# inside stellar: features that power great developer tools

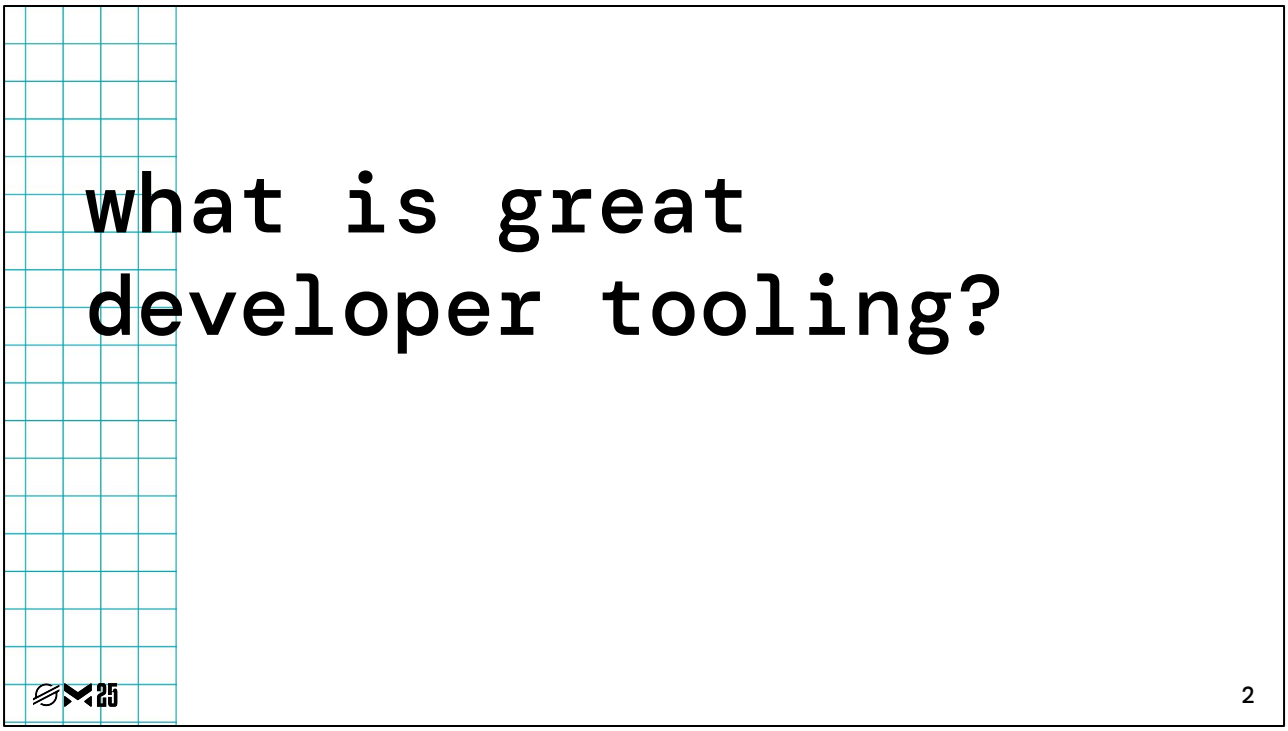
\+ features for building developer tools & products for Stellar

leigh mcculloch
principal software engineer
stellar development foundation



G’day everyone.

I want to start with a question, I have a question:



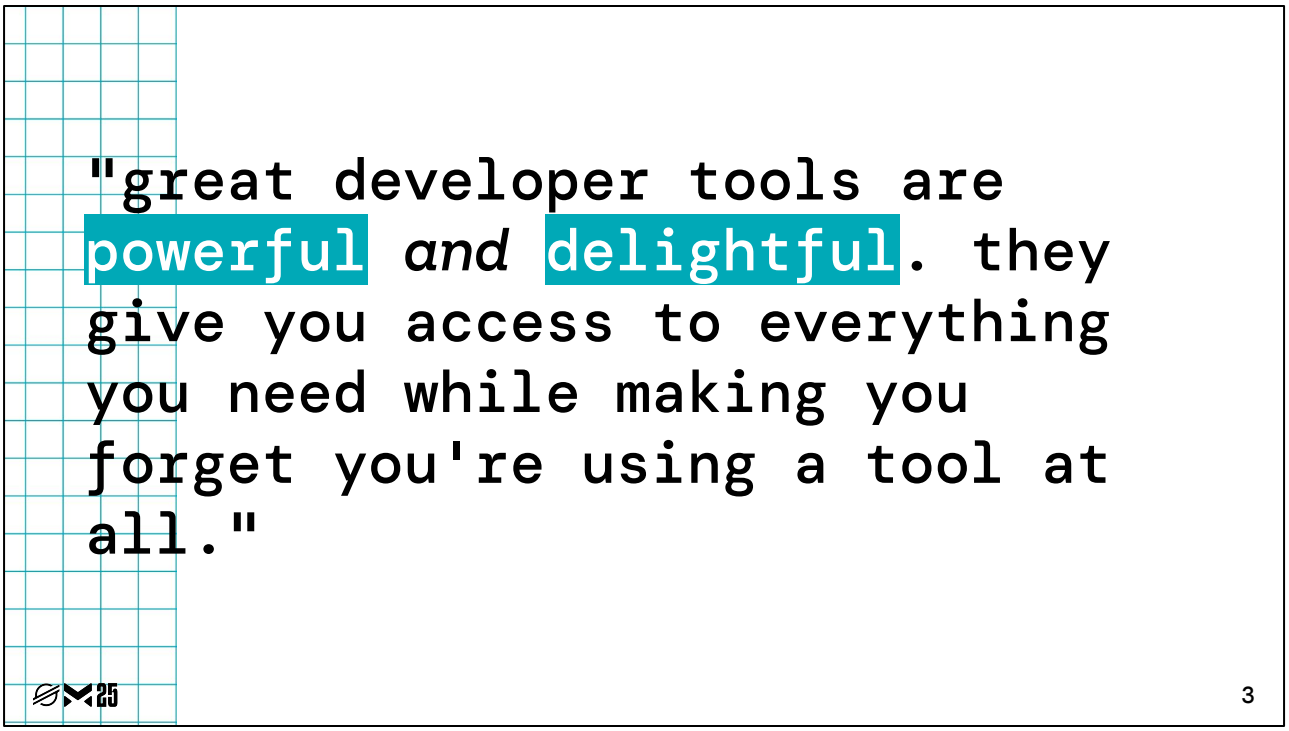

For the developers in the room, I have a question:

What is great developer tooling?

Think about the tools you use everyday. Which do you enjoy using and find yourself coming back to?

I think if we polled the room we'd fill a whiteboard with a long list of attributes. Everything from smooth user interfaces to great docs.

If we distilled all those ideas down to some common principles, I think we'd end up with something like this…




Great developer tools are powerful and delightful. They give you access to everything you need while making you forget you're using a tool at all.

In our time together we’re going to peer inside Stellar. We’re going to discover some features of Stellar that were crafted early, with the intent that they would be leveraged by developer products to create those delightful heart capturing experiences.

My hope is that at the end of our time together you will have discovered capabilities that you could take into the developer products you build. Even if you don’t identify as a tool developer, my hope is that these features might spur new ideas for you.

features

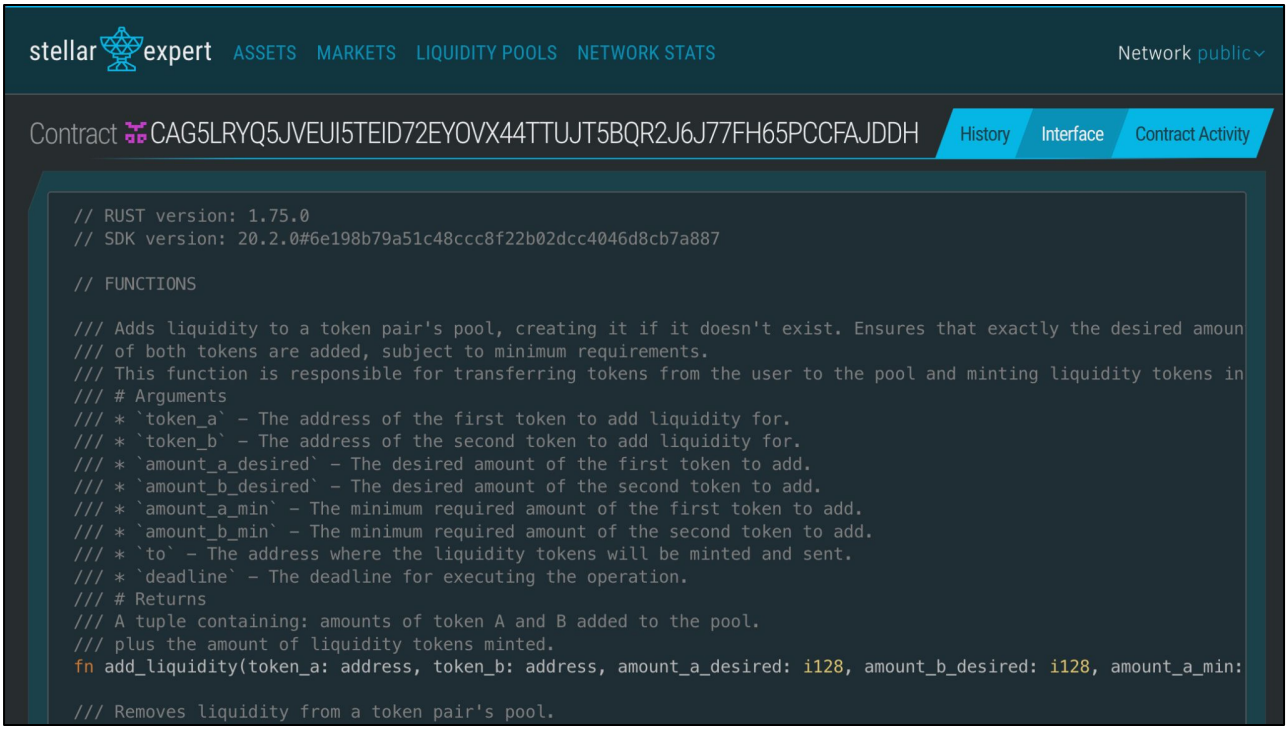
01 contract specs

02 soroban env

03 xdr json



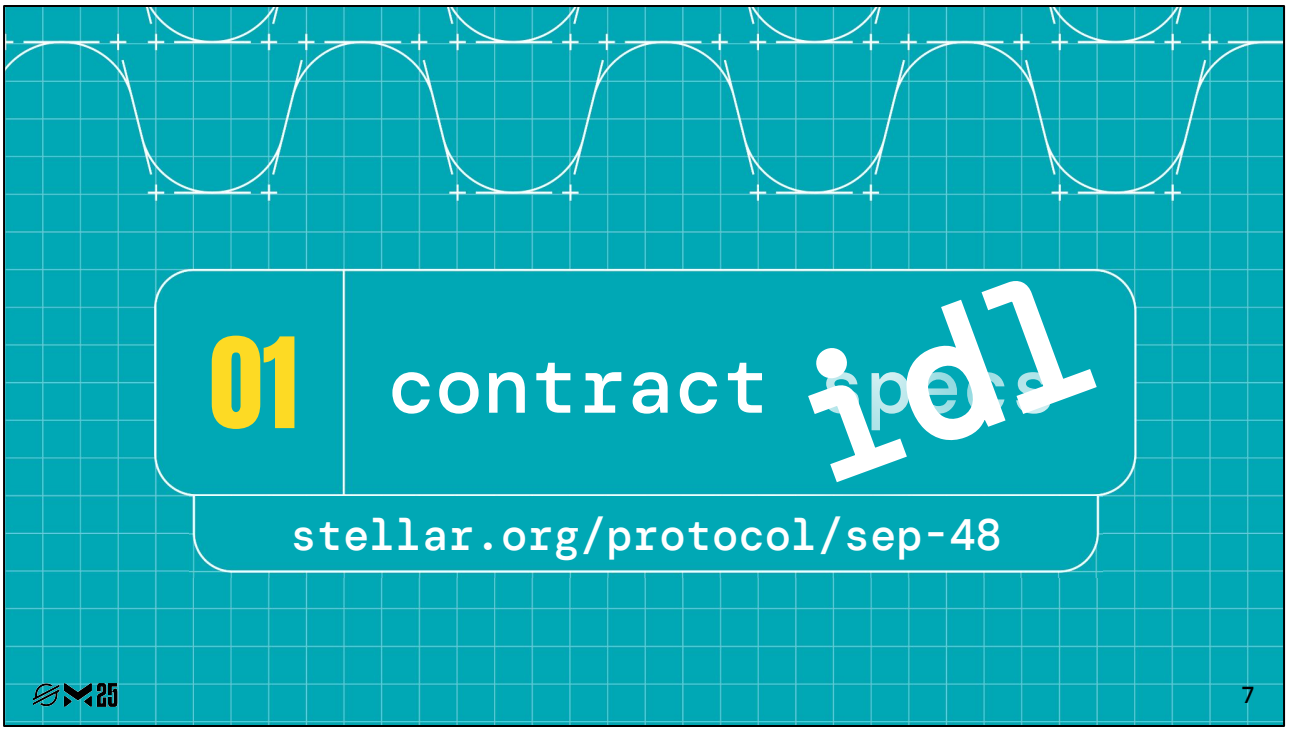
The features we're going to look at are contract specs, the soroban environment, and xdr-json.




Have you ever wondered how stellar.expert is able to render these really nice views of contracts and their functions?

Well the interface being shown here is powered by contract specs…


If you haven’t come across Stellar’s contract specs before, they are Stellar’s…IDL for contracts.




If you haven't come across Stellar's contract specs before, they are Stellar's…IDL for contracts.

IDL stands for…interface definition language.

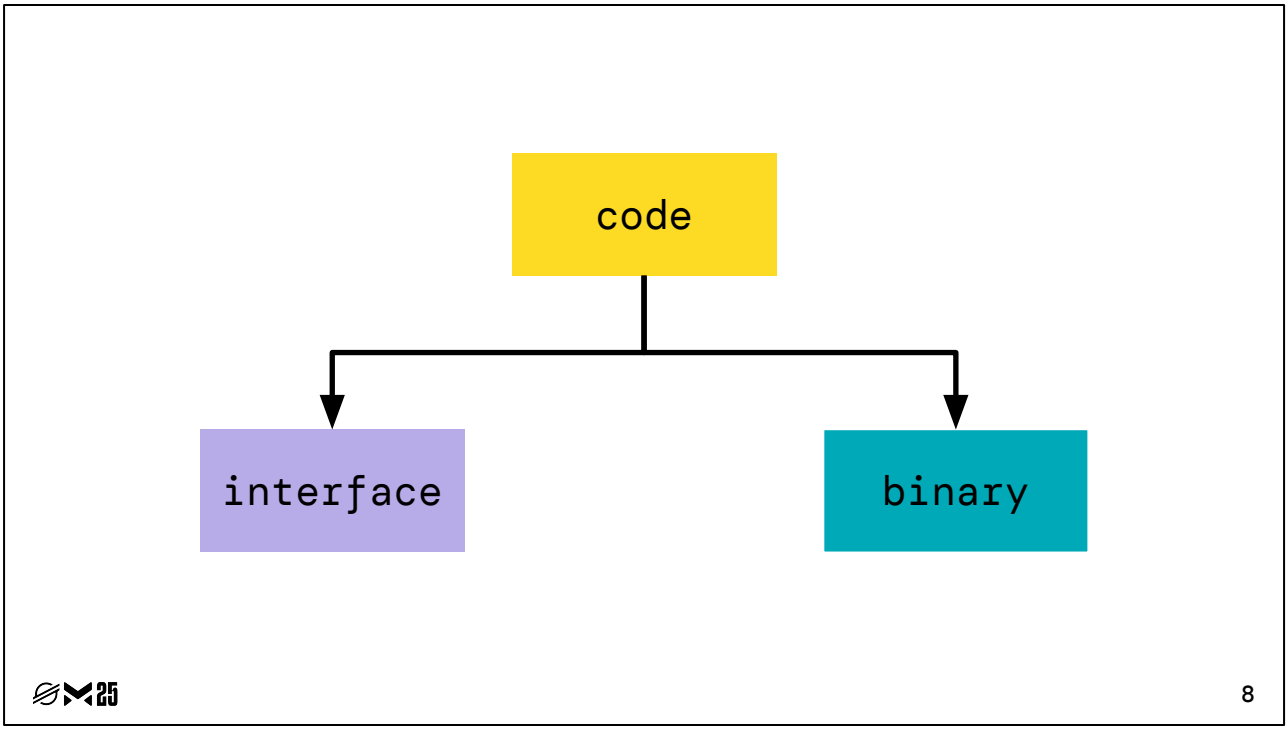
And it's a way to define the interface of a program.

For a contract that means defining the functions that can be called.

If you've built a contract for Stellar you might be surprised to hear that Stellar has an IDL, because you will have never seen it. And that's true. Stellar's IDL is not something that contract developers ever need to interact with.

But that's unique in the world of blockchain.

Many blockchains have their…contract interfaces be something that developers need to distribute separately to their contracts.



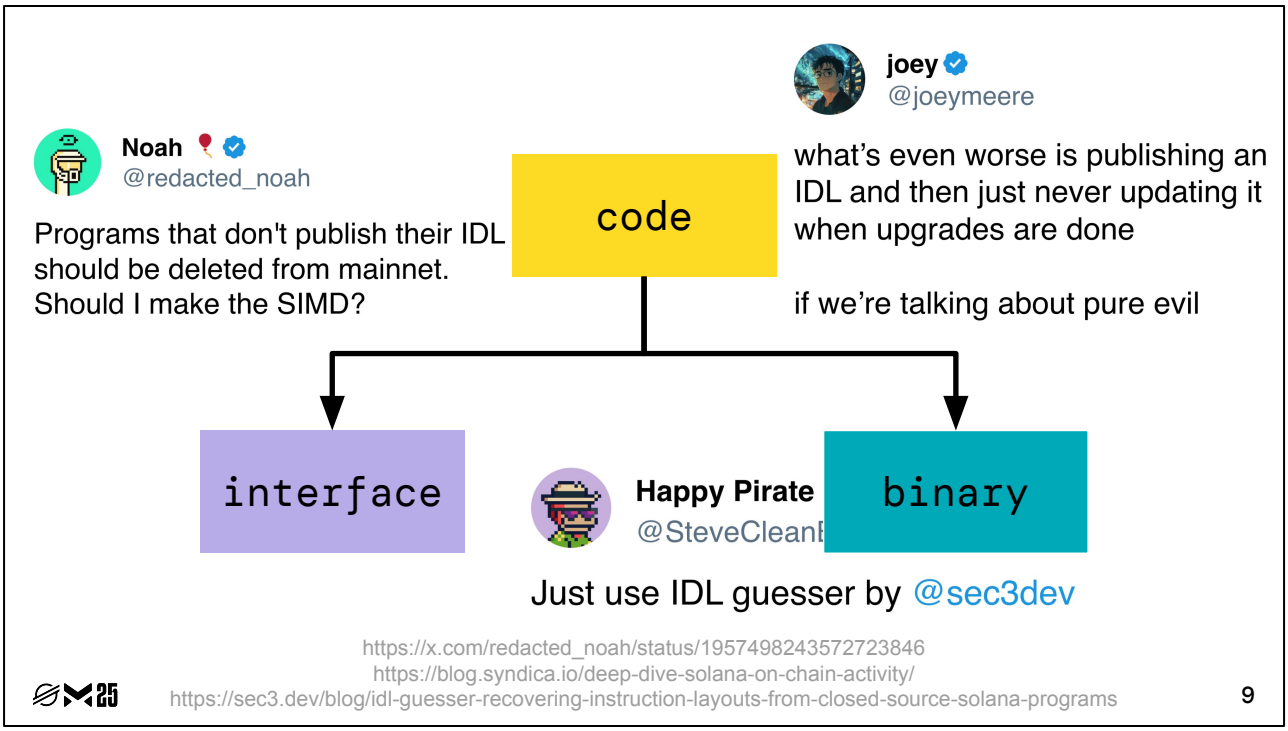



Many blockchains have their…contract interfaces be something that developers need to distribute separately to their contracts.

For EVM the interface is a JSON file that typically gets uploaded to IPFS.

For Solana the interface is also a JSON file that gets uploaded to the Solana network.

But this model hasn't played out so well…




But this model hasn't played out so well…

Earlier this year Syndica stated that only 20% of the top 1000 programs on Solana have interfaces published.

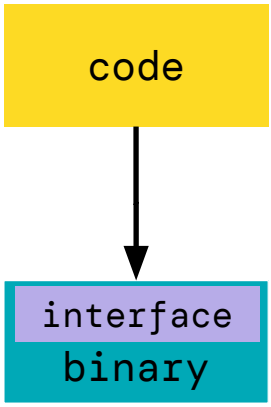
The auditor Sec3 stated in response to that, that "it's not uncommon to see the published IDLs are outdated and mismatch the deployed on-chain programs."

If you follow crypto Twitter you may have seen some of the conversation about status quo; people calling out how poor of an experience this is and calling for change.

It is such a problem that developers are using tools to guess what the interface of a Solana program is.

Ref: https://blog.syndica.io/deep-dive-solana-on-chain-activity/ (https://archive.is/egCyq)
Ref: https://sec3.dev/blog/idl-guesser-recovering-instruction-layouts-from-closed-source-solana-programs (https://archive.is/KioPz)
Ref: https://x.com/redacted_noah/status/1957498243572723846 (https://archive.is/MXpwz)
Ref: https://x.com/joeymeere/status/1957529263646642297 (https://archive.is/lvIG5)
Ref: https://x.com/SteveCleanBrook/status/1957520978969276809 (https://archive.is/FC0YC)

Stellar has taken a different approach…






Stellar has taken a different approach…

Stellar builds the interface directly into the contract binary. It's not something that developers see or distribute. It's just there, hidden inside the contract.

On Stellar contracts stand on their own on mainnet.

The spec gets inserted automatically by the Soroban Rust SDK when you run the **cargo build** command.

There's no separate process to generate it, publish it. There's no second network to rely on, like IPFS. There's no way to forget to update the IDL when upgrading.

Let's look at a real world example again. This…is the type of information that's in a Stellar contract's spec.

```rust
/// Adds liquidity to a token pair's pool, creating it if it doesn't exist.
/// Ensures that exactly the desired amounts of both tokens are added, subject
/// to minimum requirements.
/// This function is responsible for transferring tokens from the user to the
/// pool and minting liquidity tokens in return.
/// # Arguments
/// * `token_a` - The address of the first token to add liquidity for.
/// * `token_b` - The address of the second token to add liquidity for.
/// * `amount_a_desired` - The desired amount of the first token to add.
/// * `amount_b_desired` - The desired amount of the second token to add.
/// * `amount_a_min` - The minimum required amount of the first token to add.
/// * `amount_b_min` - The minimum required amount of the second token to add.
/// * `to` - The address where the liquidity tokens will be minted and sent.
/// * `deadline` - The deadline for executing the operation.
/// # Returns
/// A tuple containing: amounts of token A and B added to the pool.
/// plus the amount of liquidity tokens minted.
fn add_liquidity(token_a: Address, token_b: Address, amount_a_desired: i128, amount_b_desired:
i128, amount_a_min: i128, amount_b_min: i128, to: Address, deadline: u64) →
Result<(i128,i128,i128), CombinedRouterError>
```



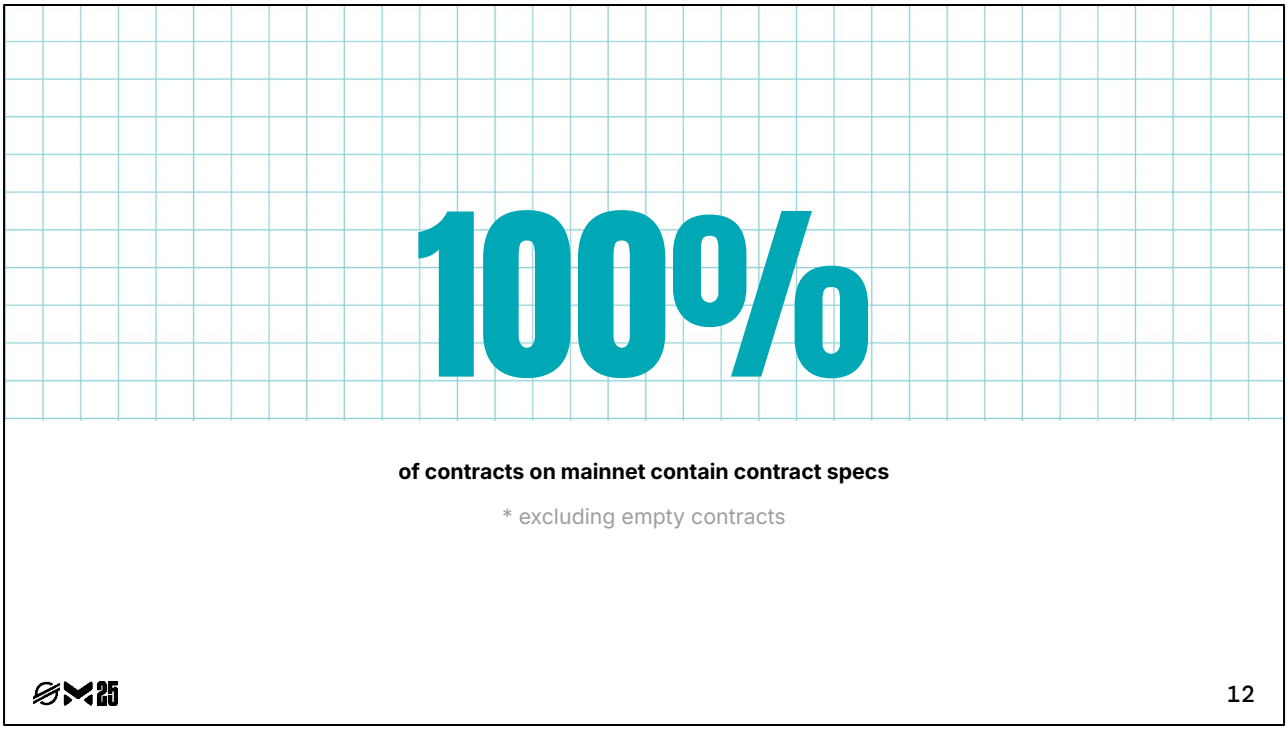
This…is the type of information that's in a Stellar contract's spec.

We have a function definition, parameter names, parameter types. And documentation!

In the whisk network upgrade that launched a couple weeks ago, there are even events too.

This example is from a real contract on mainnet.

And I'm happy to say that this is not a rare event. 100% of contracts on mainnet…




…have a spec.

That's not 100% of the top contracts, that's 100% of all contracts.

This adoption is phenomenal.

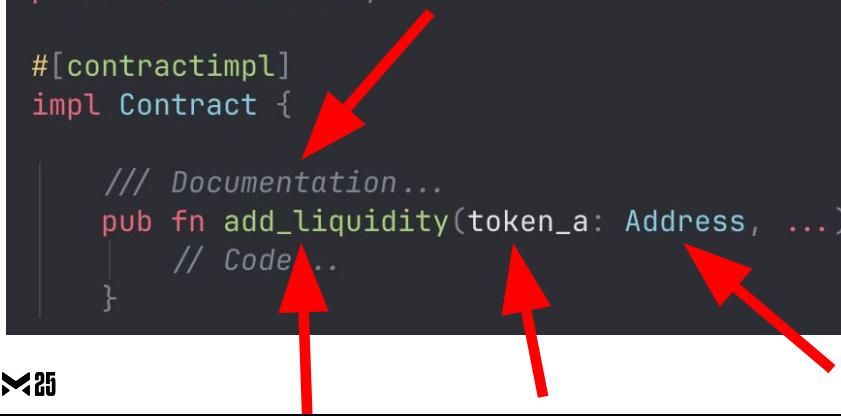
For developers of all products on Stellar, this gives you access to a rich definition of how to interact with any contract, and makes so much possible.

Ref: https://github.com/leighmcculloch/stellar-contract-wasms/blob/a1f4987ac4b267475d06fdf443f7ef9515cb1f42/analysis/spec.csv *(Note that while this CSV shows a three contracts without specs the network still has 100% publish rate because those three contracts contain zero contract functions. Contracts without functionality can't by definition have a spec since there is no interface to describe.)*

Let's see how this practically benefits users today…

rust soroban-sdk

```rust
use soroban_sdk::{contract, contractimpl, Address};

#[contract]
pub struct Contract;

#[contractimpl]
impl Contract {

    /// Documentation...
    pub fn add_liquidity(token_a: Address, ...) {
        // Code...
    }
```





Imagine I’m building a contract using the Rust SDK. I write a contract function, called add_liquidity, with its parameters and documentation.

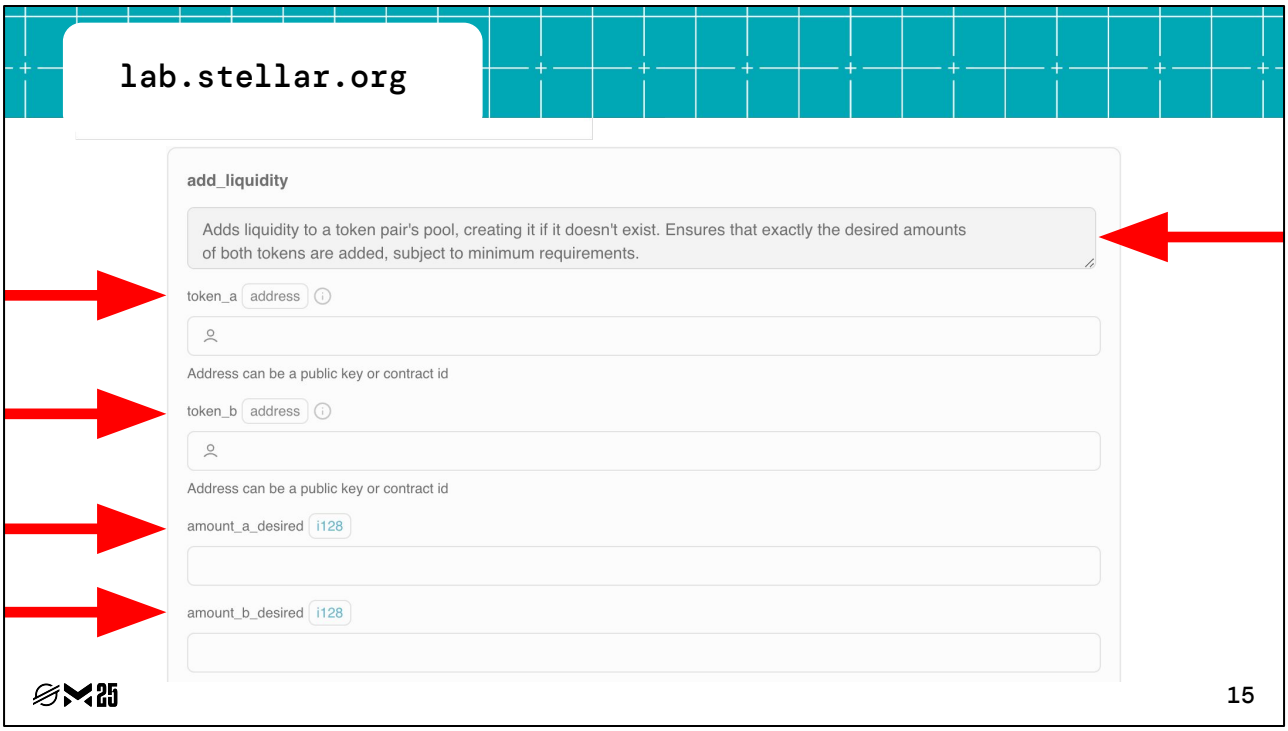
I build and deploy it, then open up the terminal to invoke it.


At the terminal I'm presented with a command line interface that is shaped by my contract.

The stellar-cli, turns my contract into a CLI of its own, with typed options dynamically built from the contract spec.

There's no code generation, no bindings generation. The Stellar CLI creates this experience dynamically at runtime.

Kudos to Willem from Aha Labs for contributing this delightful experience that uses the contract spec. It is my favourite thing about the CLI.




I load up the Stellar Lab paste in my contract ID, and I'm presented with a dynamically generated interactive UI that's been built from the spec.

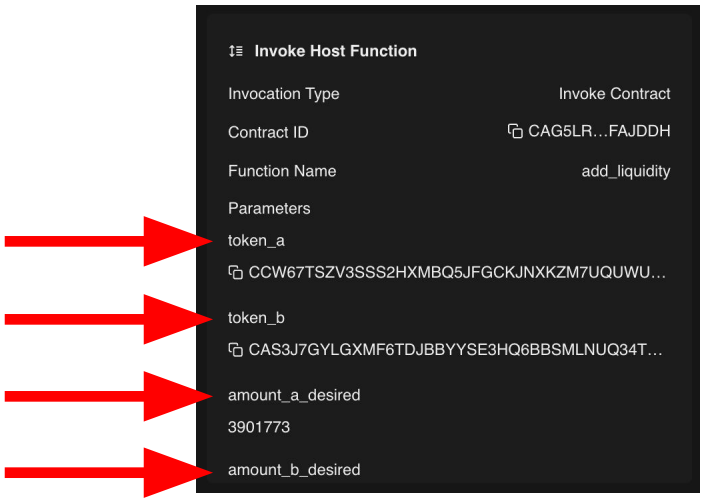
## js stellar-sdk

```
import { contract, Networks } from "npm:@stellar/stellar-sdk@14.1.1";

const client = await contract.Client.from({
  contractId: "CAG5LRYQ5JVEUI5TEID72EYOVX44TTUJT5BQR2J6J77FH65PCCFAJDDH",
  networkPassphrase: Networks.PUBLIC,
  rpcUrl: "https://mainnet.sorobanrpc.com",
});

const { result } = await client.add_liquidity({ ... });

console.log(result);
```



I build a web application using the JavaScript Stellar SDK. Given only a contract ID, the SDK turns any contract into a runtime generated client for calling the contract.

freighter






I load up my Freighter Wallet to sign a transaction, and it displays the transaction, using the contract spec to annotate the data passed to the contract, so that as the signer, I know what each of the parameters mean.

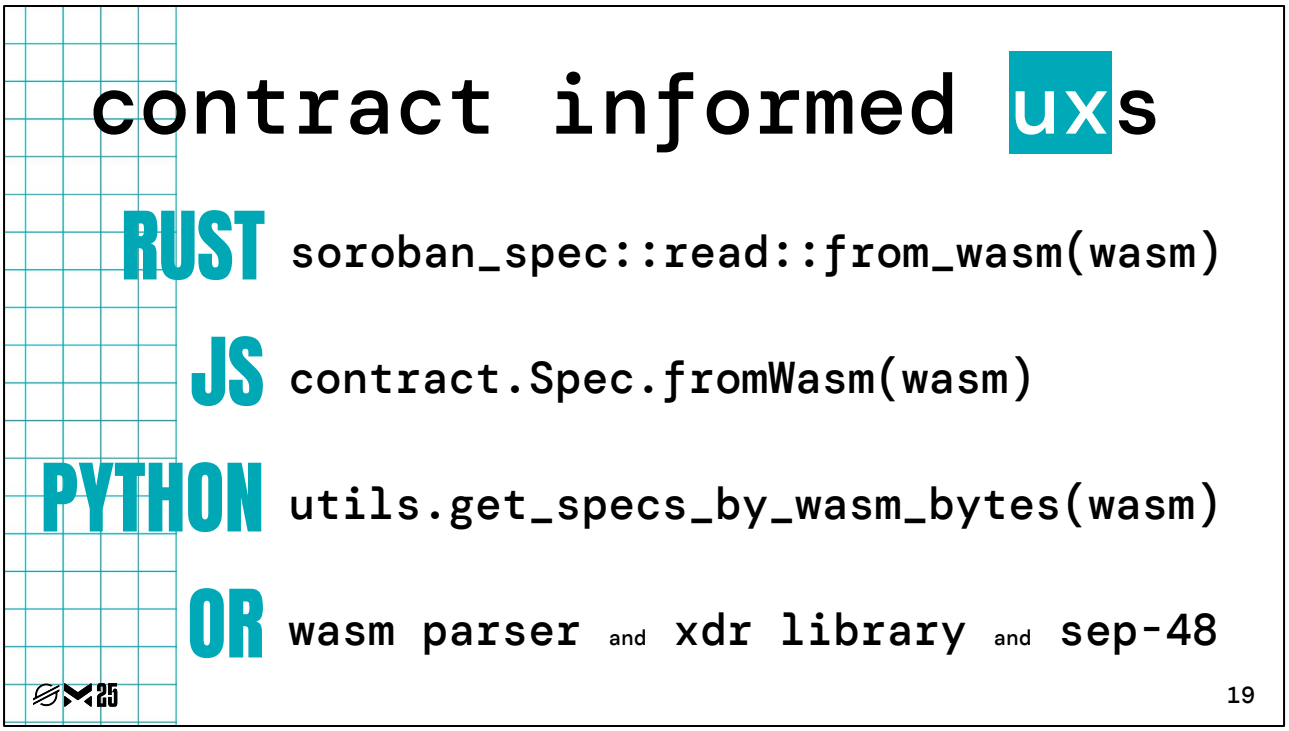
## open zeppelin monitor

```
"match_conditions": {
  "functions": [
    {
      "signature": "add_liquidity(Address,Address,i128,i128,i128,i128,Address,u46)",
      "expression": "amount_a_desired > 10000000000"
    }
  ]
}
```



I'm using the Open Zeppelin Monitor to alert me to activity on the contract.

I specify my alert rules using parameter names that the monitor matches to transaction data using the specs.




They are just a few of the ways that contract specs get used today.

Contract specs exist so that products can create amazing contract informed user experiences.

If you want to get access to the specs in your application you can use one of these utilities.

If you're building in another language, you can still get the specs, you'll need a:

A wasm parser.
And a Stellar XDR library.
And read SEP-48, it's pretty accessible, and it tells you what to do with those two things.

Next topic…


Have you ever wondered how contract tests for Stellar run in a fraction of a second while still accurately matching execution on chain?

It's all because of the soroban environment, the contract runtime of Stellar.

It exists as a Rust crate that is…embedded into stellar-core, the main node software for the Stellar network.

# execution




It exists as a Rust crate that is…embedded into stellar-core, the main node software for the Stellar network.

When I invoked my contract earlier, this is where it executed on chain, inside the soroban-environment inside stellar-core.

But the environment was designed from day one to be usable in more than just nodes.

The environment is also used in the soroban-sdk to…run tests.

## rust soroban-sdk

```rust
use soroban_sdk::{Address, Env};
use crate::{Contract, ContractClient};

#[test]
fn test_add_liquidity() {
    let env = Env::default();
    let contract_id = env.register(Contract, ());
    let client = ContractClient::new(&env, &contract_id);

    let token_a = Address::generate(&e);
    // ...
    client.add_liquidity(token_a, ..);
    // Assertions..
}
```



The soroban environment is also used in the soroban-sdk to…run tests.

When I write this test for my contract, the env value created at the beginning of the test contains the same runtime as the stellar-core node software does.

As a maintainer of the SDK I spend zero time maintaining a simulator.

As a contract developer I have confidence that the behaviour of my tests are as close as possible to the real deployment.

The soroban environment is also used in the rpc to perform…transaction simulation.


simulation without a simulator

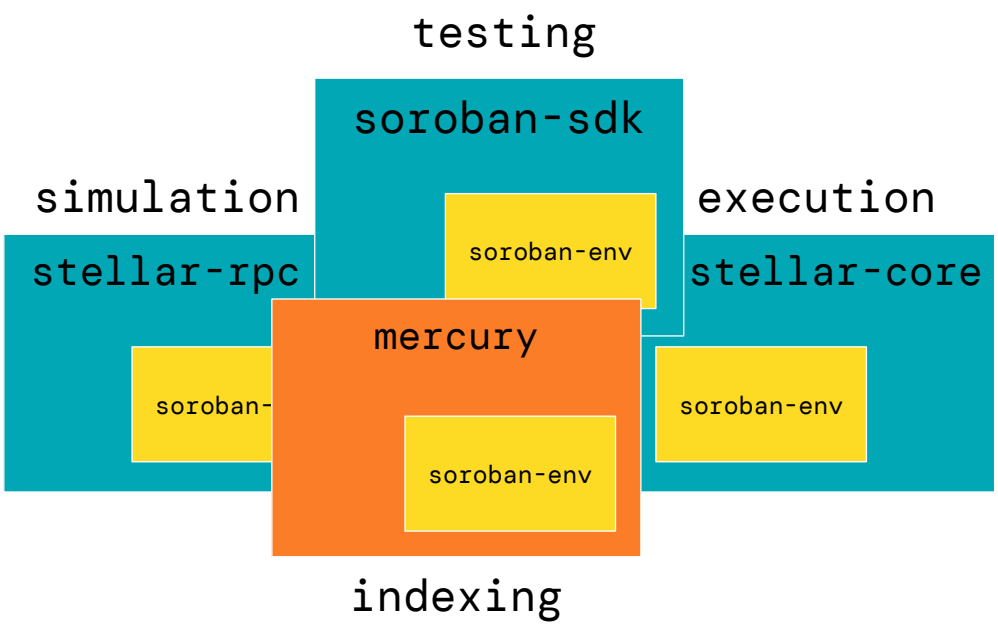


The soroban environment is also used in the stellar-rpc to perform…transaction simulation.

We call it simulation because the transaction is not running on chain, but it is still running inside the real runtime. There's no simulator.

In many blockchains there's a divide between the real runtime and the runtime you see in simulators and test environments.

But with Stellar, Soroban was built from day one to be embedded in all of these places.






Products building on Stellar can also embed the environment.

We're seeing some exciting ways that developers in the ecosystem are using the environment, in ways that we didn't imagine.

Calling out Tommaso (tdep) from xyclooLabs who is embedding the environment in mercury to power indexing.

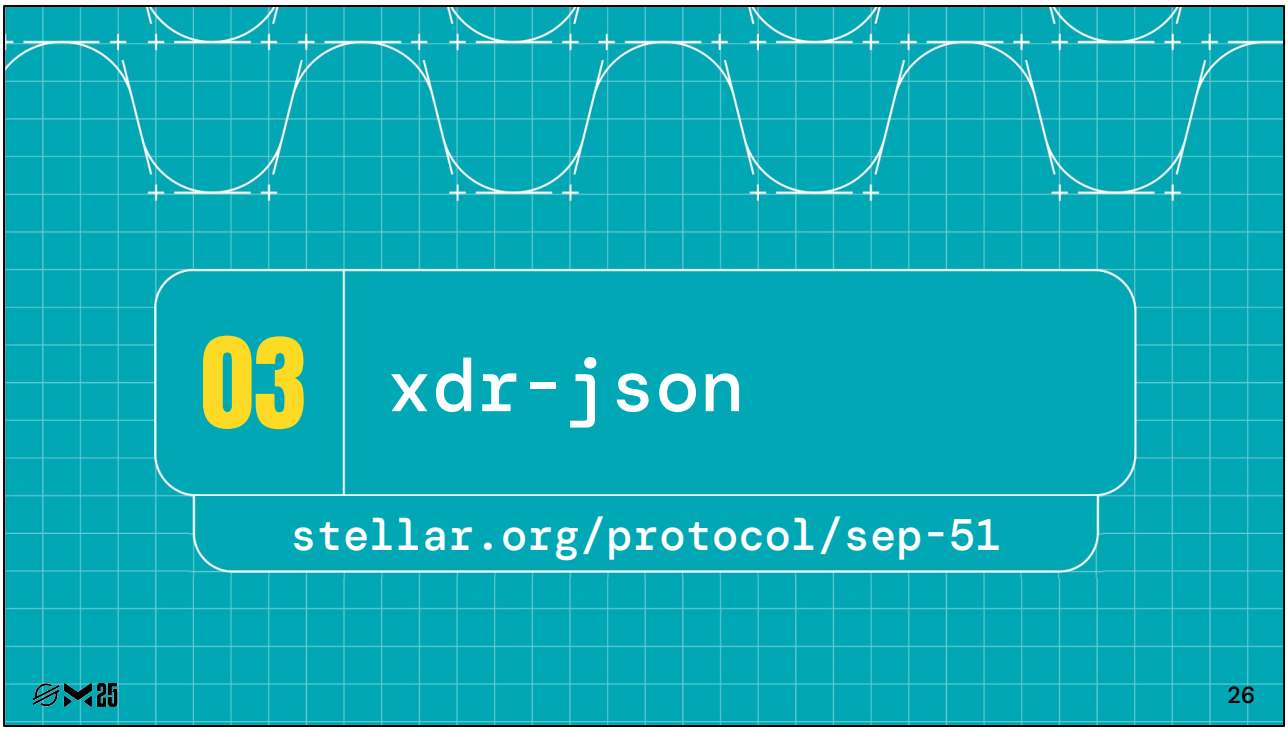

There’s so much potential to build powerful products because the soroban runtime is so readily embeddable.

If you’d like to use the soroban-env in your applications:

Check out the soroban-env-host and soroban-simulation Rust crates.

And take questions to the Stellar Developer Discord.

Moving to our last topic…




Moving to our last topic…which is xdr-json.

Have you noticed that the lab renders Stellar transactions really beautifully?

Take a look at this screenshot…




Take a look at this screenshot…

This is what you see in the lab if you give the lab a transaction in XDR.

It’s not how XDR normally looks.

Let’s take a look at XDR, and then we’ll come back to XDR-JSON.

xdr

binary format

rfc4506

used on stellar



So XDR is a binary format.

It's defined in RFC4506 and it's used by the Stellar network for everything: transactions, consensus messages, events, and for data storage.

## github.com/stellar/stellar-xdr

```
struct Transaction
{
    MuxedAccount sourceAccount;
    uint32 fee;
    SequenceNumber seqNum;
    Preconditions cond;
    Memo memo;
    Operation operations<MAX_OPS_PER_TX>;
    //...
};
```



The Stellar XDR is defined in that repo.

The definitions are written in XDR language that just happens look like C code.

All the Stellar SDKs that support working with XDR have code generated from the XDR definitions in this repo.

XDR has a lot going for it…

| | xdr |
|---|---|
| efficient | ✅ |
| simple | ✅ |
| no forward compatibility | ✅ |
| deterministic encoding | ✅ |
| readable by devs | ❌ |



XDR has a lot going for it…

It's efficient.

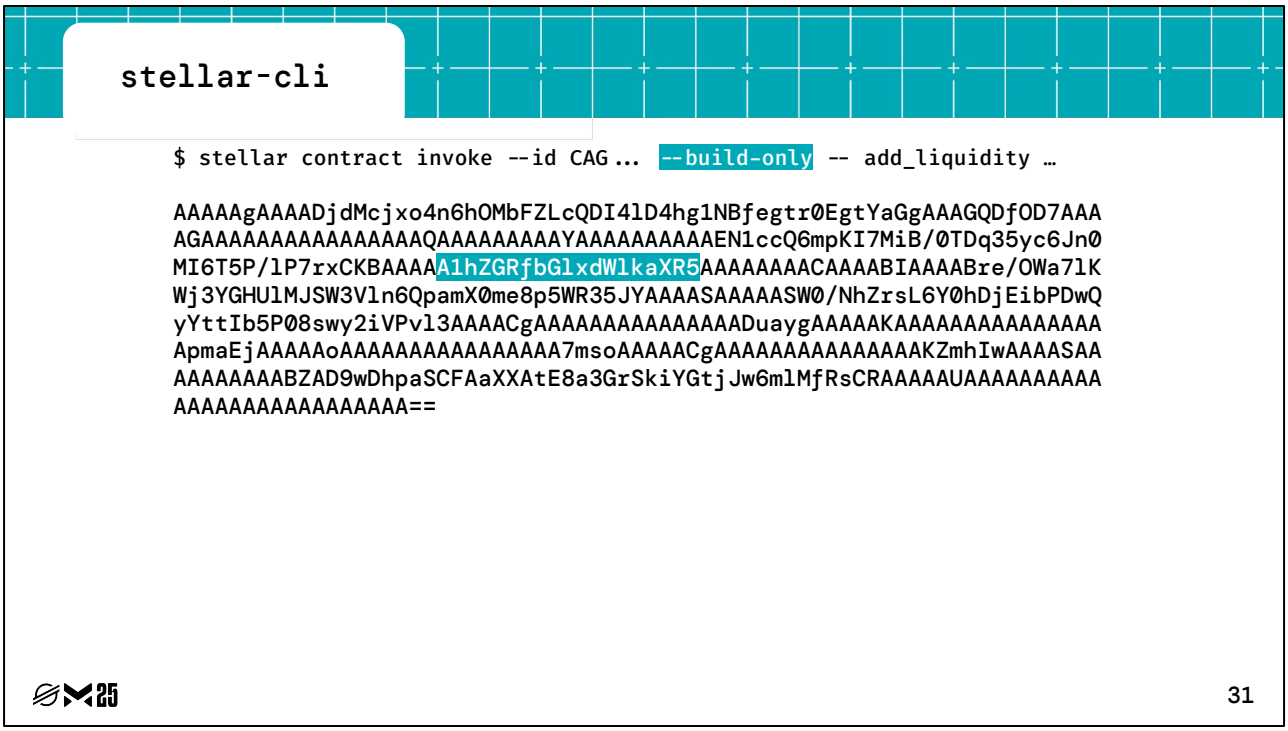
It's one of the simplest binary format.

It has no automatic forward compatibility that ensures that systems consuming transactions don't miss important details when new data is added to transactions.

It's encoding is deterministic that is really good for transaction signing because there one valid binary representation.

But it's binary, it is not easy for humans to read.

Let's see what I mean…






Think back to when we were building that contract and when we invoked the add_liquidity function using the stellar-cli.

I'd like to inspect the transaction that the invoke command created.

I run the same command, but I use the –build-only option so that it will output the transaction on the command line.

The command outputs the transaction as base64 encoded binary.

Most of the time this is how developers see XDR.

The highlighted section in the base64 is the function name, add_liquidity, and it's not readable at all.

If we decode the base64 and look at the binary…

## stellar-cli

```
$ stellar contract invoke --id CAG... --build-only -- add_liquidity …
    | base64 -d
    | xxd

00000000: 0000 0002 0000 0000 e374 c723 c68e 27ea  .........t.#..'.
00000010: 138c 6c56 4b71 00c8 e250 f886 0d4d 05f7  ..lVKq...P...M..
00000020: a0b6 bd04 82d6 1a1a 0000 0064 037c e0fb  ...........d.|..
00000030: 0000 0018 0000 0000 0000 0000 0000 0001  ................
00000040: 0000 0000 0000 0018 0000 0000 0000 0001  ................
00000050: 0dd5 c710 ea6a 4a23 b322 07fd 130e adf9  .....jJ#."......
00000060: c9ce 899f 4308 e93e 4ffe 53fb af10 8a04  ....C..>O.S.....
00000070: 0000 000d 6164 645f 6c69 7175 6964 6974  ....add_liquidit
00000080: 7900 0000 0000 0008 0000 0012 0000 0001  y...............
00000090: adef ce59 aee5 2968 f760 61d4 94c2 525b  ...Y..)h.`a...R[
000000a0: 7565 9fa4 296a 65f4 99ef 29e5 6477 e496  ue..)je...).dw..
000000b0: 0000 0012 0000 0001 25b4 fcd8 59ae c2fa  ........%...Y...
000000c0: 6348 438c 489b 3c3c 10c9 8b6d 21be 4fd3  cHC.H.<<...m!.O.
000000d0: cb30 cb68 953e f977 0000 000a 0000 0000  .0.h.>.w........
000000e0: 0000 0000 0000 0000 3b9a ca00 0000 000a  ........;.......
000000f0: 0000 0000 0000 0000 0000 0000 a666 848c  .............f..
00000100: 0000 000a 0000 0000 0000 0000 0000 0000  ................
```



When we decode the base64 and look at the binary…

We can see a little bit more. We can see the add_liquidity function name is in there, but other than that, the rest is lost on us.

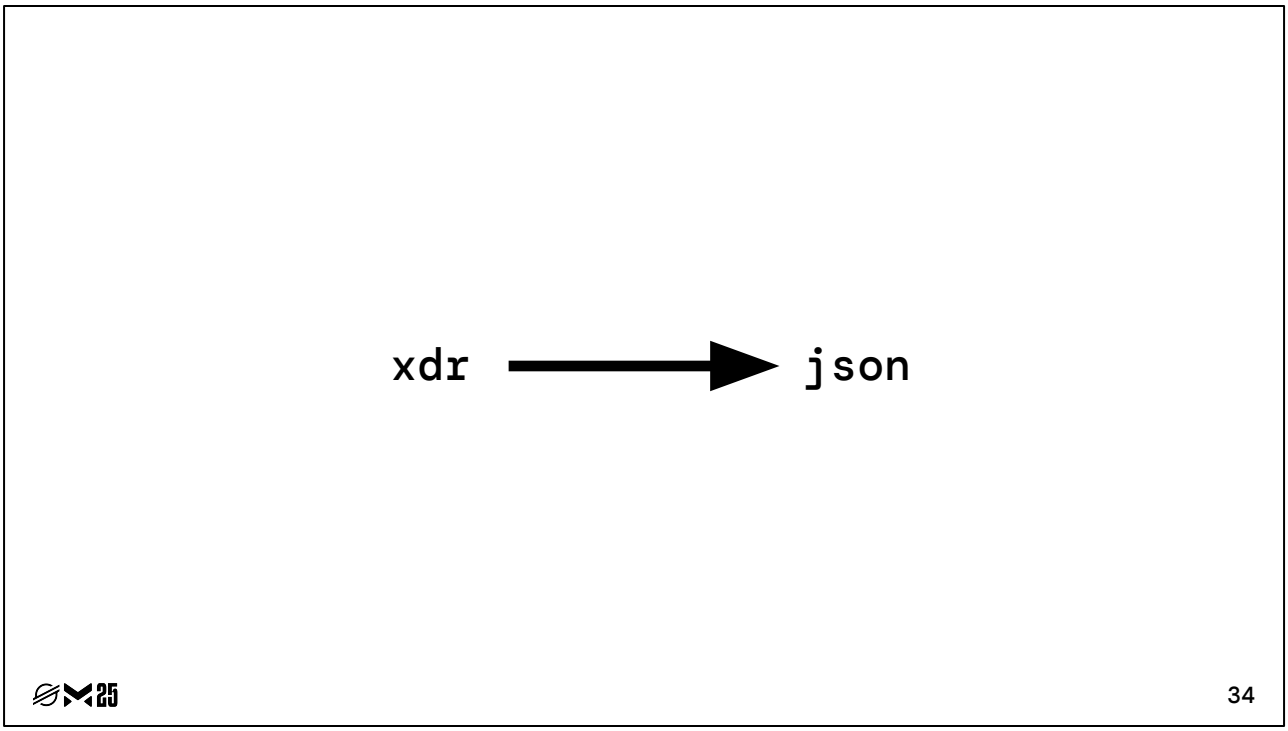
This is where…xdr-json comes in.

## xdr-json



And that's where…xdr-json comes in.

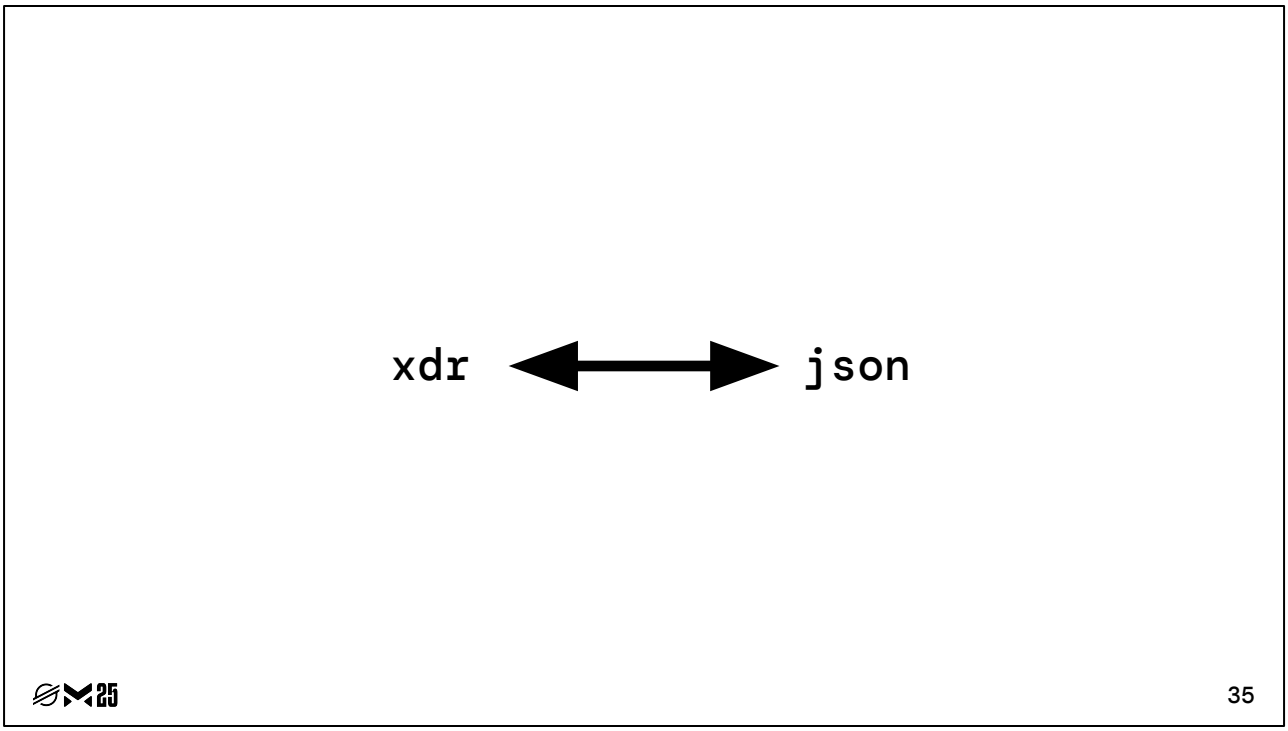
XDR JSON is defined in SEP-51 and it is a formula for what any XDR value should look like when…converted into JSON.






XDR JSON is defined in SEP-51 and it is a formula for what any XDR value should look like when…converted into JSON.

The mapping supports lossless round trip conversion, so every XDR value can be represented as JSON, and then converted…back into the exact original binary.






The mapping is a lossless round trip conversion, so every XDR value can be represented as JSON, and then converted…back into the exact original binary.

It works for the entire Stellar XDR library, all 433 types.

Let’s have a look at how this creates some delightful experiences.

Imagine I’m back inspecting that transaction.

## stellar-cli

```
$ stellar contract invoke --id CAG... --build-only -- add_liquidity …
    | stellar tx decode
    | jq '.tx.tx.operations[0].body'
```


```
{
  "invoke_host_function": {
    "host_function": {
      "invoke_contract": {
        "contract_address": "CAG5LRYQ5JVEUI5TEID72EYOVX44TTUJT5BQR2J6J77FH65PCCFAJDDH",
        "function_name": "add_liquidity",
        "args": [
          {
            "address": "CCW67TSZV3SSS2HXMBQ5JFGCKJNXKZM7UQUWUZPUTHXSTZLEO7SJMI75"
          },
          {
            "address": "CAS3J7GYLGXMF6TDJBBYYSE3HQ6BBSMLNUQ34T6TZMYMW2EVH34XOWMA"
          },
          {
            "i128": "1000000000"
          },
```




Imagine I'm back inspecting that transaction.

This time I pipe the binary to the stellar tx decode command which gives me back JSON.

Because the transaction is JSON, I have the whole JSON ecosystem of tools available to me.

So I use tools like the jq tool, to extract out a specific field within the JSON.

I decide I want to edit the transaction before its submitted…

## stellar-cli

```
$ stellar contract invoke --id CAG... --build-only -- add_liquidity ... | stellar tx edit
```


```
{
  "$schema": "https://stellar.org/schema/xdr-json/v23.0.0/TransactionEnvelope.json",
  "tx": {
    "tx": {
      "operations": [ {
        "body": {
          "invoke_host_function": {
            "host_function": {
              "invoke_contract": {
                "function_name": "add_liquidity",
                "args": [
                  { "address": "CCW67TSZV3SSS2HXMBQ5JFGCKJNXKZM7UQUWUZPUTHXSTZLEO7SJMI75" },
                  { "address": "CAS3J7GYLGXMF6TDJBBYYSE3HQ6BBSMLNUQ34T6TZMYMW2EVH34XOWMA" },
                  { "i128": "1000000000" },
                  { "i128": "2791736460" },
                  { "i128": "1000000000" },
                  { "i128": "2791736460" },
                  { "address": "GBMQAP3QBYNFUSBBIBUXLQFUJ4NNY2WSSITANNRSODVGSTD5DMBEIQWM" },
                  { "u64": "0" }
                ],
```




I decide I want to edit the transaction before its submitted…

I rerun the command but this time I pipe the result to the stellar tx edit command.

It opens my editor with the tx decoded into XDR-JSON with a schema line injected at the top.

Editors like VSCode and VIM that support a JSON language server use the schema to provide…

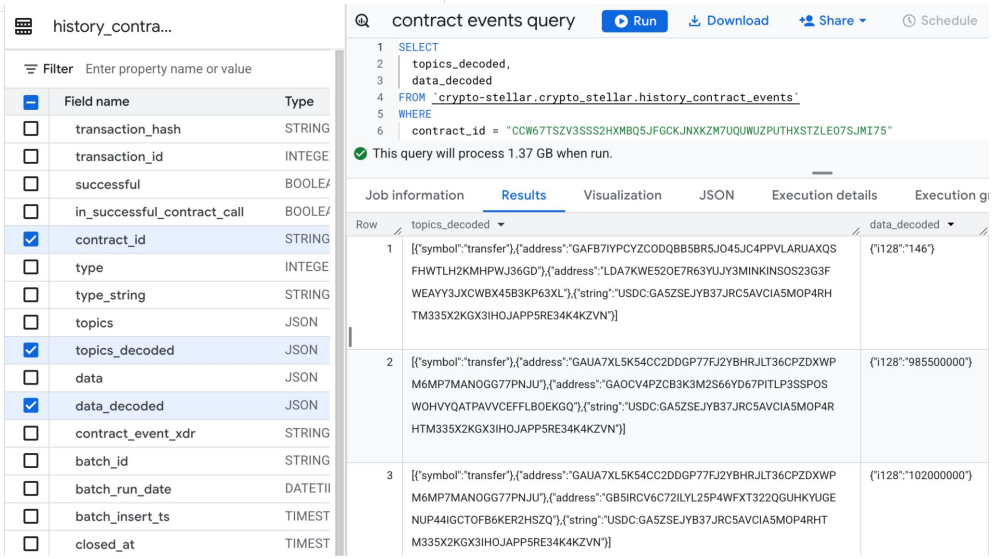

Editors like VSCode and VIM that support a JSON language server use the schema to provide…

Auto completion, documentation, validation, warnings.

Here the IDE is showing me all the possible values that I can specify for a function argument.

Outside of the terminal…

## hubble (bigquery)






Outside of the terminal…

I open up Hubble, the Stellar BigQuery dataset.

I can use the XDR-JSON to query XDR values like contract storage and contract events.

Here in this screenshot I'm just dumping out all the JSON. But BigQuery has great support for working with JSON and I can filter on the contents of the JSON, or extract fields out of it.

It's really powerful.

## stellar-xdr mcp

```
Hi AI, what does this mean?
AAAAAgAAAADjdMcjxo4n6hOMbFZLcQDI4lD4hg1NBfegtr0EgtYaGgAAAGQDfOD7AAAAGAAAAAAAAAAAAAAAAAQAA
AAAAAAAYAAAAAAAAAAEN1ccQ6mpKI7MiB/0TDq35yc6Jn0MI6T5P/lP7rxCKBAAAAA1hZGRfbGlxdWlkaXR5AAAA

  ✓ mcp_stellar-xdr_xdr_guess {"xdr":"AAAAAgAAAADjdMcjxo4n6hOMbFZLcQDI4lD4hg1NBfegtr0E…

    FeeBumpTransactionInnerTx, TransactionEnvelope

  ✓ mcp_stellar-xdr_xdr_decode {"type":"TransactionEnvelope","xdr":"AAAAAgAAAADjdMcjxo…

    {"tx":{"tx":{"source_account":"GDRXJRZDY2HCP2QTRRWFMS3RADEOEUHYQYGU2BPXUC3L2BEC2Y…

This is a Stellar blockchain transaction envelope for adding liquidity to a smart
contract. The transaction:

• Sources from account GDRXJRZDY2HCP2QTRRWFMS3RADEOEUHYQYGU2BPXUC3L2BEC2YNBVUEY
• Calls add_liquidity function on contract
CAG5LRYQ5JVEUI5TEID72EYOVX44TTUJT5BQR2J6J77FH65PCCFAJDDH
• Supplies amounts of two tokens: 1,000,000,000 and 2,791,736,460 units
• Includes two user addresses and a pool address
• Has a fee of 100 stroops and sequence number 251323047867318296
• Is unsigned (no signatures included)
```



Chatting to my AI agent that's connected to the Stellar XDR MCP, I can ask questions about what a tx does and even ask the agent to make changes to the transaction, or create entirely new XDR values.

All of this possible because the Stellar XDR MCP gives the AI agent XDR JSON and JSON Schema.

And agents know how to work with JSON.

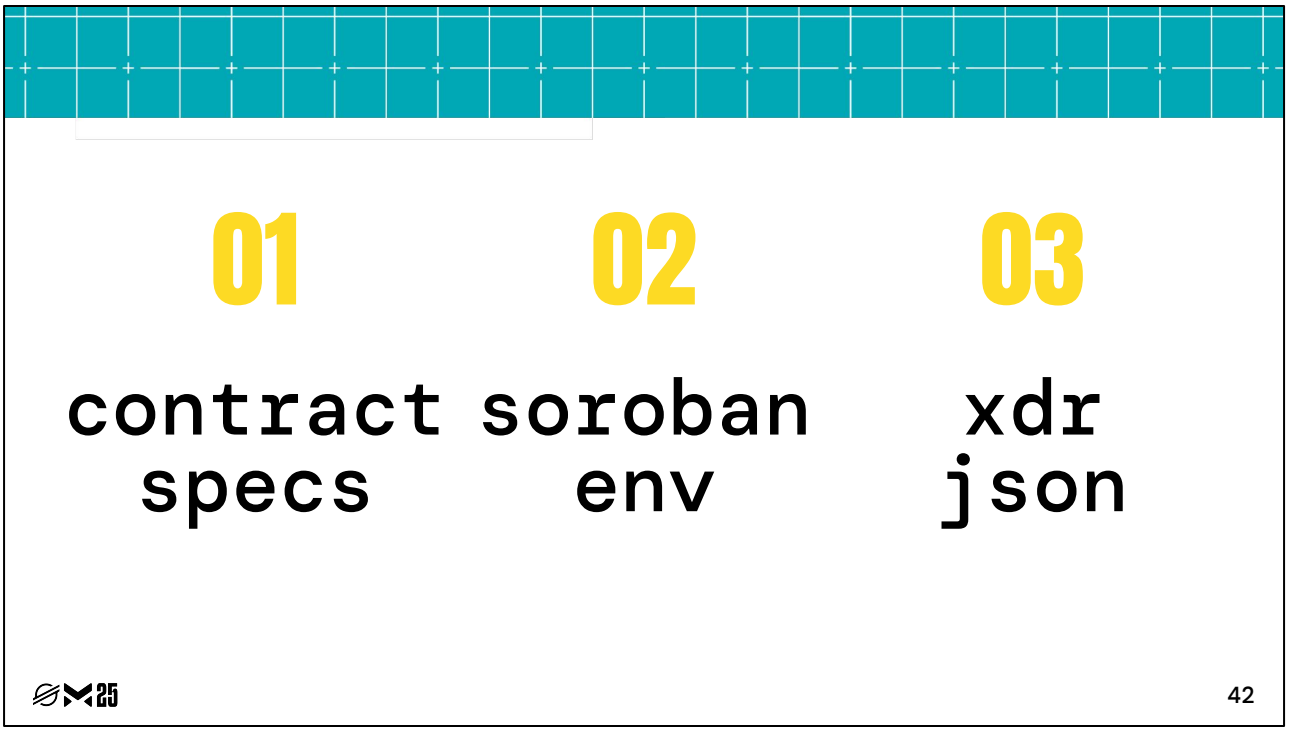

XDR-JSON, a great default for working with XDR, in any product. It creates for a delightful experience when interacting with raw network data.

If you're building any developer experience, you're bound to touch XDR, and would benefit from using XDR-JSON as an encoding format.

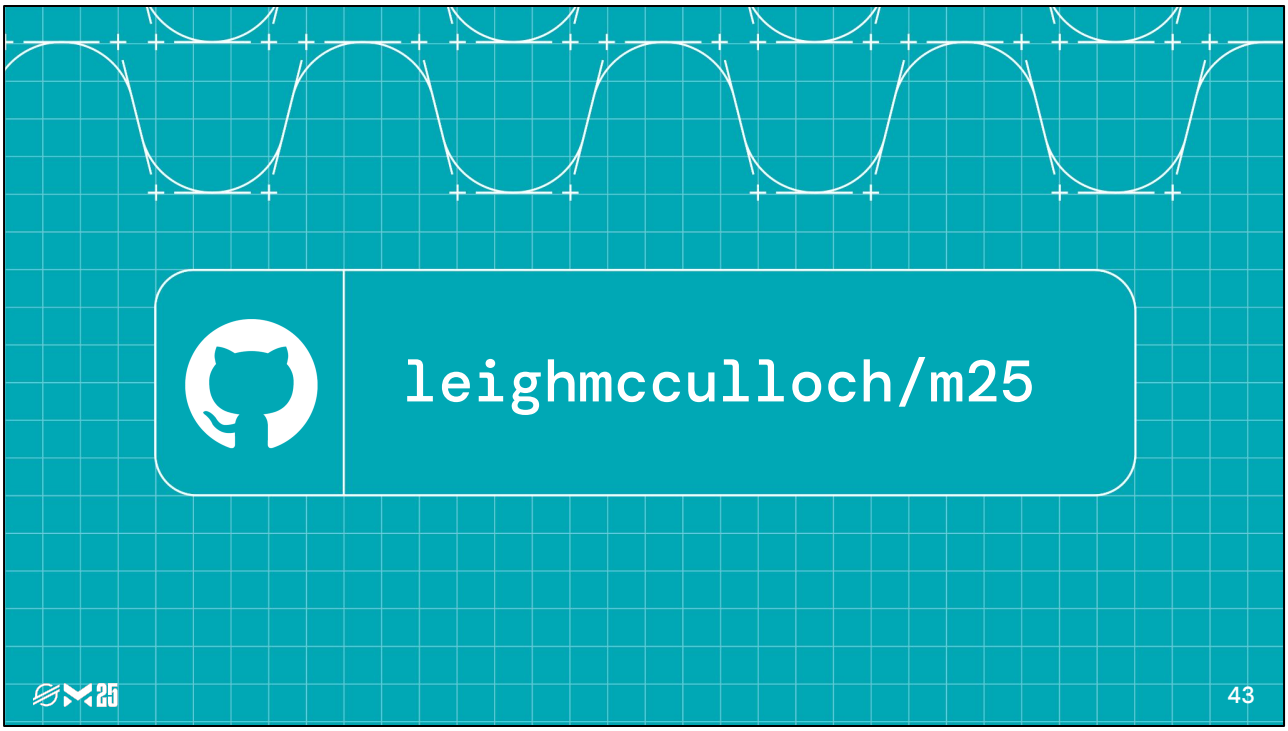
It's available in three languages today.

Rust, JS, and Go.

And hopefully more in the future.




To wrap up, these features are a boost to developer products, making them more powerful and delightful to use.




I want to leave you with code. Checkout this GitHub repository for examples of how to use all these features.

Ref: https://github.com/leighmcculloch/m25