Radix DeFi White paper

Delivering Security, Interoperability, Incentives and Scale for DeFi

13 August 2020 v1.0

RADIX

ISSUED BY THE RADIX FOUNDATION LIMITED

REG No: 10864928
Registered office: Argyle Works, 29-31 Euston Rd, London, England, NW1 2SD

Table of Content

Introduction

Friction in today's financial system costs the world 0.05% of global GDP (approximately $71Bn per annum), and delivers financial products and services that are often inflexible and benefit primarily those at the top of the food chain. Radix believes this is the most important problem that the right permissionless DLT network can solve.

The concept of Decentralised Finance (DeFi) offers an alternative, where innovative new financial applications can be quickly developed and easily accessed by all. The right permissionless DLT network can enable this new era of financial innovation and democratization, providing an open infrastructure of programmable assets that can substantially replace the closed, walled gardens of traditional banking infrastructure today. Delivering on the promise of DeFi requires a platform that is equal to the task of becoming the new financial layer for the connected world. DeFi as it exists today (as a ~$4Bn market) covers only a narrow set of financial services for a narrow set of users who are already excited about crypto. But the growth of this early DeFi market is already constrained by today's technology. Apps are reaching the limits of performance of available DLT platforms, and frequent exploits keep the majority of mainstream users away. These technology limitations will only become more urgent to solve as DeFi grows into an anticipated $64Bn market over the next two years.

Radix intends to remove the technology barriers limiting the expansion of DeFi by building a layer-1 protocol that directly addresses the technology needs of DeFi, both for today's applications and for the future of general financial services for the world. This is a full stack approach, re-engineering consensus, distributed virtual machines, executable on-ledger code, DeFi component building, DeFi application building and developer incentives.

Re-shaping finance will be a 40 year sprint. To do so, the foundations have to be right.

Summary

This paper describes how the Radix platform provides an integrated technology solution to multiple problems that create significant barriers to DeFi's potential.

The growth of DeFi rests on the shoulders of developers who are building the new decentralized applications that will replace traditional closed systems. The barriers holding back DeFi are precisely those that hold back those developers. We believe there are four problems that DeFi developers face today that a DLT platform must solve before DeFi can truly go mainstream:

1. **Reduce smart contract app hacks, exploits, and failures**
2. **Build interoperable DeFi dApps faster**
3. **Incentivize a decentralized development community**
4. **Scale public ledger dApps without breaking DeFi composability**

While these challenges also apply to the majority of use cases for permissionless DLT platforms, DeFi needs a purpose-built solution all the more urgently because users' personal wealth is at stake in every transaction.

These problems are each associated with four crucial layers of a complete DeFi platform stack; Radix addresses them layer by layer.

Starting at the bottom of the stack, developers need a new **consensus and network** design to make DeFi scalable without breaking composability; something that "scalable" networks such as Ethereum 2.0, Near, Polkadot, Cosmos and Avalanche all fail at. This problem is the focus of our **Cerberus**[1] consensus protocol.

On that foundation, a new **development environment** is needed to let DeFi developers confidently create a new kind of smart contract that lowers the risks of hacks, exploits, and code failures. Then to accelerate the building of interoperable DeFi apps, an on-ledger system is needed that provides access to modular **DeFi "lego bricks"**. And finally, to rapidly grow a community around these developer tools, we need a system that creates a decentralised, self-incentivizing **developer ecosystem**, similar to blockchain's current self-incentivizing network infrastructure. These three problems are the focus of the **Radix Engine**[2] and two of its unique features: the **Component Catalog** and the **Developer Royalty System**.

Barriers to DeFi Growth	Platform Layers	Radix Solutions
Scale Dapps without breaking DeFi composability	**Consensus/Network**	**← Cerberus**
Avoid smart contract app hacks, exploits, and failures	**Development Environment**	**← Radix Engine**
Build interoperable DeFi dApps faster	**DeFi "Lego Bricks"**	**← Component Catalog**
Incentivize a decentralized development community	**Developer Ecosystem**	**← Developer Royalties**

We believe Radix's comprehensive, integrated bottom-to-top approach is what uniquely makes it the first layer-1 DLT protocol uniquely suited for DeFi.

[1] The initial Radix mainnet launch will include a simplified version of Cerberus consensus. The capability described in this paper pertains to the full version of Cerberus intended for a later mainnet update.

[2] The initial Radix mainnet launch will include Radix Engine v1. The Radix Engine features described in this paper pertains primarily to the following Radix Engine v2.

1. Reduce Smart Contract Hacks, Exploits, and Failures

“Smart contract” has become a generic term to refer to application code that a developer writes to deploy to a distributed ledger. Following Ethereum’s pioneering use of smart contracts to make blockchains more programmable, there has been an explosion of new smart contract blockchain platforms that largely replicate the specific implementation adopted by Ethereum: that is, a general purpose Turing-complete language and virtual machine running on-ledger. Tezos, EOS, NEO, Tron, Hashgraph, Hyperledger, and more generally follow this model.

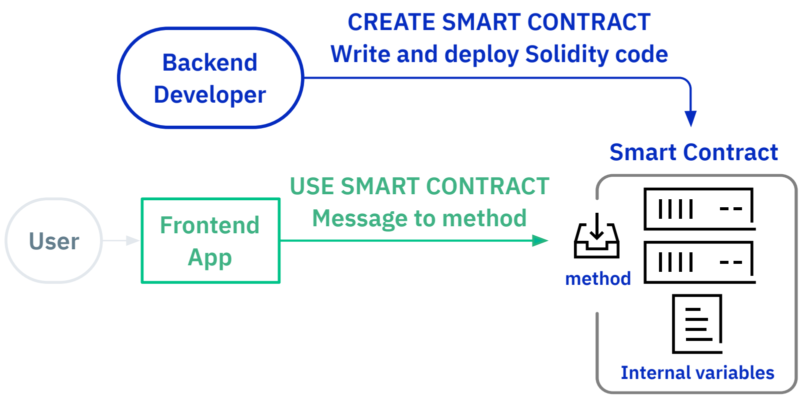
Building a reliable financial application, however, is a different class of problem than building a game, web service, or other general application. A DeFi application deployed to a DLT network is expected to run **autonomously, trustlessly,** and **irreversibly** while managing millions of dollars in assets. Developers building to specialized requirements like these typically use specialized development environments to make it as easy as possible to avoid bad results.

The lack of finance-appropriate smart contract development environments has led to an unfortunate sequence of high-profile losses of funds in DeFi. Simple programming errors and programming language quirks that normally would only cause annoyance in a general application have instead created documented openings for exploits and system failures. Even where there are no bugs in smart contract logic, connections between multiple complex DeFi applications have made it nearly impossible to design smart contract code to cover all edge cases that could be manipulated by an adversary.

The Radix Engine development environment is designed specifically for the creation of logic that defines predictable, correct results on-ledger in response to requests. This form of DLT programmability is based on Finite State Machines (FSMs), a class of solution that is common in mission-critical embedded systems where predictable correctness is the first priority. To make clear the difference from traditional Ethereum-style smart contracts, we give Radix Engine smart contracts a name more suggestive of their function: **Components**.
Let’s compare the typical smart contract approach to Radix Engine’s Components.

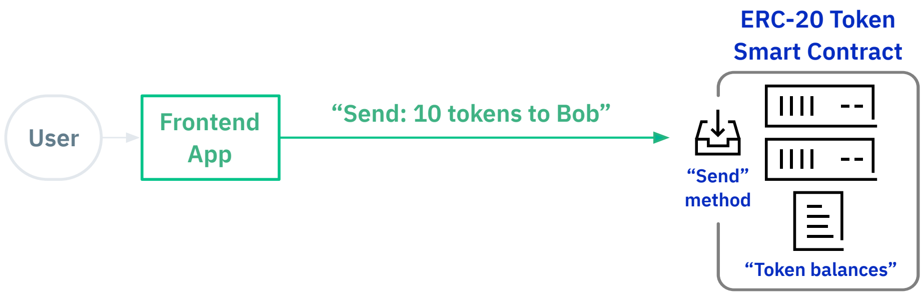
The Ethereum Smart Contract/Method Model

An Ethereum smart contract can be thought of as a black box deployed by a developer to the network. Inside that box you can imagine a little general purpose computer server running some code. To make that server-in-a-box do something, it offers “methods” that users or other apps can call by sending a signed message. The contract has its own internal variables inside the box that it can update based on the messages it receives via its methods.



In this paper we refer to a ***“backend developer”*** *as the creator of apps/systems, like DeFi apps, that an app created by a* ***“frontend developer”*** *may use. In many cases a backend developer may of course also develop their own frontend app.*

Those internal variables are used by the developer to represent all sorts of things. For example, an ERC-20 smart contract creates something that behaves like a supply of tokens by maintaining an internal list of balances. “Sending” a token to somebody really means using a “send” method that the contract code translates into twiddling its internal Token balance variables – ie. reducing the sender’s balance and increasing the recipient’s.

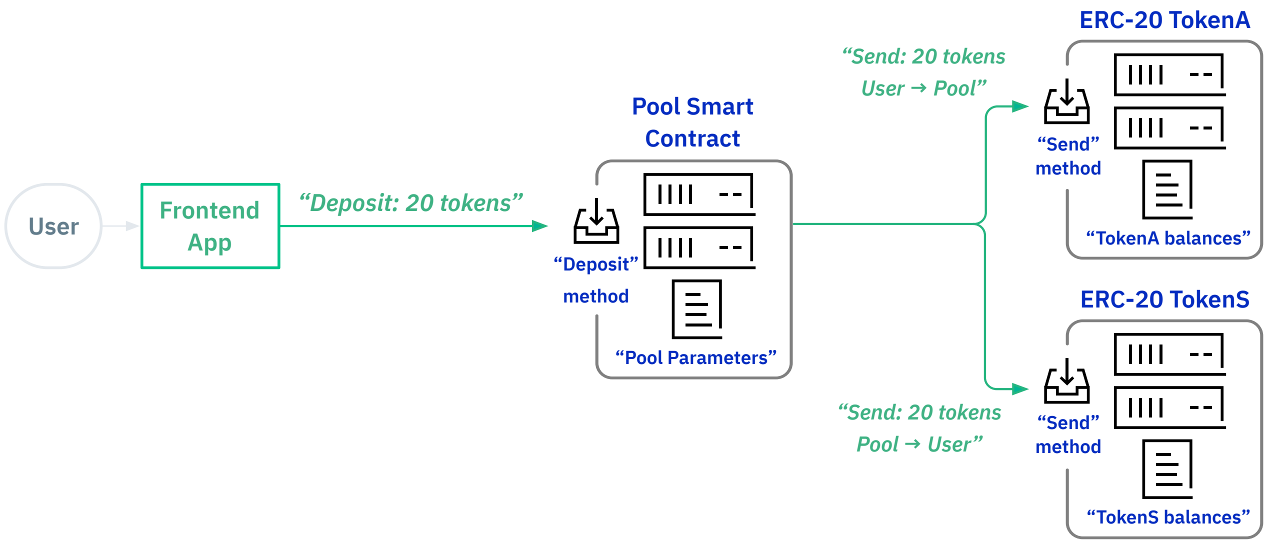


An ERC-20 smart contract represents the idea of a supply of tokens by carefully maintaining an internal list of balances – rather like balances maintained by a bank for its users.

This model is quite flexible, allowing *anything*, in theory, to run on a decentralized platform – thus the Ethereum vision of the “world computer”.

One problem however is that it puts a significant burden on the developer to ensure that their “representation” of tokens, or whatever else, within their smart contract is always correct, and that updates to the internal variables match intuitive expectations. This is doubly critical with DeFi, where potentially very expensive unexpected outcomes are immutably committed to a trustless ledger. And the situation becomes much more complex with DeFi applications where one transaction involves multiple composed smart contracts. In this case, one contract may call the methods on other contracts, with each updating their respective internal variables to produce a combined result.

For example, even a simple liquidity “pool” smart contract can quickly get complex, with results scattered across multiple contracts. If we want our pool to accept an existing (ERC-20) TokenA into the pool and mint a corresponding calculated amount of a TokenS representing a share of the pool, we end up with something like this:



For financial transactions, this doesn't behave like what we would expect for an open platform of highly composable assets. What we would prefer to think of as "tokens moving in and out of a pool" is instead expressed as negotiated balance updates across a network of black-box contracts. It starts to look a bit like the traditional world, with siloed banks communicating with each other, but each keeping privately-held books.

With greater numbers of connections between black boxes, the network of contract calls explodes and correct behavior becomes more and more difficult to reason about and predict. A wide variety of exploits of Ethereum DeFi smart contracts come down to an adversary manipulating the fact that there is nothing fundamental stopping smart contract code from doing *anything* to its internal state, often with unintuitive results that may cascade through the system.

The Radix Engine Component/Action Model

The Radix form of smart contracts, Components, are built in a way that more closely models real-world expectations for finance (and other transactional systems that ledgers are good for). Components are built from finite state machine logic, and define their behavior by **Actions** that directly translate a discrete existing input (or "before") state to an output (or "after") state.

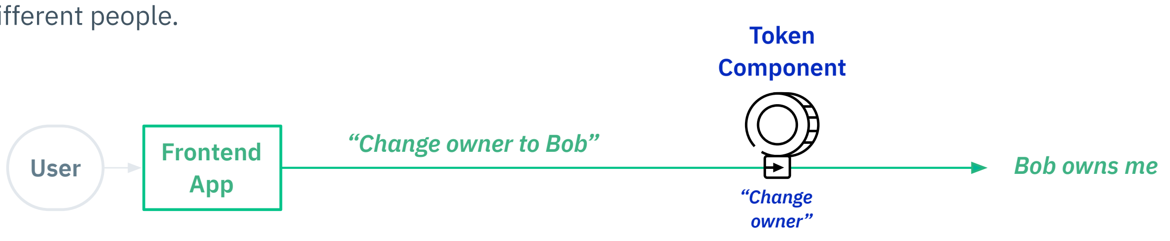
More concretely, this means that Components are defined by *what it is possible for that Component to do* via its Actions. By defining Components by their Actions in this way, we gain two important attributes when trying to avoid bad results.

First, Components can behave more intuitively like physical assets or other finance building-block "primitives", rather than as black boxes, making their behavior via Actions easier to reason about, design, and analyze. Second, *usage* of Components is similarly more intuitive and predictable – and we gain the ability for creators of Component transactions (whether a front-end app, or by reference from another Component) to directly set their own definitions *of what should be possible* for that transaction, creating explicit guard rails on what can and cannot happen in creating the final output state.

Explaining how Components and Actions work is perhaps best done through a few examples:

A User-created Token

Take the example again of a token. On Radix, developers don't need to use a monolithic ERC-20-style smart contract that keeps a list of all balances; we model each *individual indivisible token* (each "Satoshi" in Bitcoin terms) as a discrete independent Component. That token Component's primary available Action is: "*change my owner, if you have the right to do so*". So for example, the "before" input state may be *owned by Alice*, the "after" output state *owned by Bob* – if the conditions of ownership change defined in the token Component's Action are met. This means that the token Component is defined by what it is possible for it to do: to be owned by different people.

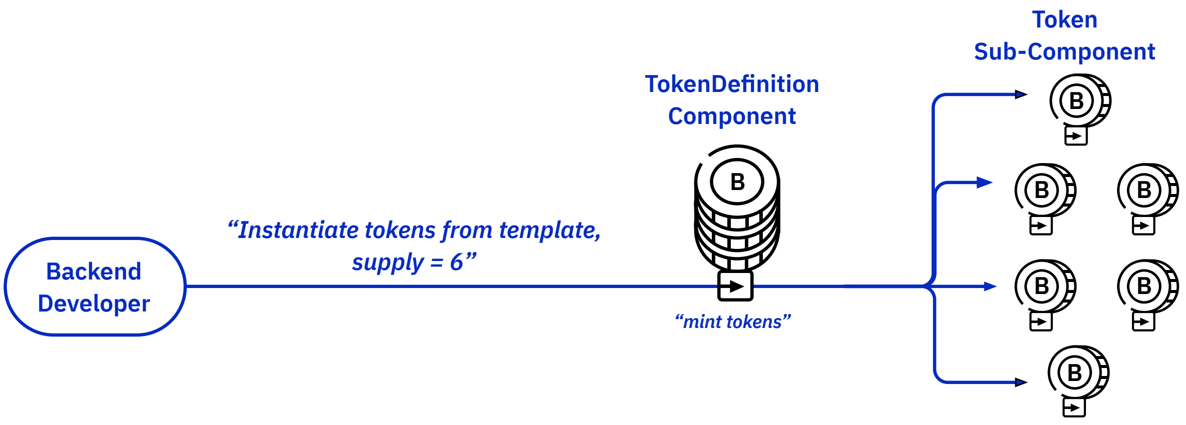


Using the FSM-based Radix Component model, each token can be its own independent Component with Actions (like "change owner") that define their behavior to match intuitive expectations.

While the Radix ledger may, behind the scenes, store a large number of these indivisible tokens collectively for efficiency, logically each token *acts intuitively like physical coins*. That is, the defining capability of each is that its ownership can be passed from one person another. There is no question of it being accidentally cloned, or a glitch in smart contract logic causing tokens to no longer be accessible.

The above picture is a little incomplete however; before we can begin sending tokens around, we need to first configure and create a supply of them. We do this using an important feature of Components: the ability to define **sub-Components**. These are Components that act independently (they have their own **Actions** and individual state), but are defined *within* a top-level "parent" Component.

A supply of user-created tokens is defined in this way. A single TokenDefinition Component[3] describes the attributes of the overall token supply (name, symbol, max supply, etc.), but also includes the definition and behavior of a sub-Component: the individual Tokens.



A developer uses TokenDefinition, customizing its parameters. The TokenDefinition Component then provides the "mint tokens" Action that creates indivisible individual Token sub-Components.

These individual tokens are created with a "mint tokens" Action on the TokenDefinition, creating token copies from the sub-Component "template". That done, the tokens can each be used as independent physical-like things just as we described above – accessed via API – while remaining associated with the parent TokenDefinition and its configuration.

A Liquidity Pool that Operates With Other Components Atomically

Not only may Components and their Actions be used individually, multiple of them may be combined within a single transaction. In this case, the mapping from input to output is *collective*, defined by all of the included Actions simultaneously. For example, let's consider a "liquidity pool" Component. A simple version could be defined by these Actions:

Deposit Action: *"I am the owner of some reserves of TokenA and mint a proportional pool share TokenS to anyone who sends TokenA in."*

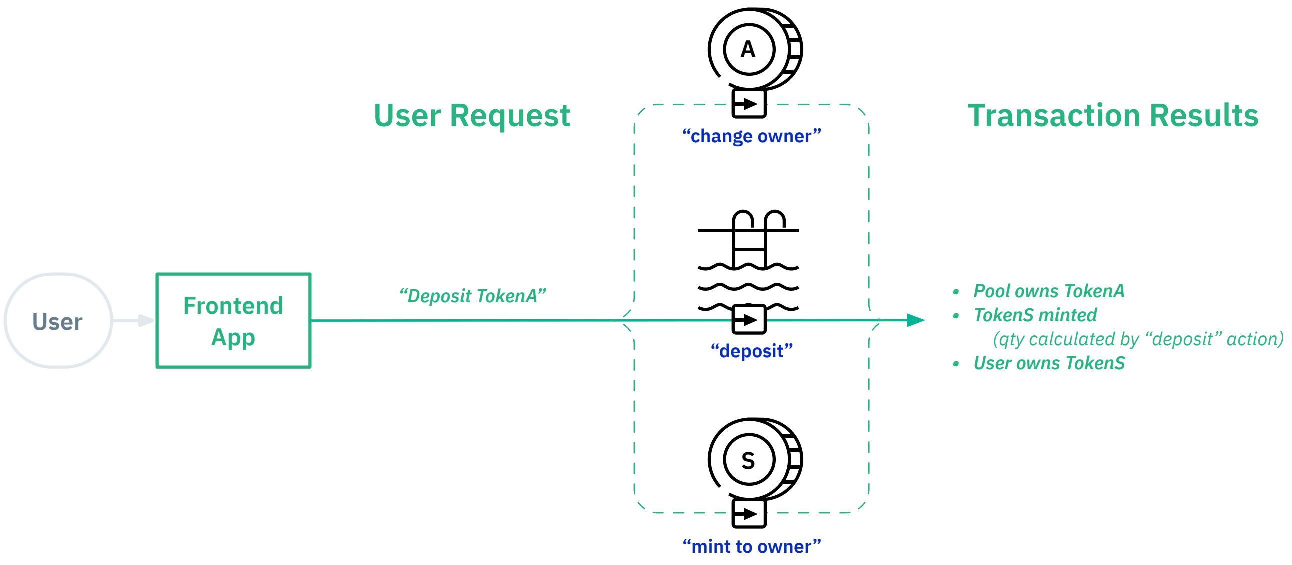
Withdraw Action: *"I am the owner of some reserves of TokenA and send a proportional amount of TokenA to anyone who sends me pool share TokenS (which I burn)."*

While these are pseudo-code descriptions of the Actions' definitions, you can see that the words in blue indicate Actions of other existing Components (TokenA, TokenS) that need to be included in a pool transaction.

[3] The TokenDefinition Component is a standard Component available on Radix. Unlike many smart contract platforms, not only many user-created tokens use this available functionality, the native RADIX platform utility token uses it as well so that both behave identically.

A nice property of FSM-driven Actions is that the input → output mappings of these multiple Actions can be combined and operated simultaneously (rather than chained together sequentially as with smart contracts). When multiple Components are required for a single transaction, *all* of the Action-driven state changes of the various Components in the transaction mesh together like gears in a gearbox. As long as all of the gears of all of the relevant Components can all successfully turn together (the input and output mappings do not conflict), the transaction is successful. If they can't, the entire transaction safely and correctly fails.

To compare with the Ethereum case above, a Radix Engine pool deposit transaction looks more like this:



A pool Component Action (deposit) includes the logic specifying that it expects to become the owner of some TokenA Components, mint a corresponding quantity of TokenS "share" Components, and make the user the owner of those.

This behaves more like an open platform of intuitively composable assets that we want!

From a front-end developer's perspective, using the "deposit" Action of the Pool Component is no more complex than using a method on a traditional smart contract. But that Action itself can define the other Components that need to be involved (some specific TokensA and TokensS) – that is, which Components need to mesh their gears together for the transaction to be successful.

In this example, the "deposit" includes changing of ownership of a discrete set of relevant tokens using their Actions. Intuitively, the "mint" or "change owner" Actions may have their own intuitive rules about who can do those things, or who the recipient can be. If the Pool tries to do something that fails to meet those rules, the user's request to the Pool correctly fails with clear rationale.

A Transaction Request with Consumer Safety Limits

To see how the consumer of Component Actions can explicitly prevent bad results, let's take the example of a Token Swap Component. This Component accepts TokenA and returns an amount of TokenB equal to a market price (perhaps provided by an oracle or some market maker logic – the specifics don't matter here). Its primary Action might look like:

> **Swap Action:** *"Any receipt of TokenA will create a send of TokenB by the equation: TokenA * [currently defined B/A rate]"*

A user wants to perform a swap using this Action, but would prefer to set his own clear limits on the acceptable swap exchange rate rather than rely completely on whatever the rate happens to be when the transaction goes through.

The Action model allows the user (i.e. the consumer of the Token Swap "swap" Action) to also specify their own Action-style rules on the input → output mapping, as follows:

> **User request:** *"I am **sending** TokenA to the "swap" Action of the Token Swap Component, **which must result** in a **send** to me of TokenB of at least [desired limit]"*

The Action-like conditions of this user request create another "gear" (for that transaction only) that must mesh with those of the relevant Components and successfully turn along with them, or else the entire transaction will correctly fail. This means that the user can place absolutely clear and direct guard rails on their usage of any Component, without having to understand the internal details of the Component's functionality. And these safety guard rails are available not just to front-end users but to the referred usage of Components by other Components, allowing more confident composed DeFi usage that limits exposure to faults in systems built by others.

As Components become more and more complex, and more Components are composed together as is common in DeFi applications, this gives both users and creators of these apps a powerful tool to eliminate the possibility of many unexpected and expensive outcomes, even when bugs or design flaws are in play.

An On-ledger Oracle for Off-ledger Data

Another useful piece of DeFi functionality is an on-ledger data oracle. In the token swap example above, the market price data that this Component needs could be provided by an Oracle Component that offers an Action like:

> **Get price Action:** *"When requested, I provide the current contents of a key-value pair representing the requested pair price"*

This might seem strange as the input and output states are identical; no output state change has been directly specified by this Action. But when used by the Swap Token's Action for example, the Oracle's "gear" meshes with that of the swap Action to determine *how the swap Action* defines an output state change (the quantity of tokens that must be sent). In this way, the oracle provides a valuable service allowing off-ledger data to drive the results of autonomous, atomic on-ledger transactions.

But how did that off-ledger data get into the Oracle's key-value store to be atomically available? The Oracle cannot itself make an off-ledger request for new data because this would violate the atomic functioning of FSM-based Actions. In the simplest case, the Oracle would have another Action (only usable with the private key of the creator) that updates it. The creator could push updates to this data as frequently as desired, or could do so more responsively.

Creating Radix Components

Creating new Components will use a new, specialized language that we call **Scrypto**. Scrypto is a functional language, providing a style of programming better suited to defining the kind of FSM-based Components described here. Functional languages are increasingly common, particularly for building reliable high-concurrency systems. Scrypto's syntax should be familiar to developers who have worked with functional languages, and provide a set of programming primitives particularly suited to creating Component/Action logic.

Put together, we believe the Component model is a much more purpose-built solution to making the Radix network programmable for the world of finance assets and applications.

2. Build Interoperable DeFi dApps Faster

A truly useful development platform includes libraries, frameworks, and other tools that allow developers to build simple, common things quickly with a minimum learning curve. These same tools also accelerate more complex builds by providing reliable, pre-built solutions for parts of the problem that other developers have encountered and solved well already. Having good standards and off-the-shelf solutions also strongly encourages interoperability between dApps that is particularly important for a DeFi ecosystem.

In DeFi, common chunks of finance-oriented functionality recur across many applications: assets (fungible or unique), shares, accounts, multi-party control, liquidity pools, swaps, purchases, and data oracles just to list a few examples. These are prime candidates for pieces of functionality that developers would like to see pre-existing, proven, well-maintained solutions.

Traditional open source methods and community collaboration are certainly good places to start to encourage these builds. Package managers often assist in the process of discovering and using pre-existing tools. But the Radix Engine gives us an exciting new possibility: **putting community collaboration and package manager-like functionality directly on-ledger.**

An on-ledger mechanism for Components to be modularly used, leveraged, updated, versioned, extended, and combined is a powerful tool for developers. Components deployed on-ledger in this way don't just contribute to the developer ecosystem; they directly extend the effective functionality of the Radix platform. And a developer need not build a fully-functional standalone dAapp Component to usefully contribute. Components that do one thing very well, and are built to be easily reused or combined with other Components, can become standards of the platform that accelerate builds and encourage interoperability for everyone.

The idea of "writing programs to do one thing well" and "writing programs to work together" was the guiding philosophy of the creators of UNIX in the 1970s. The result created the foundation of open source development and the spectacularly successful family tree of UNIX-based operating systems and applications since then. We believe that rebuilding the financial systems of the world around a decentralized platform suggests a similar philosophy, maximizing interoperability, modularity, and potential for anyone to make meaningful contributions both large and small.

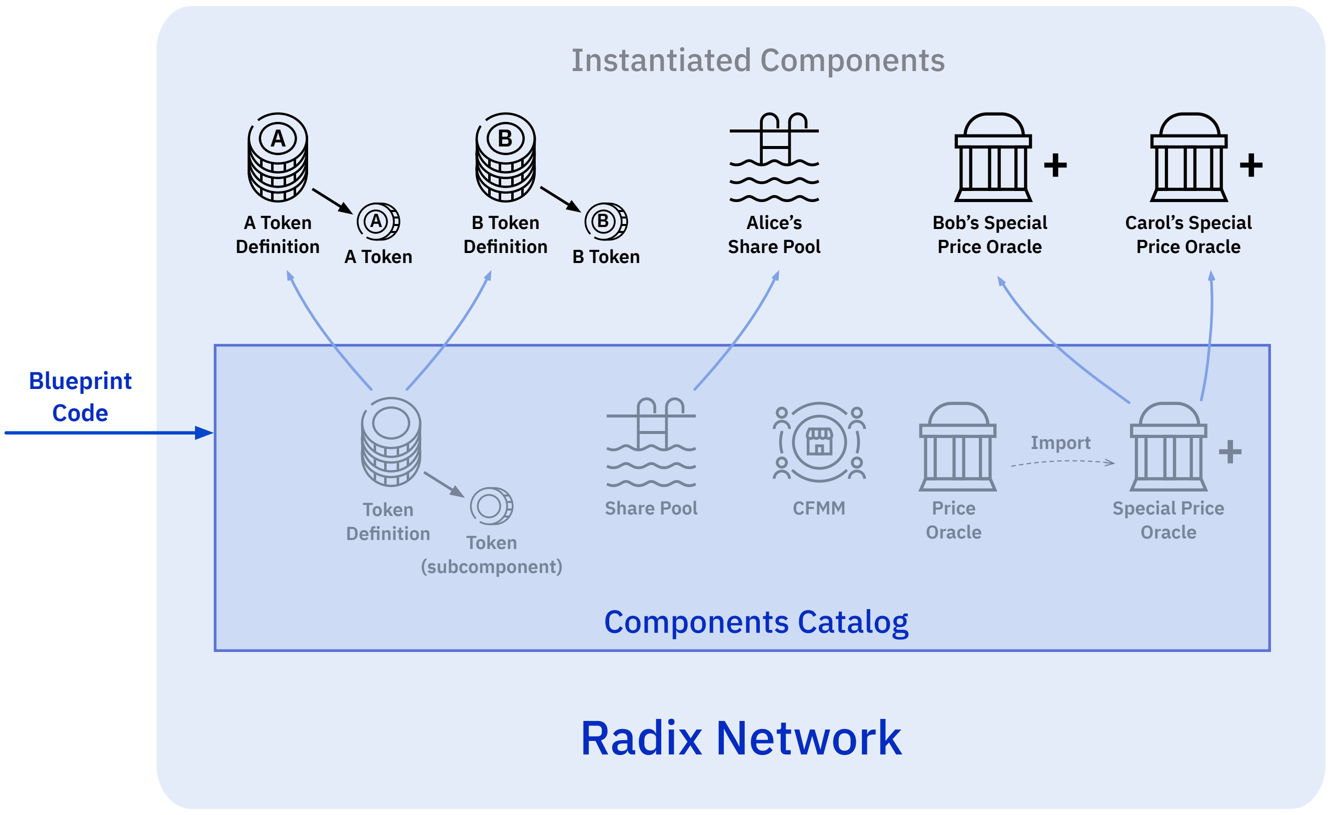
We have integrated this philosophy deeply into the way Components are deployed and used on Radix with a platform feature we call the **Component Catalog.**

The Radix Component Catalog

With typical smart contract DLTs, a developer writes some code (Solidity in the case of Ethereum) and then pushes it to the network where it becomes an active smart contract for users of the network to interact with. The Component Catalog changes this model.

When a Component's Scrypto code is pushed to the network, it is first added automatically to an on-ledger registry called the Component Catalog. Components in the Catalog are like inactive templates or blueprints that anyone may use to create multiple active Components patterned after the original in the Catalog.

To make a Component from the Catalog active for use, a developer instantiates it, creating his own **Instantiated Component** from the blueprint template. An Instantiated Component has its own unique identity on the Radix network, and its Actions become available for use by users (or other Components, as we saw previously). This means that one Component in the Catalog can be used an unlimited number of times as the template for instantiated Components that all behave in the same basic way.



New Components start life as Scrypto code that is deployed into the Component Catalog on the Radix Network itself, where they can be easily configured and instantiated – or imported into other new Components.

Instantiation is simple, done via API, and requires no Scrypto code. Most Components in the catalog will include configuration parameters allowing customization of the instantiated Component. For example, the "Token Definition" Component described earlier would let anyone instantiate their own Token Definition with its own unique name, symbol, maximum supply, etc. – and then begin minting their own tokens.

By instantiating Components from a universal on-ledger Catalog in this way, Radix makes it very quick, easy, and safe for any developer to issue assets and access other simple functionality created by others, without having to learn and write Scrypto code.

Another way of making use of Components in the Catalog is to **import** them. A developer may want to make use of the functionality of an existing Component in the Catalog – but *add to it or customize what it can do*. This is done by creating a new Component that includes an import command for the Component that provides the core functionality desired. The additional functionality is then implemented in the developer's own Scrypto code.

Importing is *not* copying the Scrypto code of the original component; it is an *on-ledger link* to the original Component (and its version). An example of this (shown above) might be a Price Oracle component that a developer wishes to customize by adding a calculation of a moving price average (and Action to access it).

The developer could create their own Special Price Oracle Component, import the original Price Oracle Component by reference to its ID in the Catalog, and add the additional Scrypto code for the price average calculation.

Both Catalog Components and Instantiated Components have their own unique Component ID, and are associated with the creator’s own unique developer ID. Components are versioned, with each new deployed update requiring a revision bump. Updating a Component does not automatically force an update onto other Components making use of it; previous revisions remain immutably available on ledger and existing Components will continue to have access to the previous revision. A developer may choose to adopt a new revision by making their own update to the Component that uses it.

Standard Platform Components

The Radix Foundation is committed to filling the Catalog with useful Components that model common elements and behaviors of DeFi as standard features of the platform. This includes things like assets (fungible or non-fungible tokens) and accounts (including multi-sig control) and will expand to include functions desired by the community that could include higher-level DeFi primitives like liquidity pools, swapping systems, purchasable assets, data oracles, and more. Each of the standard Catalog Components developed by the Radix Foundation can be instantiated as-is (for example creating a supply of custom tokens via API call) or, as we saw above, they can be modularly combined in various ways to create more complex functionality.

Using these standard Catalog Components, developers can skip reimplementing common functionality themselves, accelerating development time or reducing it to zero for very simple things like issuing a token. Catalog Components created by third parties can also become standardized features of the Radix network in exactly the same way. Whoever creates them, on-ledger Catalog Components create natural composability and interoperability for the DeFi ecosystem.

Foundation-provided Components are not enough for a thriving developer ecosystem however. Neither can we simply release a platform and throw up our hands, expecting developers to fill in functionality before adoption and business demand creates a reason to do so. Our solution is to use the unique features of the Radix Engine to build in, at a protocol level, a financial incentive for developers to invest their time in contributing useful Components to Radix, leading into the next important DeFi problem Radix solves.

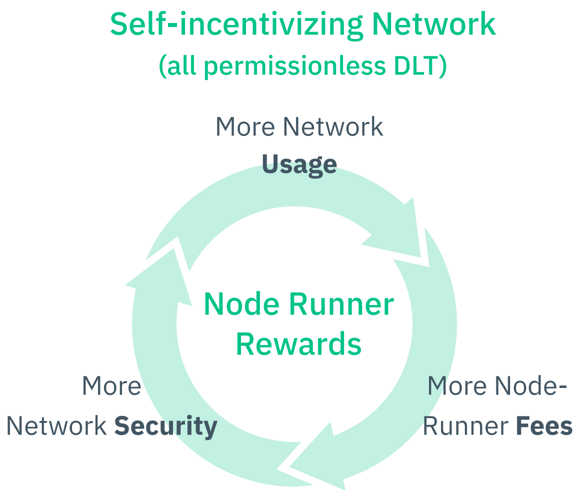
3. Incentivize a Decentralized Development Community

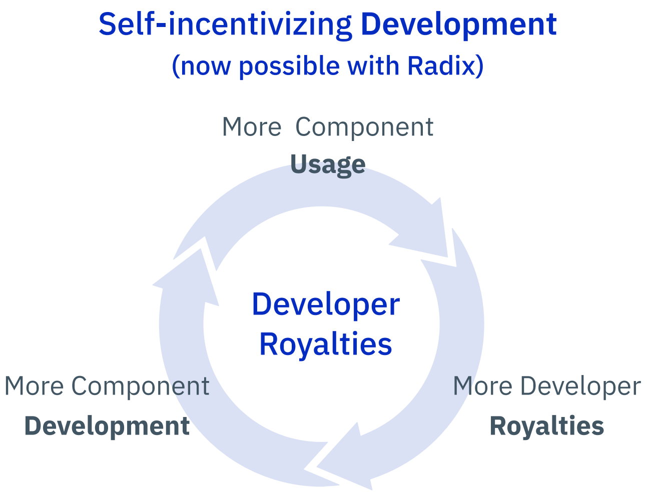
Perhaps the most important innovation of blockchain was the ability to create open networks that are economically self-incentivized – originally in the form of “mining”. With this innovation, a community of node-runners can be incentivized to participate from the earliest stages of the network, as well as to scale the network up (its security at least) to safely conduct transactions of billions of dollars of value.

But creating a permissionless DLT network suitable for DeFi requires more than just node-runners providing the low-level infrastructure. There must be a thriving developer ecosystem creating the kind of useful, interoperable Components that the Radix Engine and Component Catalog enable.

Developer Royalties

While traditional methods of growing a developer ecosystem can work, we believe that *decentralized developer self-incentivization* can create a breakthrough in rapid decentralised ecosystem growth, creating the same kind of market-based incentives as “mining” . The Radix Engine and the Component Catalog make this possible for the first time; we call it the **Developer Royalty System**.


Self-incentivizing Network
(all permissionless DLT)
More Network **Usage**
Node Runner Rewards
More Node-Runner **Fees**
More Network **Security**


Self-incentivizing Development
(now possible with Radix)
More Component **Usage**
Developer Royalties
More Developer **Royalties**
More Component **Development**

The core concept is that the developer of any Component may specify a royalty in RADIX tokens (the native Radix utility token already used for transaction fees to node-runners) for *each usage of that Component in a transaction.*

Note that this isn’t an “app store” where one must pay for access to Components; it is an entirely per-transaction-use fee that must be included within the transaction itself. This means that payment of royalties is based on the real *utility* that the Component brings to the network.

The Radix protocol itself is able to calculate and charge the correct royalty fees – in the same way it handles fees for node-runners – because the Component Catalog and the various types of Component usage are all on-ledger. If a developer adds a highly useful, modular, interoperable Component to the Catalog, Radix ensures they are rewarded for the transactions enabled by their work in a completely decentralized manner. This is a unique property of the Radix Engine development environment that allows the Radix network to self-incentivize a developer community for the first time.

How Royalty-Setting Works for Developers

The fundamental innovation behind Radix Developer Royalties is the on-ledger Component Catalog and associated on-ledger mechanisms for using Components to create applications and transactions. With these, the Radix protocol directly links royalty *definition* (by the developer) to royalty *payment* (by the user), creating an open marketplace for network utility created by developers. In this marketplace, royalty payments on a per-transaction-use basis are automatic and guaranteed by exactly the same consensus mechanisms that guarantee Radix network security.

Developers must have the tools to participate in this on-ledger marketplace in whatever way they prefer. We start by allowing the developer to set a different royalty for each of the different types of Component usage that are possible once it is added to the Catalog[4]. These are:

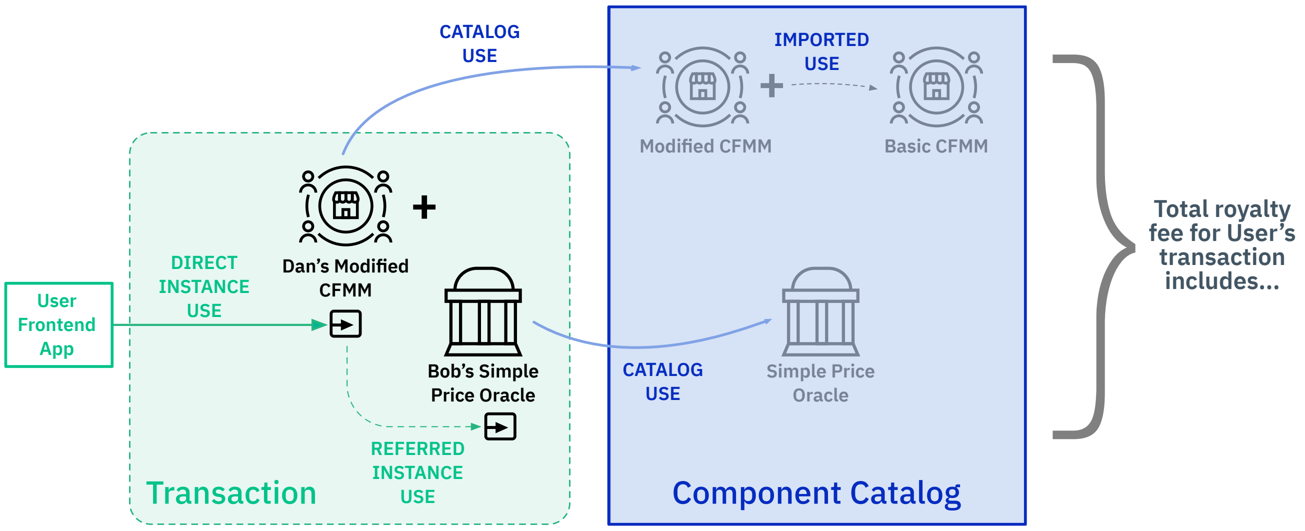
1. **DIRECT INSTANCE USE** – This is when a user transaction directly accesses an Action of an instantiated Component. For example, this might be the interface to a DeFi application like a CFMM (constant function market maker), even if this Component uses other Components “behind the scenes”.
2. **REFERRED INSTANCE USE** – When a Component is used directly, as in #1, it may specify that multiple other instantiated Components must be included in the Action for the transaction. This is “referred” instance use. For example a DeFi application might use an instantiated oracle Component to get access to its data by reference for a certain transaction.
3. **CATALOG USE** - Each instantiated Component is associated with an original template Component in the Catalog. The catalog use royalty is paid when one of these Catalog Components it is the template basis for an active, Instantiated Component that is used as in #1 and #2.
4. **IMPORTED USE** - This refers to the usage when a developer “imports” an existing Catalog Component into his own new Catalog Component. As in #3, importing itself is not paid, but rather the imported Component is considered used when another Catalog Component that has imported it is used as in #3.

Note that there are two types of usage here, indicated by the green and blue colors. A developer would specify the royalties for CATALOG and IMPORTED USE when the Component is **deployed** to the Catalog. These can be thought of as the royalties for “behind the scenes” use of Components. Royalties for DIRECT INSTANCE and REFERRED INSTANCE USE would be specified at the time that a Component is **instantiated**. These are the royalties for the use of the Actions of each active, independent “copy” of a Component from the Catalog.

For each transaction using the Action of an instantiated Component, there will be an associated set of royalties for that instance, any other instances it refers to, and the various Catalog templates that sit behind the instances. Ultimately it is this full set of Components that made the transaction possible.

The four types of usage can be seen in this example for a single transaction that a User requests to Dan’s Modified CFMM:

[4] While fixed XRD pricing is the simplest, we anticipate developers may wish to algorithmically set pricing for each use type, perhaps to set royalties to a certain fiat price at current XRD rate using a price oracle of their choice. This is entirely feasible within the Radix protocol.



A user's transaction request to an Action of Dan's Modified CFMM (and instantiated Component) must include the summed royalty fees of the associated Components, according to type of use. These summed fees are easily determined by the Radix Engine so a User app knows what it must pay, and to ensure all royalties are paid out correctly.

Developers may tailor their royalties for each of these types of usage to suit the nature of what they build and how they expect it to be used. We believe this opens up substantial flexibility for creating decentralized on-ledger revenue streams and business models – as well as enabling royalty-free use where desired. Some use case examples are given later.

In addition, the Royalty system allows developers to set specific pricing (for any of the four types of usage) for specific other developers (via developer ID) or Components (via Component ID). For example, a developer may wish to offer discounted usage royalty pricing when accessed by a DeFi app that he expects to provide unusually high usage volume. Or the developer may simply wish to remove the royalty on his Components when accessed by other Components that he himself has built, while still applying the royalty when others use them.

Updating of royalty pricing is also possible. As with any other Component update, this would require a revision update. As described, however, previous versions (with previous royalties) continue to remain immutable and available for use. Anyone wishing to adopt a newer version of a Component would also accept the set of royalties associated with it.

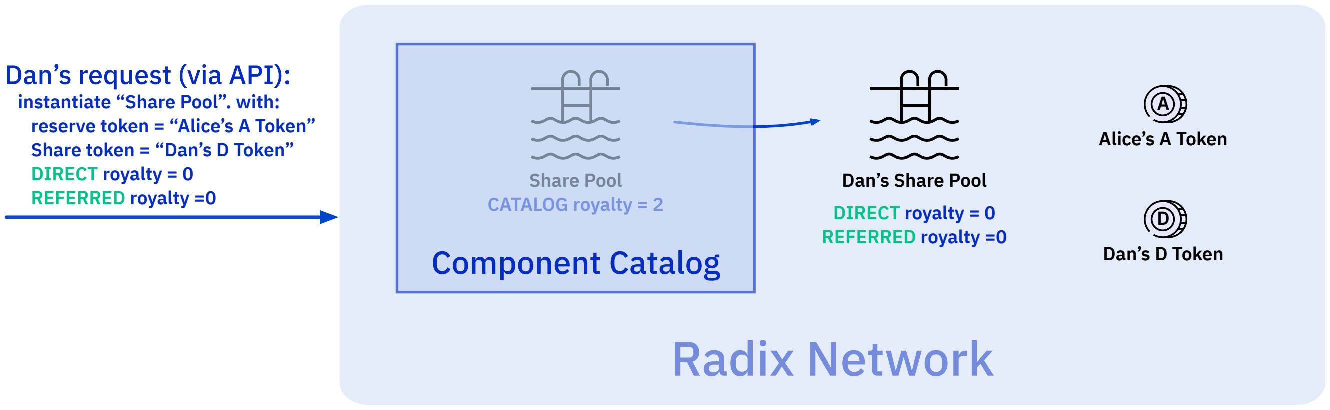
With each Component defining their associated set of royalties, the Radix protocol has everything it needs to automatically calculate and assess the royalties a transaction creator must pay for that transaction. An example may make the creation of Components and the assessment of royalties in transactions more clear.

Component Creation and Royalty Example

Let's take the example of Dan: he wishes to create a DeFi service around a Constant Function Market Maker (CFMM) Component that he has some clever ideas about.

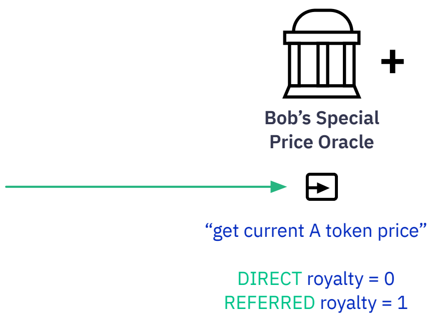
He could build it the whole thing from scratch, but first he searches the Component Catalog to see if there are existing proven solutions for any parts of the problem. One part of the CFMM is a share pool where a specified token is minted in exchange for other tokens being contributed to the pool (in proportion to the pool's reserves). Fortunately somebody already built a "Share Pool" Component that many people seem to be using happily and safely from the Component Catalog.

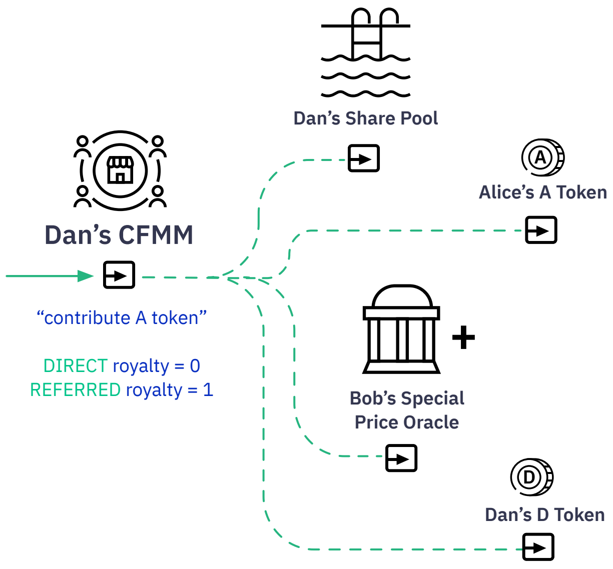
Dan instantiates the share pool for himself, configuring it by setting the reserve token that can be contributed (an "A" token already instantiated by Alice) and the share token that should be minted in exchange for contributions (a "D" token that Dan already instantiated himself).



Dan sends an API request to the Radix ledger to configure and instantiate a Share Pool from the Component Catalog. His instantiated pool is then active for use alongside existing Token Components he will use.

Dan also needs a price oracle in order to get current market pricing for A Token to use as part of his CFMM's internal economics logic. Fortunately Bob has already created and instantiated a nice special price oracle Component that Bob feeds with a variety of market data for different tokens. Dan considers the REFERRED royalty that Bob charges for the oracle to be reasonable, and so he can just use the Action of this instantiated Component as-is to get the price data he needs.



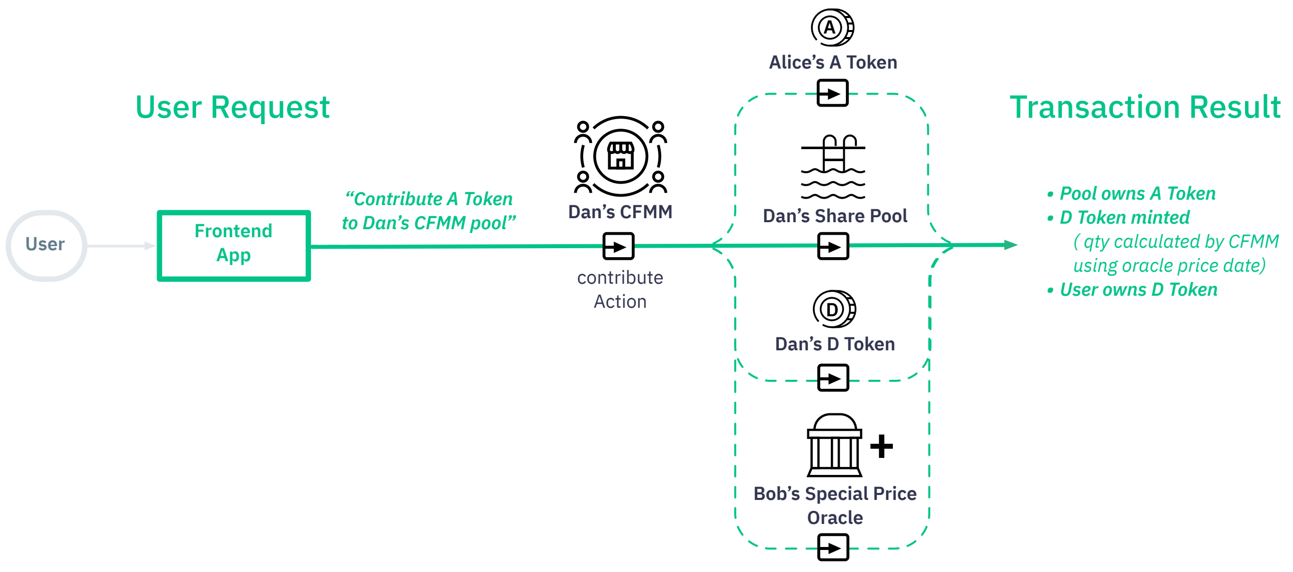


Dan then writes the Scrypto code for his CFMM, defining a "contribute A token" Action (among others). This action's Scrypto definition includes that it needs to use certain Actions on A token, his share pool, his D token, and Bob's special price oracle.

He doesn't need to *import* any of these Components; he can implement his own contribute Action so that transactions using it include these other Component's Actions as a referred use (thus applying their respective REFERRED royalties to that transaction).

Dan sets a royalty for his CFMM, pushes its Scrypto code to the network, and instantiates it. Perhaps he creates a web UI as a frontend app to enable users to create transactions using it.

A transaction using Dan's CFMM would look something like the illustration below. The user's request from Dan's frontend app, at left, is translated by the CFMM, and the other Components it refers to, into a correct set of output result state changes at right (assuming of course that the transaction satisfies the constraints of all Actions involved).



An example of a transaction where one Component (Dan's CFMM) specifies the use of four other instantiated Components that are automatically, atomically composed to create a single output state result.

The royalty the user must pay for this transaction is calculated first by adding up the royalties for each of the Components in the Catalog that were part of the transaction. This includes the CFMM Component Dan created, the Token Definition Component of which the tokens are sub-components (provided without royalty by the Radix Foundation), and both the Special Price Oracle Component and the Price Oracle Component that it imported. (See the Components shown earlier in this document.) Then the node-runner fee is added to the royalty to determine the final fee:

Developer royalties:			
Dan's CFMM	**DIRECT**	5	
CFMM	CATALOG	0	(set to zero for Dan's own use)
Alice's and Dan's Tokens	REFERRED	0	
Token Definition x2	CATALOG	0	(provided by Radix Foundation)
Bob's Special Price Oracle	REFERRED	1	
Special Price Oracle	CATALOG	0	(again, set to zero for Bob's own use)
Price Oracle	IMPORTED	0.5	
Dan's Share Pool	REFERRED	0	(set to zero for Dan's own use)
Share Pool	CATALOG	2	
Node-runner fee:			
Transaction fee		0.1	
Total transaction fee:		8.6 XRD	

While this may seem complex when listed out this way, a front-end application can use the Radix Engine to automatically determine the full required fee for a given transaction ahead of time.

The Developer Royalty System creates a strong incentive for developers to aggressively build out the most useful functionality they can think of, as early as possible, in order to maximize the adoption and usage of their Components. A developer need not build out a full, complex, application in order to reach a point where their efforts can be rewarded; it may be even more valuable to build Components that do one thing very well, and are highly modular, in order to maximize their usability by others. These Components may become effectively new standards of the platform when widely adopted.

The Developer’s Guide to the Radix Universe

The Component Catalog and Developer Royalties provide the foundations of a fully decentralized marketplace for Component development. On one side of the market, developers of Components may freely add them to the Catalog and set their own per-transaction royalties to be enforced by the Radix protocol. On the other side, developers who wish to leverage those Components have access to immutable on-ledger data telling them what the transaction fees will be for a given Component, how broadly it is being used, its version history (fully open source), and which Components are associated with a given developer ID.

While all the right data is available on-ledger, to make it easily searchable, browsable, and visualized requires a proper front-end service. The Radix Foundation is committed to providing the first option in the form of the **Developer’s Guide**. While we believe it is important for us to provide this service right away to our community, our hope is that many such services will arise to address the needs of a diverse developer community.

The Developer’s Guide will collect the contents of the Catalog and current instantiated Components and present them in an “app store” like interface. However rather than presenting paid apps to users, the Guide will allow developers to discover Components (both Catalog and Instantiated) to accelerate their own development, and distill on-ledger data into a view of the associated reputation and history of development and usage, as well as transaction royalty pricing, in order to make good integration decisions within a highly interoperable DeFi ecosystem.

The Developer’s Guide will provide another useful service to the developer community in how it presents the work of different developers. Like any truly open marketplace, bad behavior is possible. Someone could choose to copy the Components created by others, or otherwise try to introduce exploitative Components into the Catalog. This of course cannot be stopped on a permissionless ledger. But the Developer’s Guide, as an off-ledger service, can work to detect such behavior and present relevant context in the search results it provides.

Unlike an app store, no purchases are made through the Developer’s Guide, and the Radix Foundation will take no cut of royalties (nor would it, or anyone else, have the ability to) – it is a convenient interface into a fully decentralized marketplace entirely between developers and their users.

The Radix Foundation and the Open Source Radix Project

The Radix Foundation is currently developing the Radix protocol as a fully open source code base. Long term, Radix must be a community-led movement, and we will look to the examples set by other successful open source and blockchain projects to build and support this transition to community. This project is the infrastructure bedrock for Radix, including limitlessly scalable Cerberus consensus, and the Radix Engine that underpins the Component Catalog and Developer Royalty System.

Starting from this bedrock, the Developer Royalty System provides the right incentives at the application layer of Radix to rapidly grow the platform into a vibrant, interoperable, and open DeFi application ecosystem for developers and users.

The Radix Foundation's mission is to support developers at all levels, providing critical enabling functions where needed to avoid bottlenecks to adoption, while turning all of our work over to our community to extend. This of course includes aggressive development of core Components for developers to use and extend royalty-free – but also supporting Component developers through partner programs.

One particular area where the Radix Foundation can assist in early phase bootstrapping is to subsidize developer royalties. In the early stages of the network when there may be relatively few transactions (and associated fees), virtually all blockchains (including Radix) subsidize the rewards to node-runners via token supply inflation or simply a subsidy paid from a reserve. The Radix Foundation will explore ways in which it can offer the same to developers, multiplying the royalty rewards paid by users and encouraging developers to participate early in building Components.

Case Studies of Developer Royalties in Action

With the tools offered by Radix Components and the Developer Royalty system, we can imagine a variety of ways in which developers can contribute and customize royalty-based revenue streams based on usage. Here are some examples:

The Core Capability Developer

Cara has been playing with DeFi apps for a while and hears that Radix is an exciting new platform for DeFi. She doesn't want to create, launch, and support a full DeFi app – but she loves the idea of pooled liquidity and wants to build that capability for the many Radix dApps that she anticipates will want it.

She creates a Share Pool Component in the Catalog that can be configured to mint and burn a specified share token in response to deposit and redeem Actions for a specified reserve token, in the correct quantity to maintain the NAV (Net Asset Value) share represented by each share token. She expects others will create their own pools using her Component for various purposes. She sets both CATALOG and IMPORTED use royalties to a low 0.5 XRD to encourage broad use and extension by other developers.

Cara diligently responds to community security questions and feature requests, and so there is little reason for other developers to build this capability from scratch instead of simply accepting the small transaction royalty she asks for usage. Cara's Component becomes the trusted de facto share pool standard that others build their own Component code around, further increasing resistance to copycats or competitors. As Radix adoption grows, usage of Share Pool reaches a sustained 10,000 transactions per day, providing enough income for Cara to focus on independent development full-time.

The High-Value Service Developer

Hannah has a company with access to up-to-the-minute market data that she wishes to offer to Radix DeFi apps. She wants to provide this through a Component where this data can be used atomically, allowing other

Components to process transactions directly using current price data even when the transaction is composed across multiple apps.

Hannah’s company builds, deploys, and instantiates a simple Price Oracle Component that only accepts market data update transactions from her own servers. In the Catalog she sets the Component’s CATALOG and IMPORTED royalties to zero, since it is her data she wishes to monetize and she doesn’t mind free usage of her open source oracle code if it helps others. When she instantiates the Component, she also sets the DIRECT royalty to zero as she doesn’t mind direct access by end user applications; it only raises the awareness of the quality of data. Finally she sets the REFERRED royalty to 25 XRD, meaning that when DeFi applications want to make atomic decisions based on her data, this is where she wishes to derive her revenue. Because her data is only available through her own instantiated Component, it is the single on-ledger source for this data.

The DeFi Application Developer

Daiki wants to build a premier DeFi tokenized lending service on Radix, seeing it as a more secure and scalable platform and wanting to be among the first to build a user base there. The typical ways of monetizing this service are available – such as requiring an application-level payment of a fee in a preferred token with each use of his application’s Actions. However Daiki wants to accumulate XRD tokens on the early network and sees royalties as a convenient way of generating revenue from either direct or referred (ie. composed from another DeFi app) usage – without adding the additional integration complexity of an application-level fee.
He expects that his company’s frontend, support, and rapid improvements are more than enough to keep copycats at bay, but nonetheless he sets the CATALOG and IMPORTED royalties for his core Lending app Component to a very high level. When instantiating it, he sets the DIRECT and REFERRED royalties to a reasonable 5 XRD level to encourage low-friction, high-volume composed use by his frontend users and other apps.

Part of Daiki’s competitive advantage is access to excellent market price data from Hannah’s Oracle. He went to her company early on and convinced her to create a royalty exception for Daiki’s Lending app ID, allowing his app to use her data atomically at half her general royalty.

The Builder’s Guild DAO

Beatriz wishes to build a completely decentralized organization in which multiple developers work together to produce useful Radix Components and share in the aggregated royalties. She first creates a Guild token (using a free Component developed by the Radix Foundation) that represents seniority within the Guild. She creates two primary Components:

- A Guild Governance Component that enables Guild token holders to vote on the policies that grant developers membership and seniority within the Guild based on their work, as well as how new Components are deployed and updated, and their royalties set

- A Revenue Sharing Component that extends basic Account functionality to allow Guild token holders to claim proportional amounts of XRD from the Guild’s royalties that flow into its Account (which is used to deploy all of the Guild’s Components)

She deploys both of these with zero royalty in order to contribute to the open source community and encourage the creation of other DAOs, a concept she believes strongly in.

She launches the Builder’s Guild DAO, turning control over to an initial core set of developers and the Governance Component, and watches as it blossoms into a global community of contributors that develops a strong reputation for producing reliable Radix Components.

The Initial Component Offering

Itai is an independent developer that has a big idea for a DeFi app, but he needs enough capital for him to commit full-time to taking it from prototype to fully-featured platform. He is part of a network of developers that respect his skills, and so he decides to ask them for up-front funding in exchange for a portion of the royalty revenue he expects his app to produce.

Fortunately he finds Beatriz’s Revenue Sharing Component. Instead of configuring it for revenue share claims based on Beatriz’s Membership token, he configures it to use his own ItaiApp token as the basis for revenue claims on the royalty account. He distributes 50% of these tokens to his backers, keeping the rest for himself.

The Freemium Developer

Felix has built a Component that implements a clever set of algorithms that are useful to many applications. He wants to encourage the widest possible integration, and so he sets both the CATALOG and IMPORTED royalty to zero. Later, he creates an additional algorithm that is very useful to a limited subset of the developers using it. He deploys a new revision of the Component that adds a modest fee. Because Radix is an immutable ledger, developers continue to freely instantiate and import the earlier free version, but if they wish to have access to the newer more featureful version, they must also accept the new royalty with the update.
As described above, it is important to note here again that once a Component is deployed to the ledger, its developer cannot silently foist a change of royalty on users or other developers. Only through the creation of a new version of a Component can the set of royalties be updated, with the old version remaining available.

4. Scale dApps Without Breaking DeFi Composability

One clear challenge faced by DLT platforms today is scalability. The rapid expansion of DeFi apps on Ethereum has pushed the platform to its limits. While Ethereum pursues its 2.0 upgrade to help alleviate the bottlenecks by 2022, other DLT technologies have entered the picture proposing new techniques to reach greater throughput of transactions.

Posting high throughput numbers alone, however, fails to encompass the full scalability requirements of DeFi. If a DeFi DLT platform is to provide a functionally viable alternative to a global network of traditional financial systems, it must be able to support a tremendous number of DeFi apps *simultaneously*, while running each app at high throughput, and without compromising decentralization. But even this isn't enough.

Perhaps the most important feature of DeFi is the interoperability of apps and assets, often called "composability". The ability to "compose" a single transaction, making use of multiple autonomous smart contracts, is energizing much of the DeFi innovation and excitement on Ethereum today. With the ability to freely compose across any set of DeFi apps, it becomes possible to build a service that, for example, instantly provides the best exchange rate for a trade across multiple automated market makers, or allows the leveraging of a crowdsourced liquidity pool to take instant advantage of an arbitrage opportunity. Crucially, these complex operations across apps must all happen in a single "atomic" step, meaning that either the entire transaction across all apps is valid and resolved all at once, or the entire transaction safely fails. This is incredibly powerful and is the basis for how DeFi dissolves the inefficiencies of traditional financial systems, replaced by fast, customizable, and interoperable DeFi financial apps.

Despite the crucial importance of composability, **most DLT solutions seeking to increase scalability do so by significantly reducing composability**. Typical approaches for scale separate apps and transactions across "shards" where they can run faster but do not have direct, atomic access to each other. Here more sharding means less interoperability, putting scalability and composability in direct conflict. While this may be an acceptable tradeoff for simple token transfers or applications that do not need to be freely and atomically composable, it makes true DeFi at scale essentially impossible.

Radix takes a different approach, designing its full technology stack for unlimited scalability without compromising security, decentralization, or composability.

Typical Sharding for Scalability

Typical scalability solutions involve some type of sharding. Whether sharding is implemented using a hub-and-sidechain architecture (like Cosmos or Polkadot), or by breaking a block into pieces for independent processing by different nodes (like Near), the idea is the same: different apps and transactions are localized to some number of separate shards where they can be run through consensus in parallel.

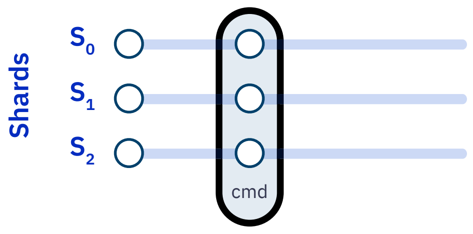
This parallelism achieves greater throughput, but the compromise is that communication between shards is made difficult. Different shards can be thought of as separate blockchains (in fact sometimes that is literally what they are), but where there is some method to send messages between them. **But if each shard conducts consensus independently, it is impossible to process a transaction across multiple shards atomically**. One way or another, cross-shard coordination must be done across multiple blocks on the different shards, often involving "receipts" or other ways of providing conditional cryptographic commitments between independent consensus processes. This makes these transactions slow, error-prone, and difficult to implement in smart contract code. Making matters worse, assignment of apps and assets to certain shards is usually static (like Ethereum 2.0), or requires significant network overhead to adjust.

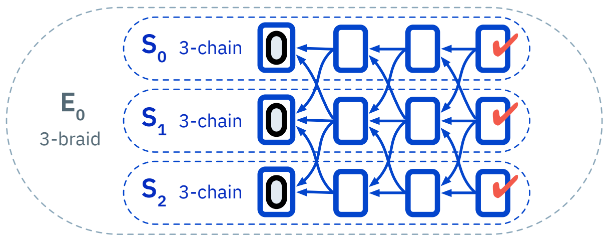
We realized early on that Radix needed to start from first principles to resolve this tension between scalability and composability. First, rather than using a static set of shards, we needed to support a practically unlimited number of shards to achieve as much parallelism as possible for a global-scale DeFi platform. Second, we needed a consensus protocol able to dynamically conduct consensus, on atomic transactions (including smart contract operations), synchronously across only the relevant shards without causing the rest of the network to stall. And third, we needed an application layer able to efficiently make use of this unlimited "shard space" and multi-shard consensus.

Cerberus Consensus

A core piece of the Radix solution is our unique Consensus algorithm, **Cerberus**. Cerberus is designed around a concept we call "pre-sharding" where, rather than trying to add sharding to a monolithic ledger, we start by splitting the ledger into a "shard space" of a number of shards so large as to be practically unlimited[5]. We can use these shards to represent anything we like, and Cerberus can "braid" secure consensus across an arbitrary number of shards as required. Our Cerberus whitepaper covers this algorithm in depth. But in short, Cerberus combines three core insights:

First, we move from the typical concept of global ordering to that of partial ordering. Virtually all DLTs assume global ordering wherein all transactions must be placed on the same timeline. Some forms of sharding essentially create multiple globally ordered timelines, but keep fixed global ordering within each. Cerberus takes this concept even further, presuming that each transaction can specify precisely which shards are relevant (and thus must be ordered) *for a specific transaction*. This requires a specialized application layer that can specify how shards are used and relate to each other, which we will get to later.





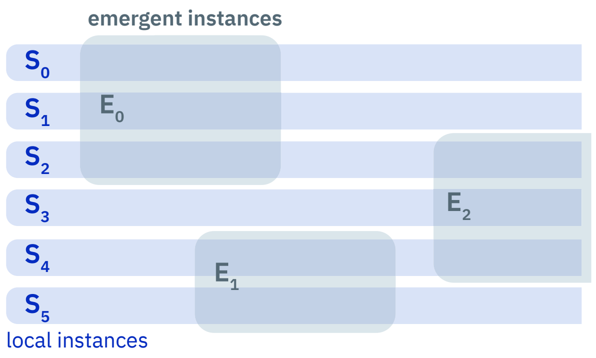
emergent Cerberus BFT

Second, now that we know which shards must be included for a transaction, we design a new form of BFT-style consensus called "braiding". Typical BFT-style consensus uses 2 or 3 phases of signed commitments between nodes in order to confidently finalize a transaction. Cerberus' braided consensus runs a single 3 phase BFT instance (called a "3-chain") within *each* shard, but braids these instances together with commitments provided by the leaders of the other related shards. The result is

[5] In concept this is similar to a typical extremely large space of public keys such that even free, random usage uses a practically insignificant amount of the available space.

an "emergent" 3-braid consensus that ensures all relevant shards can atomically commit to the multi-shard transaction.

Third, we design the protocol so that dynamic 3-braid consensus processes may run in parallel. Each shard, with its local BFT instance, can run completely independently, as can any emergent multi-shard instance (as needed for a given transaction) that isn't related to any other at the time.



We combine these insights to create Cerberus, a new consensus algorithm designed specifically for large, diverse networks of simultaneous applications and transactions. Cerberus provides linear scalability through parallelism; more network demand can be served by ever greater numbers of economically-incentivized node-runners. As the Cerberus-based network grows, atomic composability is never compromised because direct consensus between shards happens seamlessly in response to each transaction.

Radix Engine Application Layer

The application layer of a DLT provides the interface between a developer's smart contract code and the underlying ledger and so Cerberus requires a specialized application layer. For Radix this is the Radix Engine, and the important part for scalability is the Engine's bottom-most layer: Radix VM.

Radix VM provides the partial ordering that Cerberus requires to braid consensus on a per-transaction basis – including transactions driven by Component Action logic. It does this by defining transactions as a related set of changes to state that are encoded across shards. For example, the ownership of two different tokens (two pieces of state) may be located on two different shards, and so an atomic transaction that transfers the ownership of both of those tokens must correctly "partially order" that transaction relative to other transactions on those two specific shards, at that moment. That is, we must ensure that changing the owner of either token doesn't conflict with another request to change the owner of those tokens. All other requests to the network at that time don't matter.

Radix VM also must solve another substantial problem for DeFi scalability: **concurrency**. Even if Cerberus gives us excellent ability to parallelize transactions of unrelated things, many DeFi (and other) applications are intended to handle many concurrent transactions for *highly related* things. DeFi "pools" of token reserves (such as those that power Uniswap) are an excellent example of this, where every transaction is calculated using the state of the token reserves that are themselves changed by each transaction. Many requests competing to use the pool simply cannot be parallelized.

This demonstrates that there are two fundamentally different modes of achieving high throughput of transactions on our network – one where parallelization is possible (many unrelated transactions) and one where serialization is required (many highly related transactions). These different modes are well studied, and we require two different tools to solve them: optimistic requests, and pessimistic requests.

> **Optimistic requests** specify the full details of a state change up-front based on what the current state is believed to be. As long as nothing conflicting changes about the current state by the time the network processes the transaction, the transaction sails through. But there is always the chance that our optimism is punished and our transaction rejected if any of our assumptions are no longer valid, forcing us to retry. For most simple transactions on a DLT network this is quite unlikely and Cerberus

can naturally parallelize extremely effectively, but for something like a popular DeFi pool, transactions will constantly conflict, fail, and need to be retried.

Pessimistic requests means that a transaction locks a required part of the current state up front so that we can be confident that any well-formed request will not fail due to an unforeseen change of state. This means that requests that update a given piece of state are entirely serialized. When many requests are trying to update the same state (as with a DeFi pool), grouping all of that state together and applying serialization is the optimal solution.

So what we would really like is a hybrid solution: Simple transactions should use optimistic requests and be parallelized across many Cerberus shards as much as possible to maximize throughput. Complex transactions involving lots of related state (such as a high-volume DeFi pool) should use *pessimistic requests* allowing them to be serialized to make updates as quickly as possible to state that's collected on few (or one) shards. Radix Engine makes this hybrid solution possible through two elements: **Action-based requests** and **dynamic localization**.

As we've described, all Radix Engine transactions take the form of requests to on-ledger Actions of Components. Actions define how a given request creates an output (some *changes* to state, on their relevant shards) based on an input (some current state, on their relevant shards). Action-based requests allow both optimistic and pessimistic usage.

Let's take the example of our Token Swap Component from above, where a user can send TokenA and receive TokenB at a current exchange rate. Our swap Action was this:

Swap Action: "Any receipt of TokenA will create a ***send*** of TokenB by the equation TokenA * [currently defined B/A rate]"

Say I own 10 TokenA and I want to send 5 of them to the Token Swap Component to receive the correct number of TokenB in return. We can use the swap Action in two ways to do this:

Optimistic request: Using the specified "swap" Action, I will send these specific 5 TokenA from my account to the reserve account; I see the current exchange rate set within the Token Swap Component (1A → 2.7B), and *so these specific* 13.5 TokenB from the reserve account will be sent back to me.

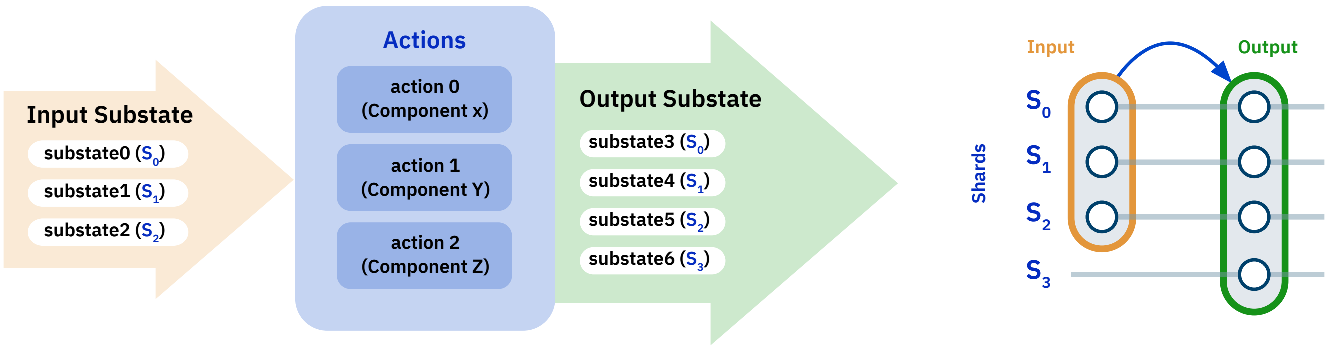
In this case, my request specifies the full transaction, including both the input state (the specific tokens) and the output state change (two transfers). The transaction is valid as long as 1) I still have those specific 5 Token B, 2) I've correctly calculated the number of TokenB, and 3) the reserve still has those specific 13.5 Token B. Here the Action essentially is only *validating* that I've specified everything correctly. If not, it fails.

Pessimistic request: Using the specified "swap" Action, I will send any 5 TokenA from my account to the reserve account.

In this case, the request only provides the minimum the Action needs. I can see that the Token Swap Component's swap Action must be told where to find a sufficient quantity of TokenA, the desired quantity to send, and that this should lead to a corresponding quantity of Token B coming back to me. But I leave it to the Action to construct (and *validate*) the final transaction and outputs. The transaction is valid as long as 1) I still have *any* 5 TokenA, and 2) the reserve still has any 13.5 TokenB.

This may seem complicated, but in both optimistic and pessimistic cases, client software can automatically construct these requests based on the user simply saying “I want to send 5 of my TokenA to the Token Swap Component’s swap Action”. The rest is completely determined by the Action’s definition (and in the optimistic case, the current ledger state known to the client). Radix will provide such client software as part of its libraries for easy integration.

A request to multiple Actions may be composed together in a single request (optimistic or pessimistic), collecting the summed inputs and producing a summed output. Cerberus only needs to know the shards on which to find the input, where to put the output, and the Actions to be used.



At left, three Actions are specified by a client along with required input substate across three shards – and the resulting output substate including those shards plus a new one identified by one of the Actions. At right, the resulting transaction on the relevant shards.

With this Action-based model giving us the ability to create both optimistic and pessimistic requests as needed, our complete scalability solution only needs Radix VM to make some intelligent decisions about how state is stored across shards. We wish to separate unrelated state (for optimistic, parallelized use), but group up highly related state (for pessimistic, serialized use). How to map different pieces of state to shards?

We could use a very simple model where we map Bitcoin UTXO-like states to individual shards:. Here, each shard would have a fixed lifecycle, becoming active with a UTXO input, and then forever dormant as a UTXO output (as the input to some other UTXO shard). This approach would map very nicely to optimistic usage, making shards highly granular and unrelated for simple things like token transfers.

However this approach breaks down in situations where we generally want pessimistic transactions. A complex DeFi app could potentially touch an extremely large number of token-UTXO shards, forcing all of those shards to lock during each round of consensus. We would prefer to group such highly-related state into one shard (or a minimum number of shards) to simplify consensus and maximize throughput of serialized operation.

Fortunately the fact that we have a practically unlimited shard space, and consensus that is able to work across that shard space dynamically, allows Radix VM to perform dynamic localization. Localization means grouping a complex set of states into a single shard. Radix VM is able to dynamically shift the expression of state from more distributed, granular, parallelizable form to localized, collective, serializable form as needed. This means that there is no fixed mapping from any particular piece of state (such as the ownership of a token, or a current exchange rate) to particular shards. Radix VM is able to arbitrarily create, and change, that mapping as required to maximize throughput.

In total, Radix Engine’s massive available parallelism, combined with its ability to dynamically optimize the performance of individual transactions, allow it to finally deliver the kind of multi-mode scalability needed for DeFi. More than simply delivering a high throughput number in special circumstances, Radix is designed for real-world, large-scale application usage. **Rather than fixing scalability by breaking composability, Radix delivers both without compromises making it the only layer-1 protocol designed to scale a highly interconnected, high-demand DeFi ecosystem.**

RADIX