Modular rollup theory (through the lens of the OP Stack)

AKA you just watched Karl's talk and now you want to understand how this whole modular rollup architecture actually works

Kelvin Fichter

Building the Optimism Collective

Modular rollup theory (through the lens of the OP Stack)

AKA you just watched Karl's talk and now you want to understand how this whole modular rollup architecture actually works

Kelvin Fichter

Building the Optimism Collective

Some context on this talk

Some context on this talk

I'll be talking about the theory behind modular rollup architecture.

Some context on this talk

I'll be talking about the theory behind modular rollup architecture. The OP Stack is a software stack that turns this theory into practice.

Some context on this talk

I'll be talking about the theory behind modular rollup architecture. The OP Stack is a software stack that turns this theory into practice. I don't like talking about theory alone, so I'll be using the stack to keep this talk grounded.

Some context on this talk

I’ll be talking about the theory behind modular rollup architecture. The OP Stack is a software stack that turns this theory into practice. I don’t like talking about theory alone, so I’ll be using the stack to keep this talk grounded.

Also I’m going to use TypeScript types, TypeScript is god tier don’t @ me.

Some context on this talk

I'll be talking about the theory behind modular rollup architecture. The OP Stack is a software stack that turns this theory into practice. I don't like talking about theory alone, so I'll be using the stack to keep this talk grounded.

Also I'm going to use TypeScript types, TypeScript is
tier don't @ me.

Some context on this talk

I'll be talking about the theory behind modular rollup architecture. The OP Stack is a software stack that turns this theory into practice. I don't like talking about theory alone, so I'll be using the stack to keep this talk grounded.

Also I'm going to use TypeScript types, TypeScript is

tier don't @ me. I hope you enjoy!

Section 1

Modular rollups 101

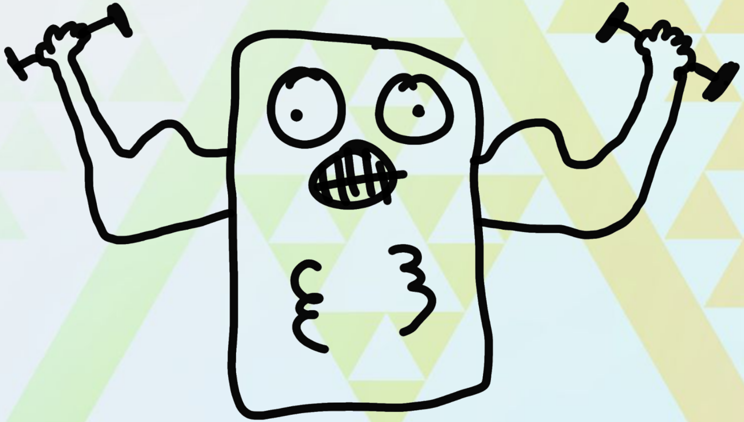
Some brief history

Some brief history

Back in 2020, everyone was building **monolithic rollups**.

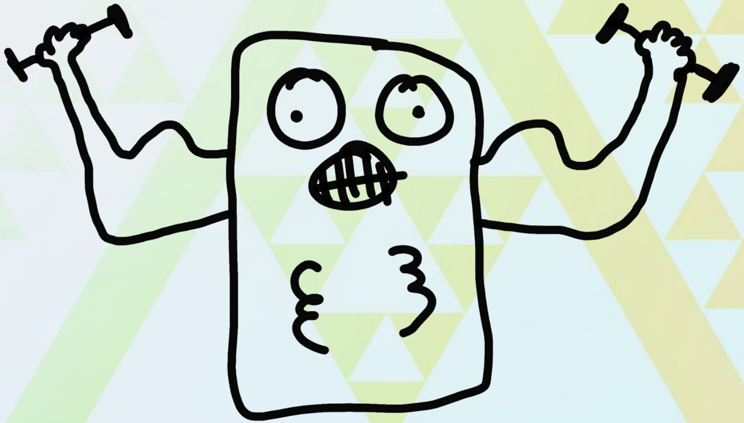
Some brief history

Back in 2020, everyone was building **monolithic rollups**.



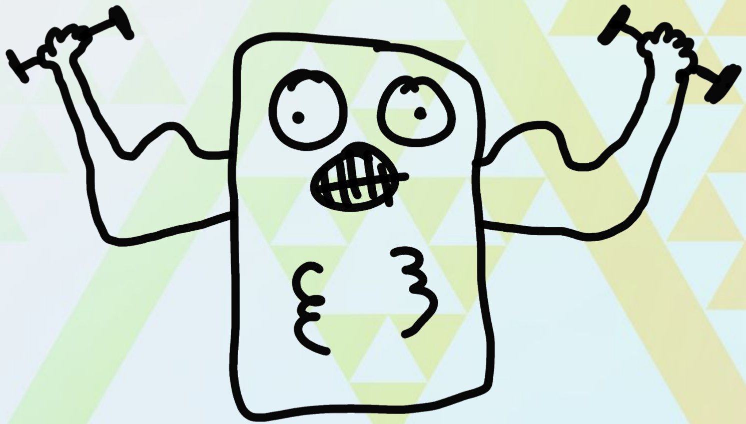
Some brief history

Back in 2020, everyone was building **monolithic rollups**.
Rollups were defined (and limited) by our proof systems.



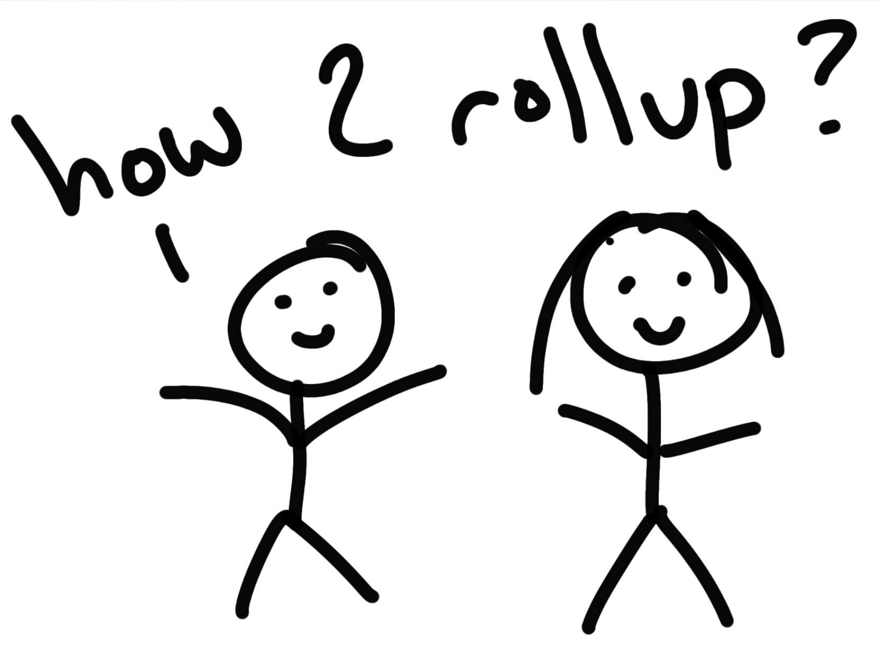
Some brief history

Back in 2020, everyone was building **monolithic rollups**. Rollups were defined (and limited) by our proof systems. We did this because we had no clue what we were actually building.



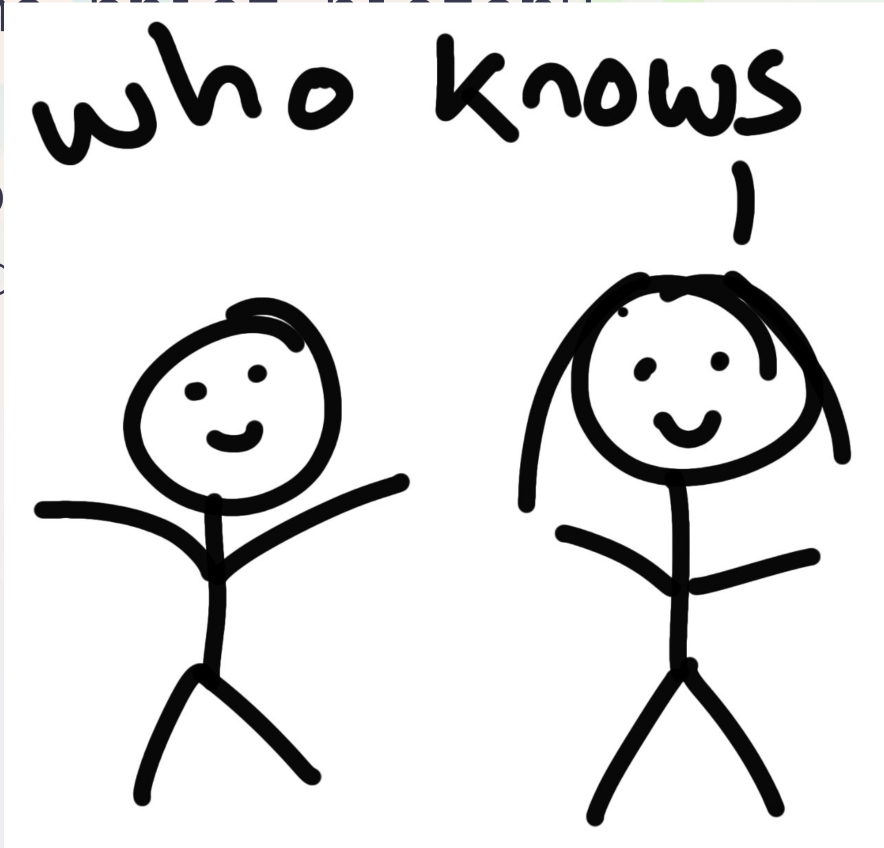
Some brief history

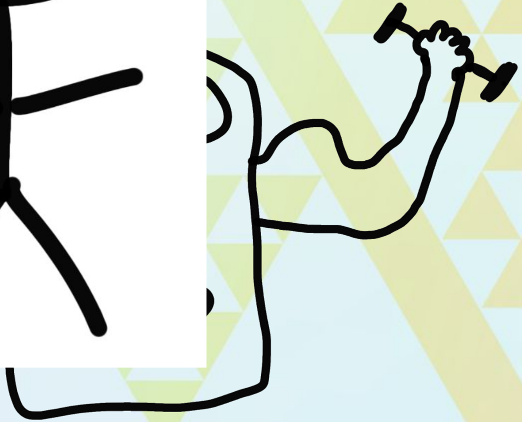
Back in 2020, everyo s.
Rollups were defined s.
We did this because ally
building.



Some brief history

Back in 2020, everyo … s.
Rollups were defined … s.
We did this because … ally
building.

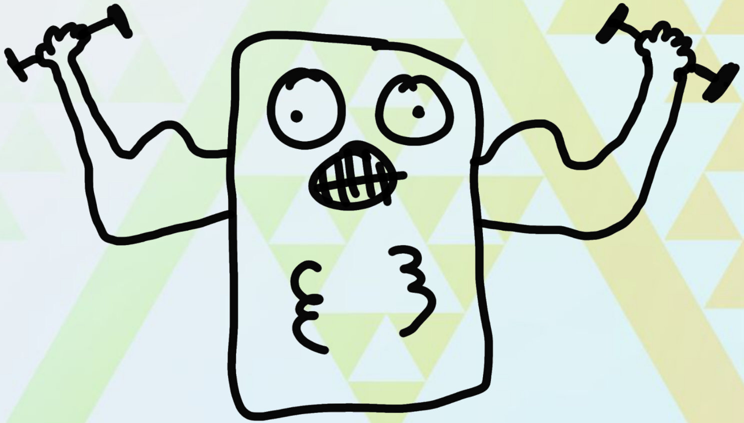




Some brief history

Back in 2020, everyone was building **monolithic rollups**. Rollups were defined (and limited) by our proof systems. We did this because we had no clue what we were actually building.

Mental models are important!



Aside

Isn’t that funny? We can work on things for a long time before we really start to understand what we’re actually building.

Aside

Isn’t that funny? We can work on things for a long time before we really start to understand what we’re actually building. Anyway.

Then we finally kinda got it

Then we finally kinda got it

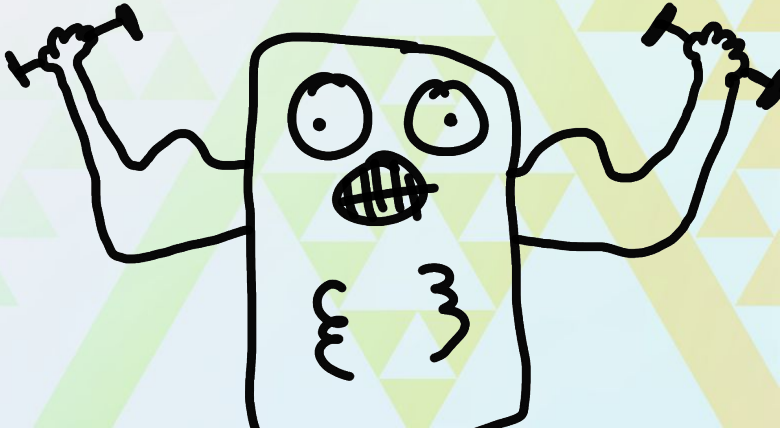
Between 2021 and 2022, we started to understand rollups more deeply.

Then we finally kinda got it

Between 2021 and 2022, we started to understand rollups more deeply. We first came to realize that the proof should be fully separated from execution.

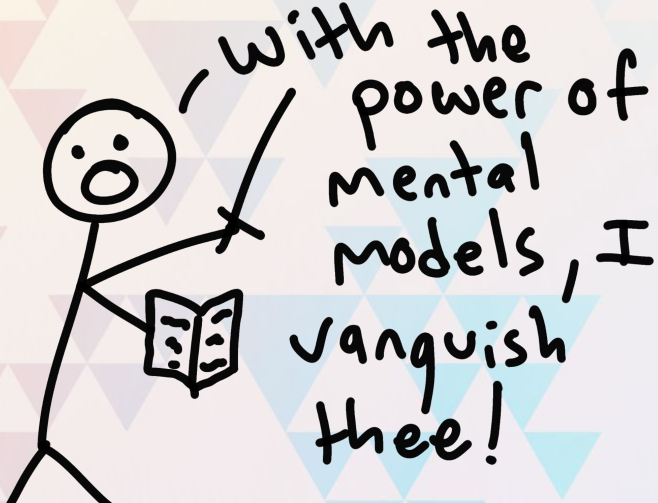
Then we finally kinda got it

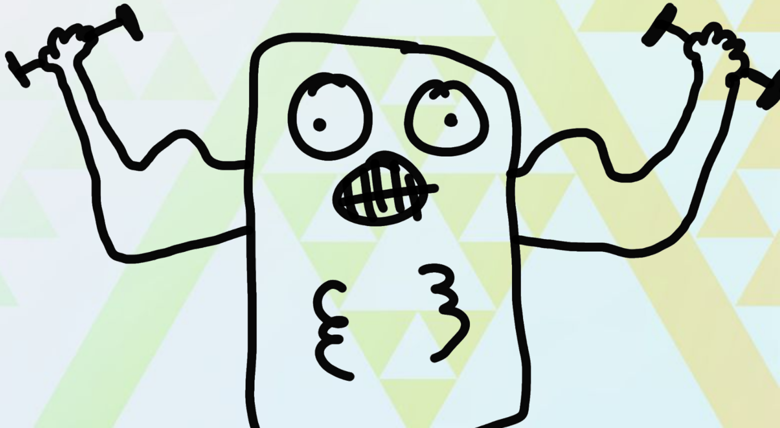
Between 2021 and 2022, we started to understand rollups more deeply. We first came to realize that the proof should be fully separated from execution.



Then we finally kinda got it

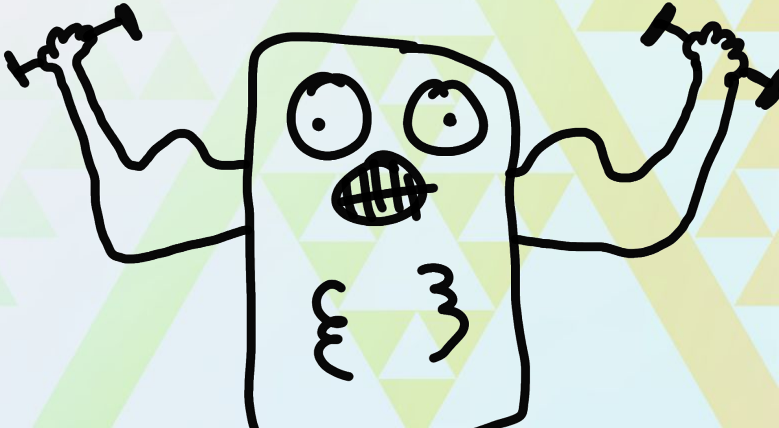
Between 2021 and 2022, we started to understand rollups more deeply. We first came to realize that the proof should be fully separated from execution.





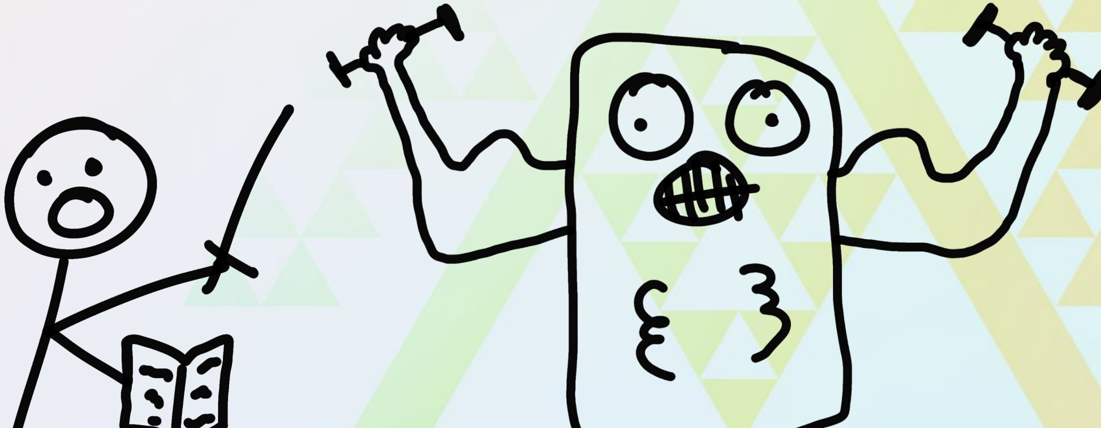
Then we finally kinda got it

Between 2021 and 2022, we started to understand rollups more deeply. We first came to realize that the proof should be fully separated from execution.



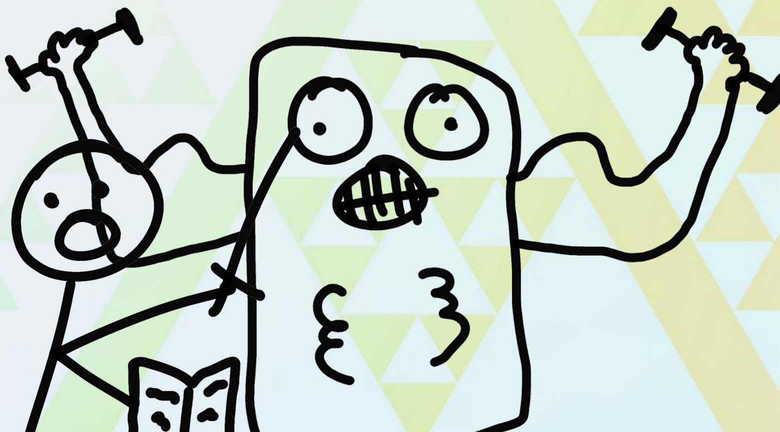
Then we finally kinda got it

Between 2021 and 2022, we started to understand rollups more deeply. We first came to realize that the proof should be fully separated from execution.



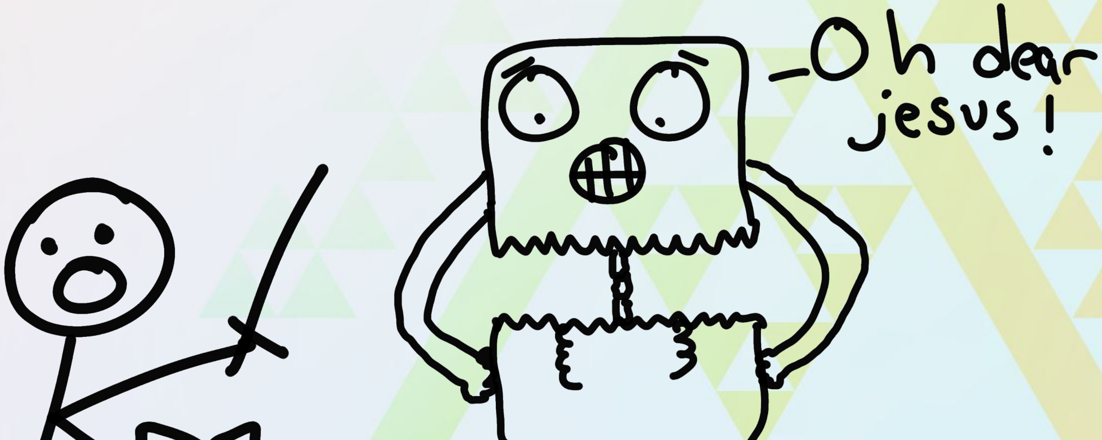
Then we finally kinda got it

Between 2021 and 2022, we started to understand rollups more deeply. We first came to realize that the proof should be fully separated from execution.



Then we finally kinda got it

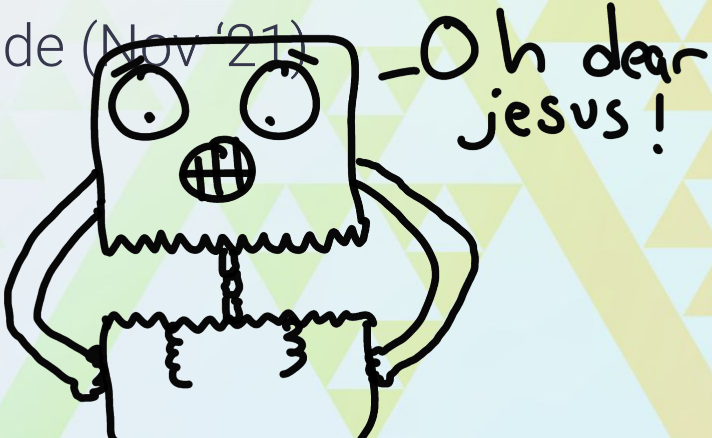
Between 2021 and 2022, we started to understand rollups more deeply. We first came to realize that the proof should be fully separated from execution.



Then we finally kinda got it

Between 2021 and 2022, we started to understand rollups more deeply. We first came to realize that the proof should be fully separated from execution.

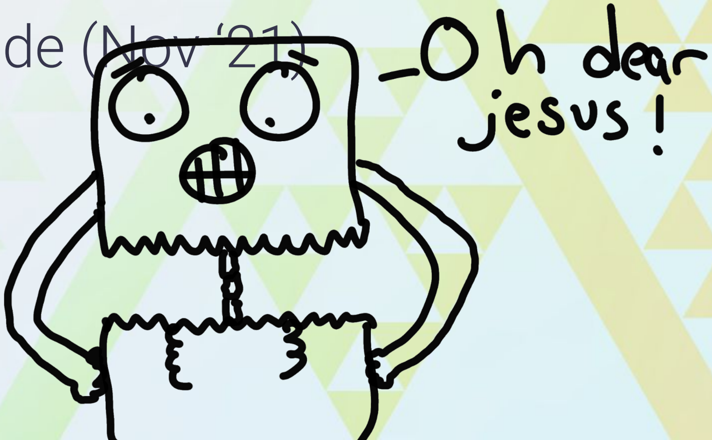
- Optimism's EVM Equivalence upgrade (Nov '21)



Then we finally kinda got it

Between 2021 and 2022, we started to understand rollups more deeply. We first came to realize that the proof should be fully separated from execution.

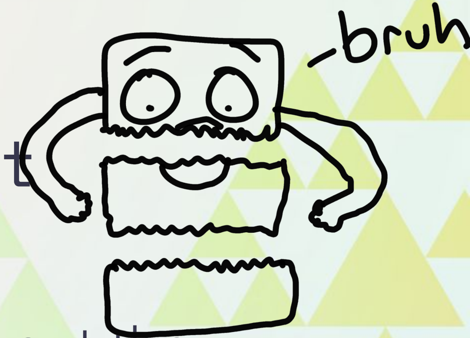
- Optimism's EVM Equivalence upgrade (Nov '21)
- Arbitrum's Nitro upgrade (Aug '22)



Then we finally kinda got it

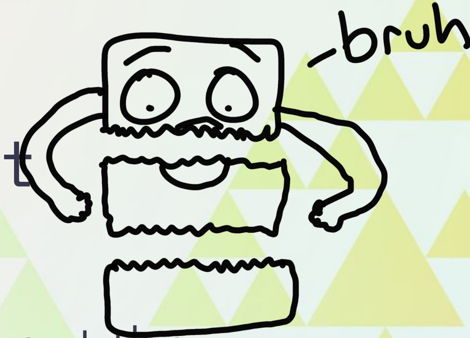
We then also came to realize that we could break out the data availability layer.

Then we finally kinda got it



We then also came to realize that we could break out the data availability layer.

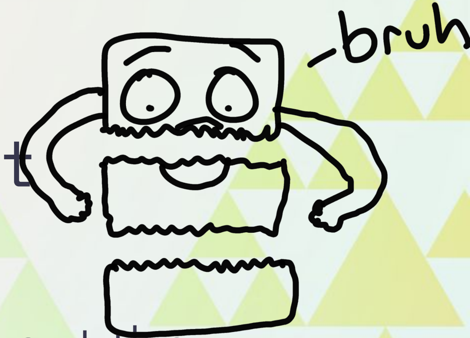
Then we finally kinda got it



We then also came to realize that we could break out the data availability layer.

- Metis forked Optimism and added a DA committee

Then we finally kinda got it



We then also came to realize that we could break out the data availability layer.

- Metis forked Optimism and added a DA committee
- Arbitrum releases Nova with a DA committee

Rollups were becoming modular!

Rollups were becoming modular!

We generally saw rollups beginning to break down into three primary layers.

Rollups were becoming modular!

We generally saw rollups beginning to break down into three primary layers.

- Consensus

Rollups were becoming modular!

We generally saw rollups beginning to break down into three primary layers.

- Consensus
- Execution

Rollups were becoming modular!

We generally saw rollups beginning to break down into three primary layers.

- Consensus
- Execution
- Settlement

Isn’t that just modular blockchains?

Isn't that just modular blockchains?

Yes.

Isn't that just modular blockchains?

Yes. But it was the theory of modular blockchains actually being put into practice.

Isn’t that just modular blockchains?

Yes. But it was the theory of modular blockchains actually being put into practice. Instead of fun charts describing how different pieces might fit together, this was software actually fitting these pieces together!

It was time to make things official

It was time to make things official

This was modular blockchain design being put into practice, but it was messy and haphazard.

It was time to make things official

This was modular blockchain design being put into practice, but it was messy and haphazard.

You know what time it is!

It was time to make things official

This was modular blockchain design being put into practice, but it was messy and haphazard.

You know what time it is!

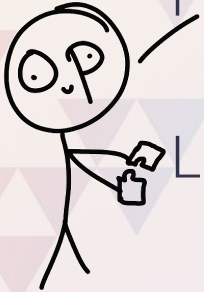
Formalization time!

It was time to make things official

This was modular blockchain design being put into practice, but it was messy and haphazard.

You know what time it is!

Formalization time!



Like, loosely formalized. I never graduated college.

Section 2

The Consensus Layer

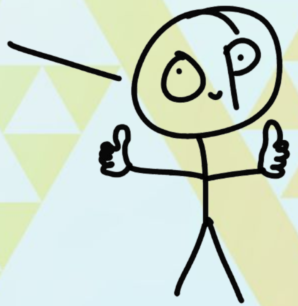
Beep boop, bias warning

Beep boop, bias warning

I'm going to start using the abstractions that we defined as part of the OP Stack.

Beep boop, bias warning

I'm going to start using the abstractions that we defined as part of the OP Stack. I think these abstractions are good.



Beep boop, bias warning

I’m going to start using the abstractions that we defined as part of the OP Stack. I think these abstractions are good. Deal with it!

The three primary layers

The three primary layers

- Consensus

The three primary layers

- Consensus
  - Data Availability

The three primary layers

- Consensus
    - Data Availability
    - Derivation

The three primary layers

- Consensus
    - Data Availability
    - Derivation
- Execution

The three primary layers

- Consensus
  - Data Availability
  - Derivation
- Execution
- Settlement

The three primary layers

- **Consensus**
  - **Data Availability**
  - **Derivation**
- Execution
- Settlement

We have two sub-components

We have two sub-components

Data availability layer

We have two sub-components

Data availability layer
Derivation layer

What is the data availability layer even?

What is the data availability layer even?

It's where you post the data.

What is the data availability layer even?

It's where you post the data. Alright, fine, we can get slightly more formal.

What is the data availability layer even?

It’s where you post the data. Alright, fine, we can get slightly more formal. It’s an ordered list of blobs.

What is the data availability layer even?

It's where you post the data. Alright, fine, we can get slightly more formal. It's an ordered list of blobs. Preferably an immutable append-only list, but that's an implementation detail.

What is the data availability layer even?

It's where you post the data. Alright, fine, we can get slightly more formal. It's an ordered list of blobs. Preferably an immutable append-only list, but that's an implementation detail.

type DA = bytes[]

Some examples of DA layers

Some examples of DA layers

- Ethereum (via calldata)

Some examples

```go
// AdvanceL1Block advances the internal state of L1 Traversal
func (l1t *L1Traversal) AdvanceL1Block(ctx context.Context) error {
	origin := l1t.block
	nextL1Origin, err := l1t.l1Blocks.L1BlockRefByNumber(ctx, origin.Number+1)
	if errors.Is(err, ethereum.NotFound) {
		l1t.log.Debug("can't find next L1 block info (yet)", "number", origin.Number+1, "origin", origin)
		return io.EOF
	} else if err != nil {
		return NewTemporaryError(fmt.Errorf("failed to find L1 block info by number, at origin %s next %d: %w", origin, origin.Nu
	}
	if l1t.block.Hash != nextL1Origin.ParentHash {
		return NewResetError(fmt.Errorf("detected L1 reorg from %s to %s with conflicting parent %s", l1t.block, nextL1Origin, ne
	}
	l1t.block = nextL1Origin
	l1t.done = false
	return nil
}
```

Some examples of DA layers

- Ethereum (via calldata)
- Ethereum (via 4844)

Some examples of DA layers

- Ethereum (via calldata)
- Ethereum (via 4844)
- Celestia

Some examples of DA layers

- Ethereum (via calldata)
- Ethereum (via 4844)
- Celestia
- A stack of post-its

Derivation is the more interesting one

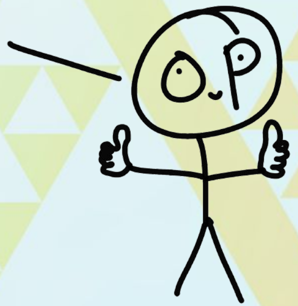
Derivation is the more interesting one

The derivation layer takes the data availability layer and the current state of the rollup and produces Engine API payloads.

Derivation is the more interesting one

The derivation layer takes the data availability layer and the current state of the rollup and produces Engine API payloads.

Why Engine API?



Derivation is the more interesting one

The derivation layer takes the data availability layer and the current state of the rollup and produces Engine API payloads.

Why Engine API? One of those OP Stack opinionated things.

Derivation is the more interesting one

The derivation layer takes the data availability layer and the current state of the rollup and produces Engine API payloads.

Why Engine API? One of those OP Stack opinionated things. Already standard in Ethereum clients and makes block building easier.

Let’s formalize it

Let's formalize it

Derivation has a relatively simple function signature.

Let's formalize it

Derivation has a relatively simple function signature.

$$\mathrm{derive}(S_{prev}, DA) \Rightarrow \begin{Bmatrix} \text{payload} \\ \text{or} \\ \text{null} \end{Bmatrix}$$

Derivation in Bedrock

Derivation in Bedrock

Optimism derives data from three locations:

Derivation in Bedrock

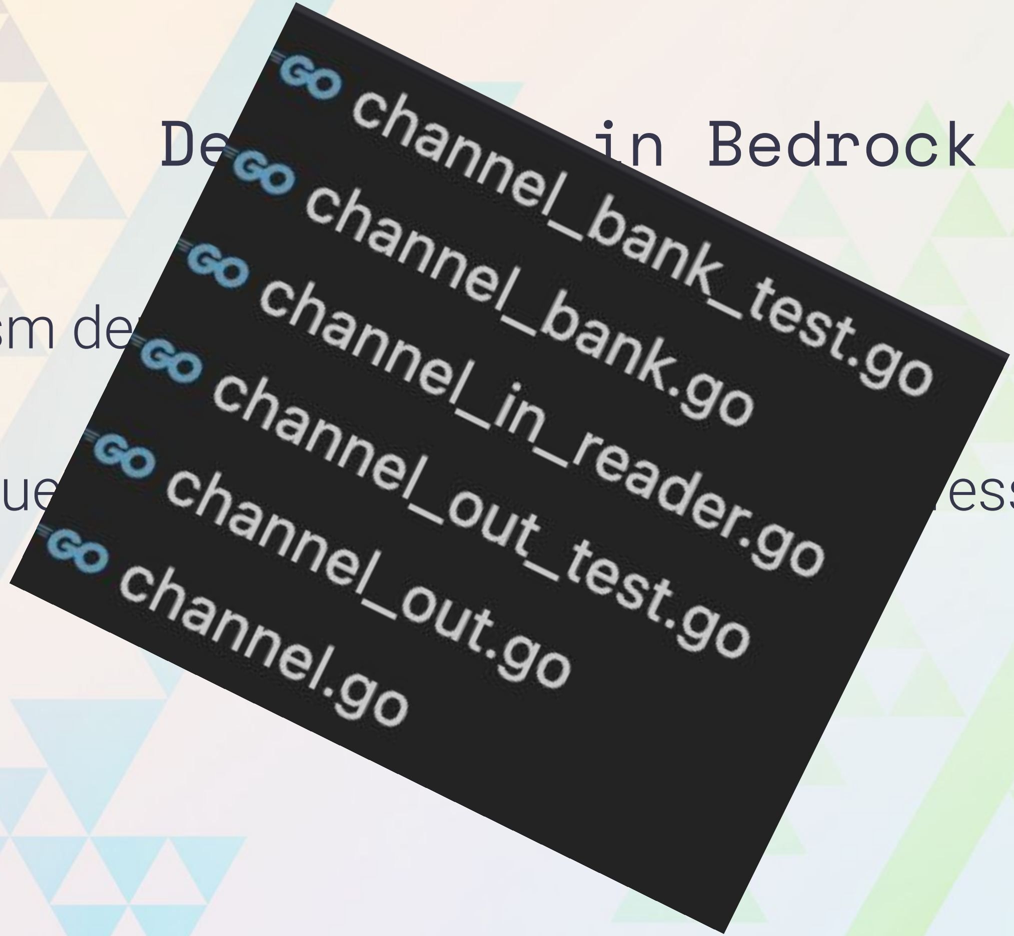
Optimism derives data from three locations:

1. Sequencer data posted to a specific address

De in Bedrock

Optimism de

1. Seque ess



Derivation in Bedrock

Optimism derives data from three locations:

1. Sequencer data posted to a specific address
2. Deposits sent to the Portal contract

Derivation in Bedrock

```go
// UserDeposits transforms the L2 block-height and L1 receipts into the transaction inputs for a full L2 block
func UserDeposits(receipts []*types.Receipt, depositContractAddr common.Address) ([]*types.DepositTx, error) {
	var out []*types.DepositTx
	var result error
	for i, rec := range receipts {
		if rec.Status != types.ReceiptStatusSuccessful {
			continue
		}
		for j, log := range rec.Logs {
			if log.Address == depositContractAddr && len(log.Topics) > 0 && log.Topics[0] == DepositEventABIHash {
				dep, err := UnmarshalDepositLogEvent(log)
				if err != nil {
					result = multierror.Append(result, fmt.Errorf("malformatted L1 deposit log in receipt %d, log %d: %w", i, j, err))
				} else {
					out = append(out, dep)
				}
			}
		}
	}
	return out, result
}
```

Derivation in Bedrock

Optimism derives data from three locations:

1. Sequencer data posted to a specific address
2. Deposits sent to the Portal contract
3. L1 block data itself

Deri in Bedrock

Optimism derives

1. Sequencer da
2. Deposits ser
3. L1 block dat

```go
// L1InfoDeposit creates a L1 Info deposit transaction based on the L1 block,
// and the L2 block-height difference with the start of the epoch.
func L1InfoDeposit(seqNumber uint64, block eth.BlockInfo) (*types.DepositTx, error) {
	infoDat := L1BlockInfo{
		Number:         block.NumberU64(),
		Time:           block.Time(),
		BaseFee:        block.BaseFee(),
		BlockHash:      block.Hash(),
		SequenceNumber: seqNumber,
	}
	data, err := infoDat.MarshalBinary()
	if err != nil {
		return nil, err
	}

	source := L1InfoDepositSource{
		L1BlockHash: block.Hash(),
		SeqNumber:   seqNumber,
	}
	// Set a very large gas limit with `IsSystemTransaction` to ensure
	// that the L1 Attributes Transaction does not run out of gas.
	return &types.DepositTx{
		SourceHash:          source.SourceHash(),
		From:                L1InfoDepositerAddress,
		To:                  &L1BlockAddress,
		Mint:                nil,
		Value:               big.NewInt(0),
		Gas:                 150_000_000,
		IsSystemTransaction: true,
		Data:                data,
	}, nil
}
```

Derivation in Bedrock

Optimism derives data from three locations:

1. Sequencer data posted to a specific address
2. Deposits sent to the Portal contract
3. L1 block data itself

Each of these get translated into Engine payloads

This abstraction is ridiculously powerful.

This abstraction is ridiculously powerful.

Want to build a rollup?

This abstraction is ridiculously powerful.

Want to build a rollup?

- Read sequenced transactions directly from tx data

This abstraction is ridiculously powerful.

Want to build a rollup?

- Read sequenced transactions directly from tx data
- Read deposit transaction data from events

This abstraction is ridiculously powerful.

Want to build a rollup?

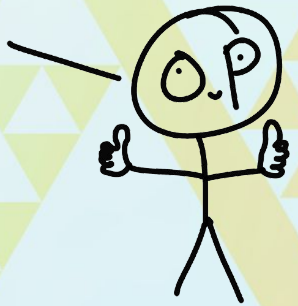
- Read sequenced transactions directly from tx data
- Read deposit transaction data from events
- Read block data and system generate transactions

This abstraction is ridiculously powerful.

But that's not all you can build.

This abstraction is ridiculously powerful.

Let's look at a toy example.



This abstraction is ridiculously powerful.

Let’s look at a toy example.

Any time there’s a Uniswap swap event, we derive an L2 transaction that includes the assets and amount swapped.

This abstraction is ridiculously powerful.

Let’s look at a toy example.

Any time there’s a Uniswap swap event, we derive an L2 transaction that includes the assets and amount swapped.

Each transaction updates a value in a smart contract that keeps a running tally of total volume. What does that kinda sound like?

This abstraction is ridiculously powerful.

An indexer!

This abstraction is ridiculously powerful.

An indexer!

Are indexers just rollups?

This abstraction is ridiculously powerful.

An indexer!

Are indexers just rollups?
Who knows.

This abstraction is ridiculously powerful.

An indexer!

Are indexers just rollups?
Who knows.
Whatever.

This abstraction is ridiculously powerful.

An indexer!

Are indexers just rollups?
Who knows.
Whatever.
Anyway, you get it.

This abstraction is ridiculously powerful.

An indexer!

Are indexers just rollups?
Who knows.
Whatever.
Anyway, you get it.
You can do a lot with this.

Section 4

The Execution Layer

It’s what you think it is

It's what you think it is

It's the interesting part of your state transition function.

Also represented as a function

Also represented as a function

$$\mathrm{execute}(S_{prev}, \mathrm{payload}) \Rightarrow S_{next}$$

Derivation and execution work together

Derivation and execution work together

These two layers work together to form the state transition function loop.

Derivation and execution work together

These two layers work together to form the state transition function loop.

1. Wait for a new element in the DA layer list

Derivation and execution work together

These two layers work together to form the state transition function loop.

1. Wait for a new element in the DA layer list
2. Run derivation function

Derivation and execution work together

These two layers work together to form the state transition function loop.

1. Wait for a new element in the DA layer list
2. Run derivation function
    a. If it returns null, return to step 1

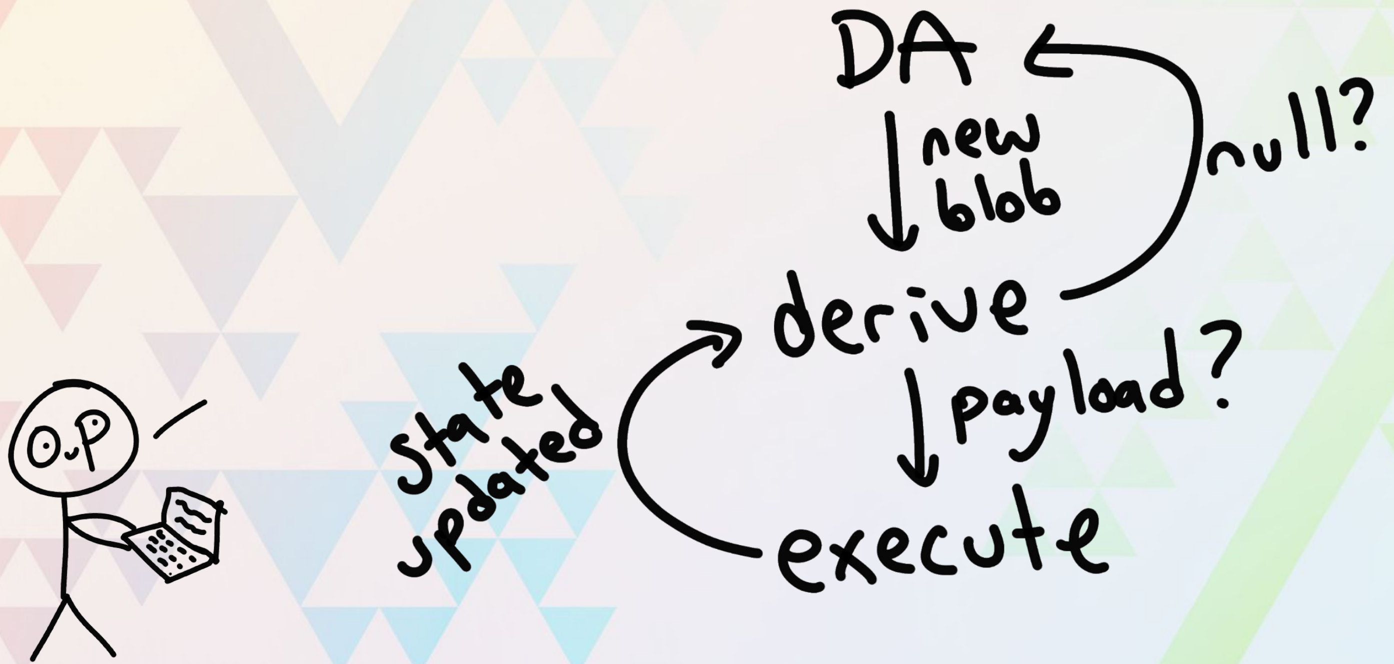
Derivation and execution work together

These two layers work together to form the state transition function loop.

1. Wait for a new element in the DA layer list
2. Run derivation function
    a. If it returns null, return to step 1
    b. If it returns a payload, pass it into the execution function, update the state, return to step 2

Derivation and execution work together

Here's that same loop drawn out:



Execution in Bedrock

Execution in Bedrock

It’s just the EVM!

Execution in Bedrock

It’s just the EVM! Mostly.

Execution in Bedrock

It’s just the EVM! Mostly.

- Smallest possible diff to make it rollup-compatible

Execution in Bedrock

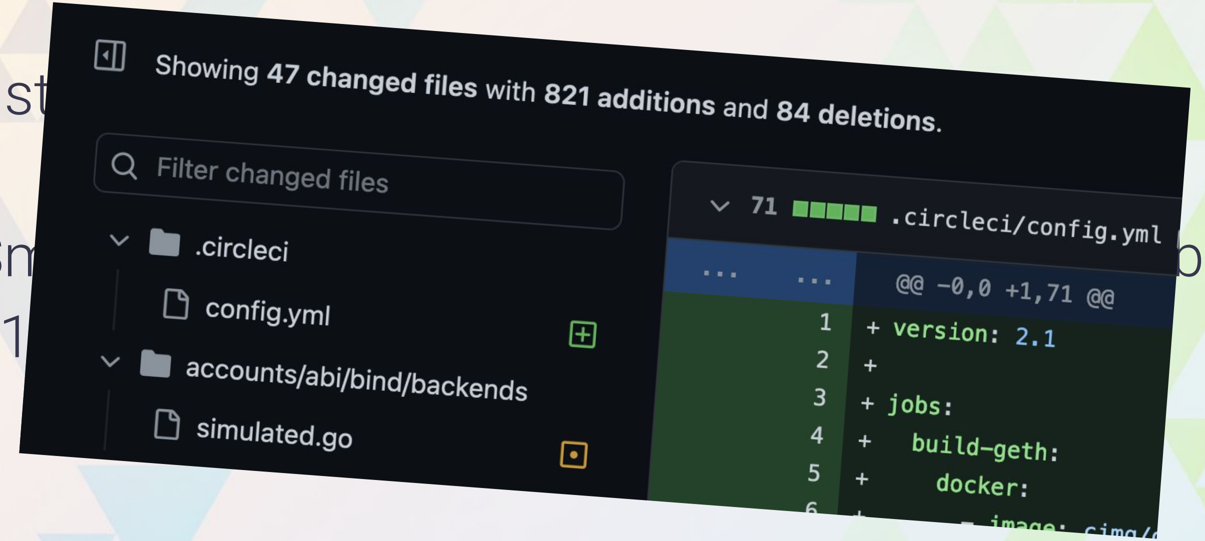
It’s just the EVM! Mostly.

- Smallest possible diff to make it rollup-compatible
- <1k lines of code in a single commit

Execution in Bedrock

It's just

- Sm ble
- <1



Execution in Bedrock

It’s just the EVM! Mostly.

- Smallest possible diff to make it rollup-compatible
- <1k lines of code in a single commit
- Support for multiple clients

Execution in Bedrock

It's just the EVM! Mostly.

- Smallest possible diff to make it fully compatible
- <1k lines of code in a single co[illegible]
- Support for multiple clients

Just because we’re doing the EVM doesn’t mean you have to do the EVM too

Just because we're doing the EVM doesn't mean you have to do the EVM too

You have an immense amount of flexibility with this design.

Just because we're doing the EVM doesn't mean you have to do the EVM too

You have an immense amount of flexibility with this design.

- Bitcoin?

Just because we're doing the EVM doesn't mean you have to do the EVM too

You have an immense amount of flexibility with this design.

- Bitcoin?
- Game Boy?

Just because we're doing the EVM doesn't mean you have to do the EVM too

You have an immense amount of flexibility with this design.

- Bitcoin?
- Game Boy?
- Python interpreter?

Just because we're doing the EVM doesn't mean you have to do the EVM too

You have an immense amount of flexibility with this design.

- Bitcoin?
- Game Boy?
- Python interpreter?

The sky's the limit.

Section 5

Settlement

Settlement

Settlement

Is it even a real thing?

Settlement

Is it even a real thing? Yes.

Settlement

Is it even a real thing? Yes. Kinda.

Settlement

Here’s how we’ll define it for the sake of the OP Stack:

Settlement

Here’s how we’ll define it for the sake of the OP Stack:

Settlement is a view that another chain has of your chain.

Settlement

Here’s how we’ll define it for the sake of the OP Stack:

Settlement is a view that another chain has of your chain.

It’s about making claims about the state of your chain to another chain and being able to back those claims up.

Settlement needs a function too!

Settlement needs a function too!

You can make all sort of claims, but most commonly you'll make a claim about the "state root" of the L2.

Settlement needs a function too!

You can make all sort of claims, but most commonly you'll make a claim about the "state root" of the L2.

$$\text{valid}(S_p, S_n, DA, \text{derive}, \text{execute}) \Rightarrow \text{boolean}$$

How do we make this function work?

$$\mathrm{valid}(S_p, S_n, DA, \mathrm{derive}, \mathrm{execute}) \Rightarrow \mathrm{boolean}$$

How do we make this function work?

$$\mathrm{valid}(S_p, S_n, DA, \mathrm{derive}, \mathrm{execute}) \Rightarrow \mathrm{boolean}$$

Look at this carefully.

How do we make this function work?

$$\text{valid}(S_p, S_n, DA, \text{derive}, \text{execute}) \Rightarrow \text{boolean}$$

Look at this carefully. State is a given, so that’s fine.

How do we make this function work?

$$\text{valid}(S_p, S_n, DA, \text{derive}, \text{execute}) \Rightarrow \text{boolean}$$

Look at this carefully. State is a given, so that's fine. Derivation and execution could be implemented on-chain, but we bypass that with fault proofs or validity proofs.

How do we make this function work?

$$\text{valid}(S_p, S_n, DA, \text{derive}, \text{execute}) \Rightarrow \text{boolean}$$

But how do we access the data availability layer?

It’s another function!

It’s another function!

Remember, our DA takes the form:

type DA = bytes[]

It's another function!

We want a function to access the DA:

It's another function!

We want a function to access the DA:

getBlobByIndex(idx) => bytes

Oooo important formalization

Oooo important formalization

`getBlobByIndex` formalizes something important.

Oooo important formalization

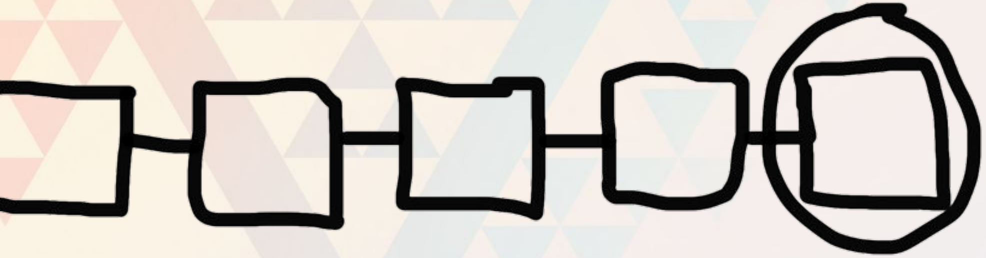
`getBlobByIndex` formalizes something important. First, the ability to resolve this function clearly depends on the actual availability of the DA.

Oooo important formalization

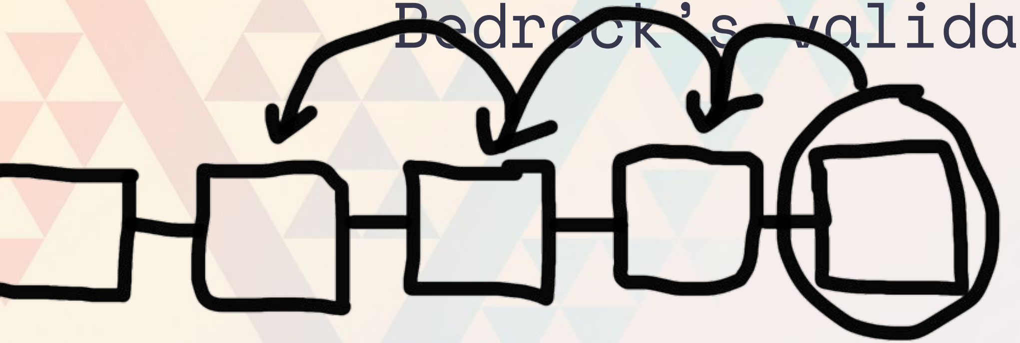
`getBlobByIndex` formalizes something important. First, the ability to resolve this function clearly depends on the actual availability of the DA. Second, this function also depends on the mechanism by which we prove that the blobs are correct.

Bedrock's validation function

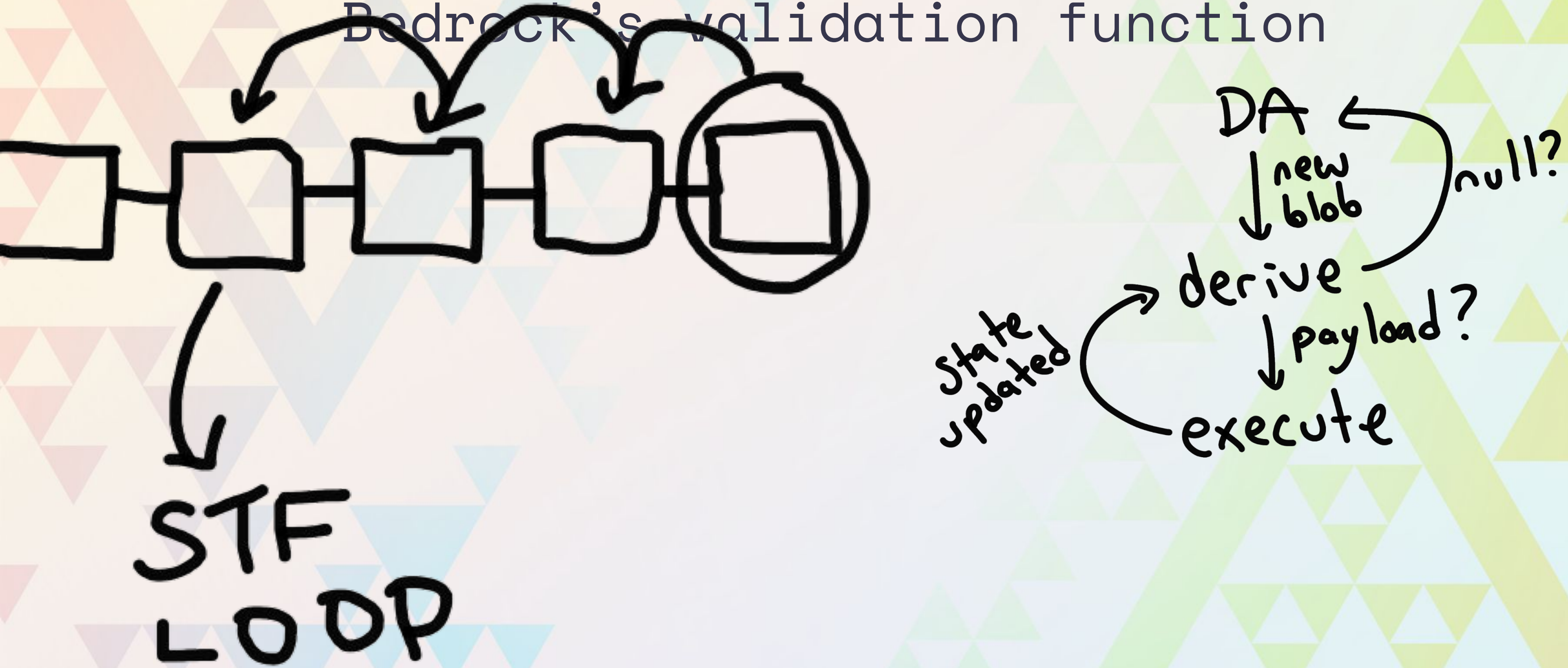
Bedrock’s validation function



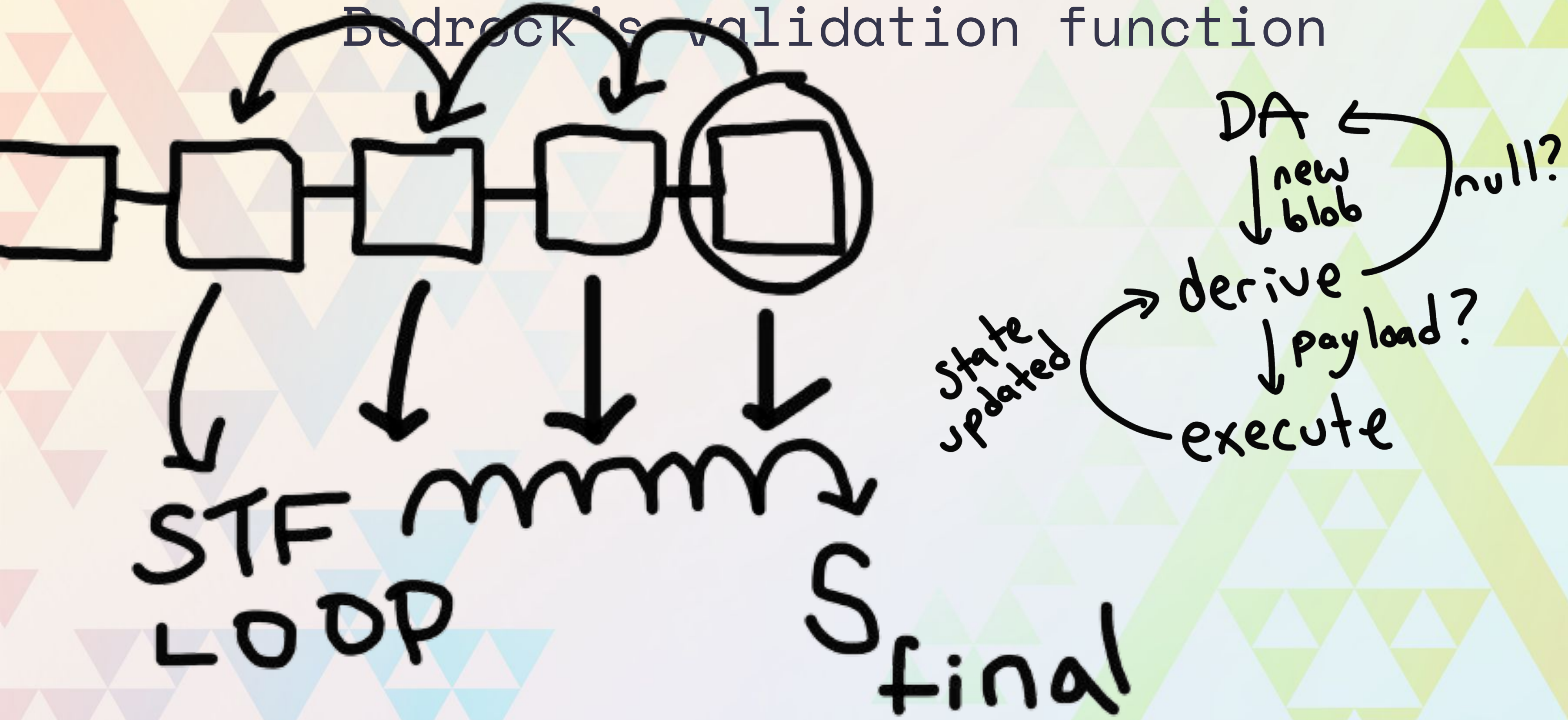
Bedrock's validation function



Bedrock's validation function

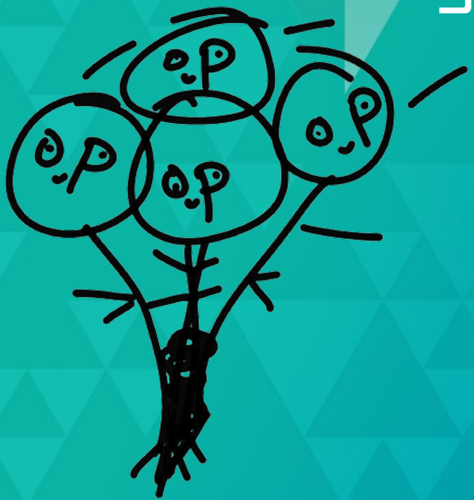


Bedrock's validation function



Section 2

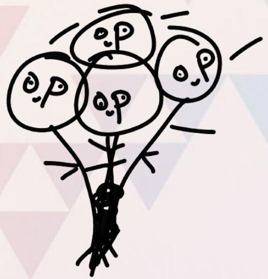
Bringing it all back together



Whew.

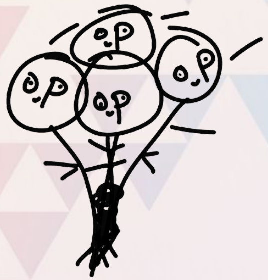
Whew.

Lot’s of content there.



Whew.

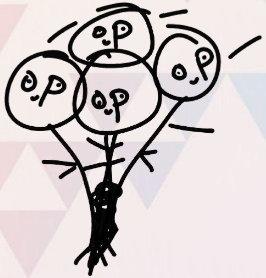
Lot’s of content there. But not too many components!



Recapping the components

Recapping the components

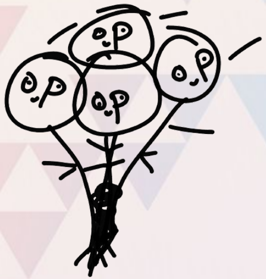
type DA = bytes[]



Recapping the components

type DA = bytes[]

$$\text{derive}(S_{prev}, DA) \Rightarrow \left\{ \begin{matrix} \text{payload} \\ \text{or} \\ \text{null} \end{matrix} \right\}$$

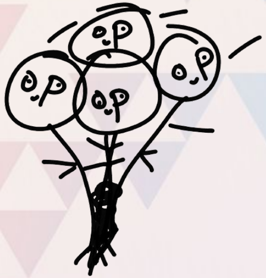


Recapping the components

type DA = bytes[]

$\text{derive}(S_{prev}, DA) \Rightarrow \left\{ \begin{array}{c} \text{payload} \\ \text{or} \\ \text{null} \end{array} \right\}$

$\text{execute}(S_{prev}, \text{payload}) \Rightarrow S_{next}$



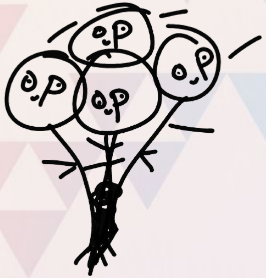
Recapping the components

type DA = bytes[]

$\text{derive}(S_{prev}, DA) \Rightarrow \left\{ \begin{array}{c} \text{payload} \\ \text{or} \\ \text{null} \end{array} \right\}$

$\text{execute}(S_{prev}, \text{payload}) \Rightarrow S_{next}$

$\text{valid}(S_p, S_n, DA, \text{derive}, \text{execute}) \Rightarrow \text{boolean}$



Recapping the components

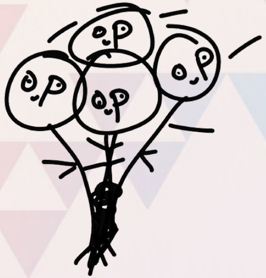
type DA = bytes[]

derive$(S_{prev}, DA) \Rightarrow \left\{ \begin{array}{c} \text{payload} \\ \text{or} \\ \text{null} \end{array} \right\}$

execute$(S_{prev}, \text{payload}) \Rightarrow S_{next}$

valid$(S_p, S_n, DA, \text{derive}, \text{execute}) \Rightarrow$ boolean

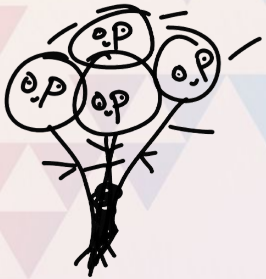
getBlobByIndex(idx) ⇒ bytes



Build your dream chain

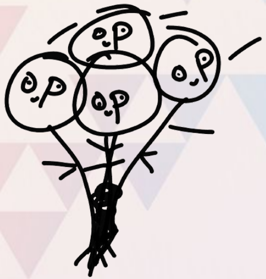
Build your dream chain

- Bitcoin Plasma?



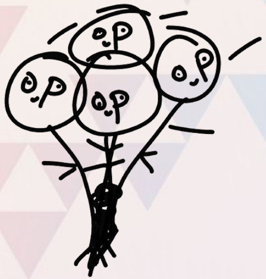
Build your dream chain

- Bitcoin Plasma?
- Bridge Rollup with multiple DAs and settlement layers?

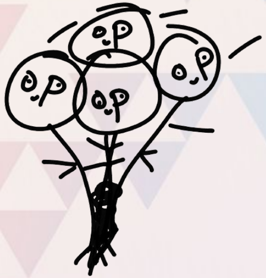


Build your dream chain

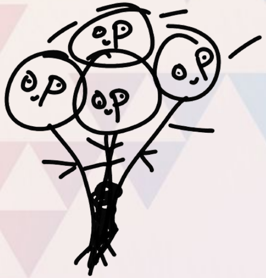
- Bitcoin Plasma?
- Bridge Rollup with multiple DAs and settlement layers?
- Another parallelized VM?



Literally build whatever, just fit the APIS!



That’s the whole talk



ty and remember to enjoy life

Kelvin Fichter

Building the Optimism Collective

@kelvinfichter

bedrock specs

y and remember to enjoy life

Kelvin Fichter

Building the Optimism Collective

@kelvinfichter

bedrock specs

getintouch

y and remember t

Kelvin Fichter

Building the Optimism Collective

@kelvinfichter