Algorand School

2022

Cosimo Bassi
Solutions Architect at Algorand Inc.

cosimo.bassi@algorand.com

Algorand

Our journey today:

understanding Algorand Consensus and Algorand Networks, how to use Algorand Dev Tools, how to develop decentralized applications on the Algorand Virtual Machine

Algorand

Agenda

- Algorand Consensus
- Algorand Decentralized Governance
- Algorand Networks
- Algorand Transactions
- Algorand Accounts
- Algorand Virtual Machine
- Algorand Smart Contracts on Layer-1
- Smart Signatures & Smart Contracts
- TEAL
- PyTEAL
- dApp Example & Use Cases

ALGORAND CONSENSUS

Pure Proof of Stake (PPoS)

What is a blockchain?

A **blockchain** is a **public ledger** of transactional data, **distributed** across a system of multiple **nodes** in a network.

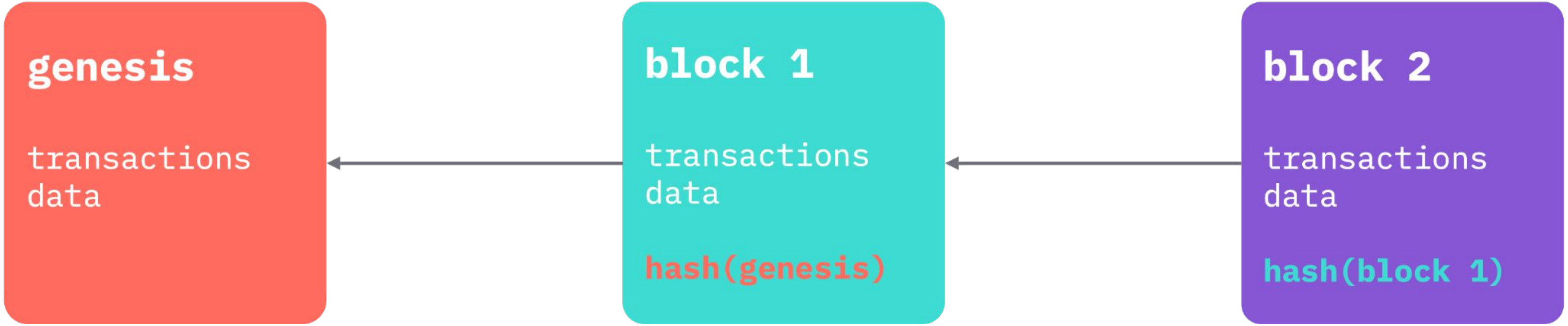
All these nodes work together, using the **same set of software and rules**, to **verify** the transactions to add to the finalized ledger. This set of rules is called **consensus protocol**.

The ledger is **publicly verifiable**, **permissionless** and **tamper-proof**.

What is a blockchain?

The "**block**" refers to a set of transactions that are proposed and verified by the other nodes and eventually added to the ledger.

The "**chain**" refers to the fact that each block of transactions also contains proof (a cryptographic hash) of what was in the previous block.



The architecture of consensus

1. How to choose the **proposer for the next block** for a **public and permissionless** blockchain?

2. How to ensure that **there is no ambiguity** in the choice of the next block?

3. How to ensure that the **blockchain stays unique** and has **no forks**?

4. How to ensure that **consensus mechanism itself can evolve** over time while the blockchain is an immutable ledger?

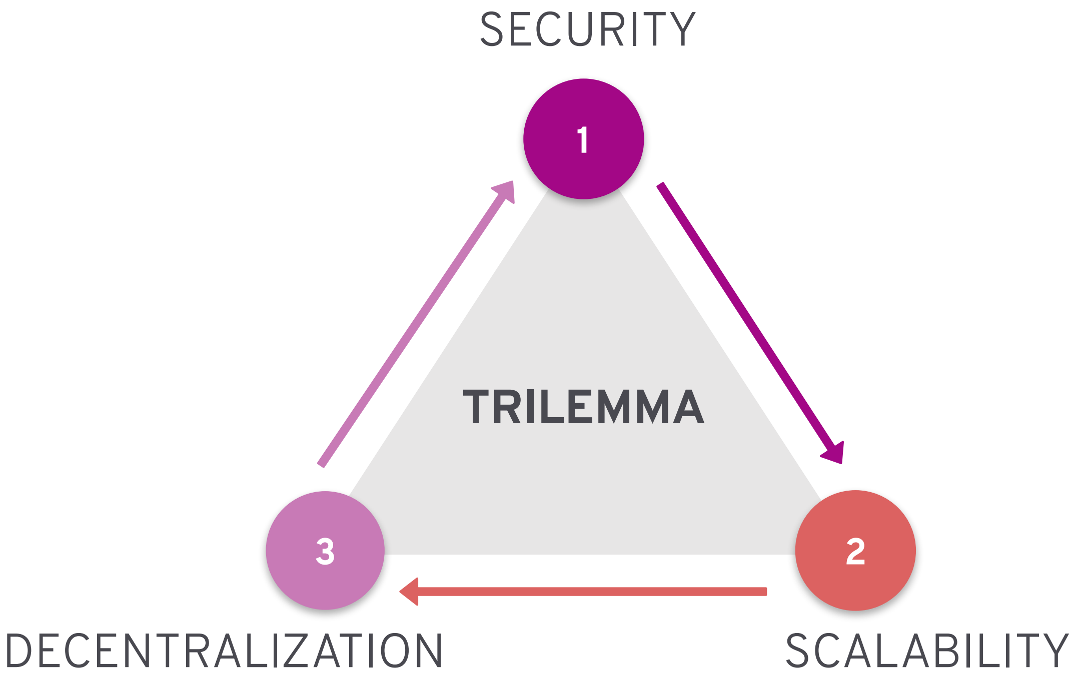
Temple of Concordia

Valley of Temples (Agrigento), 440-430 B.C.

Consensus mechanisms

		CRITICAL ISSUES
PoW	**PROOF OF WORK** Miners **compete with each other** to append the next block and earn a **reward for the effort**, fighting to **win an expensive computational battle**.	• Huge electrical consumption • Concentration of governance in few mining farms • Soft-forking of the blockchain
BPoS	**BONDED PROOF OF STAKE** Validators **bind their stake**, to **show their commitment** in validating and appending a new block. Misbehaviors are punished.	• Participating in the consensus protocol makes users’ stakes illiquid • Risk of economic barrier to entry
DPoS	**DELEGATED PROOF OF STAKE** Users **delegate the validation** of new blocks to a **fixed committee**, through weighted voting based on their stakes.	• Known delegate nodes, therefore exposed to DDoS attacks • Centralization of governance

Is the *Blockchain Trilemma* unsolvable?



Algorand PPoS Consensus

- **Scalable** billions of users
- **Efficient** 1000 TPS (10x work in progress)
- **Fast** < 5s per block
- **Low fees** 0.001 ALGO per txn
- **No Soft Forks** prob. $< 10^{-18}$
- **Instant Transaction Finality**
- **Minimal hardware node requirements**
- **No delegation or binding of the stake**
- **No minimum stake**
- **Carbon negative**
- **Secure with respect DDoS**
- **Network Partitioning resilience**

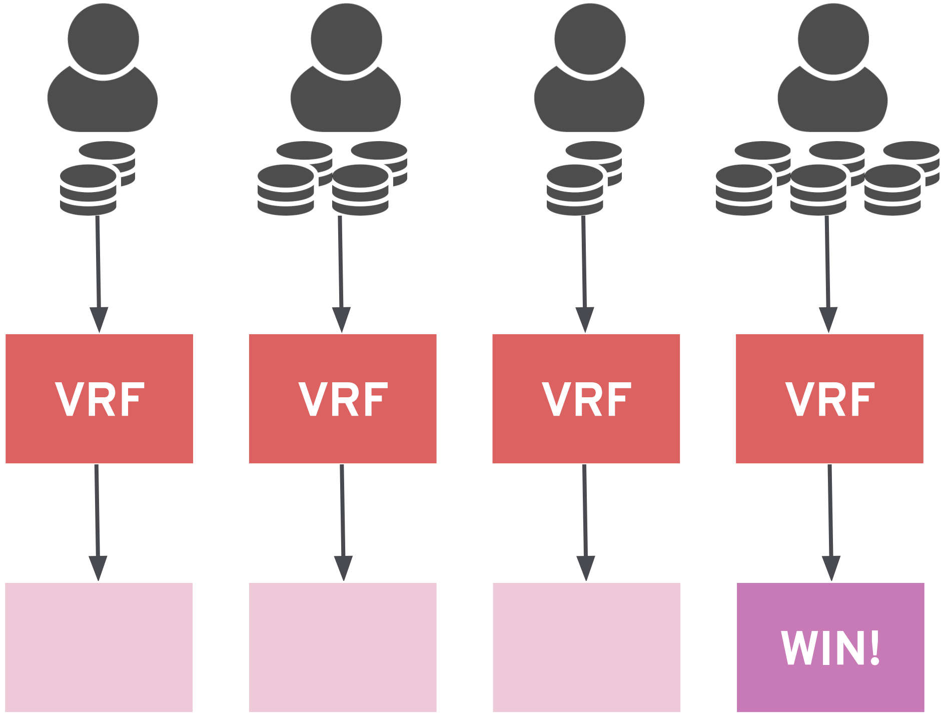
Silvio Micali

Algorand Founder

Professor MIT, Turing Award, Gödel Prize

Digital Signatures, Probabilistic Encryption, Zero-Knowledge Proofs, Verifiable Random Functions and other primitives of modern cryptography.

Who chose the *next block*?



1. Each **online ALGO** could be assimilated to a **ticket** participating in a safe and secret **cryptographic sortition**

2. For **each new block**, tickets' draw is performed in a **distributed, parallel and secret** and manner, directly on online accounts' hardware (in microseconds)

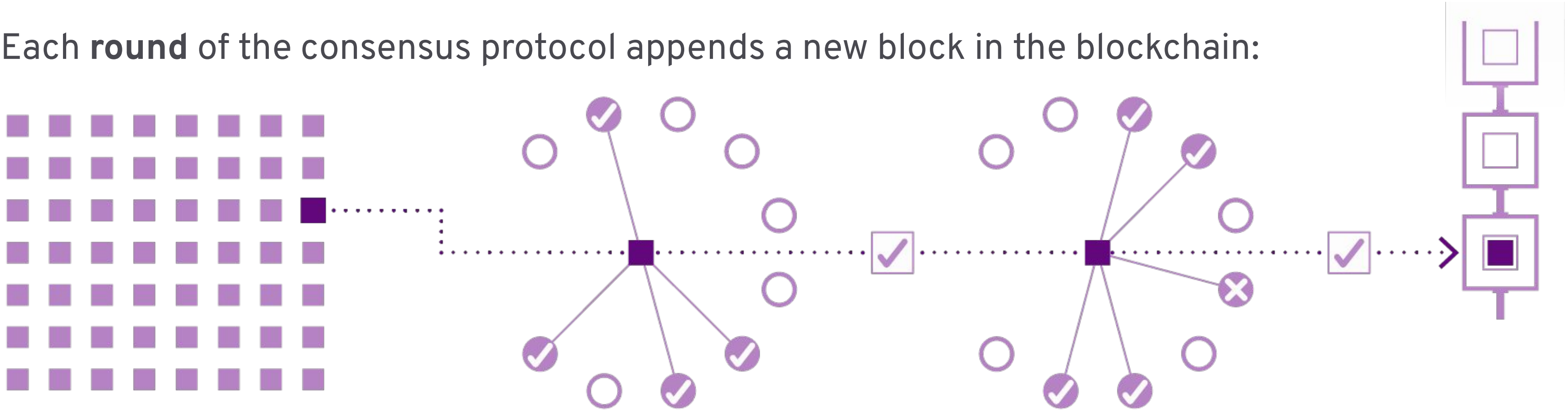
3. The **winner is revealed** in a safe and **verifiable way** only after winning the draw, proposing the next block

A glimpse on “simplified” VRF sortition

1. A **secret key (SK)** / **public verification key (VK)** pair is associated **with each ALGO in the account**
2. For each new round ***r*** of the **consensus protocol** a **threshold *L(r)*** is defined
3. **Each ALGO** in the account **performs a VRF**, using its own **secret key (SK)**, to generate:
    a. a pseudo-random number: $Y = VRF_{SK}(seed)$
    b. the verifiable associated proof: $\rho_{SK}(seed)$
4. If $Y = VRF_{SK}(seed) < L(r)$, that specific ALGO **“wins the lottery”** and **viraly propagates the proof** of its victory $\rho_{SK}(seed)$ to other **network’s nodes**, through “gossiping” mechanism
5. Others node can use the **public verification key (VK)** to verify, through $\rho_{SK}(seed)$, that the number **Y** was generated by **that specific ALGO, owned by the winner of the lottery**

Pure Proof of Stake, in short

Each **round** of the consensus protocol appends a new block in the blockchain:



An account is elected to **propose** the next **block**	A **committee** is elected to **filter** and **vote** on the **block proposals**	A **new committee** is elected to reach a quorum and **certify** the **block**	The **new block** is **appended** to the **blockchain**

Through the **cryptographic lottery**, an **online account** is elected with probability directly proportional to its stake: **each ALGO corresponds to an attempt** to win the lottery!

Pure Proof of Stake security

Algorand's decentralized Byzantine consensus protocol can tolerate an **arbitrary number of malicious users** as long as honest users hold a **super majority of the total stake** in the system.

1. The adversary **does not know which users he should corrupt**.
2. The adversary **realizes which users are selected too late** to benefit from attacking them.
3. Each **new set of users will be privately and individually elected**.
4. During a network partition in Algorand, the adversary is **never able to convince two honest users to accept two different blocks** for the same round.
5. Algorand is able to **recover shortly after network partition** is resolved and guarantees that new blocks will be generated at the same speed as before the partition.

Pure Proof of Stake, some numbers

BLOCKS	> 18 M with 0 downtime
BLOCKCHAIN SIZE	~ 900 GB
ADDRESSES	> 18 M with ~ 6 M monthly active addresses
AVG. BLOCK FINALIZATION	~ 4,4 sec per block
TXNS VOLUME MONTHLY PEAK	~ 40 M transactions (March 2021)
TPS WEEKLY PEAK	~ 1150 transactions per second

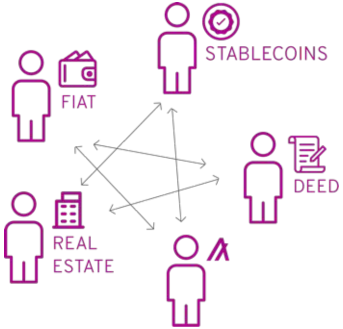
** up to January 2022*

Algorand Layer-1 primitives

Algorand Standard Assets (ASA)

- SECURITIES
- CURRENCIES
- STABLECOINS
- UTILITY TOKENS
- CERTIFICATIONS
- REAL ESTATE

Atomic Transfers (AT)



Algorand Smart Contracts (ASC1)

- FAST
- SECURE
- LOW COST

DECENTRALIZED GOVERNANCE

Governing ALGO

ALGO Decentralized Governance

Decentralized Governance over **Algorand Ecosystem Resource Pool (AERP)**, including Ecosystem Support, Participation Incentives and Contingent Rewards (total of 3,2B ALGOs) previously entrusted to the Algorand Foundation.

Decentralized Governors

- **Vote on-chain**, each quarter, by **staking their ALGO-votes** in governance.
- **Decide how the AERP should be utilised and distributed**, to support the long term development of the Algorand network.
- **Are rewarded for their efforts**, based on their stake in governance.

The ALGO

GENESIS BLOCK	Main Net, June 2019
GENESIS HASH	wGHE2Pwdvd7S12BL5FaOP20EGYesN73ktiC1qzkkit8=
TOTAL SUPPLY (fixed)	10 B ALGO
MINIMAL UNIT	1 microALGO = 10^{-6} ALGO
CIRCULATING SUPPLY	~ 6,5 B ALGO
PPoS PARTICIPATING STAKE	~ 2,1 B ALGO
GOVERNANCE STAKE	~ 3,2 B ALGO
GOVERNORS	~ 65 k

** January 2022*

ALGORAND NETWORKS

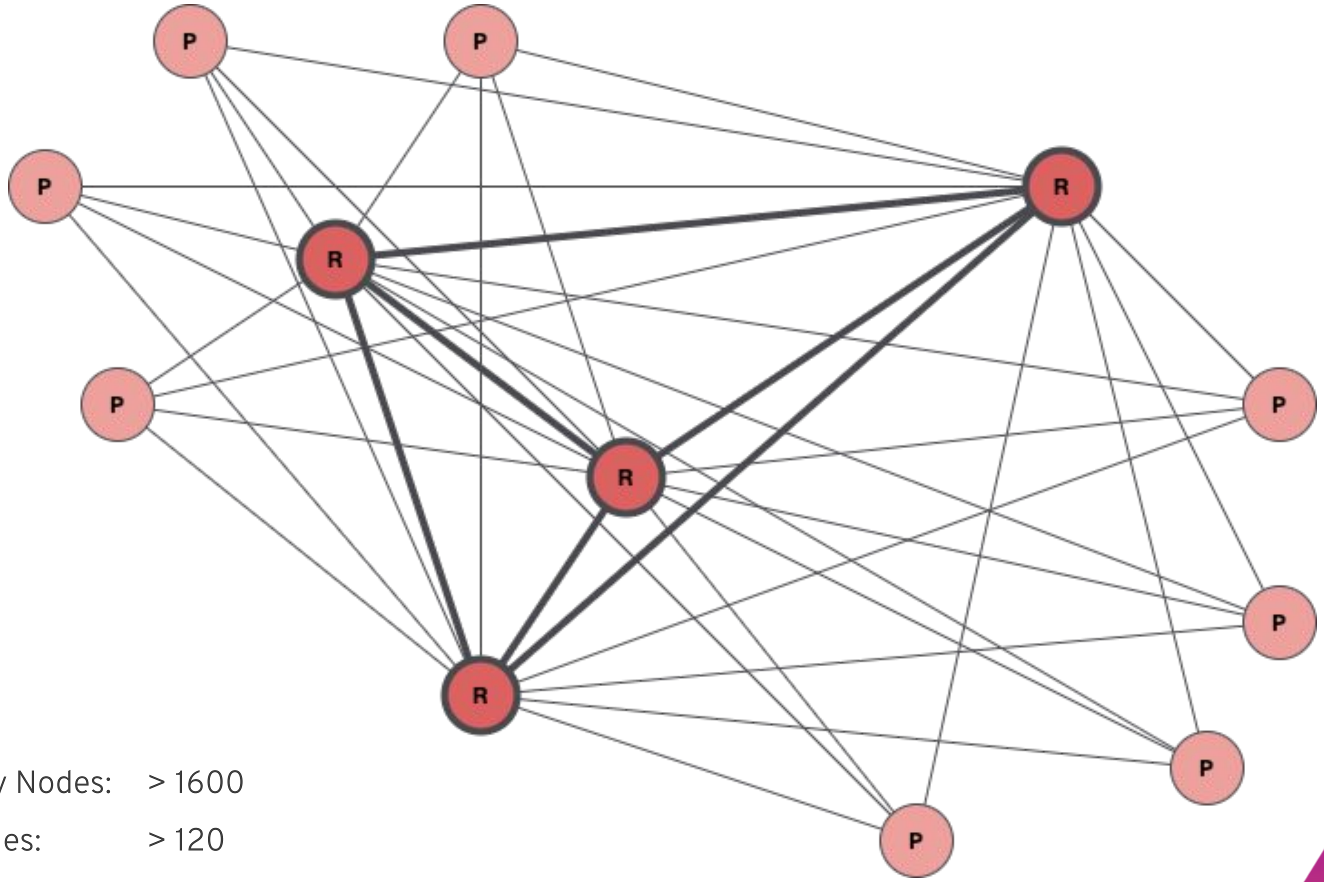
Nodes, Indexer and APIs

Algorand Node configurations

1. **Non-Relay Nodes**
   - Participate in the PPoS consensus (if hosting participation keys)
   - Connect only to Relay Nodes
   - Light Configuration: store just the lastest 1000 blocks (Fast Catch-Up)
   - Archival Configuration: store all the chain since the genesis block

2. **Relay Nodes**
   - Communication routing to a set of connected Non-Relay Nodes
   - Connect both with Non-Relay Nodes and Relay Nodes
   - Route blocks to all connected Non-Relay Nodes
   - Highly efficient communication paths, reducing communication hops

Example of Algorand Network topology



Node Metrics

- Non-Relay Nodes: > 1600
- Relay Nodes: > 120

Algorand

Access to Algorand Network

The **Algorand blockchain is a distributed system of nodes** each maintaining their **local state** based on validating the history of blocks and the transactions therein. **Blockchain state integrity** is maintained by the **consensus protocol** which is implemented within the **Algod daemon** (often referred to as the node software).

An application **connects to the Algorand blockchain** through an **Algod client**, requiring:

- a valid Algod REST API endpoint IP address
- an Algod token from an Algorand node connected to the network you plan to interact with

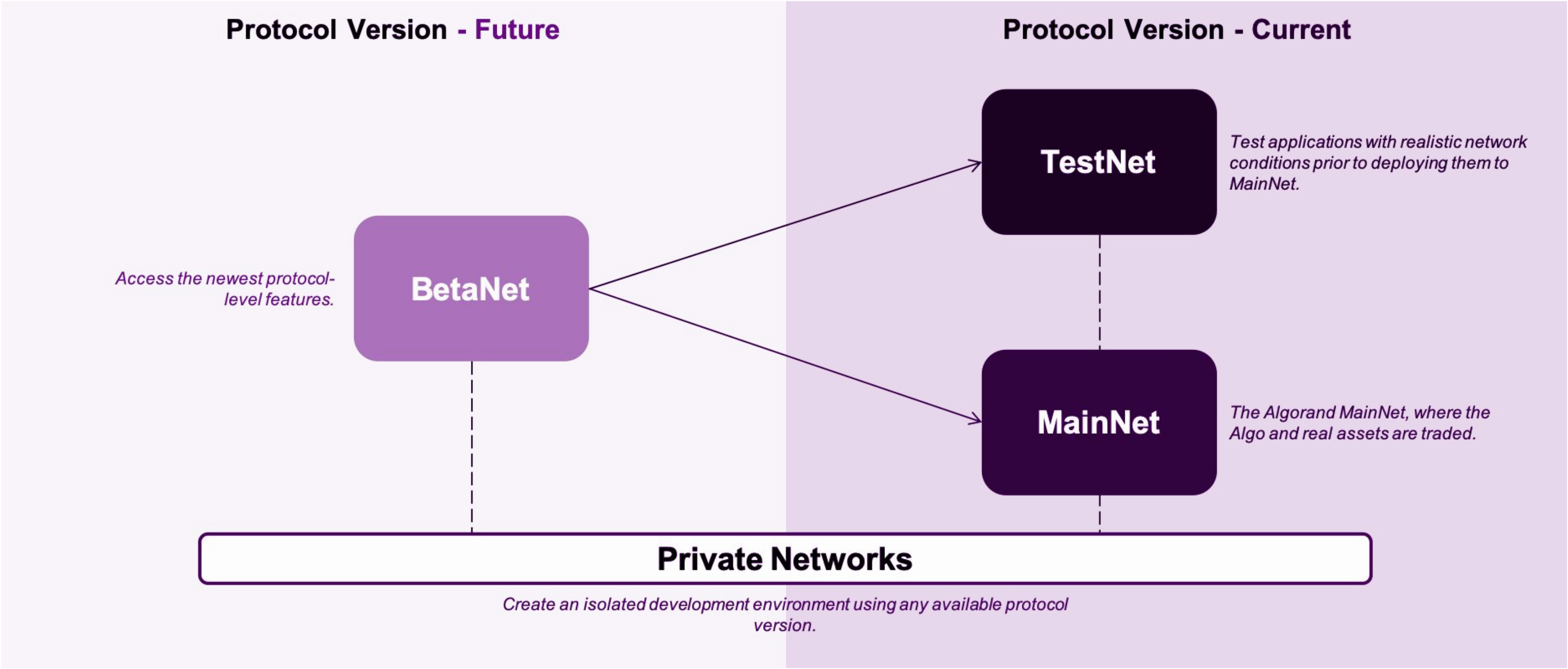
These **two pieces of information** can be provided by **your local node** or by a **third party node aaS**.

How to get an Algod Client?

There are **three ways** to get a REST API Algod **endpoint IP address / access token**, each with their respective pros and cons depending on development goals.

	Use a third-party service	Use Docker Sandbox	Run your own node
Time	**Seconds** - Just signup	**Minutes** - Same as running a node with no catchup	**Days** - need to wait for node to catchup
Trust	1 party	1 party	Yourself
Cost	Usually free for development; pay based on rate limits in production	Variable (with free option) - see node types	Variable (with free option) - see node types
Private Networks	❌	✅	✅
`goal`, `algokey`, `kmd`	❌	✅	✅
Platform	Varies	MacOS; Linux	MacOS; Linux
Production Ready	✅	❌	✅

Algorand Networks



Algorand Node - Writing on the blockchain

1. Install (Linux, MacOS, Windows)
2. Choose a **network** (MainNet, TestNet, BetaNet, PrivateNet)
3. Start & Sync with the network, Fast Catchup

genesis.json (mainnet)

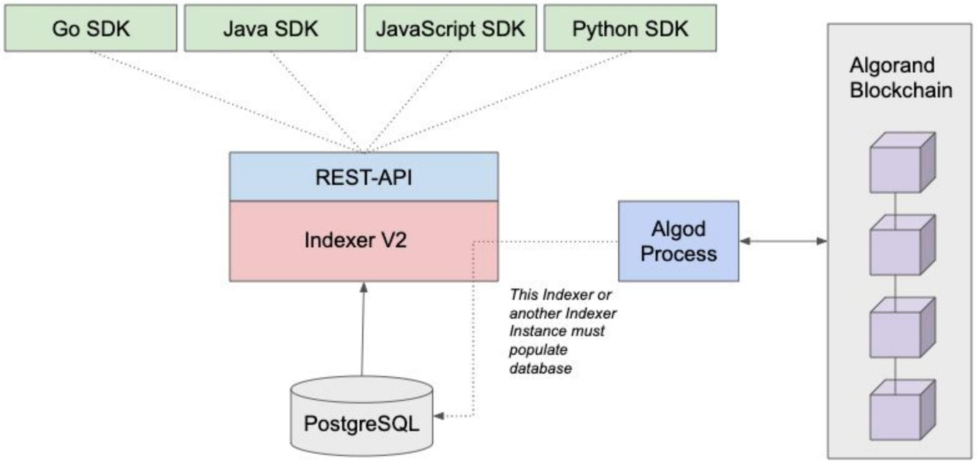

```
{
  "alloc": [
    {
      "addr": "737777777777777...77777777UFEJ2CI",
      "comment": "RewardsPool",
      "state": {
        "algo": 10000000000000,
        "onl": 2
      }
    },
    {
      "addr": "Y76M3MSY6DKBRHBL7C3...F2QWNPL226CA",
      "comment": "FeeSink",
      "state": {
        "algo": 1000000,
        "onl": 2
      }
    },
    ...
  ],
  "fees": "Y76M3MSY6DKBRHBL7C3NNDX...F2QWNPL226CA",
  "id": "v1.0",
  "network": "mainnet",
  "proto":
"https://github.com/algorandfoundation/specs/tree/5615ad
c36bad610c7f165fa2967f4ecfa75125f0",
  "rwd": "7377777777777777777777...77777777UFEJ2CI",
  "timestamp": 1560211200
}
```


Interacting with Algorand Nodes

1. CLI utilities: **goal**, **kmd** and **algokey**
2. REST API interface: **algod V2**, **kmd**, **indexer**
3. Algorand SDKs: **JavaScript**, **Python**, **Java** o **Go**

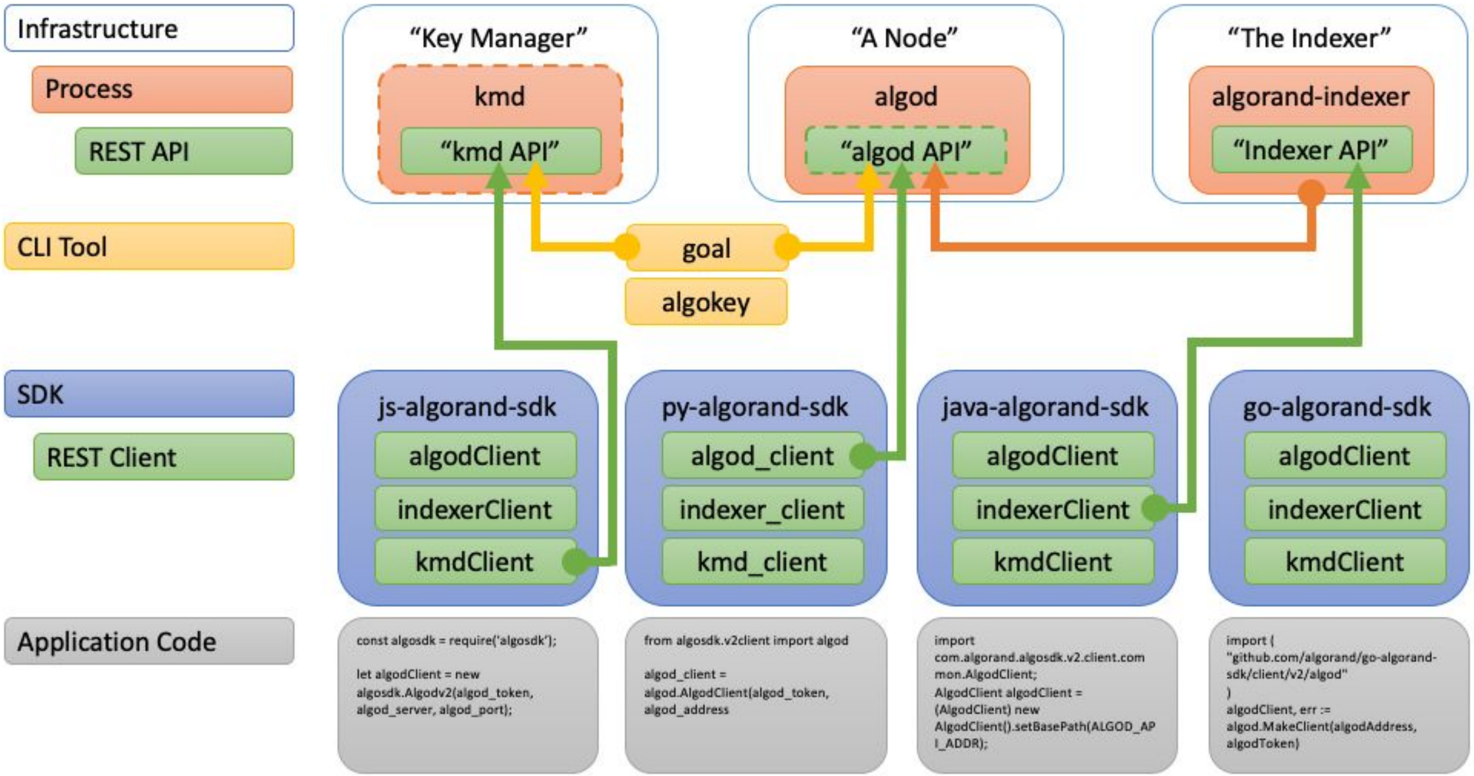
Algorand

Algorand Indexer - Reading from the blockchain



The **Indexer** provides a **REST API interface** of API calls to **query the Algorand blockchain**. The Indexer REST APIs retrieves blockchain data from a **PostgreSQL database**, populated using the Indexer instance connected to an **Archival Algorand node that reads blocks' data**. As with the Nodes, the Indexer can be used as a **third-party service**.

How to interact with Algorand Node and Indexer



Algorand SDKs

Java

JavaScript

Python

Go

Algorand Community SDKs

C#

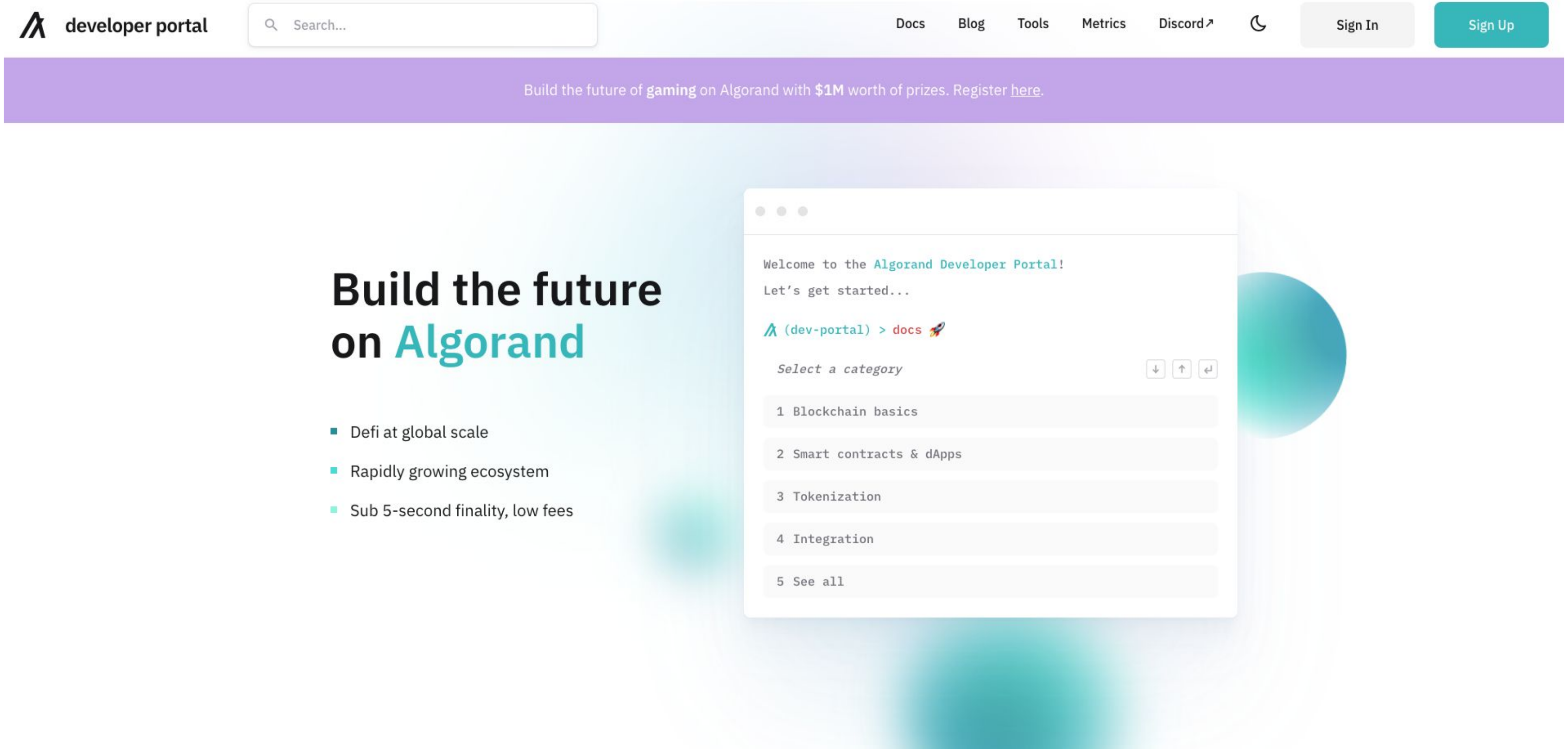
Rust

Dart

PHP

Swift

Algorand Developer Portal



Awesome Algorand

Contribute on GitHub

awesome

Algorand

⚡ An awesome list about everything related to the Algorand Blockchain.
Algorand is an open-source, proof of stake blockchain and smart contract computing platform.

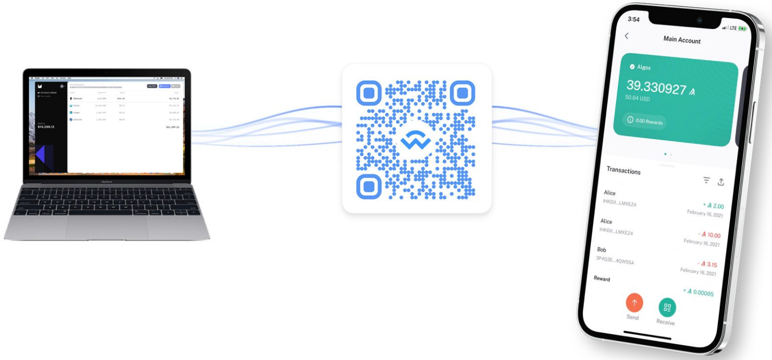
visitors 1462 | Web 2.0 Website | Web 3.0 Website | CI passing

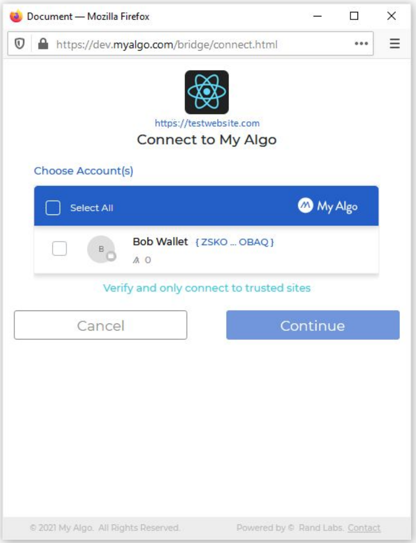
Contents

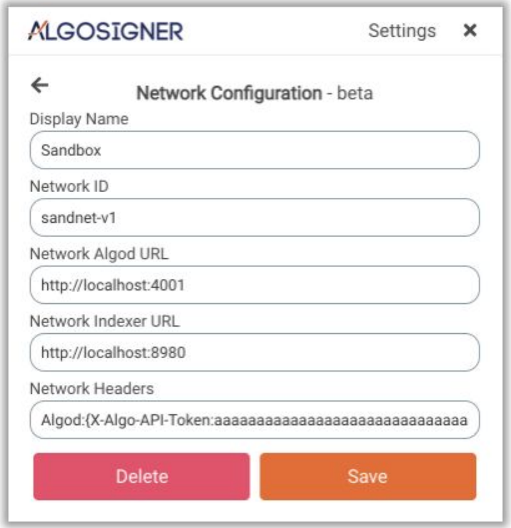
- Official
- Wallets
- Blockchain Explorers
- Learning
  - Tutorials
- Development
  - Dart
  - Go
  - PHP
  - Python

Algorand Wallets

WalletConnect







Mobile Wallet + Wallet Connect

MyAlgo Wallet

AlgoSigner

Algorand Explorers

AlgoExplorer

GOALSEEKER

ALGORAND TRANSACTIONS

Core element of blocks

Changing blockchain state

Transactions are the **core element of blocks**, which define the **evolution** of distributed ledger **state**. There are **six transaction types** in the Algorand Protocol:

1. Payment
2. Key Registration
3. Asset Configuration
4. Asset Freeze
5. Asset Transfer
6. Application Call

These six transaction types can be specified in particular ways that result in more granular perceived transaction types.

Signature, fees and round validity

In order to be approved, Algorand's transactions must comply with:

1. **Signatures**: transactions must be correctly signed by its sender, either a **Single Signature**, a **Multi Signature** or a **Smart Signature / Smart Contract**
2. **Fees**: in Algorand transactions fees are a way to protect the network from DDoS. In Algorand Pure PoS fees are not meant to pay "validation" (as it happens in PoW blockchains). In Algorand you can **delegate fees**.
3. **Round validity**: to handle transactions' idempotency, letting Non-Archival nodes participate in Algorand Consensus, transactions have an intrinsic validity of 1000 blocks (at most).

Browse through a transaction

Transactions are characterized by two kind of **fields** (codec):

- **common** (header)
- **specific** (type)

Field	Required	Type	codec	Description
Fee	*required*	uint64	`"fee"`	Paid by the sender to the FeeSink to prevent denial-of-service. The minimum fee on Algorand is currently 1000 microAlgos.
FirstValid	*required*	uint64	`"fv"`	The first round for when the transaction is valid. If the transaction is sent prior to this round it will be rejected by the network.
GenesisHash	*required*	[32]byte	`"gh"`	The hash of the genesis block of the network for which the transaction is valid. See the genesis hash for MainNet, TestNet, and BetaNet.
LastValid	*required*	uint64	`"lv"`	The ending round for which the transaction is valid. After this round, the transaction will be rejected by the network.
				the account that pays the fee and amount.
				ype of transaction. This value is automatically generated e developer tools.

Field	Required	Type	codec	Description
Receiver	*required*	Address	`"rcv"`	The address of the account that receives the amount.
Amount	*required*	uint64	`"amt"`	The total amount to be sent in microAlgos.
CloseRemainderTo	*optional*	Address	`"close"`	When set, it indicates that the transaction is requesting that the Sender account should be closed, and all remaining funds, after the fee and amount are paid, be transferred to this address.

Payment Transaction example

Here is a transaction that sends 5 ALGO from one account to another on MainNet.

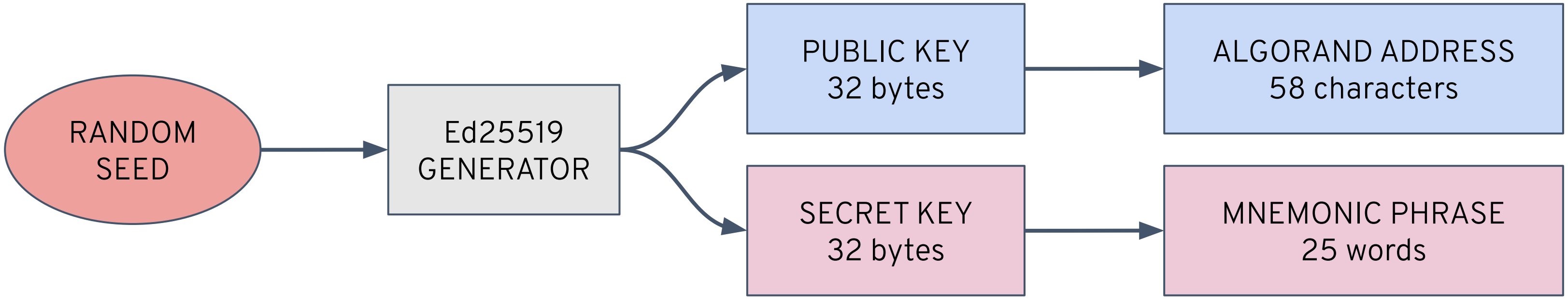

```json
{
  "txn": {
    "amt": 5000000,
    "fee": 1000,
    "fv": 6000000,
    "gen": "mainnet-v1.0",
    "gh": "wGHE2Pwdvd7S12BL5FaOP20EGYesN73ktiC1qzkkit8=",
    "lv": 6001000,
    "note": "SGVsbG8gV29ybGQ=",
    "rcv": "GD64YIY3TWGDMCNPP553DZPPR6LDUSFQOIJVFDPPXWEG3FVOJCCDBBHU5A",
    "snd": "EW64GC6F24M7NDSC5R3ES4YUVE3ZXXNMARJHDCCCLIHZU6TBEOC7XRSBG4",
    "type": "pay"
  }
}
```

ALGORAND ACCOUNTS

Transactions' Authorization

Signatures

Algorand uses **Ed25519** high-speed, high-security elliptic-curve signatures.

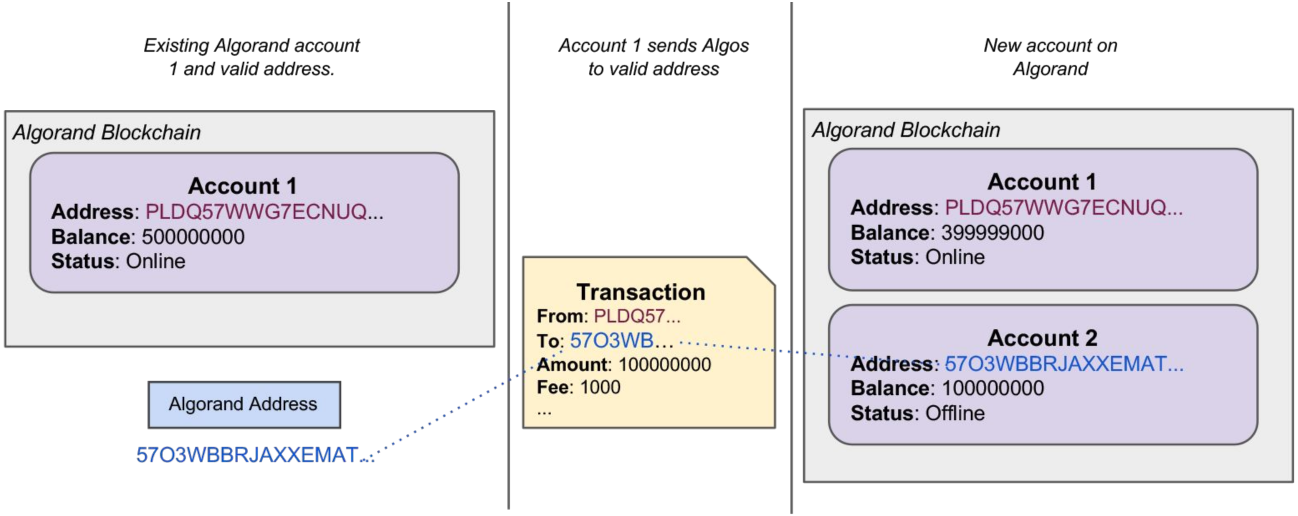


- **ADDRESS:** the **public key** is transformed into an **Algorand Address**, by adding a 4-byte checksum to the end of the public key and then **encoding it in base32**.

- **MNEMONIC:** the 25-word **mnemonic** is generated by converting the **private key bytes** into **11-bit integers** and then **mapping** those integers to the bip-0039 English word list.

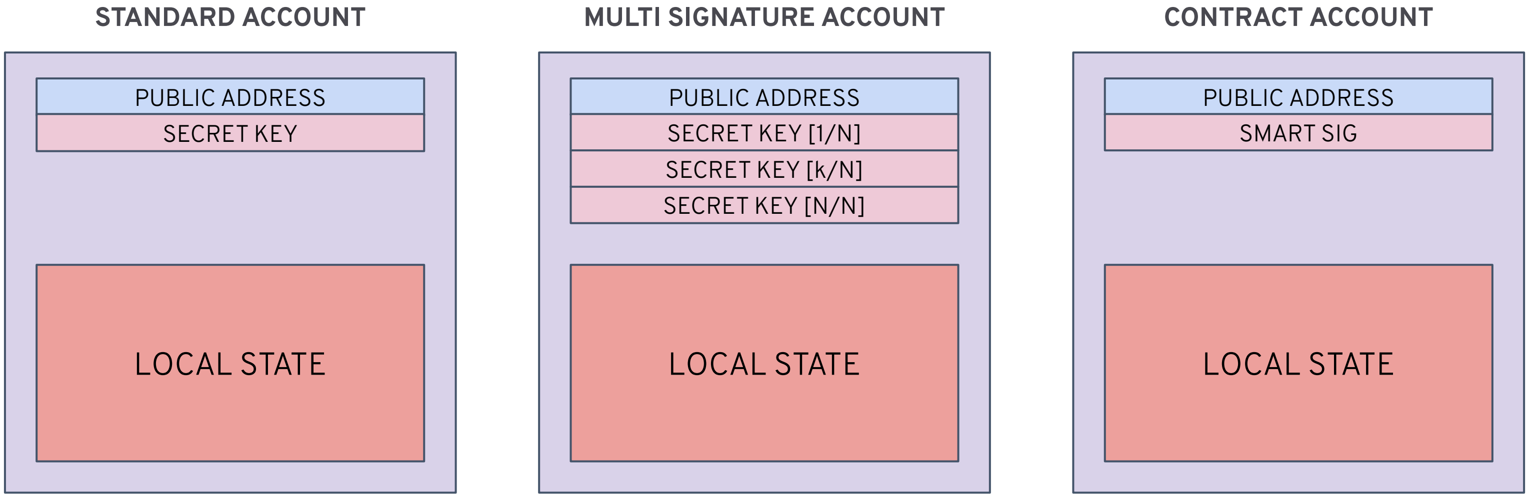
Algorand Accounts

Accounts are **entities on the Algorand blockchain** associated with specific **on-chain local sate**. An **Algorand Address** is the **unique identifier** for an **Algorand Account**.



All the potential keys pairs "already exists" mathematically, we just keep discovering them.

Transactions Authorization and Rekey-To



Algorand Rekeying: powerful **Layer-1 protocol feature** which enables an Algorand account to **maintain a static public address** while dynamically **rotating the authoritative private spending key(s)**. Any Account can Rekey either to a Standard Account, MultiSig Account or LogicSig Contract Account.

Algorand

ALGORAND VIRTUAL MACHINE (AVM)

Programming on Algorand

What's a *Smart Contract* ?

Smart Contracts are **deterministic** programs through which complex **decentralized** trustless **applications** can be executed on the **AVM**.

What's the *AVM* ?

The **Algorand Virtual Machine** is a **Turing-complete** secure **execution environment** that runs on Algorand **consensus layer**.

What the AVM *actually* does?

Algorand Virtual Machine purpose: **approving** or **rejecting** transactions' effects **on the blockchain** according to Smart Contracts' logic.

AVM **approves** transactions' effects if and only if:

1. There is <u>a single non-zero value</u> on top of AVM's stack;

AVM **rejects** transactions' effects if and only if:

1. There is <u>a single zero value</u> on top of AVM's stack;
2. There are <u>multiple values</u> on the AVM's stack;
3. There is <u>no value</u> on the AVM's stack;

Algorand

How the AVM *works*?

Suppose we want the AVM to **check** the following assertion:

$$1 + 2 = 3$$

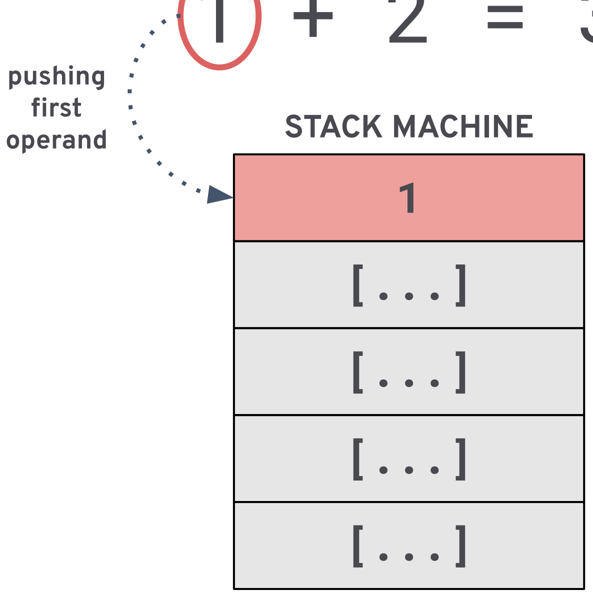
STACK MACHINE

[...]
[...]
[...]
[...]
[...]

How the AVM *works*?

Suppose we want the AVM to **check** the following assertion:

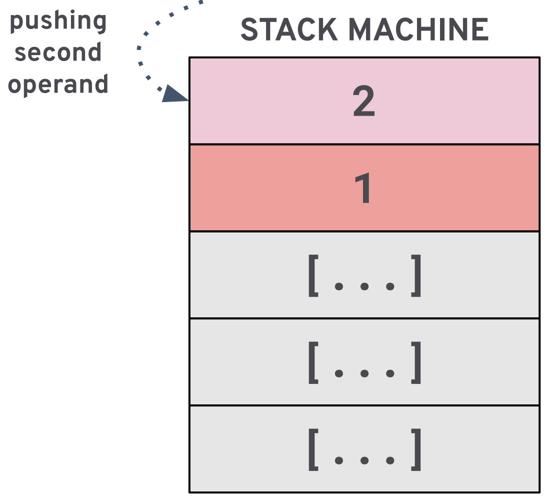
$$1 + 2 = 3$$



How the AVM *works*?

Suppose we want the AVM to **check** the following assertion:

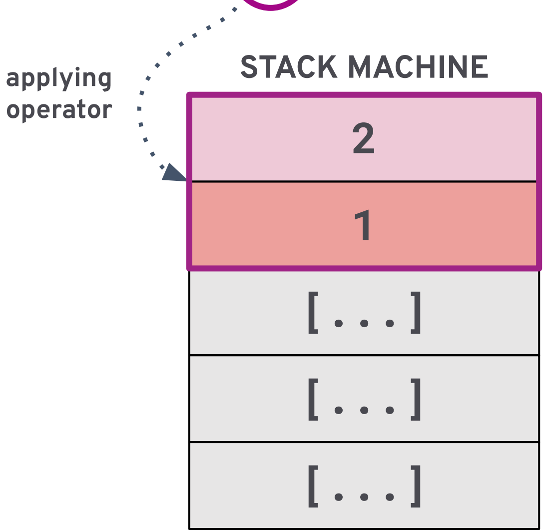
$$1 + 2 = 3$$



How the AVM *works*?

Suppose we want the AVM to **check** the following assertion:

$$1 + 2 = 3$$



Algorand

How the AVM *works*?

Suppose we want the AVM to **check** the following assertion:

$$1 + 2 = 3$$

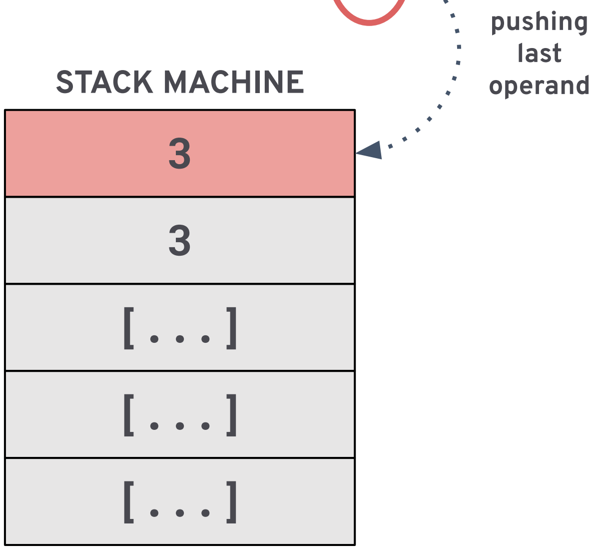
STACK MACHINE

3
[...]
[...]
[...]
[...]

How the AVM *works*?

Suppose we want the AVM to **check** the following assertion:

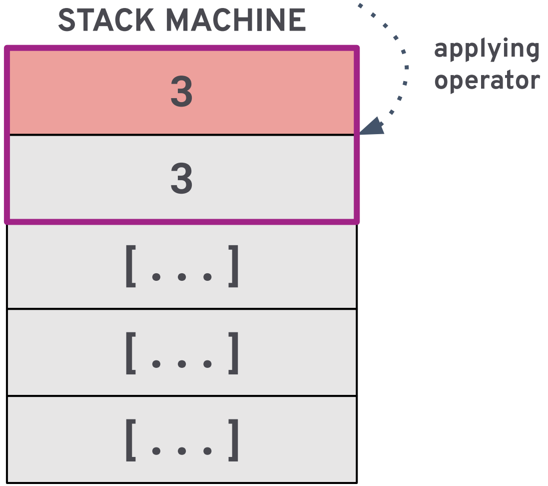
$$1 + 2 = 3$$



How the AVM *works*?

Suppose we want the AVM to **check** the following assertion:

$$1 + 2 = 3$$



How the AVM *works*?

Suppose we want the AVM to **check** the following assertion:

$$1 + 2 = 3$$

STACK MACHINE

`true`
`[...]`
`[...]`
`[...]`
`[...]`

AVM architecture

TRANSACTION

1. Sender
2. Receiver
3. Fee
4. FirstValid
5. LastValid
6. Amount
7. Lease
8. Note
9. TypeEnum
10. …

TRANSACTION ARGS

[0]:	Bytes
[i]:	Bytes
[255]:	Bytes

Stateless properties

APP ARG ARRAY

[0]:	UInt64/ Bytes
[i]:	UInt64/ Bytes
[15]:	UInt64/ Bytes

APP GLOBAL K/V PAIRS

[0]:	UInt64 / Bytes
[i]:	UInt64 / Bytes
[63]:	UInt64 / Bytes

ACCOUNT ARRAY

[0]:	Bytes
[i]:	Bytes
[3]:	Bytes

APP LOCAL K/V PAIRS

[0]:	UInt64 / Bytes
[i]:	UInt64 / Bytes
[15]:	UInt64 / Bytes

ASSET ARRAY

[0]:	UInt64
[1]:	UInt64

APP IDs ARRAY

[0]:	UInt64
[1]:	UInt64

Stateful properties

PROGRAM

```
txn TypeEnum
int 1
==
txn Receiver
addr AOC7...
==
&&
txn Fee
int 1000
<=
&&
arg 0
byte base64 "YmlhbmNvbmlnbGlv"
==
&&
txn Amount
int 42
==
txn Amount
int 77
==
||
&&
```

Processing

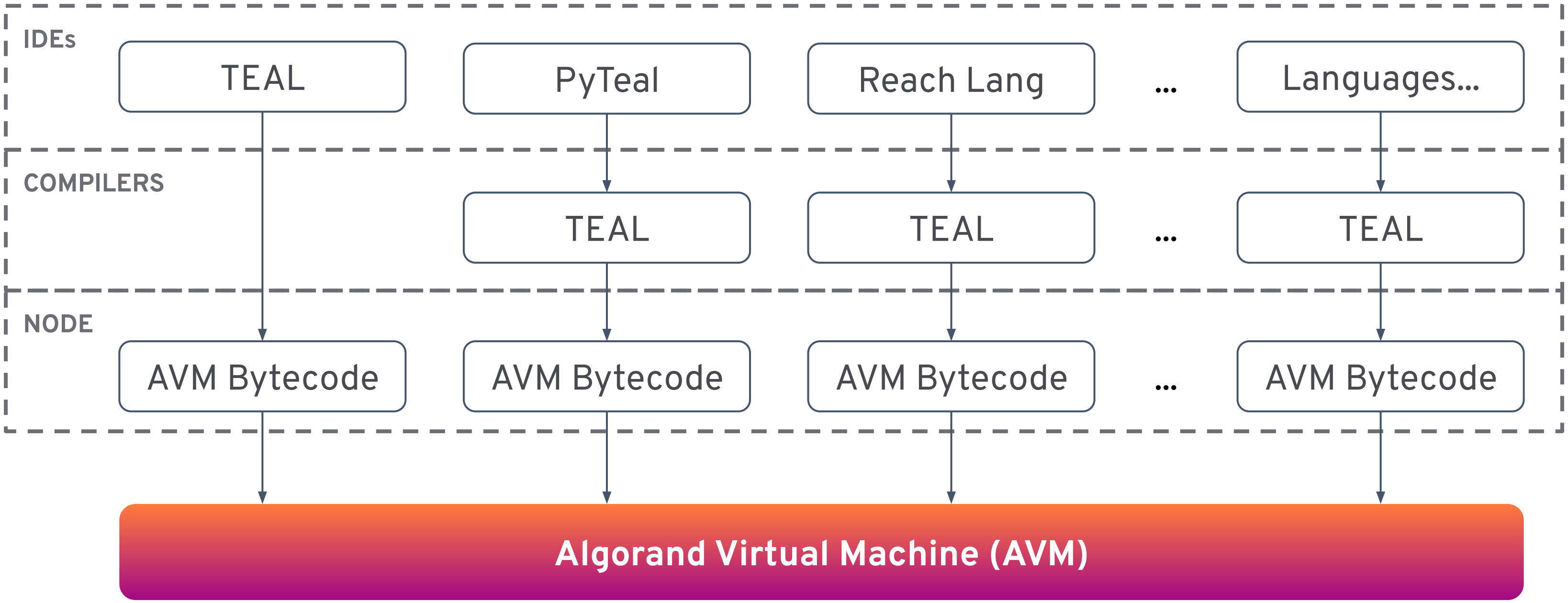
STACK MACHINE

[0]:	UInt64 / Bytes
[i]:	UInt64 / Bytes
[999]:	UInt64 / Bytes

SCRATCH SPACE

[0]:	UInt64 / Bytes
[i]:	UInt64 / Bytes
[255]:	UInt64 / Bytes

How to program the AVM?



Latest release: AVM 0.9

- The AVM now supports **looping** and **subroutines**.
- **Sharing data** between SCs when **combing contract calls** in Atomic Transfers.
- Delegating transaction fees using **pooled transaction fees** in Atomic Transfers.
- AVM **dynamic opcode cost evaluation**, allowing larger and more modular SCs.
- **Much larger** Stateful Smart Contracts (up to 8kb of program).
- More **versatile application transaction array** indexes.
- **Customizable** global and local state **key/value pairs**, to **maximize storage**.
- **Larger URLs** for Algorand Standard Assets (up to 96 bytes).
- More precision: **512 bit math operations** for byteslice arithmetic.
- Many **additional TEAL opcodes** (including more advanced math operators).

AVM vs EVM

	Algorand Virtual Machine	Ethereum Virtual Machine
TURING COMPLETENESS	YES	YES
EXECUTION SPEED	~ 4.5 sec regardless dApp complexity	> 20 sec depends on dApp complexity
ENERGY EFFICIENCY	0.000008 [kWh/txn] all final	120 [kWh/txn] not all final
EXECUTION COSTS	~ 0.001 $ (public network) regardless dApp complexity	~ 20 $ (public network) depends on dApp complexity
INTEROPERABILITY	native interoperability ASA, AT, MultiSig, RekeyTo...	user defined solutions / standards
EFFECTS FINALITY	instant	~ 6 blocks
MATHEMATICAL PRECISION	512 bits	256 bits
PROGRAMMABILITY	TEAL, PyTEAL, Reach, ...	Solidity, Viper, Reach, ...

What can be built on the AVM?

- Escrow accounts
- KYC processes
- Financial instruments (Bonds, ETFs, etc.)
- Loan payment
- Voting applications
- Auctions
- Multiparty or Delegated fund management
- Programmatic recurring fees / recurring debt
- And more…

Algorand

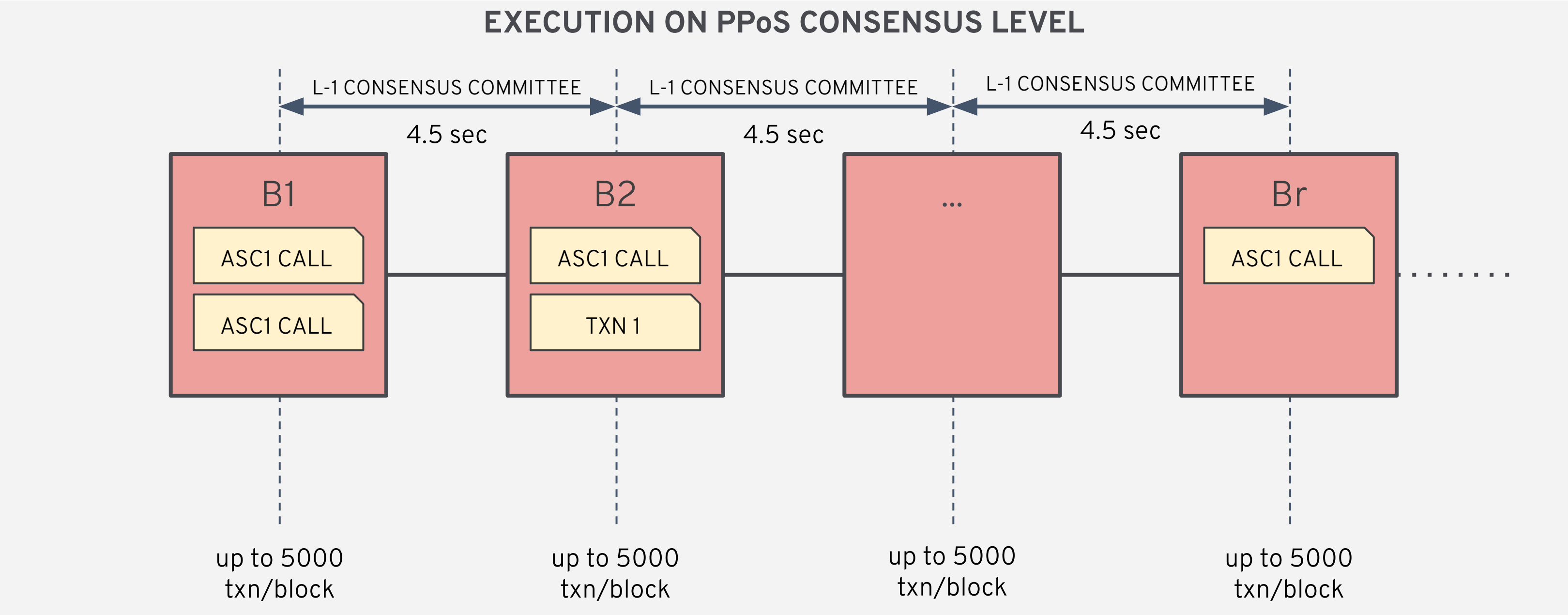
ASC1

Algorand Smart Contracts on Layer-1

What does *execution on Layer-1* mean?

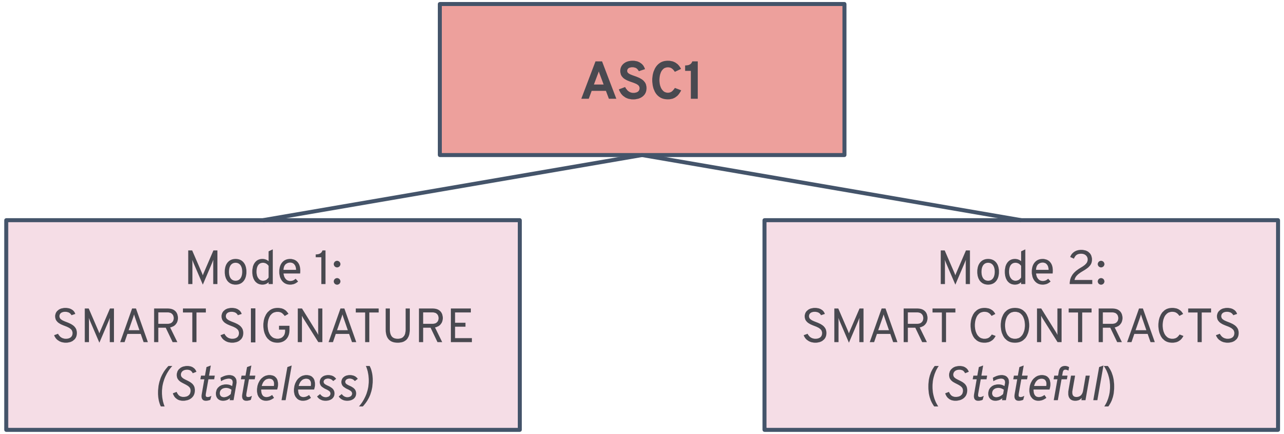
- Smart Contracts are executed *“at consensus level”*
- Benefit from network's speed, security, and scalability
- Fast trustless execution (~4.5 seconds per block)
- Low cost execution (0.001 ALGO regardless SC’s complexity)
- Instant Finality of Smart Contracts’ effects
- Native interoperability with Layer-1 primitives
- Safe high level languages (PyTeal, Reach, Clarity)
- Low energy consumption

What does *execution on Layer-1* mean?

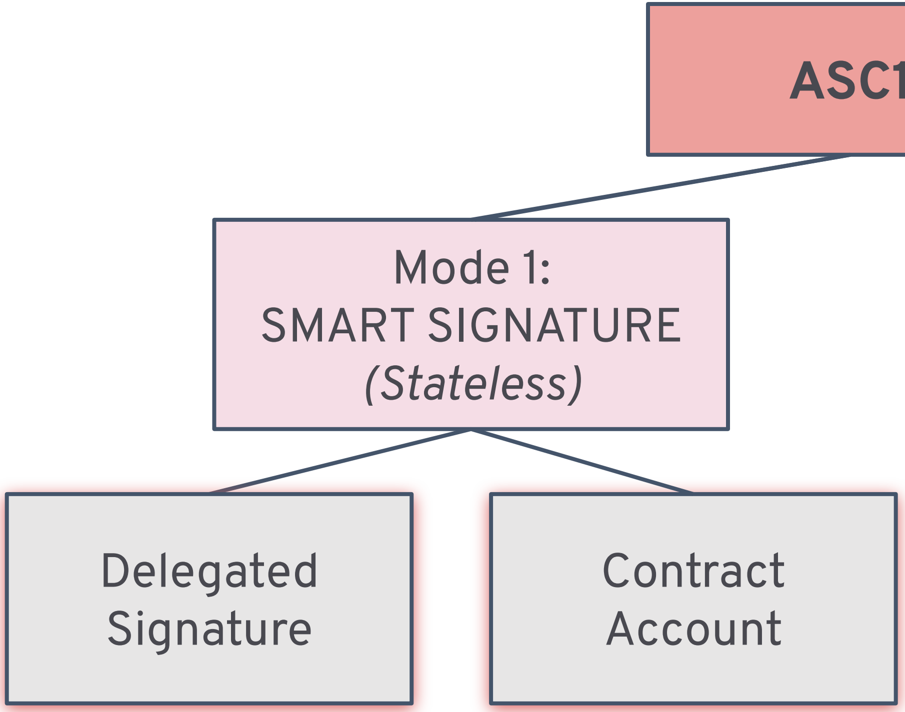


- ASC1 execution does not slow down the whole blocks production

AVM Modes


ASC1
Mode 1:
SMART SIGNATURE
(*Stateless*)
Mode 2:
SMART CONTRACTS
(*Stateful*)

Stateless ASC1

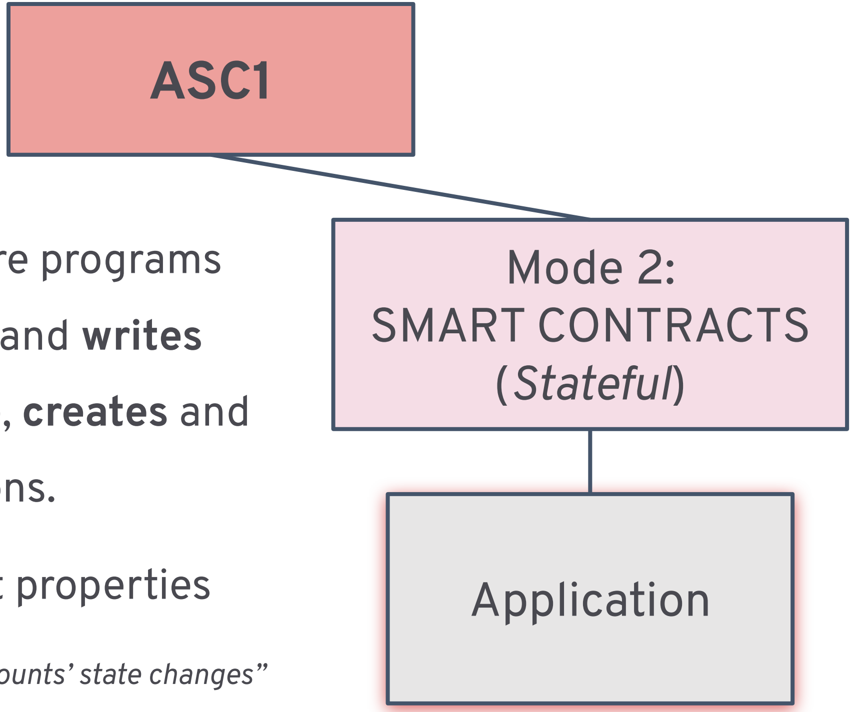


Smart Signatures are programs whose logic **governs transactions' authorization.**

Domain: transient properties

"Stateless ASC1 deal with assets' spending approvals"

Stateful ASC1



Smart Contracts are programs whose logic **reads** and **writes** **blockchain's state**, **creates** and **submits** transactions.

Domain: persistent properties

"Stateful ASC1 deal with accounts' state changes"

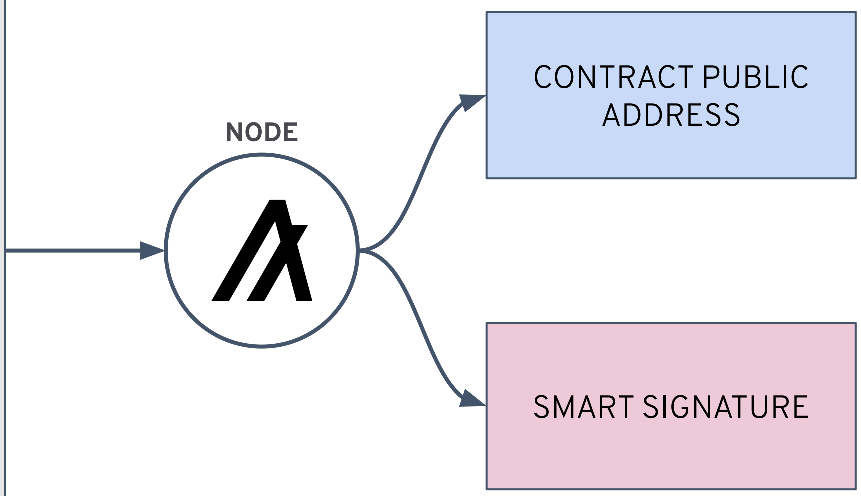
SMART SIGNATURES

Authorizing transactions through TEAL logic

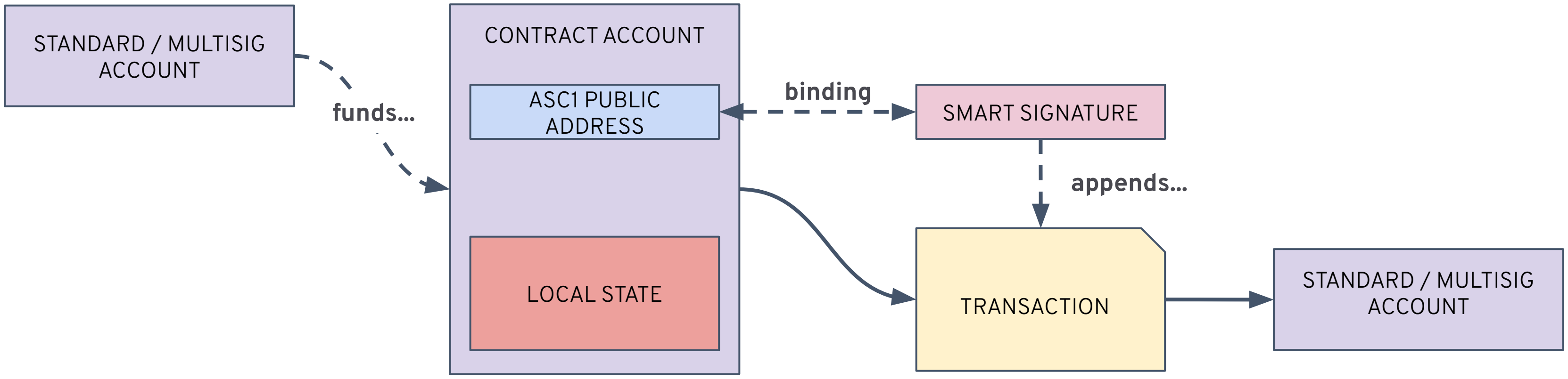
Creating Smart Signature

PROGRAM

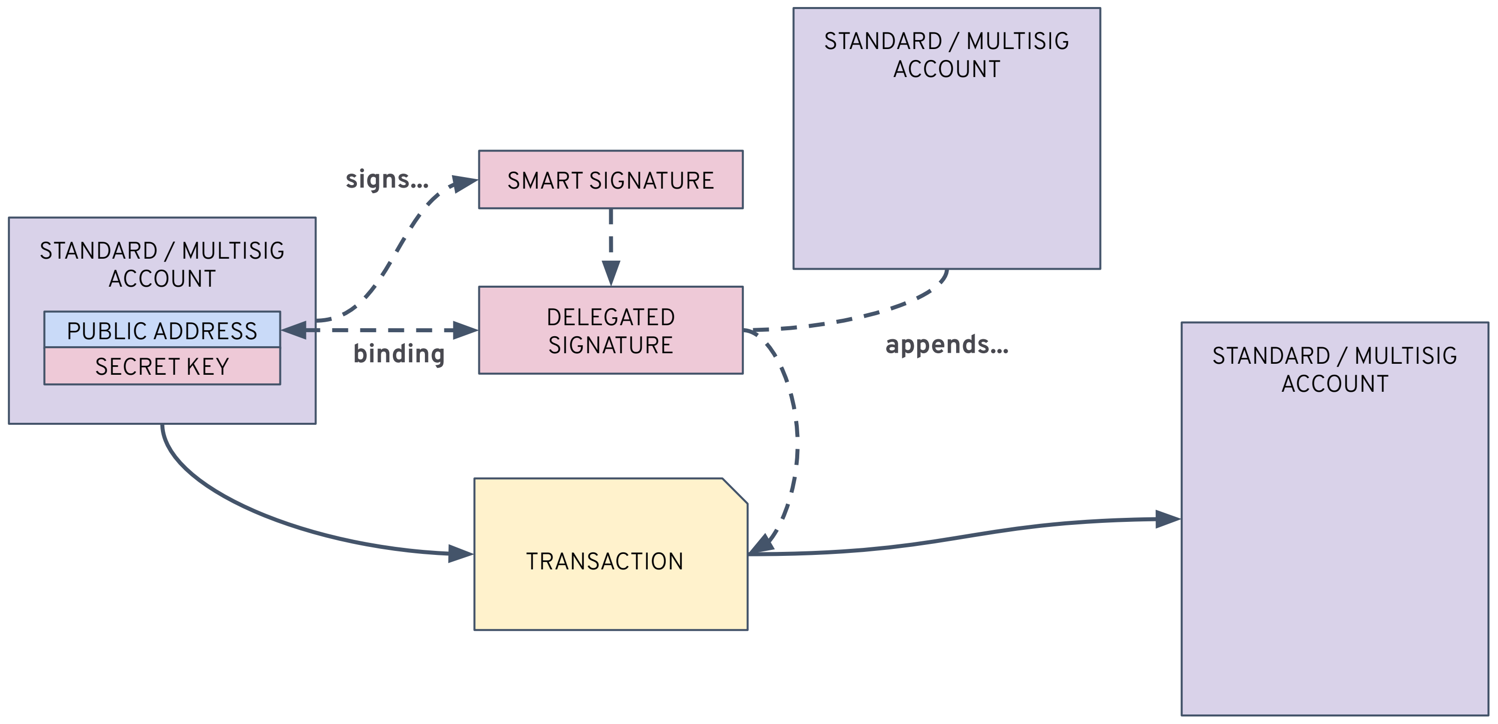
```
txn TypeEnum
int 1
==
txn Receiver
addr AOC7...
==
&&
txn Fee
int 1000
<=
&&
arg 0
byte base64 "YmlhbmNvbmlnbGlv"
==
&&
txn Amount
int 42
==
txn Amount
int 77
==
||
&&
```



Contract Account



Delegated Signature



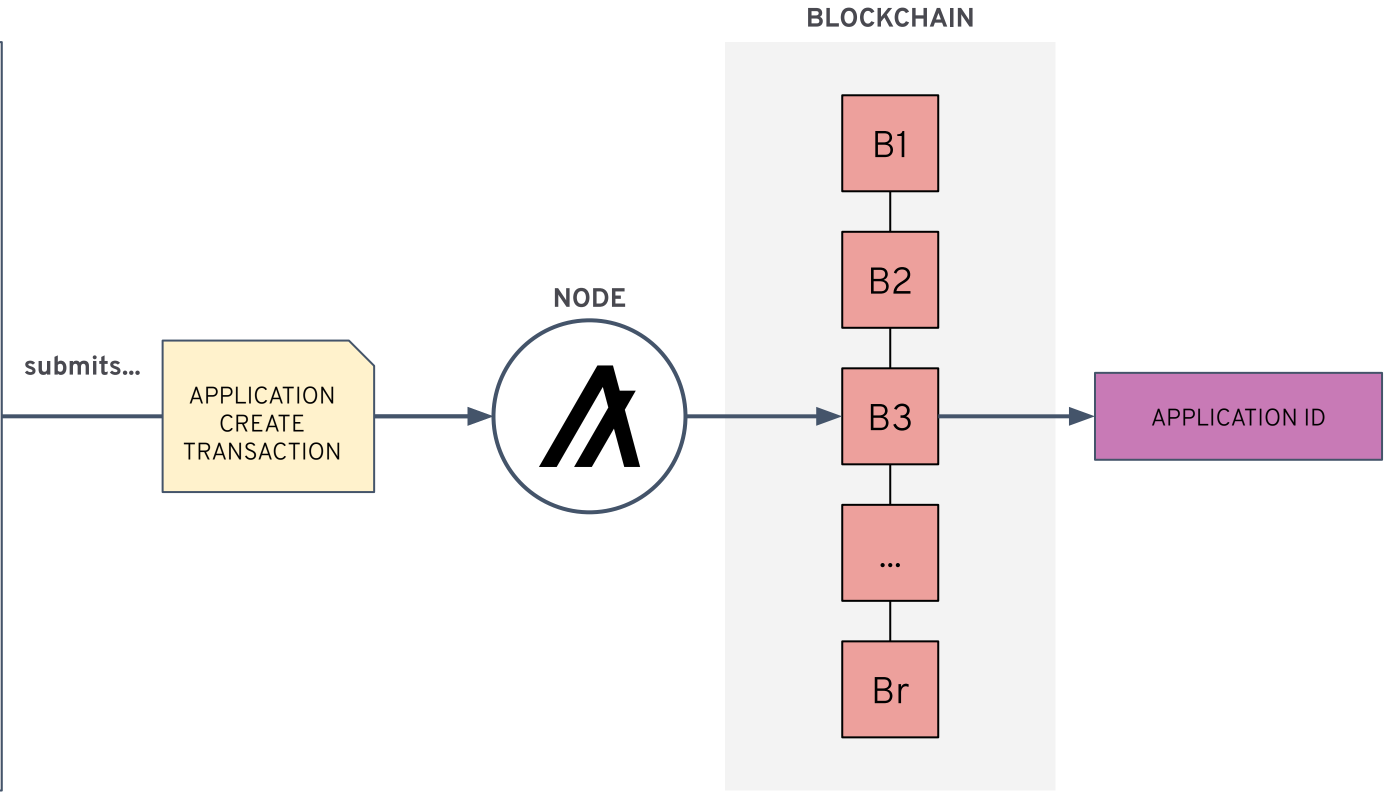
SMART CONTRACTS

Decentralized Applications on Algorand

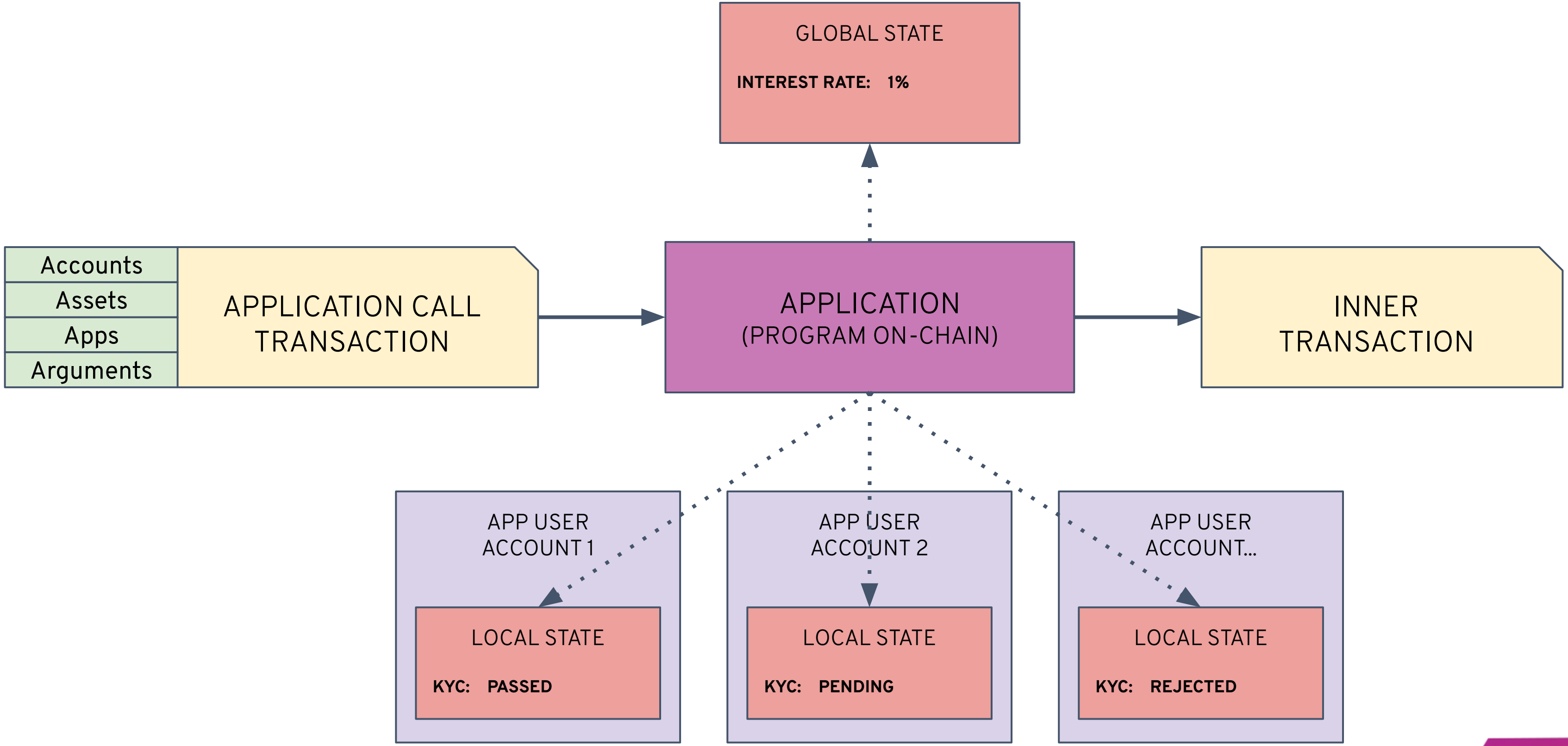
Deploying Applications on chain

PROGRAM

```
txn TypeEnum
int 1
==
txn Receiver
addr AOC7...
==
&&
txn Fee
int 1000
<=
&&
arg 0
byte base64 "YmlhbmNvbmlnbGlv"
==
&&
txn Amount
int 42
==
txn Amount
int 77
==
||
&&
```



Interacting with Applications



TEAL

AVM assembly-like language

Smart Signature Example

Suppose we want to develop a **Smart Signature** that approves a transaction **if and only if**:

1. is "*Payment*" **type** transaction;
2. the **receiver** is a specific "*ADDR*";
3. **fees** are less or equal to "*1000 microALGO*";
4. first **argument** is equal to "*bianconiglio*";
5. **amount** is equal to "*42 ALGO*";
6. or **amount** is equal to "*77 ALGO*";

Where do we start?

Smart Signature as “Transaction Observer”

Smart Signatures can be defined as a *“transactions’ observers”*: programs that meticulously check all **fields in the transaction** (or in a group of transactions) that intend to *“approve”* or *“reject”* based on TEAL logic.

To translate those 6 **semantically defined** example’s conditions into **TEAL** we need to check which transaction fields are going to be controlled by Smart Signature’s logic.

Let's start with the translation…

Translating conditions into TEAL...

1. is "*Payment*" **type** transaction;

TxType	*required*	string	`"type"`	Specifies the type of transaction. This value is automatically generated using any of the developer tools.

```
txn TypeEnum
int 1
==
```

1

Translating conditions into TEAL...

2. the **receiver** is a specific “*ADDR*”;

Receiver	*required*	Address	"rcv"	The address of the account that receives the amount.

```
txn Receiver
addr AOC7...
==
```

2

Translating conditions into TEAL...

3. **fees** are less or equal to "*1000 microALGO*";

Fee	*required*	uint64	`"fee"`	Paid by the sender to the FeeSink to prevent denial-of-service. The minimum fee on Algorand is currently 1000 microAlgos.

3

```
txn Fee
int 1000
<=
```

Translating conditions into TEAL...

4. first **argument** is equal to “*bianconiglio*”;

`arg`	push Args[N] value to stack by index

4

```
arg 0
byte base64 "YmlhbmNvbmlnbGlv"
==
```

Translating conditions into TEAL...

5. **amount** is equal to “*42 ALGO*”;

Amount	*required*	uint64	`"amt"`	The total amount to be sent in microAlgos.

```
txn Amount
int 42000000
==
```

5

Translating conditions into TEAL...

6. **amount** is equal to "*77 ALGO*";

Amount	*required*	uint64	"amt"	The total amount to be sent in microAlgos.

```
6
txn Amount
int 77000000
==
```

Logic connectors...

```
txn TypeEnum
int 1
==
```
1

```
txn Receiver
addr AOC7...
==
```
2

```
txn Fee
int 1000
<=
```
3

```
arg 0
byte base64 "YmlhbmNvbmlnbGlv"
==
```
4

```
txn Amount
int 42000000
==
```
5

```
txn Amount
int 77000000
==
```
6

STACK

[...]
[...]
[...]
[...]
[...]

Algorand

Logic connectors...

```
txn TypeEnum
int 1                                1
==
txn Receiver
addr AOC7...                         2
==
&&
txn Fee
int 1000                             3
<=
arg 0
byte base64 "YmlhbmNvbmlnbGlv"       4
==
&&
&&
txn Amount
int 42000000                         5
==
txn Amount
int 77000000                         6
==
||
&&
```

This is probably the most complex phase in **TEAL** programming, because you need to keep in mind the **state of the stack**.

This phase is drastically simplified with the use of **PyTEAL**, Python binding for TEAL, which automatically performs this concatenation, saving us the effort of thinking about the state of the stack.

STACK

[...]
[...]
[...]
[...]
[...]

Execution...

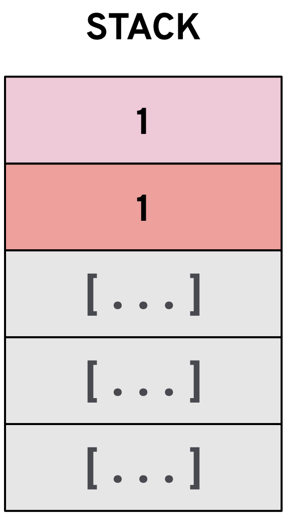
```
txn TypeEnum
int 1
==
txn Receiver
addr AOC7...
==
&&
txn Fee
int 1000
<=
arg 0
byte base64 "YmlhbmNvbmlnbGlv"
==
&&
&&
txn Amount
int 42000000
==
txn Amount
int 77000000
==
||
&&
```

STACK

1
[...]
[...]
[...]
[...]

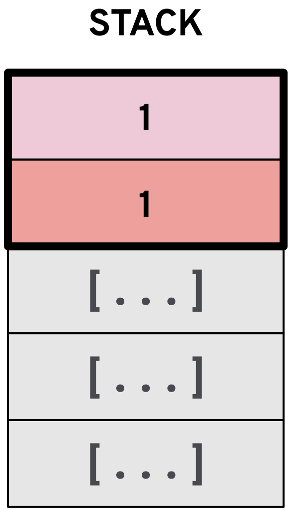
Execution...

```
txn TypeEnum
int 1
==
txn Receiver
addr AOC7...
==
&&
txn Fee
int 1000
<=
arg 0
byte base64 "YmlhbmNvbmlnbGlv"
==
&&
&&
txn Amount
int 42000000
==
txn Amount
int 77000000
==
||
&&
```



Execution...

```
txn TypeEnum
int 1
==
txn Receiver
addr AOC7...
==
&&
txn Fee
int 1000
<=
arg 0
byte base64 "YmlhbmNvbmlnbGlv"
==
&&
&&
txn Amount
int 42000000
==
txn Amount
int 77000000
==
||
&&
```



Execution...

```
txn TypeEnum
int 1
==
txn Receiver
addr AOC7...
==
&&
txn Fee
int 1000
<=
arg 0
byte base64 "YmlhbmNvbmlnbGlv"
==
&&
&&
txn Amount
int 42000000
==
txn Amount
int 77000000
==
||
&&
```

STACK

1
[...]
[...]
[...]
[...]

Algorand

Execution...

```
txn TypeEnum
int 1
==
txn Receiver
addr AOC7...
==
&&
txn Fee
int 1000
<=
arg 0
byte base64 "YmlhbmNvbmlnbGlv"
==
&&
&&
txn Amount
int 42000000
==
txn Amount
int 77000000
==
||
&&
```

STACK

AOC7...
1
[...]
[...]
[...]

Algorand

Execution...

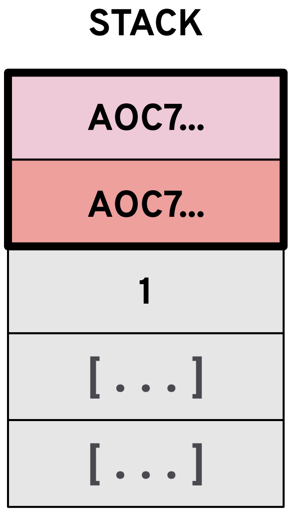
```
txn TypeEnum
int 1
==
txn Receiver
addr AOC7...
==
&&
txn Fee
int 1000
<=
arg 0
byte base64 "YmlhbmNvbmlnbGlv"
==
&&
&&
txn Amount
int 42000000
==
txn Amount
int 77000000
==
||
&&
```

STACK

AOC7...
AOC7...
1
[...]
[...]

Execution...

```
txn TypeEnum
int 1
==
txn Receiver
addr AOC7...
==
&&
txn Fee
int 1000
<=
arg 0
byte base64 "YmlhbmNvbmlnbGlv"
==
&&
&&
txn Amount
int 42000000
==
txn Amount
int 77000000
==
||
&&
```



Execution...

```
txn TypeEnum
int 1
==
txn Receiver
addr AOC7...
==
&&
txn Fee
int 1000
<=
arg 0
byte base64 "YmlhbmNvbmlnbGlv"
==
&&
&&
txn Amount
int 42000000
==
txn Amount
int 77000000
==
||
&&
```

STACK

1
1
[...]
[...]
[...]

Execution...

```
txn TypeEnum
int 1
==
txn Receiver
addr AOC7...
==
&&
txn Fee
int 1000
<=
arg 0
byte base64 "YmlhbmNvbmlnbGlv"
==
&&
&&
txn Amount
int 42000000
==
txn Amount
int 77000000
==
||
&&
```

STACK

1
1
[...]
[...]
[...]

Algorand

Execution...

```
txn TypeEnum
int 1
==
txn Receiver
addr AOC7...
==
&&
txn Fee
int 1000
<=
arg 0
byte base64 "YmlhbmNvbmlnbGlv"
==
&&
&&
txn Amount
int 42000000
==
txn Amount
int 77000000
==
||
&&
```

STACK

1
[...]
[...]
[...]
[...]

Execution...

```
txn TypeEnum
int 1
==
txn Receiver
addr AOC7...
==
&&
txn Fee
int 1000
<=
arg 0
byte base64 "YmlhbmNvbmlnbGlv"
==
&&
&&
txn Amount
int 42000000
==
txn Amount
int 77000000
==
||
&&
```

STACK

1000
1
[...]
[...]
[...]

Algorand

Execution...

```
txn TypeEnum
int 1
==
txn Receiver
addr AOC7...
==
&&
txn Fee
int 1000
<=
arg 0
byte base64 "YmlhbmNvbmlnbGlv"
==
&&
&&
txn Amount
int 42000000
==
txn Amount
int 77000000
==
||
&&
```

STACK

1000
1000
1
[...]
[...]

Execution...

```
txn TypeEnum
int 1
==
txn Receiver
addr AOC7...
==
&&
txn Fee
int 1000
<=
arg 0
byte base64 "YmlhbmNvbmlnbGlv"
==
&&
&&
txn Amount
int 42000000
==
txn Amount
int 77000000
==
||
&&
```

STACK

1000
1000
1
[...]
[...]

Execution...

```
txn TypeEnum
int 1
==
txn Receiver
addr AOC7...
==
&&
txn Fee
int 1000
<=
arg 0
byte base64 "YmlhbmNvbmlnbGlv"
==
&&
&&
txn Amount
int 42000000
==
txn Amount
int 77000000
==
||
&&
```

STACK

1
1
[...]
[...]
[...]

Execution...

```
txn TypeEnum
int 1
==
txn Receiver
addr AOC7...
==
&&
txn Fee
int 1000
<=
arg 0
byte base64 "YmlhbmNvbmlnbGlv"
==
&&
&&
txn Amount
int 42000000
==
txn Amount
int 77000000
==
||
&&
```

STACK

bianconiglio
1
1
[...]
[...]

Execution...

```
txn TypeEnum
int 1
==
txn Receiver
addr AOC7...
==
&&
txn Fee
int 1000
<=
arg 0
byte base64 "YmlhbmNvbmlnbGlv"
==
&&
&&
txn Amount
int 42000000
==
txn Amount
int 77000000
==
||
&&
```

STACK

bianconiglio
bianconiglio
1
1
[...]

Execution...

```
txn TypeEnum
int 1
==
txn Receiver
addr AOC7...
==
&&
txn Fee
int 1000
<=
arg 0
byte base64 "YmlhbmNvbmlnbGlv"
==
&&
&&
txn Amount
int 42000000
==
txn Amount
int 77000000
==
||
&&
```

STACK

bianconiglio
bianconiglio
1
1
[...]

Execution...

```
txn TypeEnum
int 1
==
txn Receiver
addr AOC7...
==
&&
txn Fee
int 1000
<=
arg 0
byte base64 "YmlhbmNvbmlnbGlv"
==
&&
&&
txn Amount
int 42000000
==
txn Amount
int 77000000
==
||
&&
```

STACK

1
1
1
[...]
[...]

Execution...

```
txn TypeEnum
int 1
==
txn Receiver
addr AOC7...
==
&&
txn Fee
int 1000
<=
arg 0
byte base64 "YmlhbmNvbmlnbGlv"
==
&&
&&
txn Amount
int 42000000
==
txn Amount
int 77000000
==
||
&&
```

STACK

1
1
1
[...]
[...]

Execution...

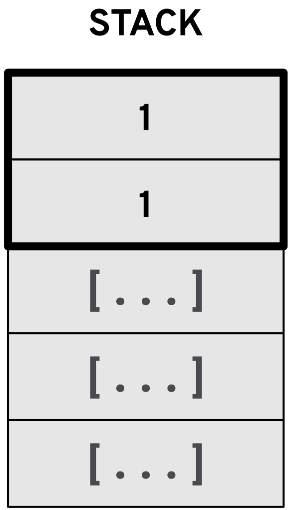
```
txn TypeEnum
int 1
==
txn Receiver
addr AOC7...
==
&&
txn Fee
int 1000
<=
arg 0
byte base64 "YmlhbmNvbmlnbGlv"
==
&&
&&
txn Amount
int 42000000
==
txn Amount
int 77000000
==
||
&&
```

STACK

1
1
[...]
[...]
[...]

Execution...

```
txn TypeEnum
int 1
==
txn Receiver
addr AOC7...
==
&&
txn Fee
int 1000
<=
arg 0
byte base64 "YmlhbmNvbmlnbGlv"
==
&&
&&
txn Amount
int 42000000
==
txn Amount
int 77000000
==
||
&&
```



Execution...

```
txn TypeEnum
int 1
==
txn Receiver
addr AOC7...
==
&&
txn Fee
int 1000
<=
arg 0
byte base64 "YmlhbmNvbmlnbGlv"
==
&&
&&
txn Amount
int 42000000
==
txn Amount
int 77000000
==
||
&&
```

STACK

1
[...]
[...]
[...]
[...]

Execution...

```
txn TypeEnum
int 1
==
txn Receiver
addr AOC7...
==
&&
txn Fee
int 1000
<=
arg 0
byte base64 "YmlhbmNvbmlnbGlv"
==
&&
&&
txn Amount
int 42000000
==
txn Amount
int 77000000
==
||
&&
```

STACK

42000000
1
[...]
[...]
[...]

Execution...

```
txn TypeEnum
int 1
==
txn Receiver
addr AOC7...
==
&&
txn Fee
int 1000
<=
arg 0
byte base64 "YmlhbmNvbmlnbGlv"
==
&&
&&
txn Amount
int 42000000
==
txn Amount
int 77000000
==
||
&&
```

STACK

42000000
42000000
1
[...]
[...]

Execution...

```
txn TypeEnum
int 1
==
txn Receiver
addr AOC7...
==
&&
txn Fee
int 1000
<=
arg 0
byte base64 "YmlhbmNvbmlnbGlv"
==
&&
&&
txn Amount
int 42000000
==
txn Amount
int 77000000
==
||
&&
```

STACK

42000000
42000000
1
[...]
[...]

Execution...

```
txn TypeEnum
int 1
==
txn Receiver
addr AOC7...
==
&&
txn Fee
int 1000
<=
arg 0
byte base64 "YmlhbmNvbmlnbGlv"
==
&&
&&
txn Amount
int 42000000
==
txn Amount
int 77000000
==
||
&&
```

STACK

1
1
[...]
[...]
[...]

Execution...

```
txn TypeEnum
int 1
==
txn Receiver
addr AOC7...
==
&&
txn Fee
int 1000
<=
arg 0
byte base64 "YmlhbmNvbmlnbGlv"
==
&&
&&
txn Amount
int 42000000
==
txn Amount
int 77000000
==
||
&&
```

STACK

42000000
1
1
[...]
[...]

Execution...

```
txn TypeEnum
int 1
==
txn Receiver
addr AOC7...
==
&&
txn Fee
int 1000
<=
arg 0
byte base64 "YmlhbmNvbmlnbGlv"
==
&&
&&
txn Amount
int 42000000
==
txn Amount
int 77000000
==
||
&&
```

STACK

77000000
42000000
1
1
[...]

Execution...

```
txn TypeEnum
int 1
==
txn Receiver
addr AOC7...
==
&&
txn Fee
int 1000
<=
arg 0
byte base64 "YmlhbmNvbmlnbGlv"
==
&&
&&
txn Amount
int 42000000
==
txn Amount
int 77000000
==
||
&&
```

STACK

77000000
42000000
1
1
[...]

Execution...

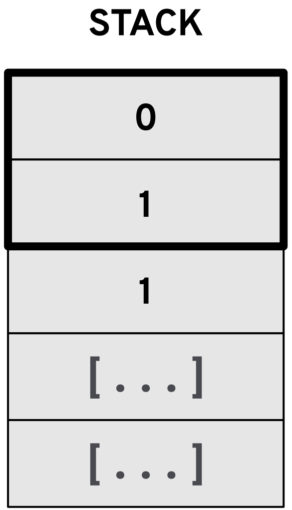
```
txn TypeEnum
int 1
==
txn Receiver
addr AOC7...
==
&&
txn Fee
int 1000
<=
arg 0
byte base64 "YmlhbmNvbmlnbGlv"
==
&&
&&
txn Amount
int 42000000
==
txn Amount
int 77000000
==
||
&&
```

STACK

0
1
1
[...]
[...]

Execution...

```
txn TypeEnum
int 1
==
txn Receiver
addr AOC7...
==
&&
txn Fee
int 1000
<=
arg 0
byte base64 "YmlhbmNvbmlnbGlv"
==
&&
&&
txn Amount
int 42000000
==
txn Amount
int 77000000
==
||
&&
```



Execution...

```
txn TypeEnum
int 1
==
txn Receiver
addr AOC7...
==
&&
txn Fee
int 1000
<=
arg 0
byte base64 "YmlhbmNvbmlnbGlv"
==
&&
&&
txn Amount
int 42000000
==
txn Amount
int 77000000
==
||
&&
```

STACK

1
1
[...]
[...]
[...]

Execution...

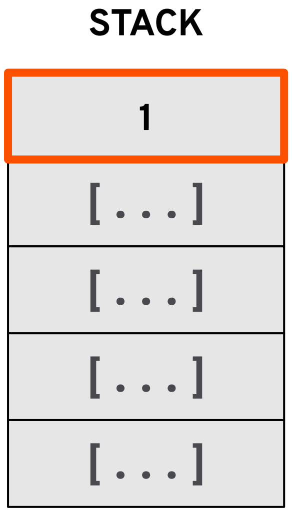
```
txn TypeEnum
int 1
==
txn Receiver
addr AOC7...
==
&&
txn Fee
int 1000
<=
arg 0
byte base64 "YmlhbmNvbmlnbGlv"
==
&&
&&
txn Amount
int 42000000
==
txn Amount
int 77000000
==
||
&&
```

STACK

1
1
[...]
[...]
[...]

Execution...

```
txn TypeEnum
int 1
==
txn Receiver
addr AOC7...
==
&&
txn Fee
int 1000
<=
arg 0
byte base64 "YmlhbmNvbmlnbGlv"
==
&&
&&
txn Amount
int 42000000
==
txn Amount
int 77000000
==
||
&&
```



Conclusion...

```
txn TypeEnum
int 1
==
txn Receiver
addr AOC7...
==
&&
txn Fee
int 1000
<=
arg 0
byte base64 "Ymlh[illegible]N[illegible]lnbG1v"
==
&&
&&
txn Amount
int 42000000
==
txn Amount
int 77000000
==
||
&&
```

STACK

1
[...]
[...]
[...]
[...]

True

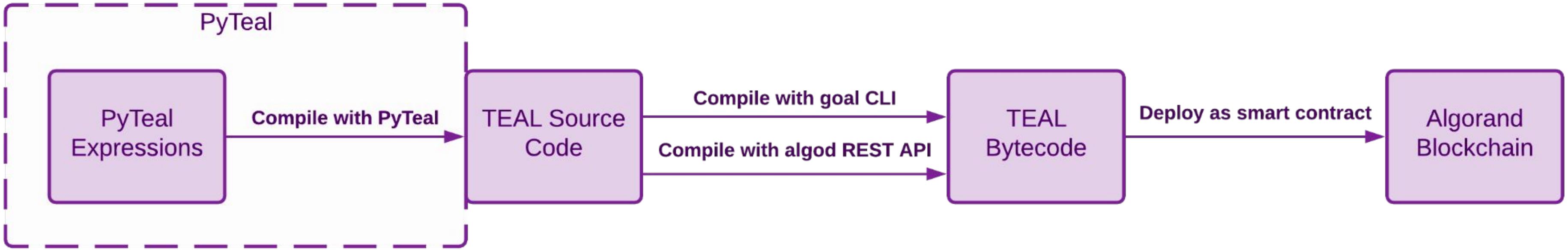
PyTEAL

Writing Smart Contracts with Python

What's *PyTEAL*?

PyTEAL is a **Python** language binding for **Algorand Virtual Machine**.

PyTEAL allows **Smart Contracts** and **Smart Signatures** to be written in Python and then compiled to **TEAL**.



Credits to [Jason Paulos]() for the [PyTEAL presentation]()

It’s easier with PyTEAL!

TEAL Source Code

```
txn Receiver
addr AOC7...
==
txn Amount
int 1000
<=
&&
```

COMPILE...

AVM bytecode

PyTEAL Source Code

```
And(
        Txn.Receiver == Addr(AOC7...),
        Txn.Amount <= Int(1000),
)
```

COMPILE...

TEAL Source Code

```
txn Receiver
addr AOC7...
==
txn Amount
int 1000
<=
&&
```

COMPILE...

AVM bytecode

Smart Signature written in PyTEAL

```python
import base64
from pyteal import *

"""Esempio ASC1 in PyTeal"""


def asc1(tmpl_receiver):
    is_payment = Txn.type_enum() == Int(1)
    is_correct_receiver = Txn.receiver() == Addr(tmpl_receiver)
    max_fee = Txn.fee() <= Int(1000)
    follow_the = Arg(0) == Bytes(
        "base64", str(base64.b64encode('bianconiglio'.encode()), 'utf-8'))
    amount_option1 = Txn.amount() == Int(42000000)
    amount_option2 = Txn.amount() == Int(77000000)
    is_correct_amount = Or(amount_option1, amount_option2)
    asc1_logic = And(is_payment,
                     is_correct_receiver,
                     max_fee,
                     follow_the,
                     is_correct_amount)
    return asc1_logic
```

PyTEAL Basics - Intro

PyTEAL expressions represent an **abstract syntax tree** (AST)

You're writing **Python code** that produces **TEAL code**.

```
from pyteal import *

program = ...
teal_source = compileTeal(program, mode=Mode.Application, version=5)
```

PyTEAL Basics - Types (1/2)

Two basic types:

- **uint64**
- **byte strings**

```
i = Int(5)
x = Bytes("content")
y = Bytes(b"\x01\x02\x03")
z = Bytes("base16", "05")
```

PyTEAL Basics - Types (2/2)

Conversion between types

- **Itob** - integer to bytes (8-byte big endian)
- **Btoi** - bytes to integer

```
Itob(i) # produces the byte string 0x0000000000000005
Btoi(z) # produces the integer 5
```

PyTEAL Basics - Math operators

Python **math** operators

```
i = Int(10)
j = i * Int(2) + Int(1)
k = And(Int(1), Or(Int(1), Int(0)))
```

PyTEAL Basics - Byte string manipulation

Byte string **manipulation**

```
x = Bytes("content")
y = Concat(Bytes("example "), x) # "example content"
z = Substring(y, Int(2), Len(y)) # "ample content"
```

PyTEAL Basics - Crypto utilities

Built-in **crypto** utilities

```
h_sha256 = Sha256(z)
h_sha512_256 = Sha512_256(z)
h_keccak = Keccak256(z)
```

PyTEAL Basics - Fields (1/3)

Fields from the **current transaction**

```
Txn.sender()
Txn.accounts.length()
Txn.application_args.length()
Txn.accounts[1]
Txn.application_args[0]
Txn.group_index()
```

PyTEAL Basics - Fields (2/3)

Fields from transactions in the **current atomic group**

```
Gtxn[0].sender()
Gtxn[Txn.group_index() - Int(1)].sender()
Gtxn[Txn.group_index() - Int(1)].accounts[2]
```

PyTEAL Basics - Fields (3/3)

Fields from **execution context**

```
Global.group_size()
Global.round() # current round number
Global.latest_timestamp() # UNIX timestamp of last round
```

PyTEAL Basics - Logs

Log publicly viewable messages to the chain

```
Log(Bytes("message"))
```

PyTEAL Basics - State (1/2)

Global - one instance per application

```
App.globalPut(Bytes("status"), Bytes("active")) # write to global key "status"

status = App.globalGet(Bytes("status")) # read global key "status"

App.globalDel(Bytes("status")) # delete global key "status"
```

PyTEAL Basics - State (2/2)

Local - one instance per opted-in account per application

```
App.localPut(Txn.sender(), Bytes("level"), Int(1)) # write to sender's local key
"level"
App.localPut(Txn.accounts[1], Bytes("level"), Int(2))  # write to other account's
local key "level"

sender_level = App.localGet(Txn.sender(), Bytes("level"))  # read from sender's
local key "level"

App.localDel(Txn.sender(), Bytes("level"))  # delete sender's local key "level"
```

PyTEAL Basics - Control Flow (1/5)

Approve the transaction and **immediately exit**

```
Approve()
```

Reject the transaction and **immediately exit**

```
Reject()
```

PyTEAL Basics - Control Flow (2/5)

Multiple expressions can be joined into a **sequence**

```
program = Seq(
    App.globalPut(Bytes("count"), App.globalGet(Bytes("count")) + Int(1)),
    Approve()
)
```

PyTEAL Basics - Control Flow (3/5)

Basic conditions can be expressed with **If**, **Then**, **Else**, **ElseIf**

```
program = Seq(
    If(App.globalGet(Bytes("count")) == Int(100))
    .Then(
        App.globalPut(Bytes("100th caller"), Txn.sender())
    )
    .Else(
        App.globalPut(Bytes("not 100th caller"), Txn.sender())
    ),
    App.globalPut(Bytes("count"), App.globalGet(Bytes("count")) + Int(1)),
    Approve(),
)
```

PyTEAL Basics - Control Flow (4/5)

Larger conditions can be expressed with **Cond**

```
program = Cond(
    [Txn.application_id() == Int(0), on_create],
    [Txn.on_completion() == OnComplete.UpdateApplication, on_update],
    [Txn.on_completion() == OnComplete.DeleteApplication, on_delete],
    [Txn.on_completion() == OnComplete.OptIn, on_opt_in],
    [Txn.on_completion() == OnComplete.CloseOut, on_close_out],
    [Txn.on_completion() == OnComplete.NoOp, on_noop],
    # error if no conditions are met
)
```

PyTEAL Basics - Control Flow (5/5)

Loops can be expressed with **For** and **While**

```
i = ScratchVar(TealType.uint64)

on_create = Seq(
    For(i.store(Int(0)), i.load() < Int(16), i.store(i.load() + Int(1)))
    .Do(
        App.globalPut(Concat(Bytes("index"), Itob(i.load())), Int(1))
    ),
    Approve(),
)
```

PyTEAL Basics - Subroutines (1/2)

Sections of code can be put into **subroutines** (Python decorators)

```
@Subroutine(TealType.uint64)
def isEven(i):
    return i % Int(2) == Int(0)

App.globalPut(Bytes("value_is_even"), isEven(Int(10)))
```

PyTEAL Basics - Subroutines (2/2)

Recursion is allowed

```
@Subroutine(TealType.uint64)
def recursiveIsEven(i):
    return (
        If(i == Int(0))
        .Then(Int(1))
        .ElseIf(i == Int(1))
        .Then(Int(0))
        .Else(recursiveIsEven(i - Int(2)))
    )
```

PyTEAL Basics - Inner Transactions (1/3)

Every **application** has control of an **account**

```
Global.current_application_address()
```

PyTEAL Basics - Inner Transactions (2/3)

Applications can **send transactions** from this **account**

```
Seq(
    InnerTxnBuilder.Begin(),
    InnerTxnBuilder.SetFields(
        {
            TxnField.type_enum: TxnType.Payment,
            TxnField.receiver: Txn.sender(),
            TxnField.amount: Int(1_000_000),
        }
    ),
    InnerTxnBuilder.Submit() # send 1 Algo from the app account to the transaction sender
)
```

PyTEAL Basics - Inner Transactions (3/3)

```
appAddr = Global.current_application_address ()
Seq(
    InnerTxnBuilder .Begin(),
    InnerTxnBuilder .SetFields(
        {
            TxnField.type_enum: TxnType.AssetConfig,
            TxnField.config_asset_name: Bytes("PyTEAL Coin"),
            TxnField.config_asset_unit_name: Bytes("PyTEAL"),
            TxnField.config_asset_url: Bytes("https://pyteal.readthedocs.io/" ),
            TxnField.config_asset_decimals: Int(6),
            TxnField.config_asset_total: Int(800_000_000),
            TxnField.config_asset_manager: appAddr,
        }
    ),
    InnerTxnBuilder .Submit(), # create a PyTEAL Coin asset
    App.globalPut(Bytes("PyTealCoinId"), InnerTxn.created_asset_id()) # remember the asset ID
)
```

A pythonic Algorand stack

- PyCharm IDE & AlgoDEA plug-in
- Algorand Sandbox Docker in dev mode
- Algorand Python SDK
- PyTEAL
- PyTest for Smart Contracts unit-tests and e2e-tests
- TEAL Debugger

DAPP EXAMPLE & USE CASES

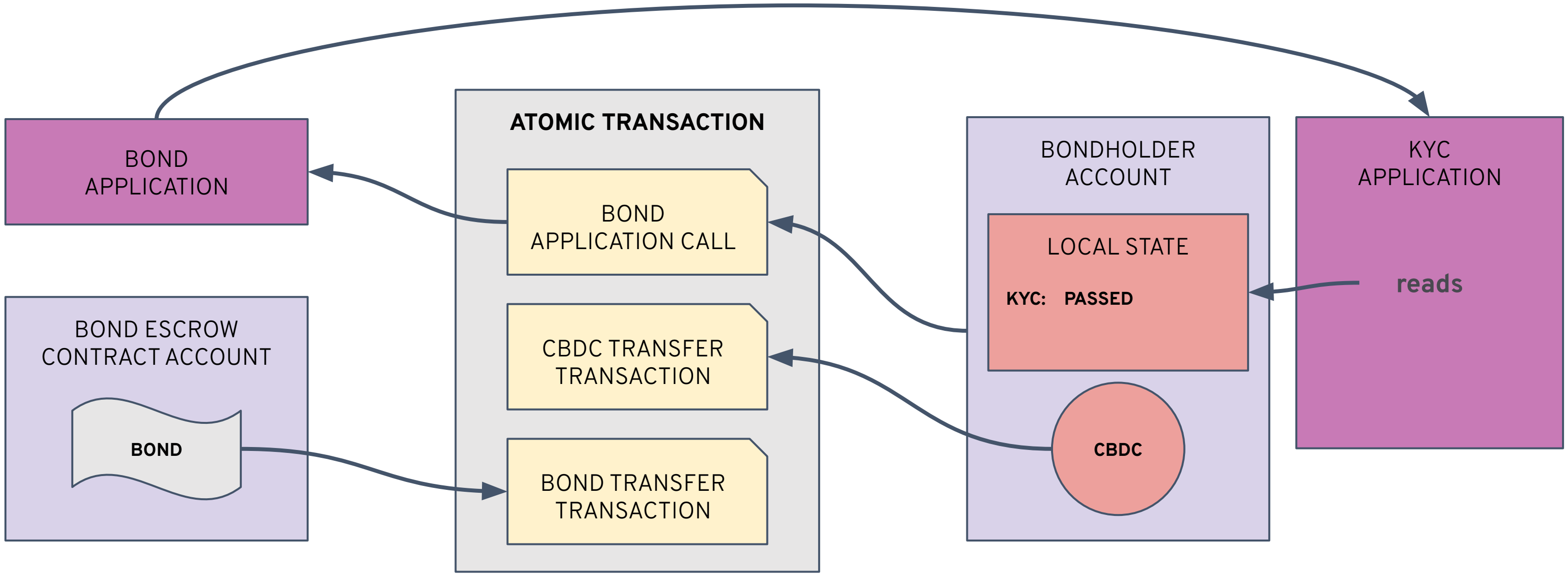
Example: zero coupon bond

Zero coupon bond

1. Bondholder purchase a bond (only if KYC has been passed)
2. Bond application checks for bond’s maturity
3. Bond Issuer re-pays the Bondholder at given rate

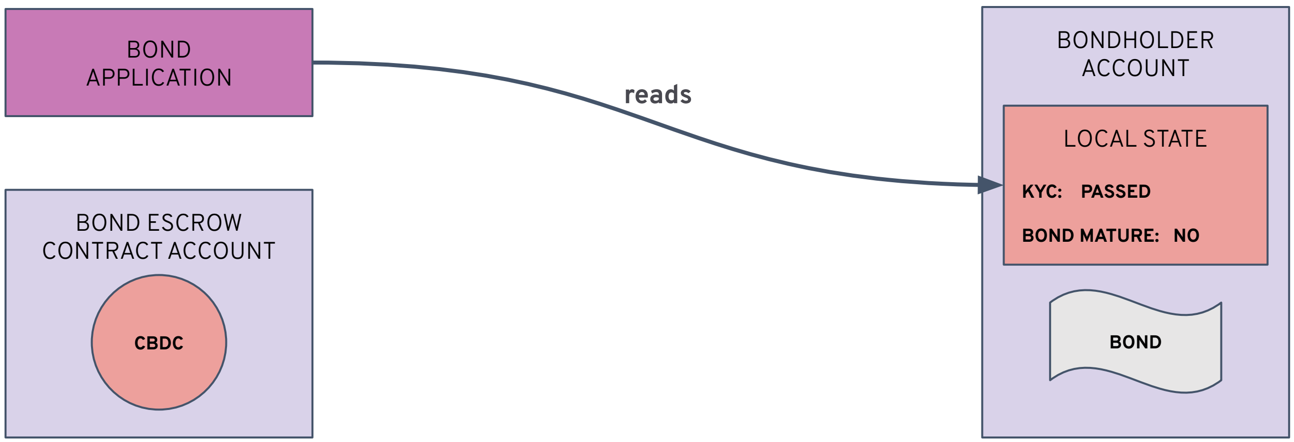
Zero coupon bond

1. Bondholder purchase a bond (only if KYC has been passed)



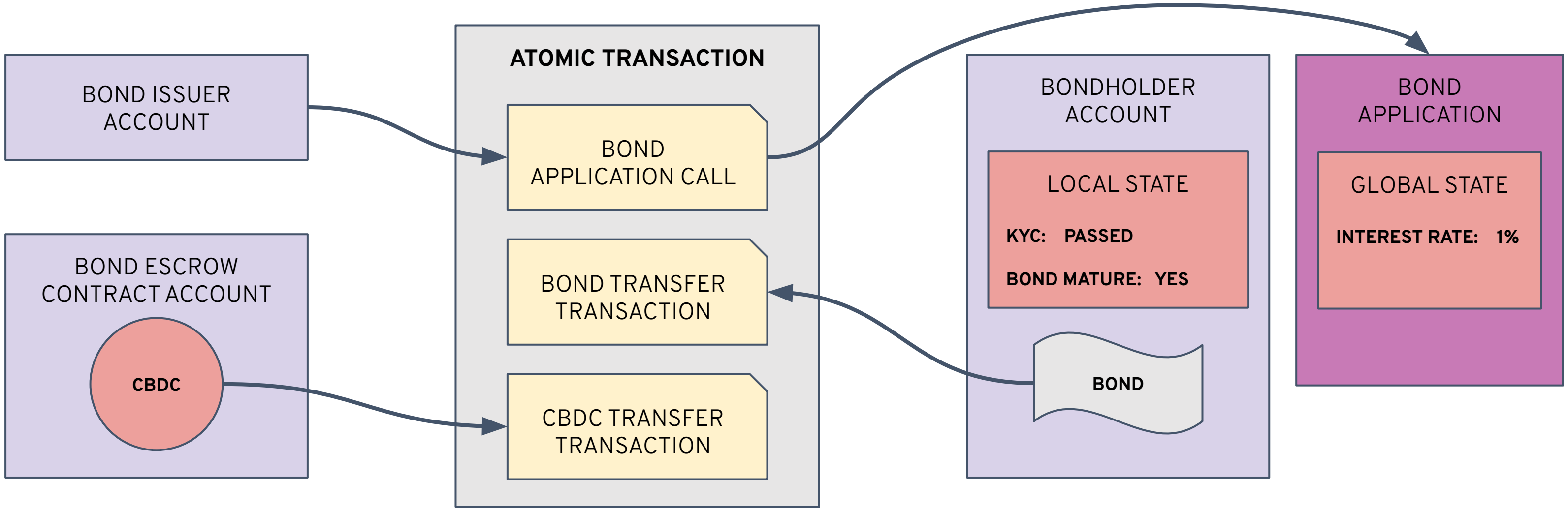
Zero coupon bond

2. Bond application checks for bond's maturity



Zero coupon bond

3. Bond Issuer re-pays the Bondholder at given rate



Q&A