Radix DeFi White Paper

Technology vision for the first layer-1 network designed for DeFi developers

4th January 2022 - v2.05

RADIX

ISSUED BY RADIX FOUNDATION LIMITED

REG No: 12106715
Registered office: Argyll House, 29-31 Euston Road, London, England, NW1 2SD

This paper is published by the RADIX FOUNDATION LIMITED.

The technology described in this paper is under development by RDX Works Limited, supported by RADIX TOKENS JERSEY LIMITED. This paper represents an update of the development of technology which may be comprised in software to be published for general use under the RADIX ® open-source license.

The Radix Foundation Limited is established to promote education research in respect of the development and management of decentralised cryptographic systems and to support knowledge creation and research in associated technologies.

The views and ambitions for the Cerberus consensus protocol and the Scrpyto language are those of the developers. The Radix Foundation does not express any view as to the viability of the technologies described herein.

Table of Contents

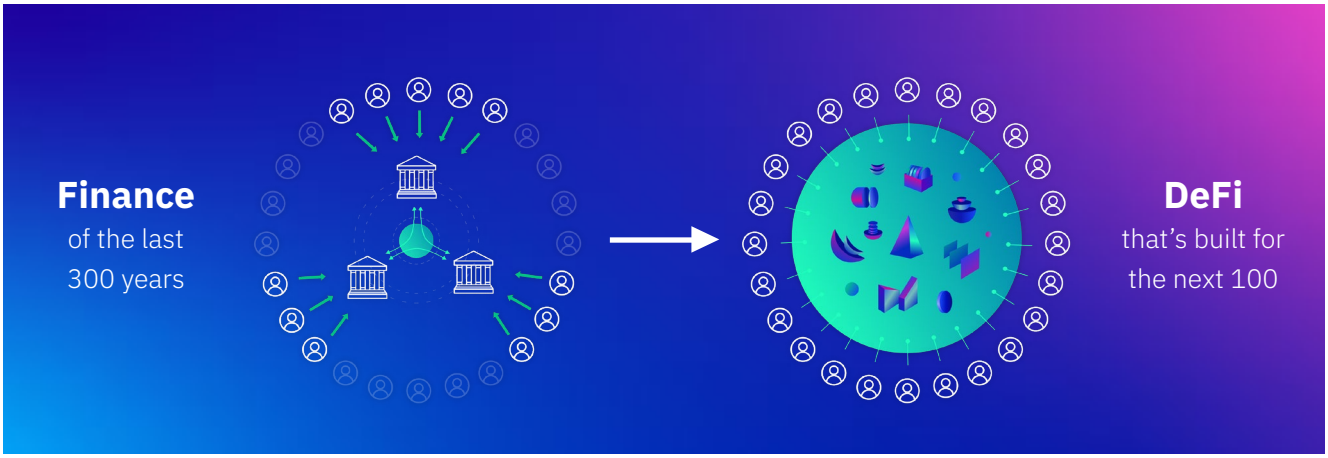
Introduction

12 years on from the invention of blockchain technology, DeFi has emerged as its most important and revolutionary use. The ability to securely manage financial assets of real value on a decentralized network means nothing less than the first steps of replacing a financial system that is inherently monopolistic with a better one that is open, democratic, and ceaselessly innovative.

DeFi has the potential to finally bring finance into the internet age. Each of us can tell something is wrong with the current system. Why do banks profit from my savings when I get near-zero interest? Why can't I access certain kinds of banking services or investments just because of who I am or where I live? Why is it so expensive to send money to other countries? How are disasters like the 2008 crash even possible? Shouldn't these things be fixable?

The reason is that today's financial system – operated through a limited number of closed institutions – is *inherently* exclusionary, slow to address user needs, and wildly inefficient and uncompetitive. It's everything the internet *isn't*.

DeFi allows us to imagine a better system than the one we've had for over 300 years. Instead of trusting our assets to closed institutions and only having access to the products and services they choose to offer, we can move to a system where everyone can hold their assets on a shared, open network. On that network, if developers with great ideas can easily create safe, powerful finance applications, then all of finance can be rebuilt in a way that is inclusive, convenient, cheap, and highly competitive. That's how we enable innovators to fix the problems we all experience.



Yet the DeFi ecosystem we have today, and its developer community, are nearly insignificant. **DeFi of today is less than 0.05% the size of the $400Tn[1] total value of global finance.**

What's holding back a world of developers from seeking all of the opportunities DeFi should offer?

Imagine what it really takes to remake global finance in a democratized model at the scale of *hundreds of trillions* of dollars and *billions* of users. It means *millions* of developers with millions of great ideas building better financial products and services. It means a system where the dApps (decentralized applications using smart contracts) that those developers write can offer just as much functionality at the same level of security that we're used to today – or better – at unlimited scale and low cost on a decentralized network.

The platforms hosting the first baby steps of DeFi can't meet this standard, and without a better solution the potential of DeFi will remain a dream.

Radix is removing the technology barriers limiting the expansion of DeFi by building a layer-1 protocol that can directly address the needs of DeFi at global scale for the next 100 years. This requires a full stack approach, re-engineering consensus, distributed virtual machines, executable on-network code, DeFi application building and developer incentives.

[1] https://www.statista.com/statistics/421060/global-financial-institutions-assets/

Summary

This paper describes how the Radix platform will provide an integrated technology solution to the multiple problems that limit DeFi's potential.

The growth of DeFi rests on the shoulders of developers who will build the dApps that replace the traditional closed financial system, and the barriers holding back DeFi are precisely those that hold back developers. We believe there are four major barriers that DeFi developers face today that Radix can solve with four crucial technologies that make up the Radix platform.

1) Asset-oriented smart contract paradigm

The problem today: On Ethereum, and other smart contract platforms today, it is extremely difficult to develop production-quality dApps. Hacks, exploits and failures of Solidity-based dApps are common - even when built and audited by experienced developers. Even simple functionality requires complex, difficult-to-analyze code to be close to production ready. Large amounts of money chase a small number of professional smart contract developers that have spent years learning to work within the paradigm.

The Radix solution: We need to move past the Ethereum conceptualization of smart contract development and give developers a new paradigm of asset-oriented smart contracts intended for DeFi. The **Radix Engine**[2] smart contract environment, together with the **Scrypto** programming language, finally make production-quality DeFi dApp development easy and safe – without limiting what the developer can build. With these tools, more complex dApps become practical, the pool of available dApp developers can rapidly grow, and developers finally stand a chance of avoiding hacks and exploits.

2) On-network DeFi "lego bricks"

The problem today: All functionality on Ethereum must be deployed as standalone smart contracts that resist modularity, reuse, and standardization. Developers deploy largely redundant reimplementations of this functionality over and over making even standards like ERC20 and ERC1155 – for functionality as simple and common as tokens and NFTs – surprisingly difficult to establish and unreliable in use. Even higher-level common DeFi functionality like liquidity pools, swaps, oracles, or multi-sig accounts are virtually impossible to make modular and reusable at the level developers desire.

[2] The initial Radix mainnet launch, Olympia, includes Radix Engine v1. The Radix Engine features described in this paper pertains primarily to features of the Radix Engine v2 to be rolled out across the Alexandria simulator and Babylon mainnet releases.

The Radix solution: We make code modularity, reuse, and standardization not just a matter of copy-and-paste, but a first class network feature. An on-network **Blueprint Catalog**[3] allows developers to contribute to and access pieces of functionality that can be directly reused, configured, combined, and extended on-network – and are actually operational and proven in use every day. The network becomes a true shared computing environment, taking open source development into the decentralized era.

3) Self-incentivizing developer ecosystem

The problem today: Across DeFi platforms today, developer incentives mostly consist of fixed, centralized funds administered by well funded foundations. Huge sums of money are wasted on projects that go nowhere, and often the best developers have difficulty finding a path to being rewarded for their contributions of what could be valuable DeFi functionality. The incentive model is so broken that great teams often spend their time chasing grants rather than focusing on building communities and truly valuable, sustainable dApps.

The Radix solution: To sustainably grow a community around Radix's developer tools, we need a system that creates a decentralized, self-incentivizing developer ecosystem – similar to how blockchains currently self-incentivize their network infrastructure. This is the purpose of Radix's on-network system of flexible **Developer Royalties**[4] which allows Scrypto developers to be rewarded for code that is truly valuable to users and other developers – in every transaction.

4) Unlimited dApp scalability

The problem today: Ethereum continues to demonstrate how slow network throughput creates high network fees and excludes users even at the DeFi's current small scale; reaching DeFi's potential of millions of dApps and billions of users is unattainable. The solutions being offered by "scalable" networks such as Ethereum 2.0, Near, Polkadot, Cosmos, and Avalanche fail to deliver improvements for practical smart contract usage and compromise unlimited atomic composability between dApps – breaking one of the most important enabling features of DeFi.

The Radix solution: To complement Radix's tools to let developers rapidly build, deploy, and be rewarded for great DeFi dApps and code, we need a network consensus design able to provide not just *more* throughput, but unlimited dApp scalability at global usage without breaking composability. This is exactly what our unique **Cerberus**[5] consensus protocol, designed in tandem with Radix Engine, delivers. It allows massive parallelization of both simple transactions and complex dApps through specialized form of sharding and a breakthrough "braided" multi-shard consensus mechanism that allows all assets and smart contracts to be freely and atomically composed on a transaction-by-transaction basis.

Radix's comprehensive, integrated bottom-to-top technology approach is what will make the Radix Public Network the only place where a world of DeFi developers will have everything they need to remake global finance.

Let's look at each of these technologies one by one.

---

[3] The Blueprint Catalog is expected to be available in the Babylon mainnet release, when Scrypto becomes deployable to the network.
[4] The Developer Royalties system is expected to be available in the Babylon mainnet release, when Scrypto becomes deployable to the network.
[5] The initial Radix mainnet launch, Olympia, includes a simplified version of Cerberus consensus. The capability described in this paper pertains to the full version of Cerberus intended for the later Xi'an mainnet update.

1. Asset-Oriented Smart Contract Paradigm

It should be obvious that creating a DeFi developer ecosystem that can remake global finance requires a highly specialized development paradigm. But today's smart contracts – conceptualized on Ethereum in 2013 before anyone knew that global-scale DeFi was the problem to solve – have failed to deliver what developers need to take DeFi to mass adoption. Nonetheless, virtually every smart contract platform since has followed the same fundamental approach as the original Ethereum model.

Today, the learning curve for Solidity (Ethereum's primary smart contract language) is *years long* to reach a "DeFi ready" level of expertise. Something as conceptually simple as "create a token" means deploying a bespoke smart contract that implements asset-like behavior from scratch. The responsibility is entirely on the developer for even the most basic, common functionality. Even worse, often smart contract development requires detailed understanding of how the platform operates, with high stakes if the developer doesn't fully grasp it.

The result is an extremely small community of experienced DeFi developers, a hugely inadequate hiring pool for entrepreneurs that limits what they can create, and DeFi dApps that are as elementary as possible to minimize the risk of expensive exploits.

We hear from Ethereum DeFi developers that they typically spend only 10% of their time building core functionality and 90% trying to make it safe for deployment. And even then, we see frequent DeFi hacks and exploits resulting in millions of dollars of unrecoverable losses.

This isn't a symptom of bad developers - *it's the result of a bad development paradigm*. Every DeFi application is concerned with the correct management of assets and ensuring only authorized users can do the correct things. But today's development paradigm forces the developer to implement the very concepts of assets and authorization from scratch without a safety net. The developer must then layer these bespoke implementations with their own dApp's logic – and safely integrate with other dApps that are burdened in the same way.

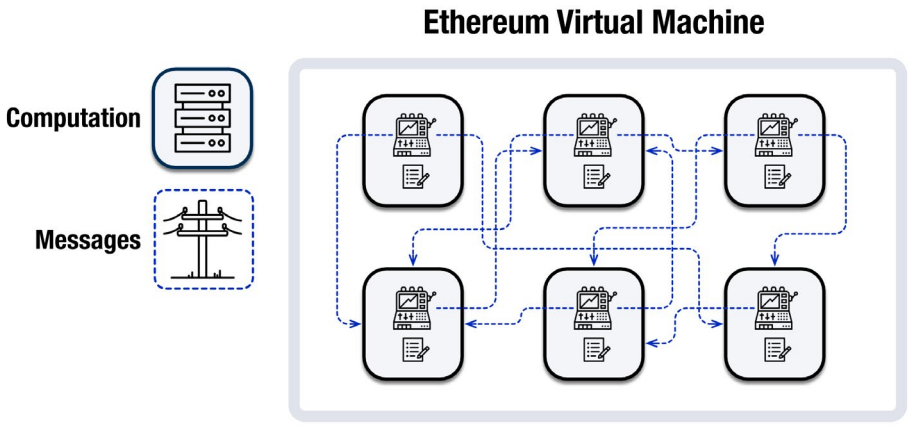
The result is that developing virtually any production-ready dApp rapidly becomes a mass of unwieldy code that is extraordinarily difficult to analyze for safe asset management and authorization. The opportunities for exploitation are nearly impossible to avoid.

Fixing these problems means creating a new smart contract paradigm: an asset-oriented smart contract environment and programming language that can finally make production-quality DeFi dApp development both practical and safe when managing millions of dollars of assets.

Today's Smart Contract Application Environment

The fundamental notion of a smart contract was established by Ethereum and its Ethereum Virtual Machine (EVM) application environment. With the EVM, the blockchain network provides a public platform that can perform computation using code that a developer deploys to the network. Each smart contract has its own private internal "state" (its set of internal data) that the smart contract logic can update.

To make these smart contracts useful, they must also be able to communicate with each other. For example, if we want a DeFi smart contract to be able to do something as simple as hold some tokens, it must be able to communicate with an independent ERC-20 contract that implements that token via a list of balances and methods to adjust those balances.



In this model, *all* functionality must be implemented as smart contracts sending messages to each other. A token? That's a smart contract that keeps track of a list of balances associated with a list of public keys. A multi-sig account? That's a smart contract that requires signatures by a list of public keys in order to take action. DeFi dApp? That's definitely a *big* smart contract.

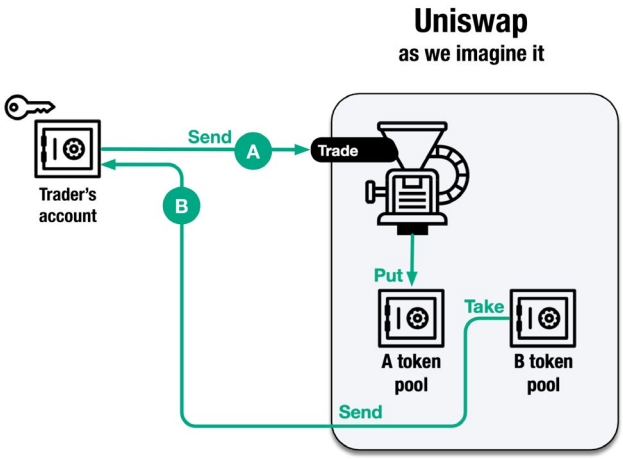
That is essentially the extent of the platform features provided by the EVM for smart contract developers. While these features are sufficient to enable smart contract functionality, we start to encounter problems when creating practical DeFi applications. Because absolutely *everything* is implemented in little smart contract silos, doing just about *anything* rapidly becomes a very complex set of messages being passed around, complex logic to ensure the right actions are taken in response to those messages and catch all possible errors, and lots of data inside each smart contract's state to keep track of everything.

To start, let's consider assets like tokens. Developers have no choice but to create each token as its own independent smart contract. That means that any time a user or smart contract needs to interact with tokens, it must do so by sending messages to those separate master smart contracts that control the internal balance sheets. Now, remember that every DeFi dApp is constantly interacting with many types of tokens, performing complex movements of tokens between users and often other smart contracts. The amount and complexity of code the developer must create starts to explode.

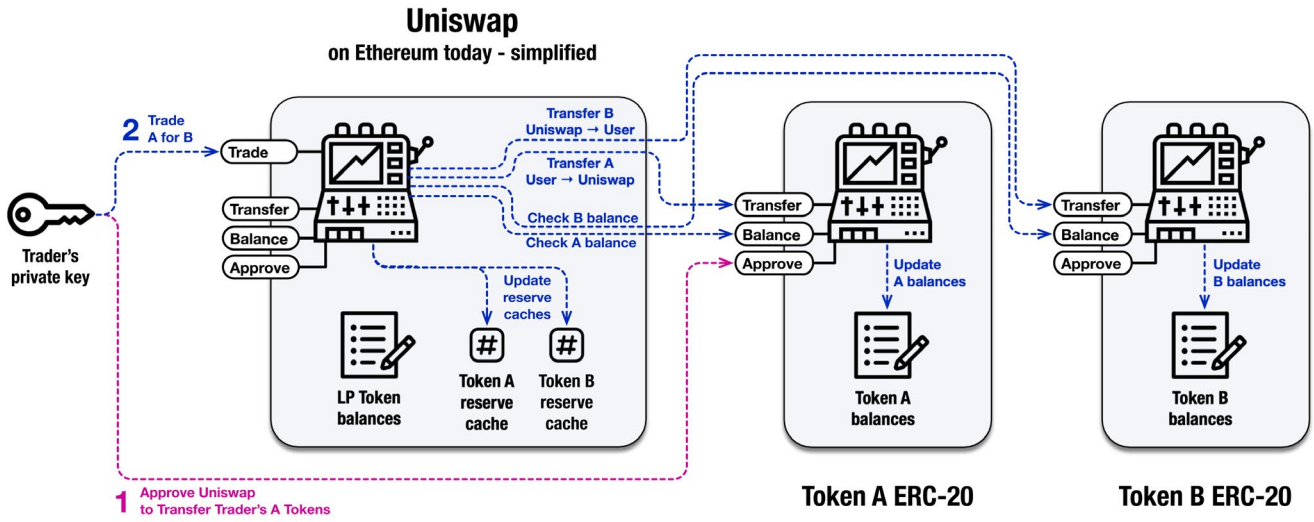
To take things a step further, let's look at a DeFi dApp that conceptually is quite simple, Uniswap. This is what it does:

- Liquidity providers (LPs) contribute pairs of tokens into a pool to be traded. LPs are given a special LP token that acts as a claim ticket for their share of the pool.
- Traders can swap between these pairs of tokens automatically from the pool, according to a clever formula, paying a small fee to the pool.

To simplify the picture even further, let's look at only Uniswap's swapping functionality. You might imagine that a swap transaction with the Uniswap smart contract would work something like this:



Conceptually, it is quite simple. Tokens are sent in, and the smart contract logic need only determine an amount of tokens to send back from its internal pool. Instead, here is a *greatly simplified* view of how just this one simple smart contract call works on the real Uniswap smart contract on Ethereum:



All of this complexity, redundancy, and opportunity for missed errors is caused by the developer needing to work around the limitations of the EVM application environment itself – notably that all of the frequent interactions with assets must be implemented in somewhat convoluted fashion at the application layer rather than in the more intuitive way that we might have imagined.

It works, and some very clever people have cobbled together the first DeFi functionality atop this architecture, but **it's terrible for the developer and perilous for the user** (just see the number of hacks and exploits listed here).

How do we create a better development environment than the EVM approach? We start with an observation: Interacting with assets isn't a special case – it's the primary subject of every single DeFi transaction and DeFi dApp, so why aren't assets a feature of the platform rather than pushing that responsibility up to the development environment?

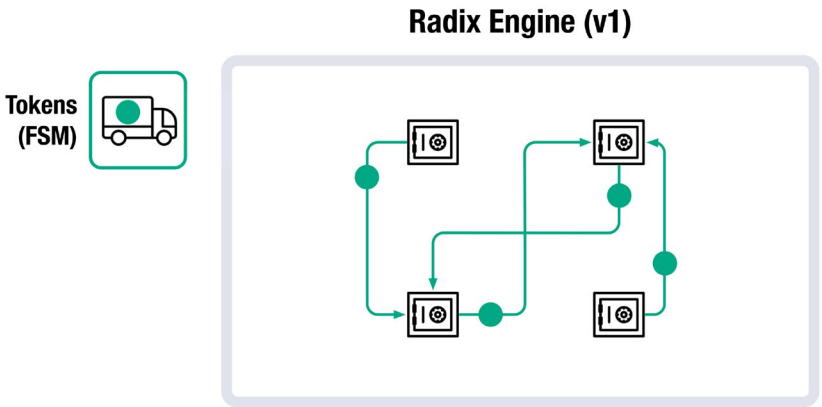
Radix Engine: An Asset-Oriented Smart Contract Environment

Radix has taken a different approach where **assets are a global feature of the platform itself**, rather than implemented over and over again at the smart contract level.

Radix calls its application environment Radix Engine. The first version of Radix Engine is already running today on the Olympia release of the Radix public network. With **Radix Engine v1**, tokens aren't implemented in smart contracts, but can be created by directly requesting them from the platform with some desired parameters.

Tokens in the Radix Engine environment aren't entries in a thousand separate balance lists; they are treated by the platform as "physical" objects that must be held in accounts and moved between them. Radix Engine accounts act like actual vaults of tokens controlled by the user, unlike on Ethereum where a user's token holdings are spread among separate smart contracts that each hold an entry for their public key. This goes for not only XRD, the Radix network's utility token, but all tokens created by users.

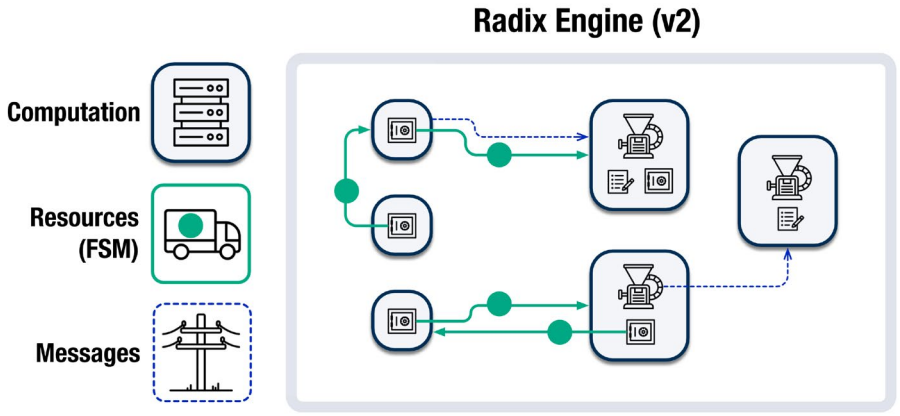
Radix Engine itself guarantees this "physicality" of behavior, using a well-constrained *finite state machine* (FSM) model. FSMs are typically used for mission-critical systems where correct results must be guaranteed. This is a good match for enforcing the correct behavior of tokens.



In this model, sending tokens isn't a matter of sending a message to a smart contract and trusting that it will update some balance entries – it is telling the platform itself *"I wish to send these tokens that I hold"*. Errors like double-accounting aren't just avoided; they *simply aren't possible.*

This is the core of the asset-oriented paradigm. Already Radix Engine v1 offers a vastly more intuitive and easy-to-use model for simple tokens, but that alone isn't sufficient for DeFi. To enable smart contracts that can serve as the basis for DeFi dApps, we need to add in decentralized *computation,* as in the EVM.

Radix Engine v2 starts with v1's asset-oriented approach, and adds computation of powerful smart contract logic written in Radix's asset-oriented programming language, Scrypto (as well as messaging between smart contracts). Radix Engine v2 also expands v1's Tokens FSM into a more powerful and general form called *Resources* (more on this below).



Compared to the EVM model, this drastically shifts the character of smart contract logic that the developer creates. Because the *Resources* FSM can be used to transact tokens (and NFTs and more!) much more easily and intuitively, smart contract code complexity drops. As with double-accounting errors, many issues with reentrancy in dApp transactions involving resources become simply impossible. And the amount of message-passing required becomes minimal, only needed to convey actual information between smart contracts rather than orchestrate movements of assets.

In short, the developer can lean heavily on REv2's *resources* to handle the great majority of what makes smart contracts today fail to be easy, safe, reusable, and composable. REv2 accomplishes this not by forcing the developer to use specialized code, but by letting them rely on resources to create code that is **simpler and safer at the same time.**

Because the result is so much different from today's style of smart contract, Radix smart contracts have a new name: *components.* Compared to today's smart contracts, *components* should have clearer function, be more modular and composable, and work more like reliable machinery.

Radix Engine and Scrypto also further encourage modularity and reusability by introducing *blueprints* that are on-network templates of useful functionality that can be instantiated over and over into components, as we'll see later.

Scrypto: An Asset-Oriented Smart Contract Language

To complement the Radix Engine application environment, we need a programming language that exposes Radix Engine's unique features while maintaining a familiar development experience with expressive logic.

Radix's solution is **Scrypto**.

Scrypto is based on Rust – and keeps most of Rust's features – but adds a range of specific functions and syntax for Radix Engine v2. It isn't just Rust running on a public DLT network; it's an asset-oriented language that allows Rust-style logic to interact not only with data (as with typical programming, and most smart contracts) but with **assets as a native, first-class citizen.**

Rather than jumping straight into what Scrypto code looks like, let's survey Scrypto's asset-oriented features, and the lifecycle of a *component* (a Scrypto smart contract). You'll see how Scrypto naturally allows the developer to focus on their own business logic and lean on Radix Engine for intuitive, safe handling of assets. The result is that building DeFi in Scrypto finally provides the ease, safety, reusability, and composability that DeFi needs to fulfill its world-changing potential.

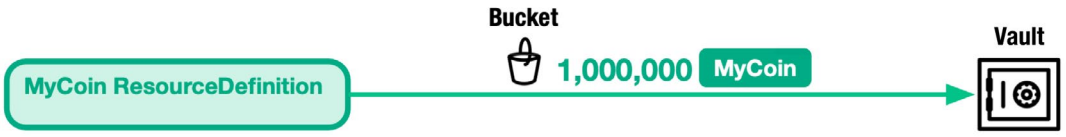
Resources, Buckets, and Vaults

Let's start with the assets themselves, such as a token that you might want to create. In Scrypto, tokens and NFTs aren't smart contracts (or components) at all. Instead they are a created as *resources* (the platform-native FSM-based "physical" assets provided by Radix Engine v2). To create a new resource, like a token or NFT, you use a built-in Scrypto function where you can specify the parameters you want for that resource.

For example, let's say you want to create a fixed supply of 1,000,000 tokens called *MyCoin*. You specify this supply and name (and other parameters, like specifying that it should be fungible) to the resource creation function which then creates a *resource definition* and returns you 1,000,000 of your new *MyCoins*. (Note: A resource definition isn't like an ERC-20 contract; it's simply a way to refer to the parameters associated with that supply of resources, wherever they may be.)

Because Radix Engine requires that resources always be "physically" located somewhere, the MyCoins returned from the resource creation function must immediately be put in a temporary container called a bucket. A bucket isn't a variable that holds a number; it behaves like *an actual container that resources can be put into or taken from*. Buckets vanish at the end of execution, however, and resources must be stored somewhere at the end of execution – so you'll also need a more permanent resource container called a *vault. Vaults* are always located within a component (more on this in a moment).

So we can picture creation of a resource like this, with newly-created tokens going into a *bucket* that is then immediately emptied into a *vault* for storage until a later transaction:



This can be done in just a couple of lines of Scrypto code.

Scrypto provides functions that allow you to do things like *take* any quantity of resources from a bucket or vault, and *put* them into other buckets or vaults. We'll see later how this makes interacting with resources much more direct and safe than the typical smart contract method.

Components and Methods

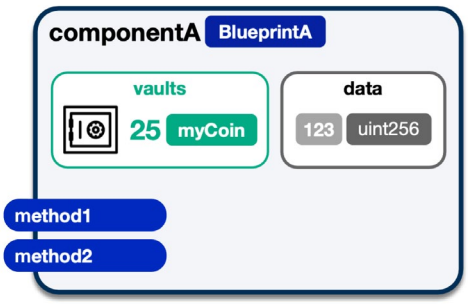
Now we have resources, but what about smart contract logic?

The Radix Engine version of a smart contract is something we call a "component". Because components are designed around resources, they don't just hold data (like ints or strings); they also hold vaults that contain all

resources owned by the component. In fact, *every* vault is owned by a component (and a component may own multiple vaults).

So the component's code defines the kind of data it holds and the kinds of vaults it holds (each vault only accepts a specific type of resource). It also defines a list of *methods* that contain all of the logic of the component and create the component's interface to the world.

Putting it all together, we can picture an example component like this, ready to be used via its *method1* and *method2:*

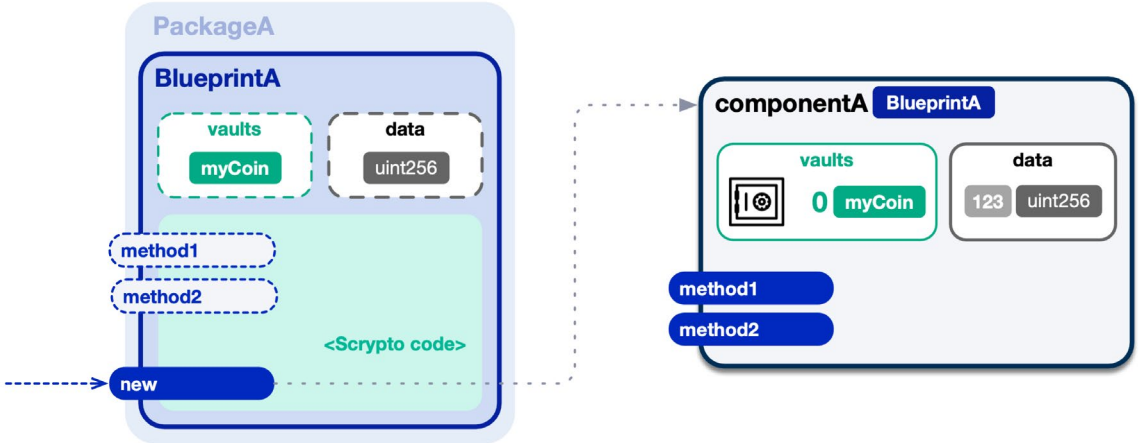


Blueprints and Packages

But wait, what is that "BlueprintA" we see next to the component's title? That is the name of the *blueprint* the component was *instantiated* from. Unlike on typical smart contract platforms, active components aren't simply deployed directly to the Radix network. All components start their life as a *blueprint* that is deployed to the Radix network and that acts like a template from which many component copies may be instantiated (each perhaps customized with input parameters).

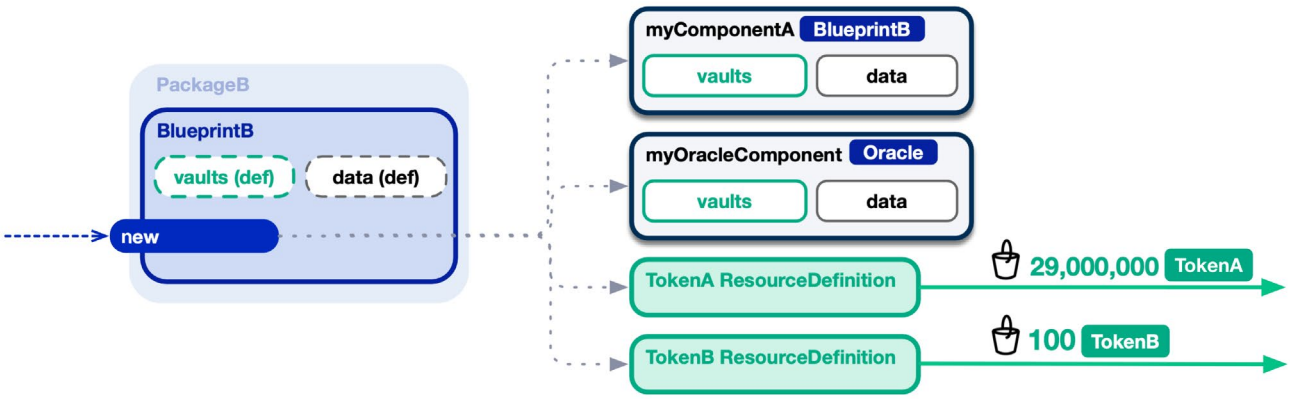
This means that the Scrypto code (including its definition of methods and types of data and vaults) is actually in the *blueprint*; the component's logic and structure is fully defined there. But the component's actual *state* - its data and resources - belong exclusively to that individual component, not the blueprint.

Once a component has been instantiated from a blueprint, it becomes active for use on the network by users (via method calls from transactions) or other components (via method calls from Scrypto code). Instantiation of a component from a blueprint happens using a function on the blueprint that performs the instantiation, like this:



(Note: You might notice one last little detail – the "PackageA" enclosing the blueprint. Multiple blueprints may be grouped together by the developer in a *package*.)

Blueprints encourage reusability of Scrypto code, and also give the developer great flexibility to perform a variety of setup and configuration actions. In fact, an instantiator function offered by a blueprint may do much more than just instantiate a single component. It may instantiate multiple different components (including from other blueprints), as well as create new resources. Here's an example (with the blueprint and component details simplified) of one blueprint instantiating two components and creating two new resources:



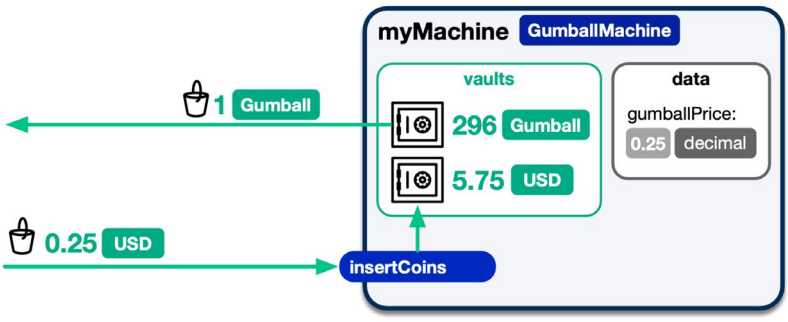
In addition to blueprints created by developers, the Radix team intends to deploy its own set of useful blueprints for anyone to use and instantiate on-network.

Using Components with Resources

Now that we can instantiate components, how do we interact with them programmatically? Similar to typical smart contracts, we use the methods offered by the component. But Scrypto methods have a significant difference: they can directly accept (buckets of) resources.

Passing resources to a component method isn't just sending a number or a reference to some tokens. Radix Engine treats it as *actually transferring* the *ownership of those tokens to the component*. Once the component has received a bucket of resources (or multiple buckets), it can *take* resources out of that bucket and *put* them elsewhere like a vault it holds, or a different bucket. The Radix Engine guarantees that the caller can no longer access that bucket – it has *transferred it away.*

The result is a much simpler and safer way of using tokens and other kinds of assets with component-based dApps on Radix. Take the example of a gumball machine component that accepts some USD tokens in exchange for a Gumball token (with a supply held in the gumball machine's vault):



A bucket of 0.25 USD is passed to the *insertCoins* method of the *myMachine* component (previously instantiated from a *GumballMachine* blueprint), and the machine's logic sees that the correct price has been paid, puts those tokens in its USD vault, takes 1 Gumball from its Gumball vault, and passes it back to the caller. The component's logic might also send back change if the user passed in too much USD.

Just how we'd expect a gumball machine to work!

On Ethereum, this would have involved the user calling a USD smart contract to give permission for the machine to withdraw on their behalf and telling the machine that they wish to input 0.25 USD, with the machine then calling the USD contract to do the withdraw, calling a Gumball contract to do the send to the user, and probably updating an internal cache of the number of Gumballs remaining for error checking. Every one of those extra smart contracts, and all of those smart contract calls, are opportunities for error – and this is just a simple gumball machine!

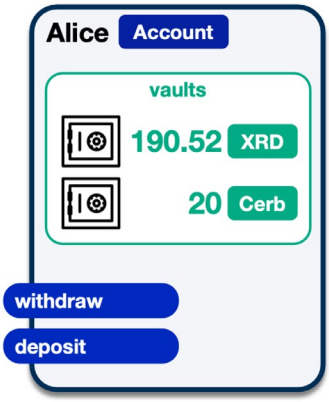
Transactions and Accounts

While you can imagine how one component might interact with another like this, at some point there must be a user transaction that kicks all of this off. How do transactions in the asset-oriented model work? What about accounts?

In short, **transactions with Radix Engine v2 are also asset-oriented**. They describe how the user wants resources that they control to flow to other components. They can even describe how to handle resources that are *returned* from a component – whether claimed by the user or passed on to another component in a composed multi-component transaction. (This also is a tremendous difference from Ethereum where a transaction is typically just a message to a smart contract that the user hopes will produce the desired result – and where composition of multiple smart contracts is not possible on the fly in transactions.)

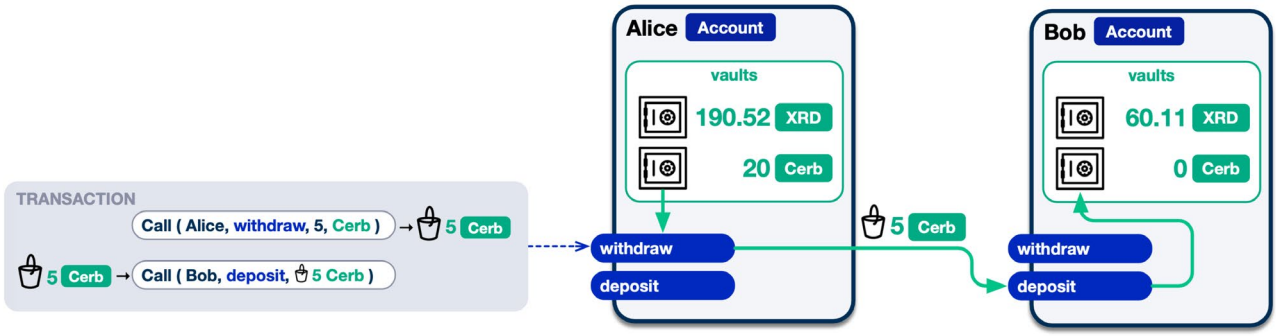
But to make sense of this type of transaction, we need to understand how accounts work.

An account in Radix Engine v2 is actually just a special kind of component that holds vaults, just like any other component. Each account component is instantiated from a special *Account* blueprint on the Radix network that provides useful methods. So Alice's account might look (in simplified form) something like this:



Let's say you're Alice and you want to send Bob 5 *Cerb* tokens. Your transaction would specify that you want to withdraw 5 Cerb from your Cerb vault (which you have permission to do via your signature) and then pass all of those tokens to the deposit method of Bob's account component. As we discussed earlier, resources must

always be located somewhere, so a bucket is used to pass them to Bob in the transaction. So the transaction contents would look something like this:



(Note: That's not the actual syntax of transactions, but it gives you the right idea.)

Notice again that when we pass the 5 Cerb to Bob, we are *actually passing a bucket containing those resources* to the deposit method, not a reference or a call to a token smart contract elsewhere.

Interacting with any other component, such as our gumball machine above or a DeFi dApp component works in just the same way. You typically withdraw some tokens from your account, and pass them to the relevant method of the component you wish to call (perhaps along with some data that it may also require as input arguments).

Authorization with Badges

Safely managing tokens and other assets is one of the recurring challenges with Ethereum and other typical smart contract platforms, but another is that of authorization. Virtually every smart contract has some methods that it wishes to protect. For example, authorization might be used on special methods reserved for the contract's owner, the rights to mint and burn tokens, or to restrict access to a whitelist of members.

Today this is typically done by keeping track of a list of account addresses (or smart contract addresses) that are allowed to do certain things. Unfortunately handling authorization in this way is inflexible and frequently creates a new vector of attack if the right checks on the list aren't performed correctly.

This problem is also solved elegantly by Radix Engine's asset-oriented design. Previously we've only mentioned that resources can be used to create tokens and NFTs, but there is another usage that we call *badges*. A resource used as a badge has the same sort of "physical" behavior as a token (and can even be a token), but badges are used for authorization. Scrypto provides special functions that easily allow components (including account components) to require *presenting* a badge to use certain of its methods.

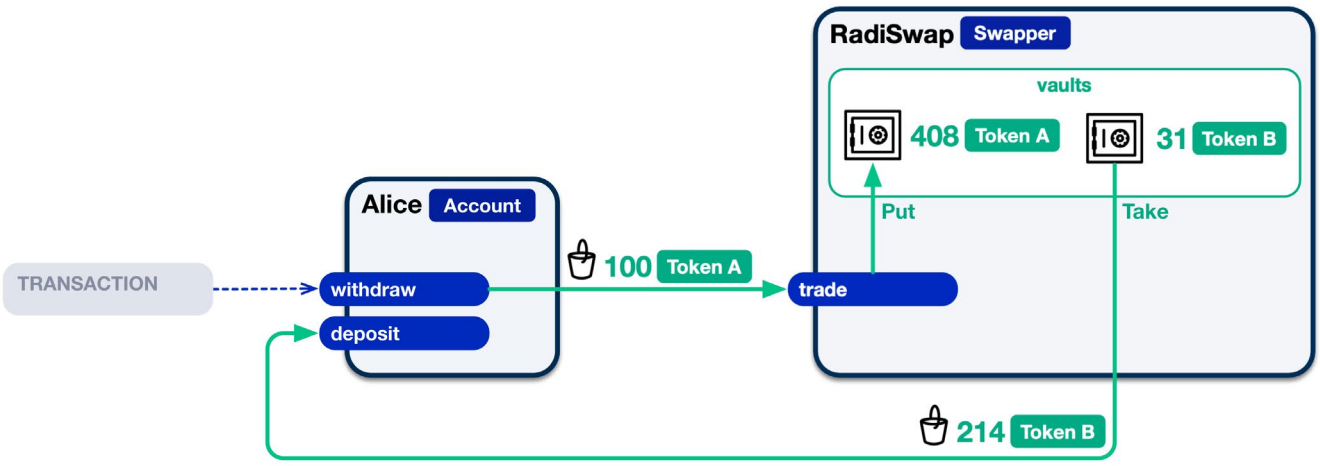
Presenting a badge isn't the same as transferring it; it provides the presentee with a reference to that badge so that it can be certain that the presenter in fact owns it without the badge changing ownership. This makes badges excellent for use in even very complex authorization patterns. Rather than checking a whitelist of addresses, a set of custom badges can be created and issued to accounts (or other components). Methods can specify the badges required to be usable. If the right badge isn't present in the transaction making a given method call, it is rejected right away.

More work remains to be done on the precise implementation of badge-based authorization, but they will offer a powerful tool to make component logic yet more safe, predictable, and flexible.

Benefits of Scrypto

Now, understanding the new tools provided by Scrypto (and Radix Engine behind the scenes), hopefully you can see the benefits of an asset-oriented approach. If a developer wishes to write something like Uniswap in Scrypto, they can focus on writing only the code that matters: the unique swapping logic. Interacting with tokens (as resources), pools (vaults), and users (just another component) is direct and simple.

In fact, let's return to the comparison between *"Uniswap as we imagine it"* and *"Uniswap on Ethereum today"* from earlier. What would a swap transaction with the Scrypto implementation look like? Something like this:



Just as we imagined it to be! Using Scrypto, the only code that needs to be written is an implementation of the *trade* method that looks at the incoming bucket of Token A resources, calculates the current exchange rate (based *directly* on the contents of its own internally-held pool vaults), and returns the right amount of Token B. All of the token manipulation performed by the *trade* method implementation happens via simple take/put functions on buckets and vaults.

The transaction itself would look something like this:

No need for the user to give blanket withdrawal permission to Uniswap here; just a direct specification of the desired movement of resources between components. Safe, flexible, and naturally composable.

This sort of simplicity and directness should fire the imagination of developers with great ideas for the next generation of DeFi, and finally make possible a tidal wave of the kind of rich, robust dApps that are needed to revolutionize and remake global finance for the better.

2. On-Network DeFi "Lego Bricks"

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 running on a decentralized network gives us an exciting new possibility: **putting community collaboration and package manager-like functionality directly onto a common network.**

This means an on-network mechanism for Scrypto Blueprints to be modularly used, leveraged, updated, versioned, extended, and combined – and creates a powerful tool for developers. Rather than simply importing raw code into their project, they may directly use pieces of Blueprint-based functionality that are *operational and proven* within the shared computing environment on the network. Blueprints deployed on-network in this way don't just contribute indirectly to a developer ecosystem; they *directly extend the available functionality of the Radix platform.*

And with such a system, a developer need not build a fully-functional standalone dApp to usefully contribute. Blueprints that do one thing very well, and are built to be easily reused or combined with other Blueprints and Components, can become de facto 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

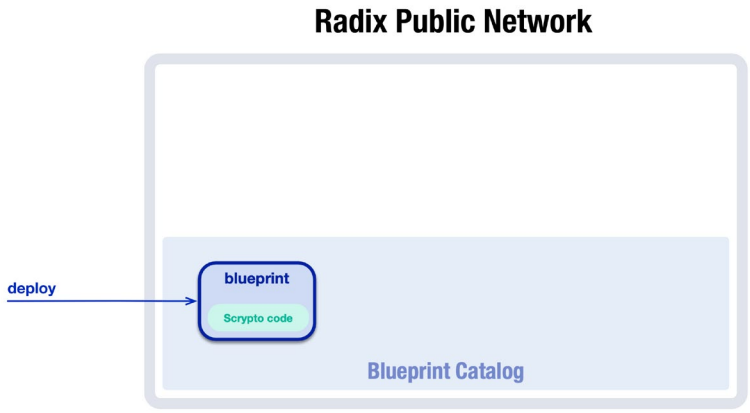
both large and small. But now we have the technology to do so directly within a decentralized computing environment.

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

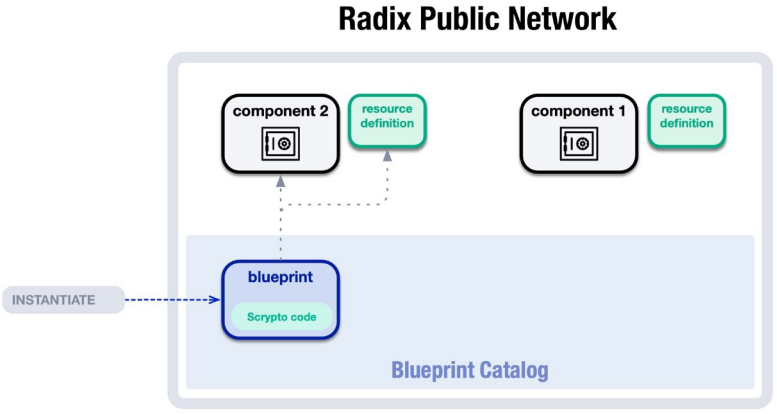
The Radix Blueprint Catalog

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 Blueprint Catalog changes this model.

When Scrypto code is deployed to the network, it starts its life as a Blueprint that is added to an on-network registry called the **Blueprint Catalog.**



As described earlier, to instantiate a Blueprint from the Catalog into an active Component for use, a developer calls a function defined in the Blueprint that brings a new Component (or multiple Components and/or resources) into existence, often using specified parameters to customize that instance. Multiple Components can be instantiated from a single Blueprint in this way.

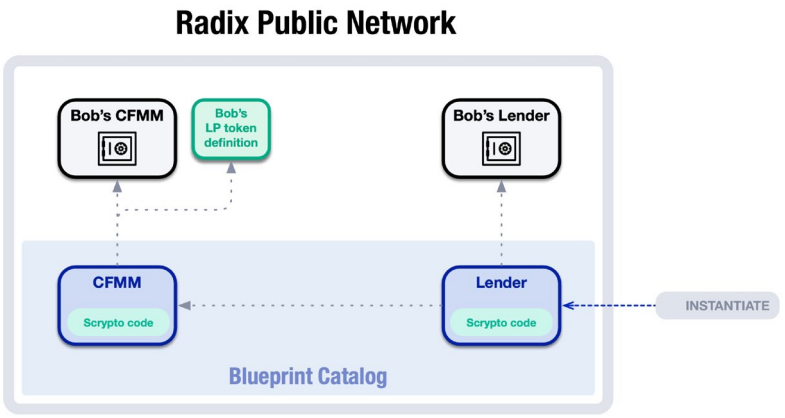


By instantiating Components from a universal on-network Catalog, Radix makes it very quick, easy, and safe for any developer to access functionality created by others. Rather than just importing some code that *seems*

useful, a developer can see that a given Blueprint has actually been instantiated by others and see how the corresponding Components are working.

A developer may even not need to learn any Scrypto to access some simple pieces of functionality. We've already seen how tokens can be created simply by requesting a customized resource from the platform, which can be done even within a transaction without Scrypto code. Similarly, Blueprints may be customized and instantiated via transaction.

Another way of making use of Blueprints in the Catalog is to instantiate them programmatically from other Blueprints or Components to modularly construct a dApp's functionality.



For example, Bob might create a *Lender* dApp Blueprint that requires use of a custom CFMM (constant function market maker DEX) to swap between assets the Lender dApp supports. If there is an existing *CFMM* Blueprint that serves Bob's needs, his *Lender* Blueprint could be written to instantiate the *CFMM* Blueprint (including configuring it) within its own instantiator function. In this way, the developer can directly include proven third-party functionality within their own dApps – all on-network.

Blueprints have their own unique address, and are associated with the creator's own unique address. Blueprints are intended to offer versioning, with each new deployed update requiring a revision bump. Updating a Blueprint does not automatically force an update onto other Components or Blueprints making use of it however; previous revisions remain immutably available on-network and existing Components or Blueprints will continue to have access to the previous revision. A developer may choose to adopt a new revision by making their own update. More work remains on the specifics of versioning and updating.

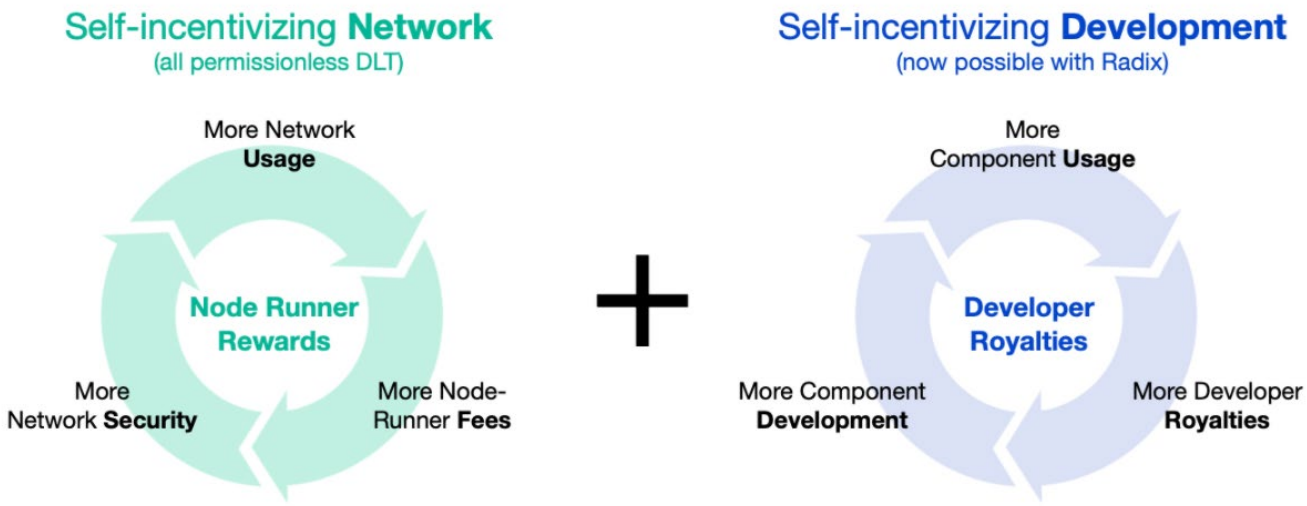
3. Self-incentivizing Developer Ecosystem

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 (or at least its security) 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 Scrypto, Radix Engine, and Blueprint Catalog enable.

Developer Royalties

While traditional methods of growing a developer ecosystem can be beneficial, like developer funds and bounties, we believe that *decentralized developer self-incentivization* can create a breakthrough in rapid decentralized ecosystem growth, creating the same kind of market-based incentives as "mining". Scrypto and the Blueprint Catalog make this possible for the first time; we call it the **Developer Royalty System.**



The core concept is that the developer of any Component/Blueprint may specify a royalty for *each usage of that Component or Blueprint 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.

Scrypto will include the ability for the developer to define their own Blueprint and Component royalty fees so that the Radix protocol (via Radix Engine) will ensure the correct payment in the transaction – in the same way it handles fees for node-runners. This is possible because the Blueprint Catalog, and the usage of Blueprints and Components are all on-network – not just off-network importing of code.

If a developer adds a highly useful, modular, interoperable Blueprint to the Catalog, or deploys a popular, useful Component, 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 for the first time enables a direct developer-to-developer marketplace for useful functionality.

How Royalty-Setting Works for Developers

The fundamental innovation behind Radix Developer Royalties is the on-network Blueprint Catalog and associated on-network 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.

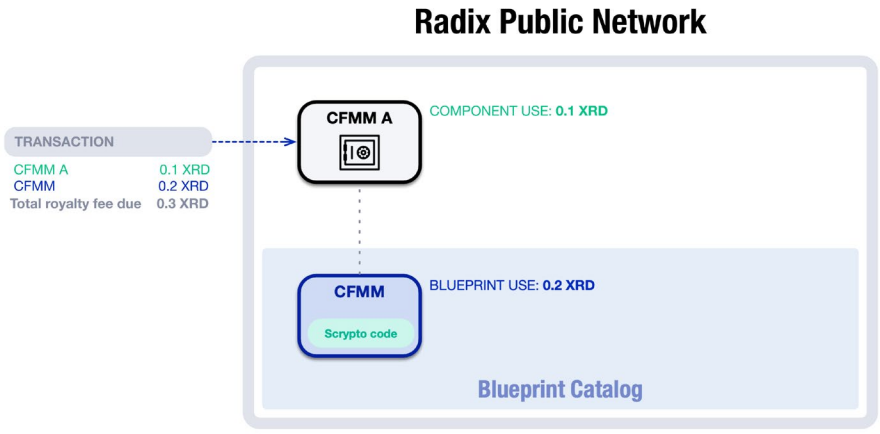
Developers must have the tools to participate in this on-network marketplace how they prefer. We start by allowing the developer to set a different royalty for different types of usage[6]. These will include:

1. **COMPONENT USE** – This is when a user transaction directly calls a method of an instantiated Component. For example, this might be the use of a DeFi application like a CFMM (constant function market maker) – either directly in the transaction itself, or if the method is called from another Component as part of a transaction.
2. **BLUEPRINT USE** - Each instantiated Component is associated with an original Blueprint in the Catalog. The Blueprint Use royalty is paid to the original Blueprint creator when a user transaction calls a method of a component that was instantiated from that Blueprint

BLUEPRINT USE royalty fees would be specified by the developer (as part of the Blueprint's code) before it is deployed to the Catalog. These can be thought of as the royalties for the "behind the scenes" value of Blueprints that have been useful for others (or the developer themself) to instantiate and use. Royalties for COMPONENT USE would be specified at the time that a Component is instantiated. These are the royalties for the use of the methods of each active, independent component, regardless of what blueprint it was instantiated by.

For a given transaction, the sum of required fees are calculated and enforced by the Radix platform when the transaction is created.

[6] While fixed XRD pricing is the simplest, we anticipate other methods should be possible in the final implementation.



Take another example of an oracle Blueprint. While the creator of a generally useful oracle Blueprint might receive a royalty for each usage of individual oracle Components instantiated from it, an oracle instantiator also can enjoy a royalty for *their specific oracle* – where that value might be derived from the data they make available through it, or through its adoption by other active dApps.

Different royalties may also be applied to different methods of Blueprints and Components. For example, some methods may be usable royalty-free, while other high-value methods may wish to have a royalty fitting to its utility.

Developers may tailor their royalties 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-network revenue streams and business models – as well as enabling royalty-free use where desired. Some use case examples are given later.

Updating of royalty pricing will also be possible. As with any other 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 Blueprint or Component would also accept the set of royalties associated with it.

With each Blueprint and 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. They would appear in a wallet in just the same way as network fees.

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 Blueprints and Components. A developer need not build out a full, complex, dApp in order to reach a point where their efforts can be rewarded; it may be even more valuable to build Blueprints that do one thing very well, and are highly modular, in order to maximize their usability by others. These Blueprints may become effectively new standards of the platform when widely adopted.

The Developer's Guide

The Blueprint Catalog and Developer Royalties provide the foundations of a fully decentralized marketplace for Scrypto code. On one side of the market, Scrypto developers may freely add Blueprints 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 Blueprints have access to immutable on-network data telling them what the transaction fees will be for a given Blueprint, how broadly it is being used, its version history (fully open source), and which Blueprints are associated with a given developer ID.

While all the right data is available on-network, 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 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 Blueprint Catalog and current instantiated Components and present them in a convenient browsing interface. However rather than presenting paid apps to users (like an "app store"), the Guide will allow developers to discover Blueprints and Components to accelerate their own development, and distill on-network data into a view of the associated developer reputation and history of development and usage, as well as transaction royalty pricing, in order to make good integration decisions.

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 Scrypto code created by others, or otherwise try to introduce exploitative Blueprints into the Catalog. This of course cannot be stopped on a permissionless ledger network. But the Developer's Guide, as an off-network service, can work to identify 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 Blueprint 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 development of core Blueprints for developers to use and extend royalty-free – but also supporting Scrypto 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 useful Scrypto code.

Case Studies of Developer Royalties in Action

With the tools offered by Scrypto 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 dApp – 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 methods 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 the Blueprint royalties to a low 0.5 XRD to encourage broad use 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 dApps. She wants to provide this through an oracle 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 dApps.

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 Blueprint royalties to zero, since it is her data she wishes to monetize (and she doesn't mind free usage of her base oracle functionality if it helps others). When she instantiates her official oracle Component from the Blueprint, she also sets Component royalty (for its key data access method) 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-network source for this data.

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 set of Guild badges that represent seniority within the Guild. She then creates two primary Components:

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

- A Revenue Sharing Component that allows Guild badge holders to claim proportional amounts of XRD from the Guild's royalties accrue its account (as the identified recipient of royalties for the Guild)

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 Blueprints and 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 badges, he configures it to use his own ItaiApp token as the basis for proportional revenue claims (based on amount of token holdings) on the royalty account. He distributes 50% of these tokens across his backers, allowing them to share directly in the returns.

4. Unlimited dApp Scalability

One clear challenge faced by DLT platforms today – especially those intended for DeFi – 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, other DLT technologies have entered the picture proposing new techniques to reach greater throughput, often measured in high "tps" (transactions per second) numbers.

Posting a high tps number alone, however, fails to meet the full scalability requirements of DeFi. To serve as an open platform for global finance, a network must provide:

- Unlimited throughput of transactions
- Unlimited throughput of DeFi dApps
- Unlimited atomic composability between dApps

Let's see how Radix provides all three through its unique Cerberus consensus algorithm, integrated closely with the Radix Engine application environment that dApps use.

Atomic Composability

Perhaps the most important feature of DeFi is the interoperability of dApps 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 smart contracts is valid and resolved all at once, or the entire transaction safely fails (if *anything* is invalid). This atomicity is incredibly powerful and is the basis for how DeFi dissolves the inefficiencies of traditional financial systems – replacing them with fast, customizable, and interoperable DeFi financial dApps.

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.

In designing Cerberus, unlimited atomic composability was a bedrock requirement that made it necessary for Cerberus to use a totally new form of sharding than has been used in blockchain or DLT to this point.

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 dApps and assets to certain shards is usually static (as in 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.

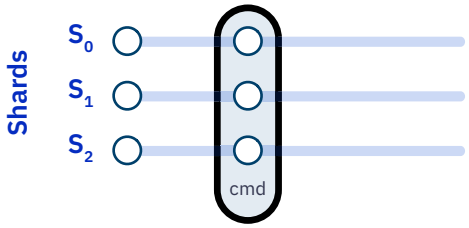
And third, we needed an application layer able to efficiently make use of this unlimited "shard space" and multi-shard consensus to process an unlimited number of transactions and dApps in parallel.

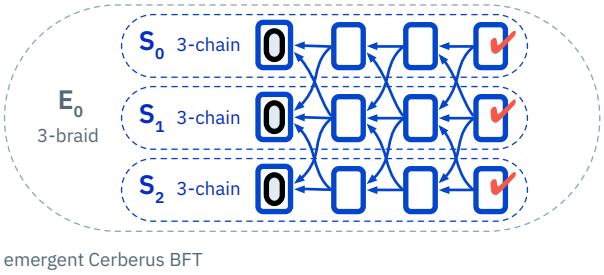
"Braided" Cerberus Consensus

The 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[7]. 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:

[7] 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.

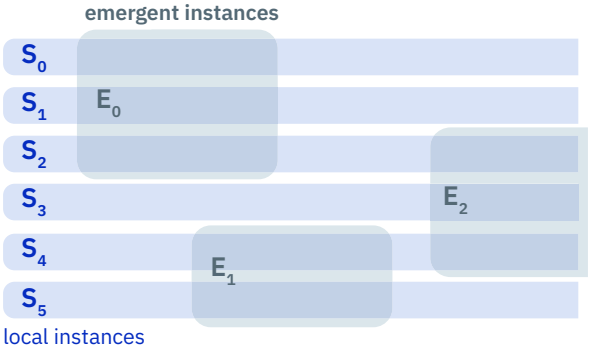
First, we move from the typical concept of global ordering to that of *partial ordering*. Virtually all DLTs assume global ordering wherein all transactions must be placed on the same timeline. Some forms of sharding essentially create multiple globally ordered timelines, but keep fixed global ordering 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.





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 any number of these instances (shards) together in a transaction using commitments created and shared by the leaders of all related shards. The result is 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 *unlimited parallelism*. This means that many things can be processed at the same time without slowing each other down – and more network demand can be served by ever greater numbers of economically-incentivized node-runners. And as the Cerberus-based network grows, atomic composability is never compromised because direct consensus between shards happens seamlessly in response to each transaction.

Scaling dApps and Smart Contracts - The Problem

To be a Layer-1 network for DeFi with full smart contract functionality, we need much more than just parallel processing of token transfers however. We need *smart contracts,* and this means understanding the relationship between the consensus layer and the *application layer*.

The application layer is what defines what the network can do. On Bitcoin, the application layer is simply the rules of the BTC token; there is no smart contract functionality. On Ethereum, the application is the Ethereum Virtual Machine (EVM) where smart contracts run, allowing the rules of the network to be programmable. Most smart contract platforms follow something very similar to the EVM application layer.

In order to provide its scalability through parallelism (using cross-shard braiding), Cerberus requires two key things from Radix's application layer:

- It must define the meaning and rules of *substates* that make up the ledger state.
- Each transaction must define which substates (and thus shards) must be included in consensus.

To understand the first, we need to know what a *substate* is. A substate is just a little record of something where some particular rules had to be followed. For example, you might want a "token substate" type that records where some particular tokens are. One bit of token substate might say "these 5 XRD are in Bob's account". But the rules of token substates would require that, for that to be true, the transaction also had to include something like "these 5 XRD are *no longer* in Alice's account". That pair of substates would describe a send of 5 XRD from Alice to Bob, and the rules ensure that no XRD can ever be accidentally lost or created. We'll talk more a little later about how Radix Engine uniquely substates for smart contracts in a very different way from Ethereum's EVM.

The reason for the second requirement is subtle but extremely important to scalability. If a key part of scalability through parallelism is only conducting consensus across the shards that are necessary (through *partial ordering*), then the application layer has to tell Cerberus *which shards are relevant* for each transaction. However, this is only possible with an application layer that knows what things are relevant – which substates matter to do what is desired.

Radix Engine Parallelizes dApps and Smart Contracts

So the big question is this: If we want Radix to process transactions – ranging from simple tokens sends to complex multi-dApp DeFi interactions – how does Radix Engine define substates and how does it define what substates are *relevant* in a given transaction?

The answer is three key choices in how Radix Engine is designed:

1. Radix Engine treats tokens as global objects at the platform level. This is important to let us parallelize the movement of assets as much as possible.

As discussed in previous articles, every DeFi application revolves around managing assets, and so Radix Engine makes assets a native feature of the platform as *resources*. Resources are the basis for tokens, coins, and even things like "badges" used for authorization.

This has enormous benefits for development, but it also means that all assets can be freely scattered around substates and shards as needed. The typical EVM model only implements tokens through individual smart

contracts, which causes an immediate bottleneck for sharding since all transactions involving a given type of token have to go through that single smart contract. Global resource-based token assets on Radix give us the flexibility we need.

2. Radix transactions are unique and based on "intent". This is important to enable high throughput through dApps without conflicts.

You might assume that, in the substate model, transactions would have to specify the substates (and thus shards) needed – as in our example above with the matched pair of substates that created a send of 5XRD from Alice to Bob. And in fact that *is* essentially how transactions work on a few UTXO-based networks like Bitcoin and Cardano.

However, this approach creates serious problems for smart contracts that process lots of transactions at the same time (ie. most of them). One transaction may be okay: my wallet can read the smart contract and determine, for example, which tokens should move in and out of it as a result and include all of those substates in the transaction. But think about *hundreds or thousands* of people using a DeFi smart contract like a Uniswap-like exchange dApp. My wallet might identify some tokens held by the smart contract that should come to me as a result of a swap and include those substates in the transaction (I don't really care which tokens come out, but the transaction must identify specific ones). But by the time the transaction is submitted and processed, it becomes very likely that somebody *else's* wallet identified the same tokens. Now suddenly my transaction is invalid and fails.

In fact, this is exactly what happened with the first DEX launched on Cardano. Many blamed this problem on Cardano's "eUTXO" style state (which has some similarities to Radix's "substates"), but really the issue wasn't with eUTXO; it was the fact that *Cardano's transactions specify the eUTXOs directly,* creating the problem that many transactions may specify *conflicting* eUTXOs.

Instead, transactions with Radix Engine specify *intent*. You can see what this looks like in my previous article about Scrypto. To use the DEX example, rather than the transaction saying...

> "these *particular* tokens (eUTXOs) should be sent and received in the swap"

... a Radix transaction says...

> "I want to send this *quantity* of tokens (I don't care which ones) into the DEX, and I expect that I will receive a certain *quantity* of other tokens (I don't care which ones) as a result."

The specific substates don't appear in the transaction submitted by the user. So how does Cerberus know which substates to use? Radix Engine itself determines them *at the time of execution* on the network.

So if my transaction says "I want to send 5 XRD", Radix Engine can (deterministically) look at the token substates available and pick substates that represent *these particular 5 XRD* when the network actually processes the transaction. This means that the substates chosen are always good at that time, and conflicts between transactions are avoided[8].

3. Each smart contract (or "component" in Radix terms) – including all of its data and the resources it owns – is assigned to a single shard at any point in time. This is important to ensuring that each component can process as many transactions as possible.

[8] Here we describe how transactions will operate once smart contract capability is added to the Radix network at the Babylon release and beyond, not the current Olympia release.

Typically the result of each execution of a smart contract (each time it's run as part of a transaction) depends on the *previous* execution in some way. For example, a DEX swap changes the balance of tokens in its pool which in turn changes the exchange rate on the next transaction. This means that there's not much reason to spread different parts of a smart contract across multiple shards (which means more complex consensus for each transaction).

Instead, the most efficient way is to run each component (a Radix Engine smart contract) – including all of its data and resources it owns – on *one shard*. Radix Engine does this by capturing the full status of each component within a single substate at any given time. Any resources that move into the component become part of that substate; and when resources move out of the component, they are split out of that substate (because remember, resources are global on Radix!).

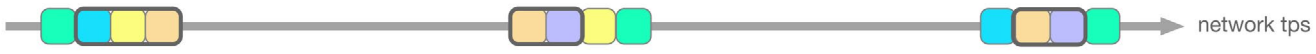
As a result, each component on Radix has full use of its own shard that can run unimpeded by anything else going on on the network. On Radix, user accounts are also components and so each account also gets its own shard automatically.

(Note: Radix Engine actually lets us be a little more clever. We don't have to exclusively assign one component to one shard, but it's the simplest way to understand how Radix Engine can optimize per-component throughput.)

Going further, a given dApp might actually be made up of *multiple* logically modular components, each with its own shard. For example, a DEX dApp might include independent components for each "pair" so that trades on each pair can run independently without slowing each other. With Cerberus' virtually infinite shardspace, this allows a dApp's developer to essentially provision as much parallelism as they need for their own application – on top of the parallelism the network offers between separate dApps.

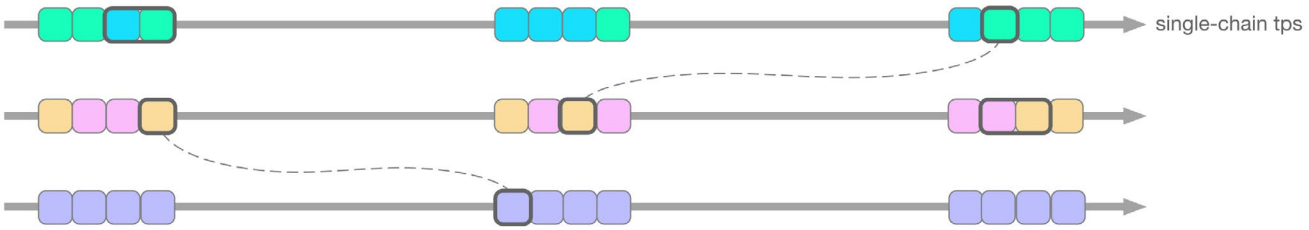
Achieving Unlimited dApp Throughput

When talking about "scalability", it's easy to lose track of what's important, particularly when it comes to the ability to scale an ecosystem of real DeFi dApps. Today on Ethereum, dApps may be composable, but throughput is extremely low because all transactions, including smart contract calls, go through a single global consensus process that is very slow.



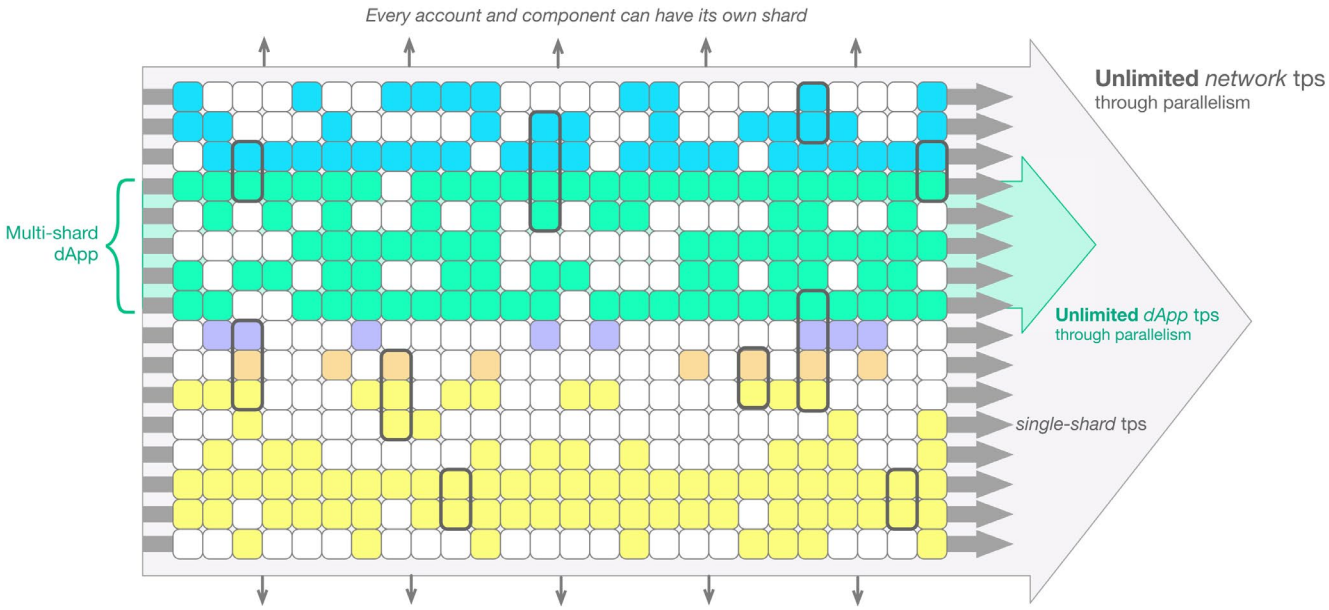
(Calls to different dApps are represented by the colored blocks, with the outlines indicating composed atomic transactions across multiple dApps.)

Most sharding concepts (such as Ethereum 2.0, Cosmos, and others) add a limited amount of parallelism through a fixed set of shards or side-chains. This increases throughput somewhat but immediately compromises (or eliminates) composability between shards/side-chains. And still these architectures put a limit on how much throughput can be achieved on a given shard, and for all of the dApps and tokens on that shard.



Combining the features of Radix Engine and Cerberus, we can create a platform uniquely designed to scale a full ecosystem of real-world DeFi dApps through massive parallelism:

- Resources are transacted in parallel without bottlenecks
- Components run in parallel at maximum single-shard speed without conflicts
- Each dApp can be parallelized for greater throughput by using multiple logically unrelated components
- Efficiency of parallelism is maximized because transactions pull together only the resources and components actually needed at that time



And all of this without compromising atomic composability because Cerberus can conduct cross-shard transactions, as needed, atomically and efficiently.

This sort of unlimited parallelism – not just for tokens but for DeFi dApps – is crucial to creating a platform that can truly scale DeFi to the level of global finance, and it is only possible through the tightly integrated design of Cerberus and Radix Engine.

RADIX