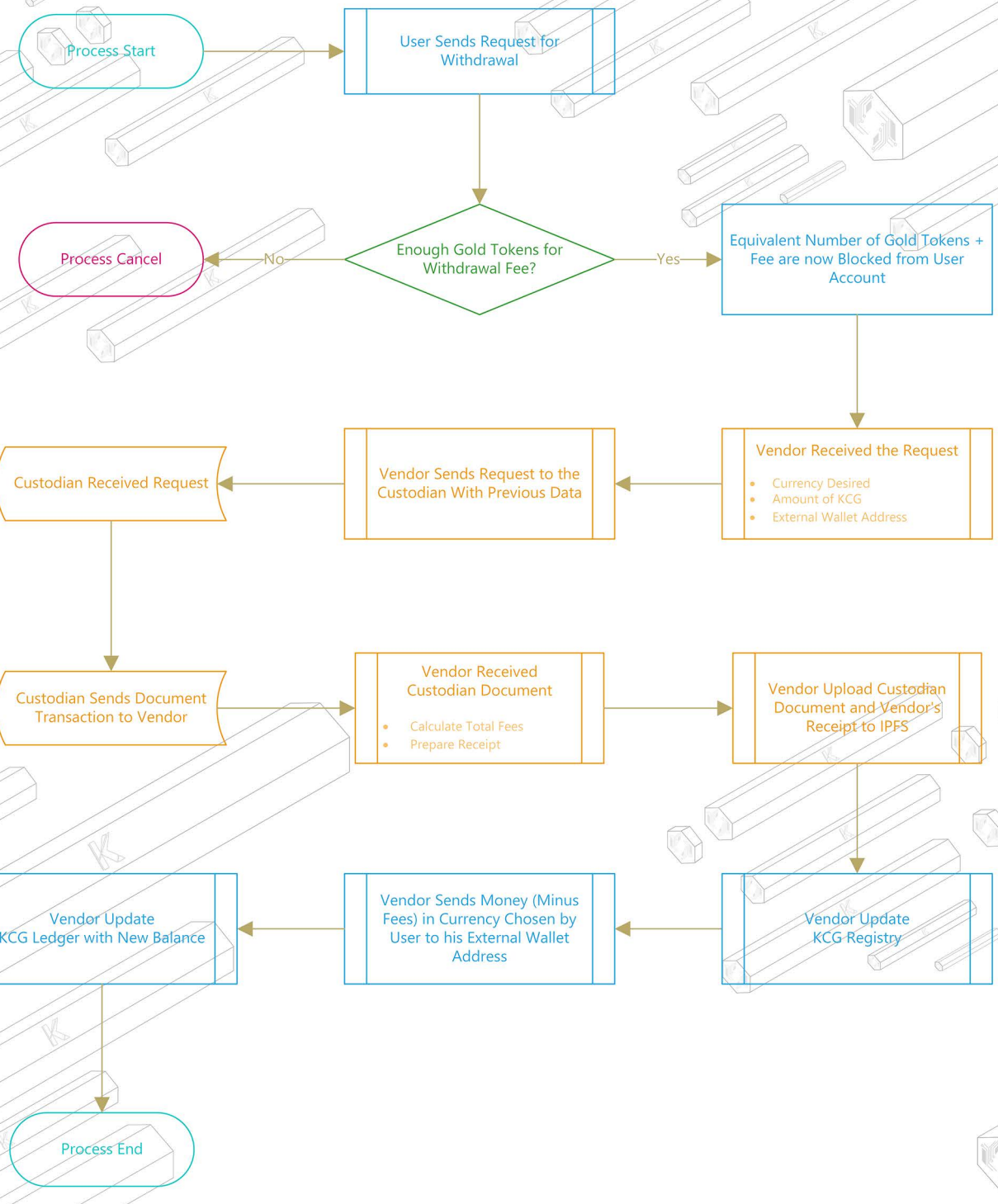
CERTIFICATE REGISTRATION WITH KCG



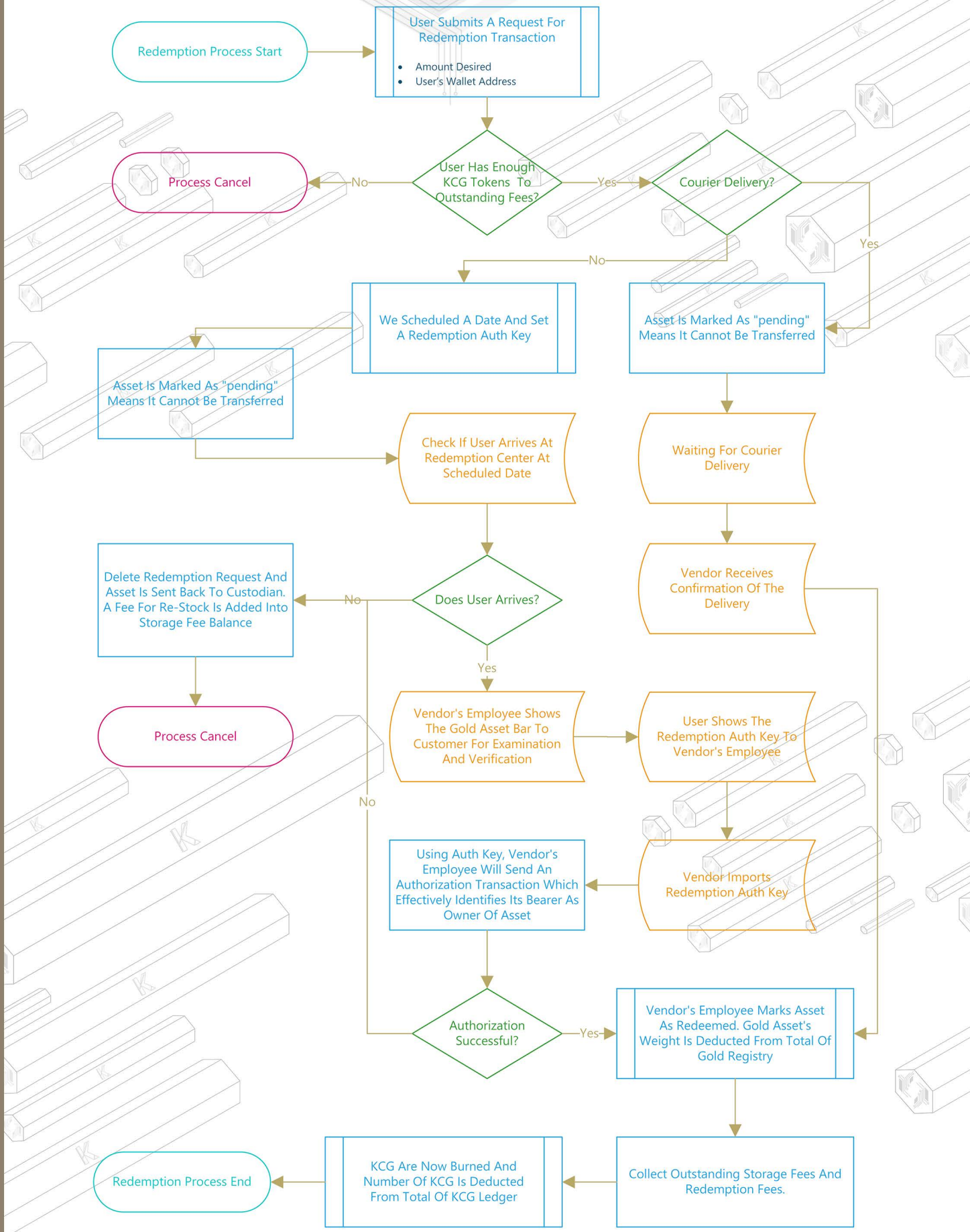
CERTIFICATE RECAST



GOLD WITHDRAWAL



GOLD REDEMPTION



SMART CONTRACTS STACK MAP

Account Types

- Karatcoin
- User
- Vendor Employee
- Vendor Admin
- Custodian Employee
- Custodian Admin
- Auditor Employee
- Auditor Admin

Participant Registries

Custodian Registry
Participating Custodian Vault

Authorized Financial Institution

Vendor Registry
Trust Company

Auditor Registry
Bureau Veritas London Local Government

Authorities in Gold Mining Areas

Participant / Administration Interfaces

Custodian Interface Contract
Allows only registered custodian to register or remove assets into the Gold Asset Registry

Vendor Interface
Allows only registered vendor to register new assets into the Gold Asset Registry contains product and order infos.

Auditor Interface Contract
Allows only registered auditors to submit audit reports into the Gold Asset Registry

Admin Interface Contract
Register Vendor, Custodian, Auditor

Delegate Vendor / Custodian / Auditor

Administrators Interface for changing

contract configuration settings

Root Level Registries

Configuration Registry
Contract that holds configuration variables used by all our contracts.

Gold Asset Registry
The registry containing all our registered Gold Asset Certificates and Audit reports details

KCG Registry
The registry containing all our registered KCG Gold tokens details

KCX Registry
The registry containing all our registered KCX validation tokens details

Certificates Registry
The registry containing all our registered Karat Certificates and relative coupons

Mint Contract
Converts Gold Asset into Gold tokens (KCG)

Service Contracts

Mint Certificate Contract
Converts Gold tokens (KCG) into Karat Certificate

Recast Certificate Contract
Converts Karat Certificate into Gold Asset tokens (KCG)

User Callable Contracts

KCG Gold tokens Contract
Transferable Karatcoin Gold tokens (KCG). Each contract represents 1gr. of Gold allocated secured in Safehouse Vaults with a corresponding serial number

KCG Ledger Contract
Ledger containing Karatcoin Gold tokens (KCG) balances

Gold Certificate Contract
Each contract represents an immutable Karat Certificate with a corresponding serial number and 2 semi annual coupons

Gold Certificate Ledger Contract
Ledger containig Karat Certificates and coupons balances

KCX token Contract
Transferable Karatcoin Validation tokens (KCX).

KCX Ledger Contract
Ledger containing Karatcoin Validation tokens balances

Rewards Ledger
Ledger containing rewards balances for stakeholders of validating blocks on Karat Blockchain

KCG TX Fees Ledger
Ledger containing fees balances of any Karatcoin Gold token transactions

Legend

- Karat Blockchain
- Only for Karat Certificates

KARATCOIN

Straight to Gold Mines