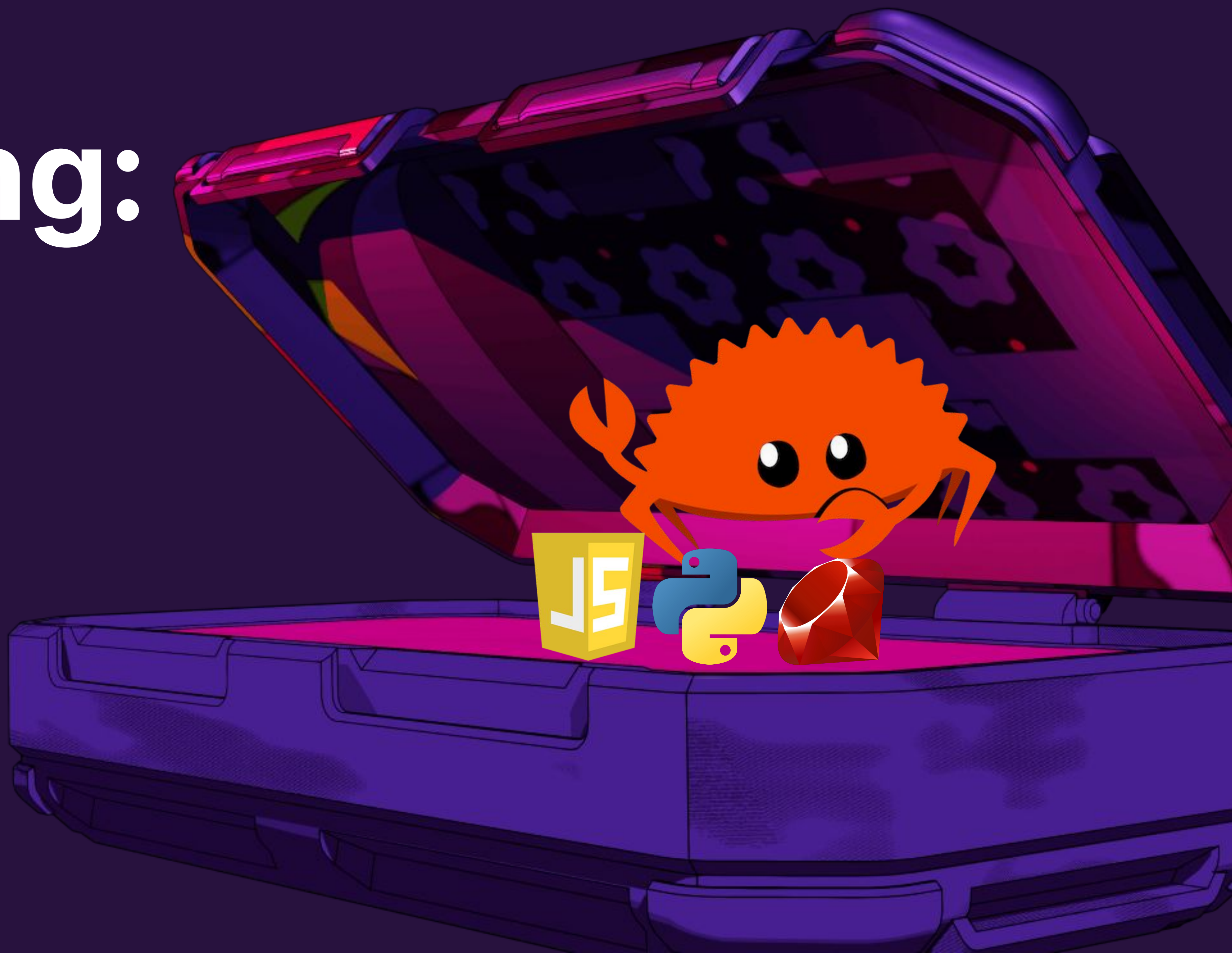
Polkadot

From High-Level to Systems Programming:

A Practical Guide to Rust



@HummusOnRails

Polkadot

@HummusOnRails

Polkadot

Maybe your language of choice is...

@HummusOnRails

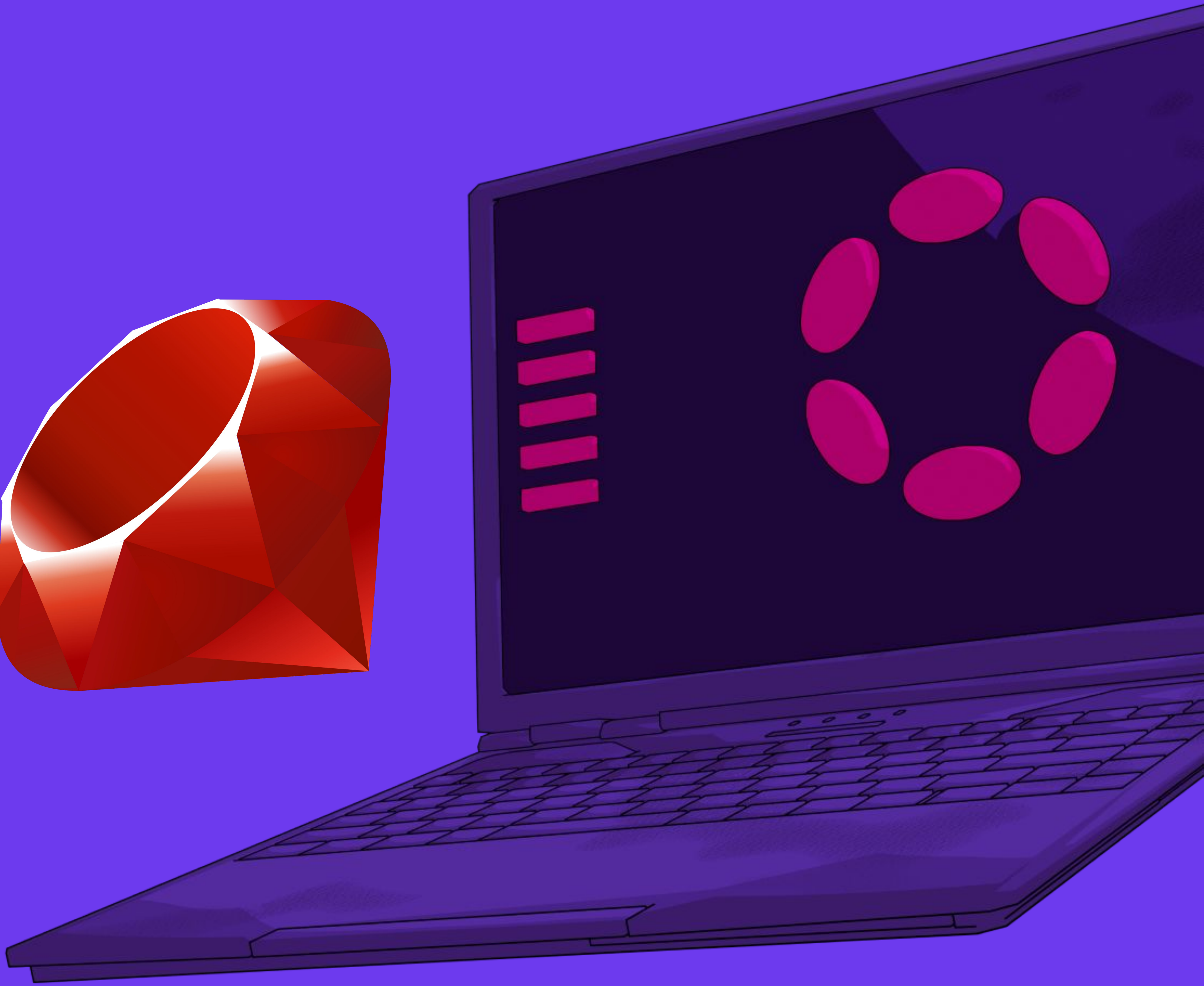
Polkadot

Maybe your language of choice is...

@HummusOnRails

Polkadot

Maybe your language of choice is...

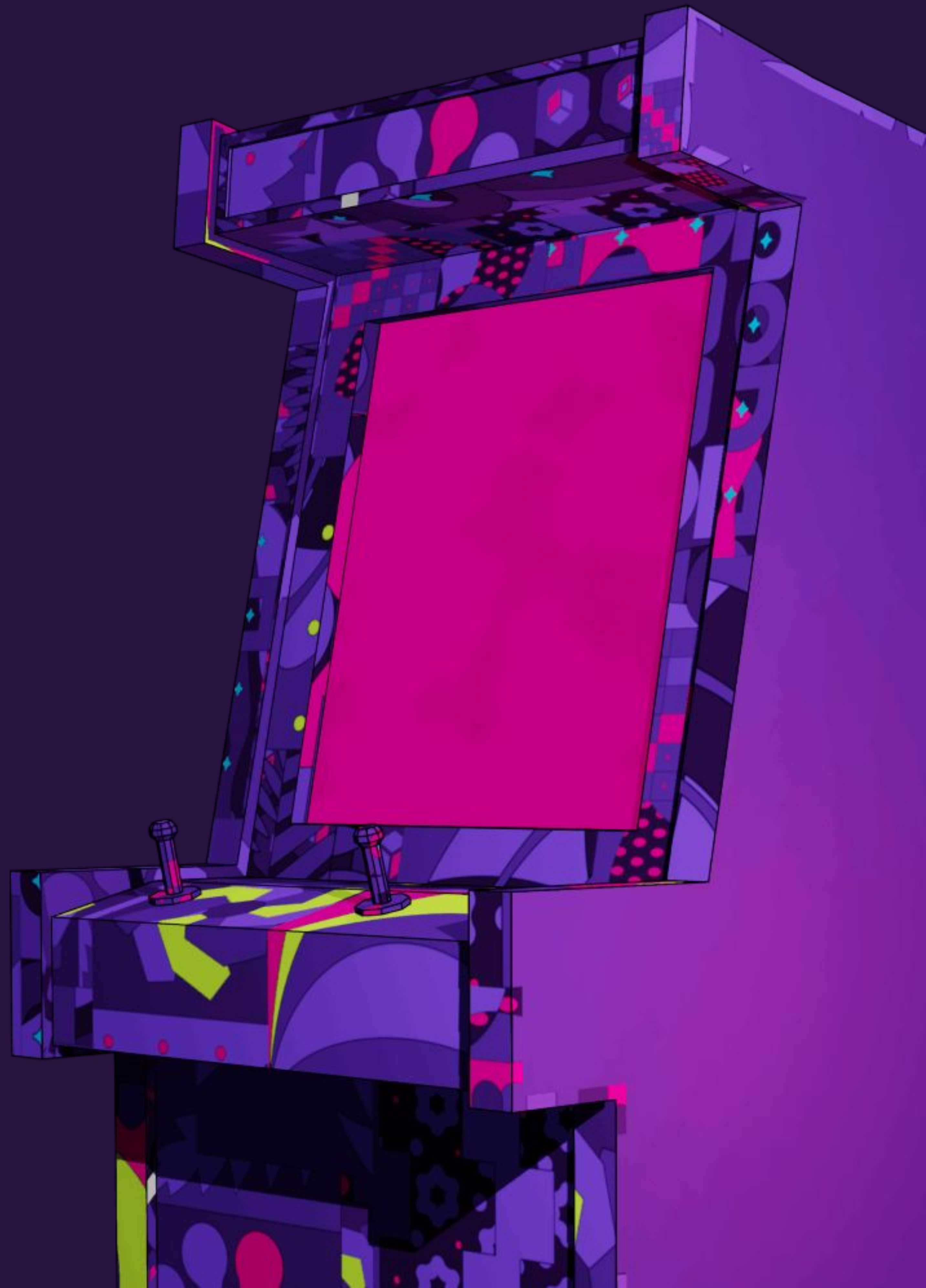


@HummusOnRails

Polkadot

And now you want to learn Rust

How do you start?



@HummusOnRails

Polkadot

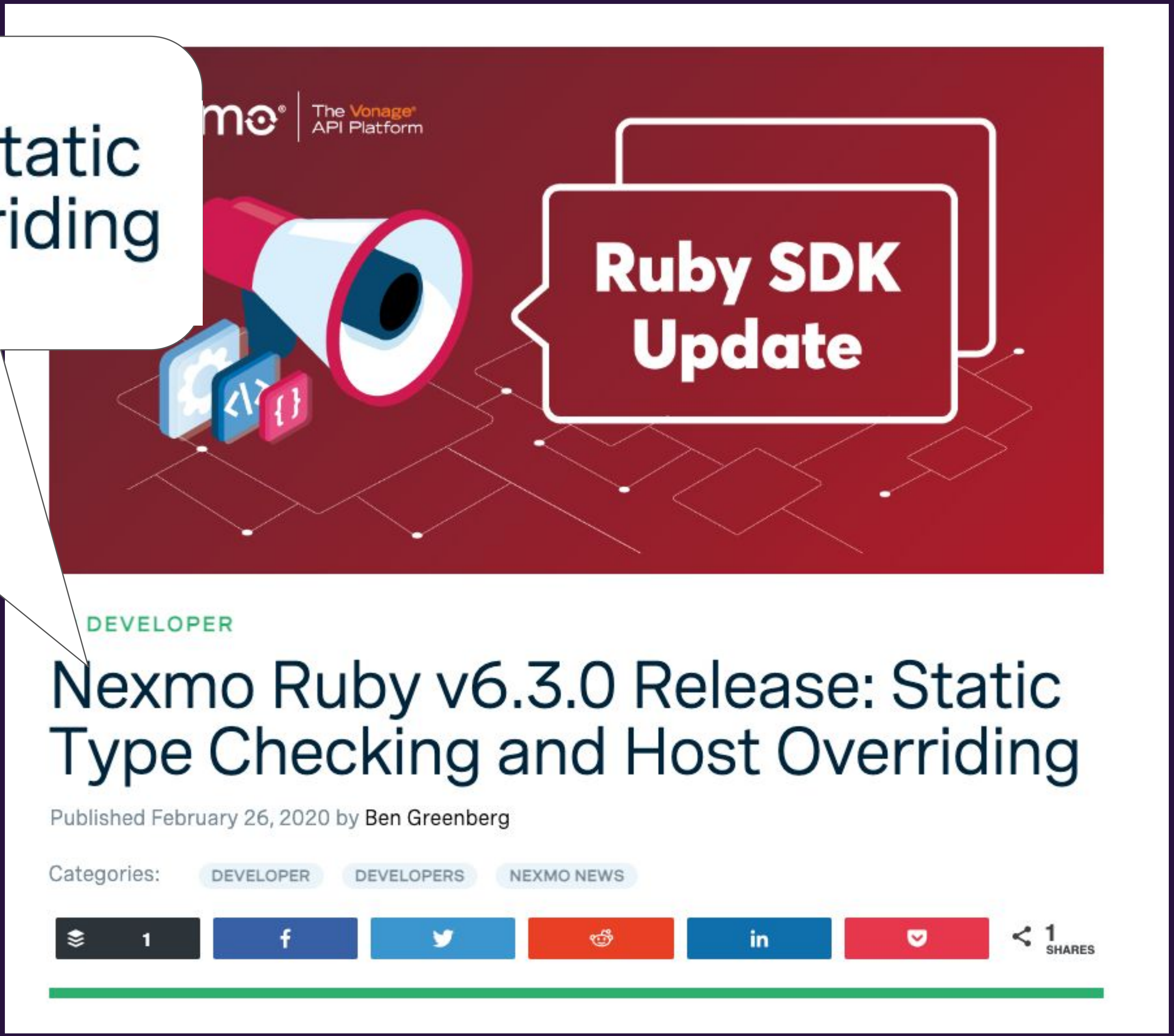
My journey to Rust started with...

@HummusOnRails

Polkadot

Nexmo Ruby v6.3.0 Release: Static Type Checking and Host Overriding

Published February 26, 2020 by Ben Greenberg



DEVELOPER

Nexmo Ruby v6.3.0 Release: Static Type Checking and Host Overriding

Published February 26, 2020 by Ben Greenberg

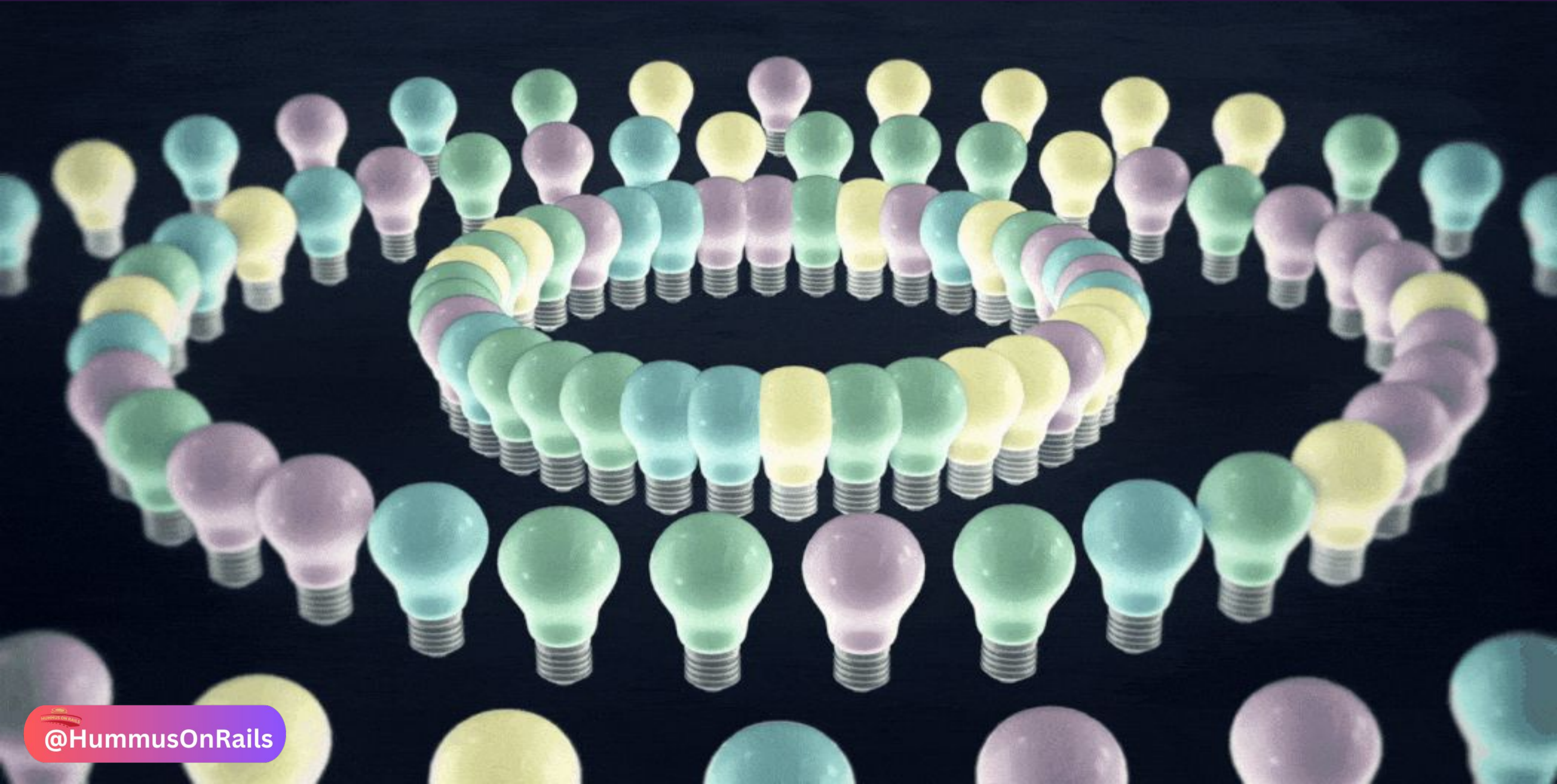
Categories: DEVELOPER DEVELOPERS NEXMO NEWS

1

1 SHARES

@HummusOnRails

Polkadot



@HummusOnRails

Polkadot

parity

@HummusOnRails

Polkadot

@HummusOnRails

Polkadot

Curriculum overview

MODULE 1

Cryptography

Communicating securely in the presence of an adversary facilitates everything from civil disobedience to online shopping. It is foundational for blockchains.

MODULE 2

Economics and Game Theory

Financial and political systems are under constant threat of attack. By understanding the strategic interactions of rational actors, such systems can be designed securely through proper incentives.

MODULE 3

Core Blockchain Concepts

A blockchain is comprised of P2P networking, distributed consensus, and many other constituents. In this module, we look at these subsystems, explore how they fit together, and understand why a blockchain may be desirable to begin with.

MODULE 4

Smart Contracts

By allowing users to deploy executable code, smart contracts allow individuals to have arbitrary interactions with one another without requiring trust.

MODULE 5

Substrate

Learn the architecture and design of Substrate, the production-ready modular framework for building blockchains, and get hands-on experience working with it.

MODULE 6

FRAME and Pallets

Learn the ins and outs of building Substrate blockchains using FRAME, the Rust-based DSL that makes it easy to compose discrete pieces of logic into a meaningful application.

MODULE 7

Polkadot, Parachains, and Cumulus

Polkadot is the heterogeneous multichain network. Learn about the rights and responsibilities of the various chains, how they operate together to share security, and how it is all governed.

MODULE 8

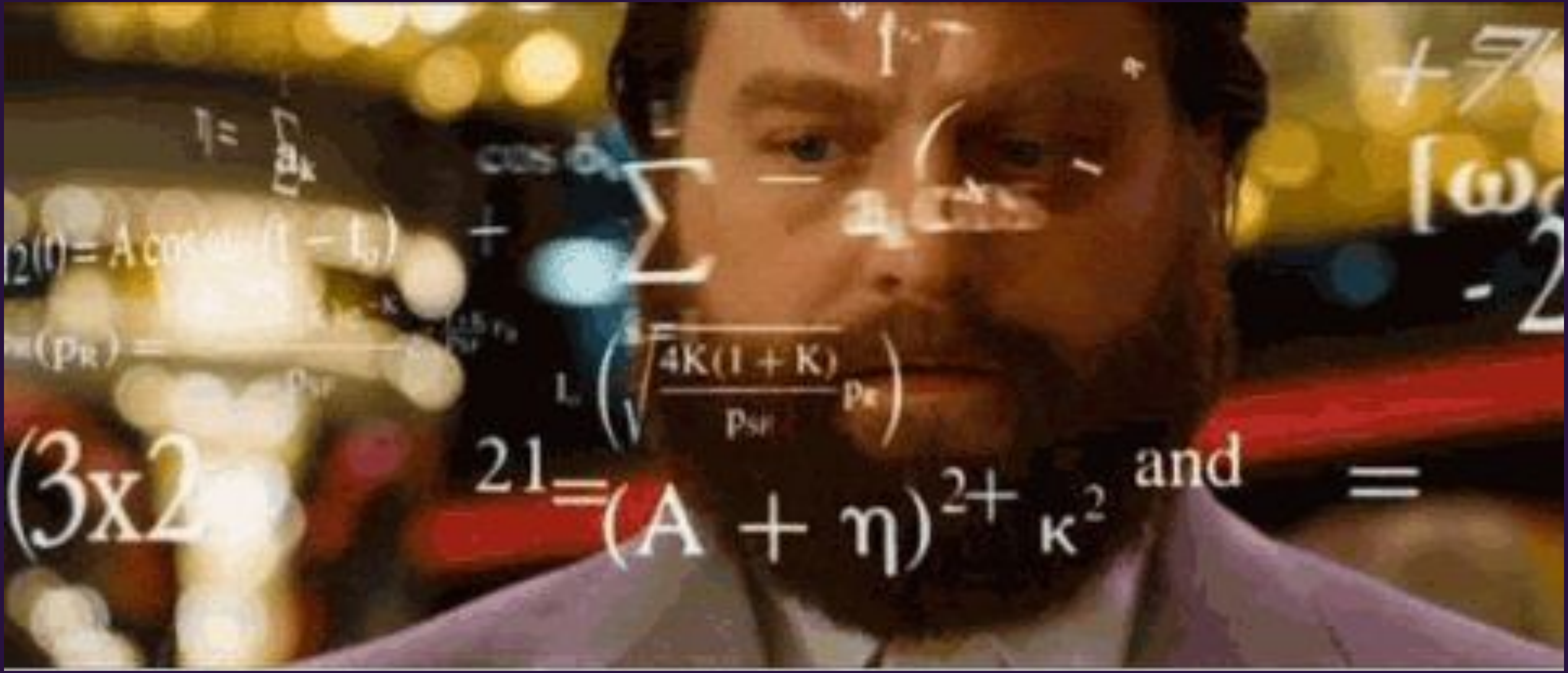
Cross-Consensus Messaging (XCM)

Explore how bespoke blockchains and smart contracts can exchange messages in the Polkadot network and beyond, and understand the associated security guarantees.

@HummusOnRails

Polkadot

My months leading up to the Academy...



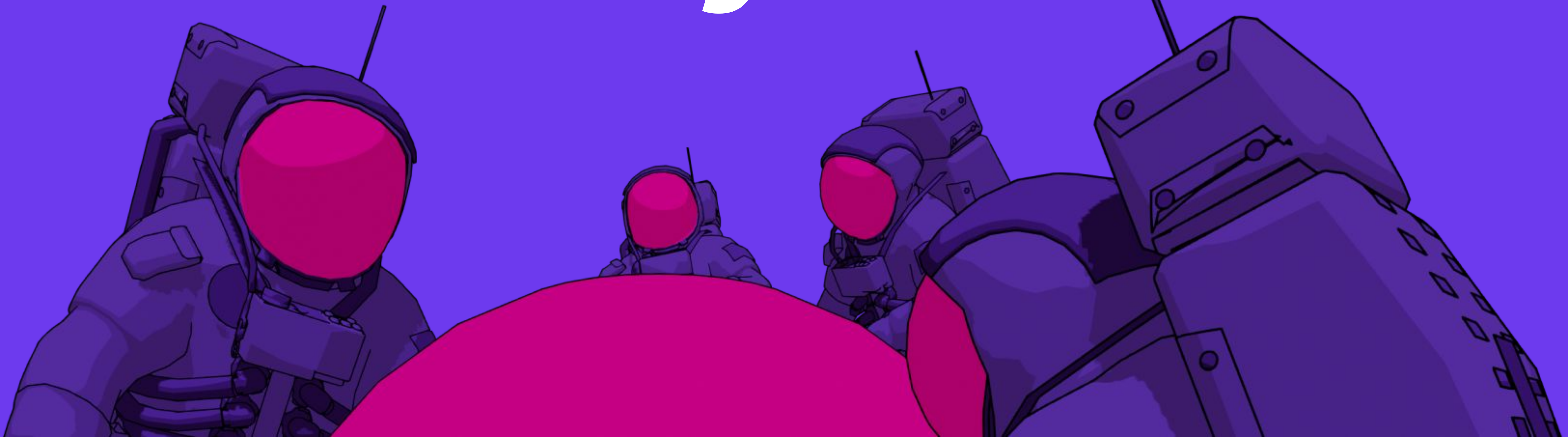
@HummusOnRails

Polkadot

@HummusOnRails

Polkadot

How will you start?



Polkadot

1

Getting Started

2

Rust Syntax

3

Ownership

4

Generics

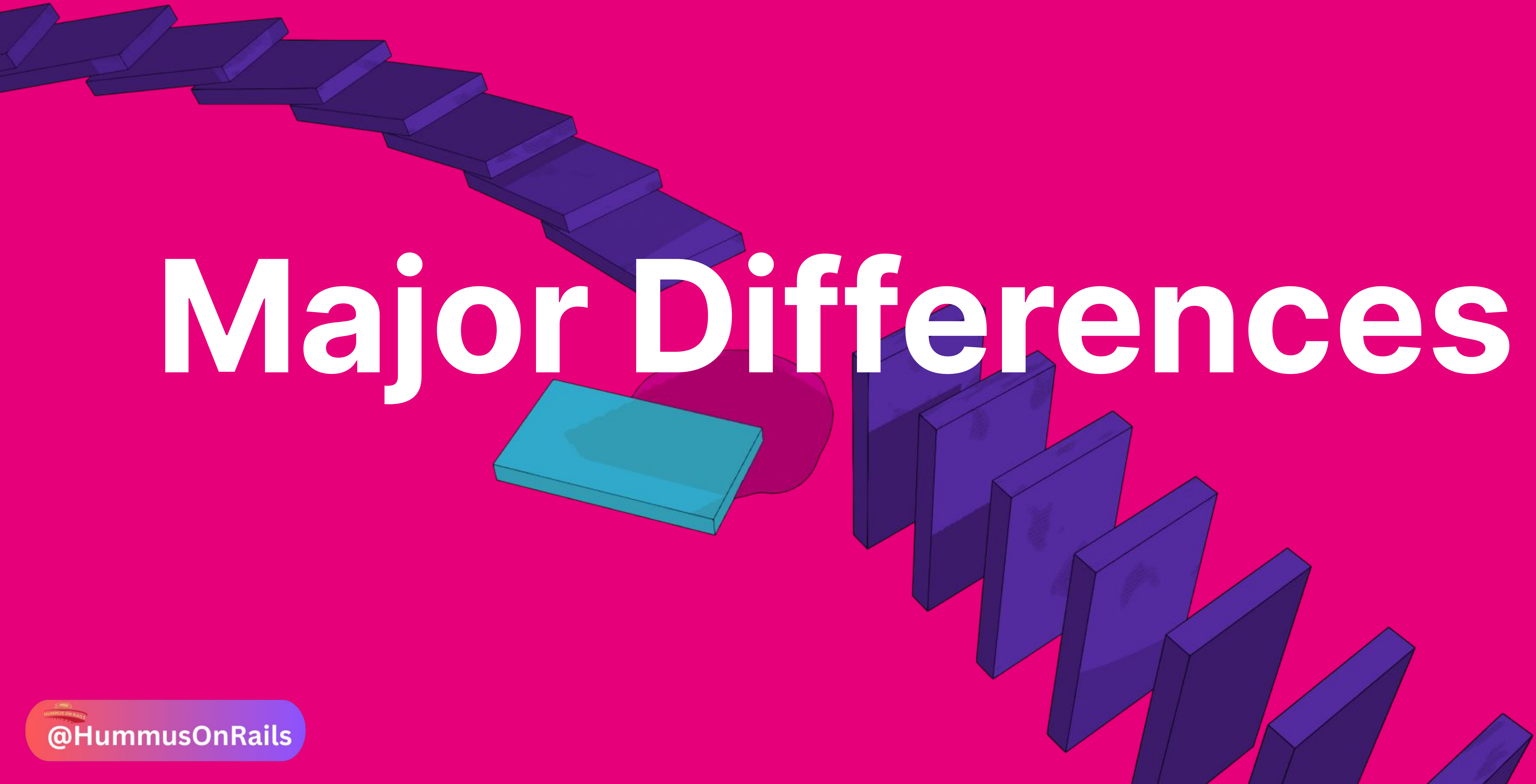
@HummusOnRails

Polkadot

Getting Started

Polkadot

Major Differences



@HummusOnRails

Polkadot

JavaScript

- Interpreted
- Dynamic typing
- Garbage collected

Python

- Interpreted
- Dynamic typing
- Garbage collected

Ruby

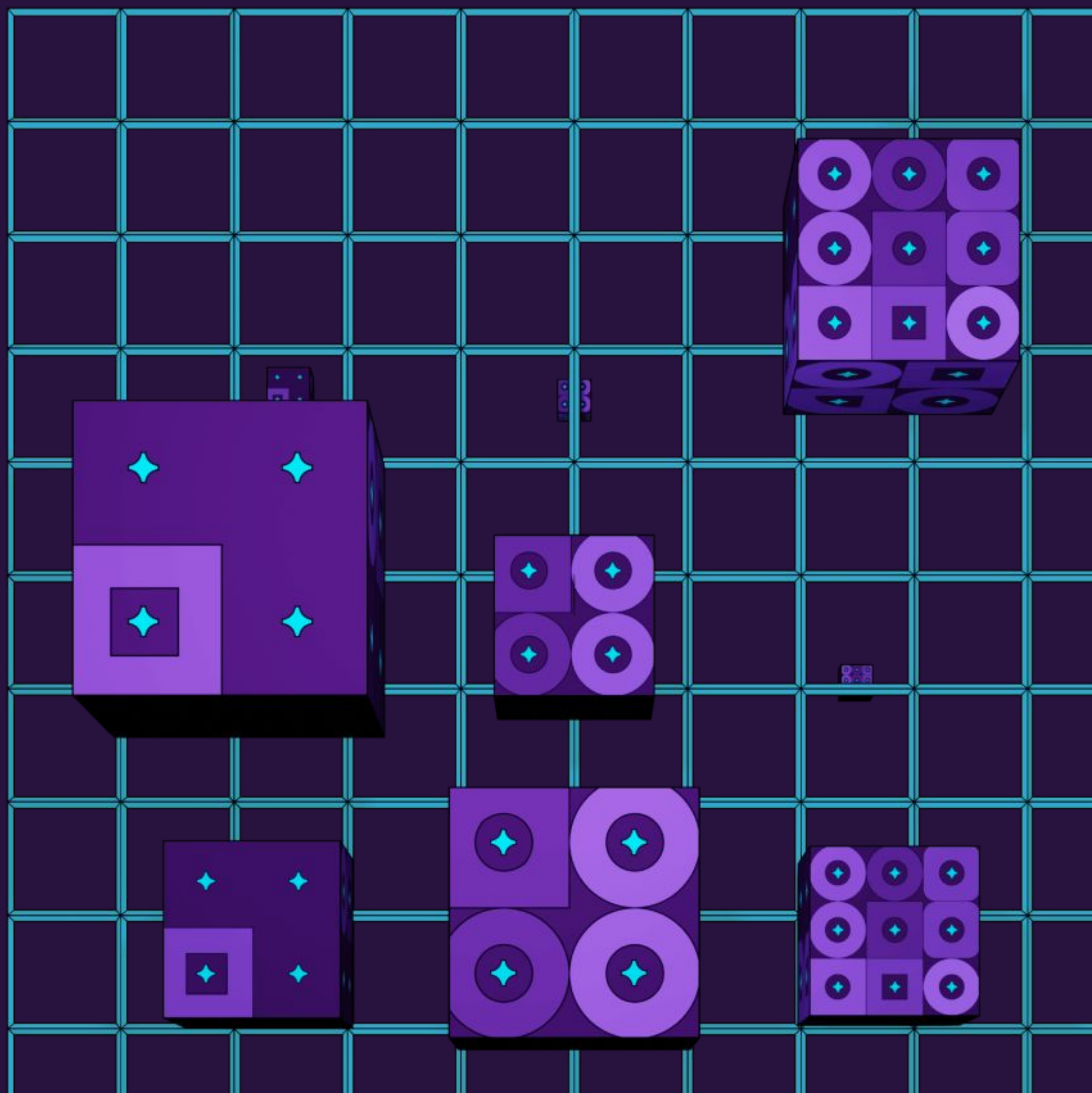
- Interpreted
- Dynamic typing
- Garbage collected

@HummusOnRails

Polkadot

Rust

- Compiled
- Static typing
- Manual memory management



@HummusOnRails

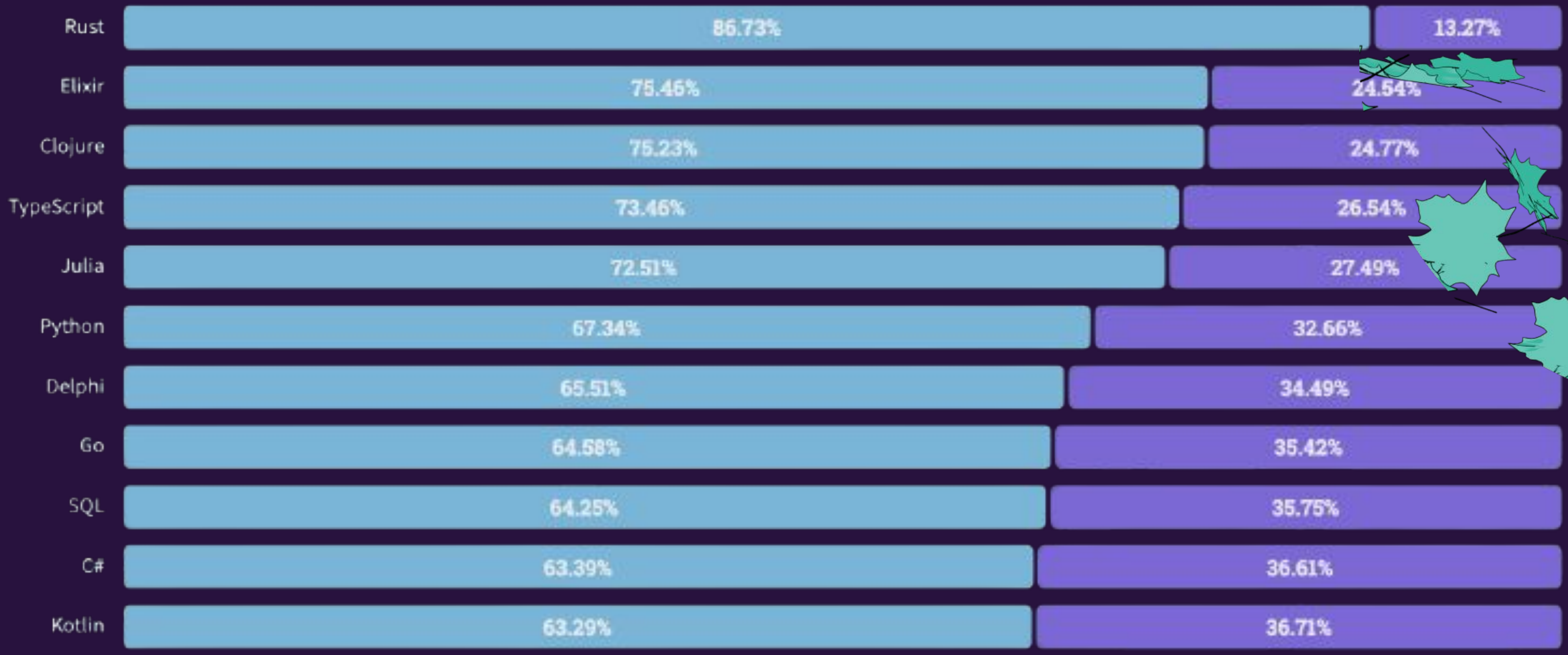
Polkadot

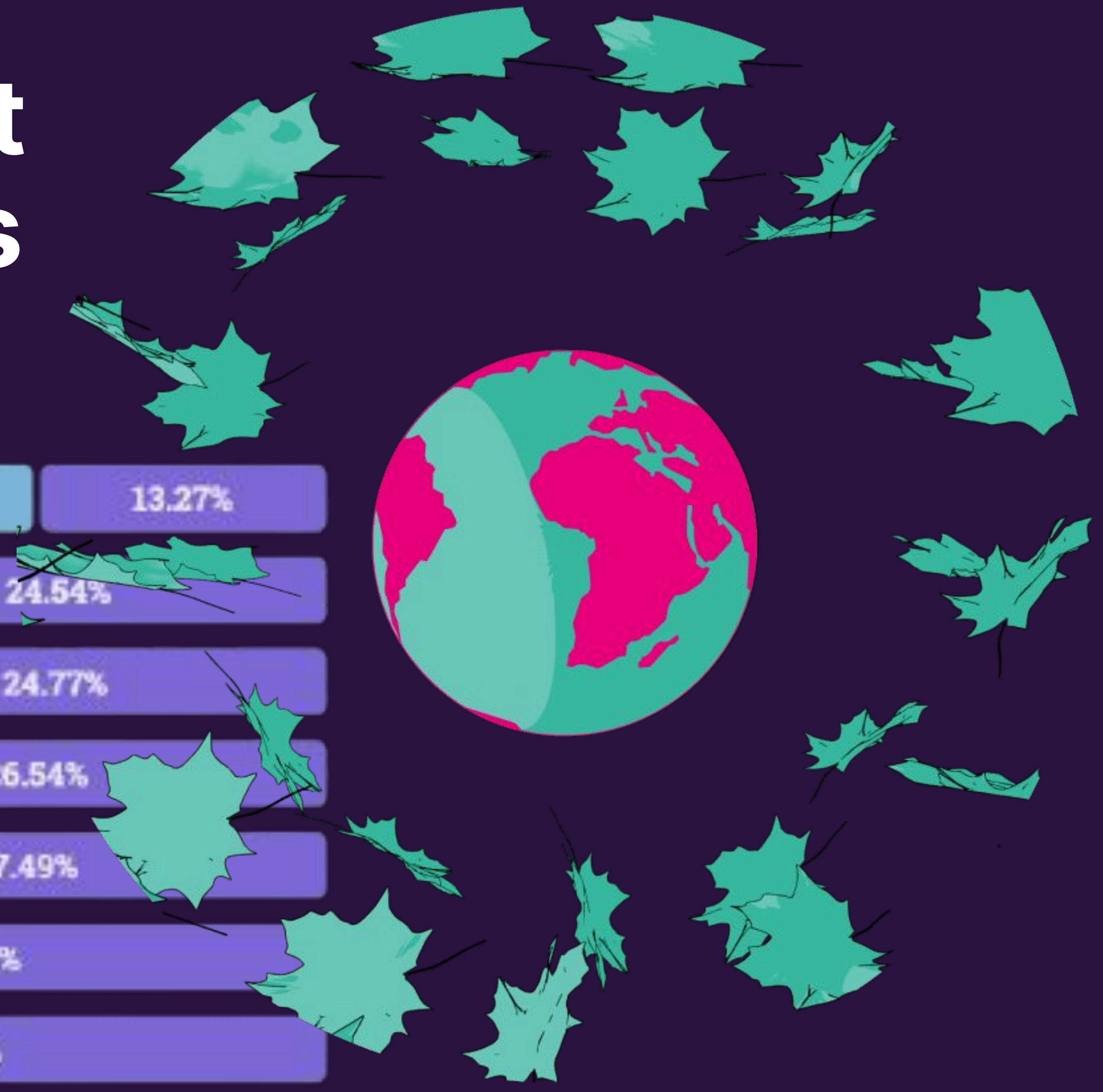
Developer Ecosystem

@HummusOnRails

Polkadot

"Rust is on its seventh year as the most loved language with 87% of developers saying they want to continue using it."





@HummusOnRails

Polkadot

Growing ecosystem of libraries and tools

To name a few...

Web development: Actix, Rocket, Tide

Async programming: Tokio, async-std

Serialization: Serde

Blockchain: Substrate, ink!

Database ORMs: Diesel, sqlx,

Cryptography and security: scale, ring, rustls

GUI development: druid, iced

Game development: Amethyst, Bevy, ggez

@HummusOnRails

Polkadot

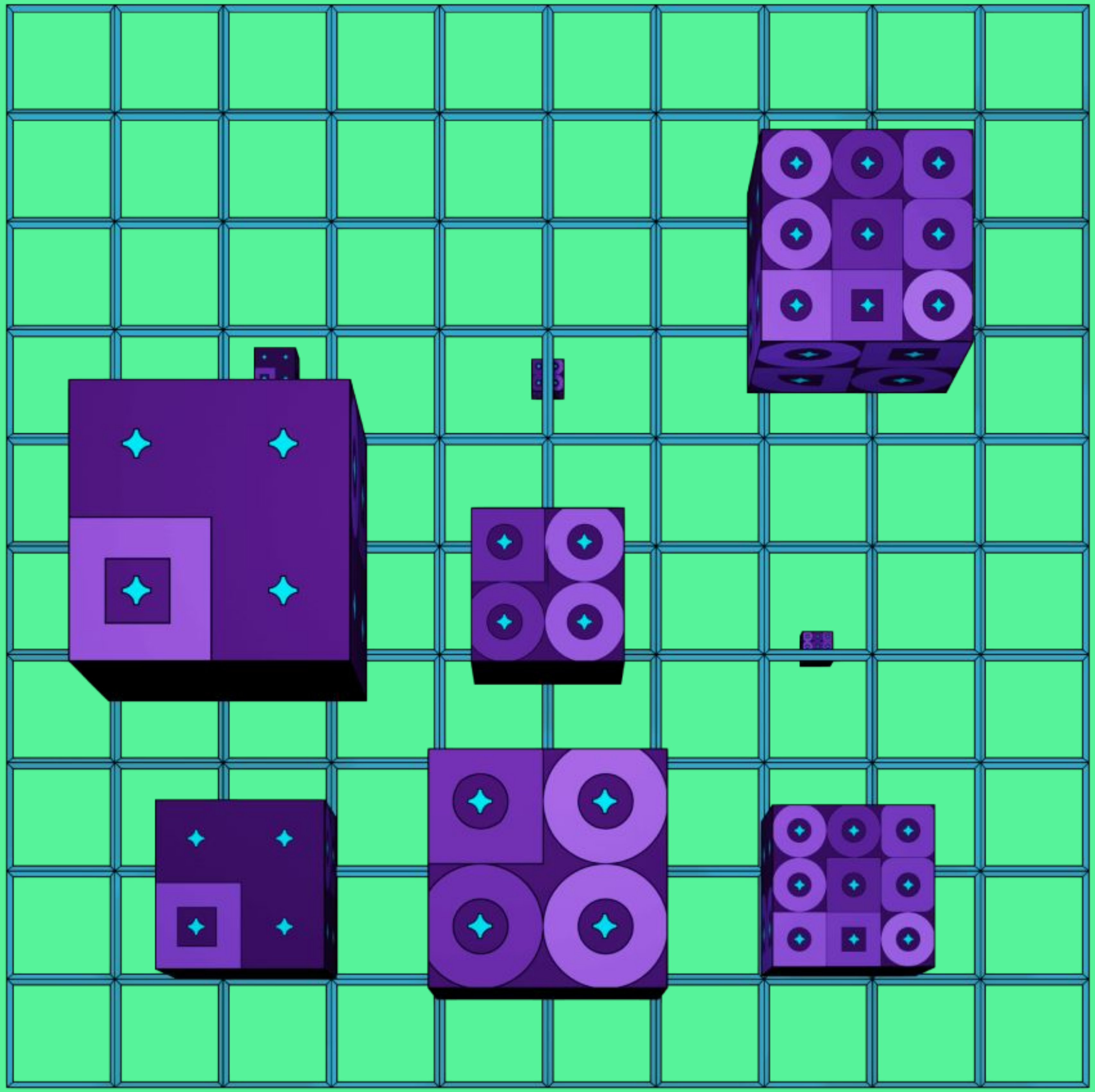
Actively being improved and developed

- Regular release cycle
- Community-driven RFC process
- Focus on ergonomics, performance and stability
- Strong emphasis on backward compatibility

Polkadot

Cargo: Package Manager and build tool

crates.io: Package registry



@HummusOnRails

Polkadot

```
# Install a specific gem
gem install <gem_name>

# Install a specific version of a gem
gem install <gem_name> -v <gem_version>

# Uninstall a gem
gem uninstall <gem_name>

# List all installed gems
gem list

# Update all installed gems to the latest version
gem update

# Update a specific gem to the latest version
gem update <gem_name>

# Search for a gem
gem search <gem_name>

# Display information about a gem
gem info <gem_name>

# Create a new gem
gem new <gem_name>
```

@HummusOnRails

Polkadot

```
# Create a new Rust project
cargo new <project_name>

# Build the Rust project
cargo build

# Build the Rust project in release mode (optimized)
cargo build --release

# Run the Rust project
cargo run

# Check the Rust project for errors without building
cargo check

# Run tests for the Rust project
cargo test

# Update dependencies specified in Cargo.toml
cargo update

# Add a new dependency to Cargo.toml
cargo add <crate_name>

# Remove a dependency from Cargo.toml
cargo rm <crate_name>

# Build and install a Rust binary
cargo install <crate_name>

# List installed Rust binaries
cargo install --list

# Uninstall a Rust binary
cargo uninstall <crate_name>
```

That’s different than Rubygems 🤔

@HummusOnRails

Polkadot

Syntax

@HummusOnRails

Polkadot

A Quick Look

@HummusOnRails

Polkadot

```
def greet(name):
    if name:
        message = "Hello, " + name +
"!" else:
        message = "Hello, stranger!"
    return message

names = ["Alice", "Bob", "", "Eve"]

for name in names:
    print(greet(name))
```

@HummusOnRails

Polkadot

```rust
fn greet(name: &str) -> String {
    let message = if !name.is_empty() {
        format!("Hello, {}!", name)
    } else {
        "Hello, stranger!".to_string()
    };
    message
}

fn main() {
    let names = vec!["Alice", "Bob", "",
"Eve"];
    for name in names {
        println!("{}", greet(name));
    }
}
```

@HummusOnRails

Polkadot

@HummusOnRails

Polkadot

Type System

@HummusOnRails

Polkadot

```ruby
def add(a, b)
  a + b
end

num1 = 5
num2 = 7
puts add(num1, num2)

str1 = "Hello, "
str2 = "World!"
puts add(str1, str2)
```

Polkadot

```rust
fn add(a: i32, b: i32) -> i32 {
    a + b
}

fn main() {
    let num1 = 5;
    let num2 = 7;
    println!("{}", add(num1, num2));

    let str1 = "Hello, ";
    let str2 = "World!";
    println!("{}", add(str1, str2));
}
```

@HummusOnRails

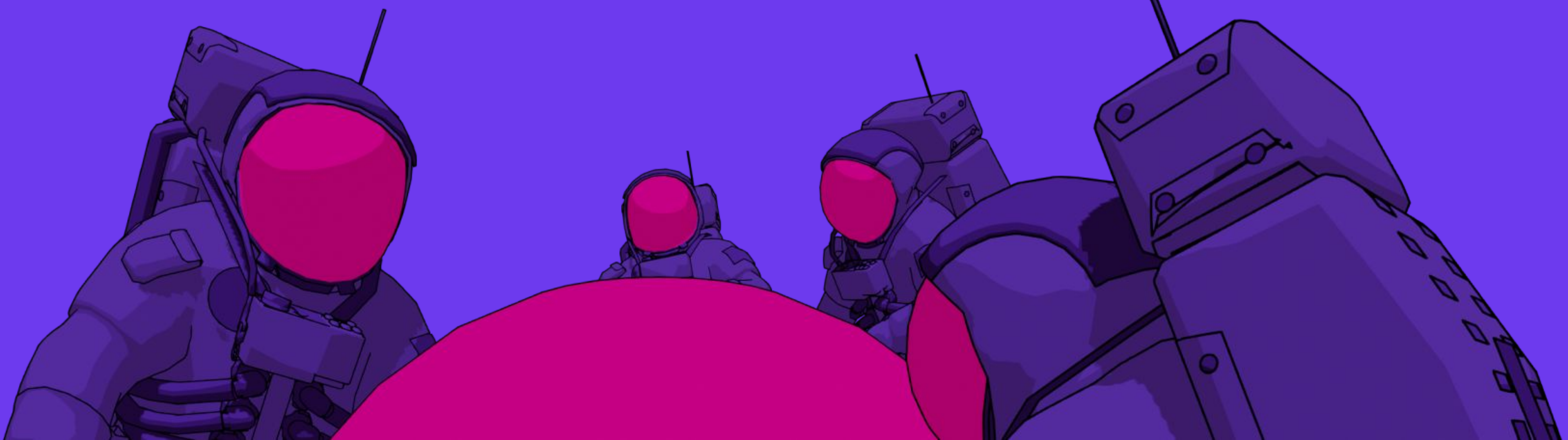
Polkadot

```
error[E0308]: arguments to this function are incorrect
 --> src/main.rs:5:5
  |
5 |     add(string1, string2)
  |     ^^^ -------  ------- expected `i32`, found `&str`
  |         |
  |         expected `i32`, found `&str`
  |
note: function defined here
 --> src/main.rs:9:4
  |
9 | fn add(a: i32, b: i32) -> i32 {
  |    ^^^ ------  ------

error[E0308]: mismatched types
 --> src/main.rs:5:5
  |
1 | fn main() {
  |           - expected `()` because of default return type
...
5 |     add(string1, string2)
  |     ^^^^^^^^^^^^^^^^^^^^^- help: consider using a semicolon here:
  |     |
  |     expected `()`, found `i32`
```

@HummusOnRails

Polkadot

Memory and Ownership



@HummusOnRails

Polkadot

Memory Management

@HummusOnRails

Polkadot

```javascript
function createLargeArray() {
  const largeArray = new Array(1e7).fill(42);
  return largeArray;
}

function main() {
  for (let i = 0; i < 5; i++) {
    const largeArray = createLargeArray();
    console.log(`Iteration ${i + 1}: Created large array`);
  }
}

main();
```

@HummusOnRails

Polkadot

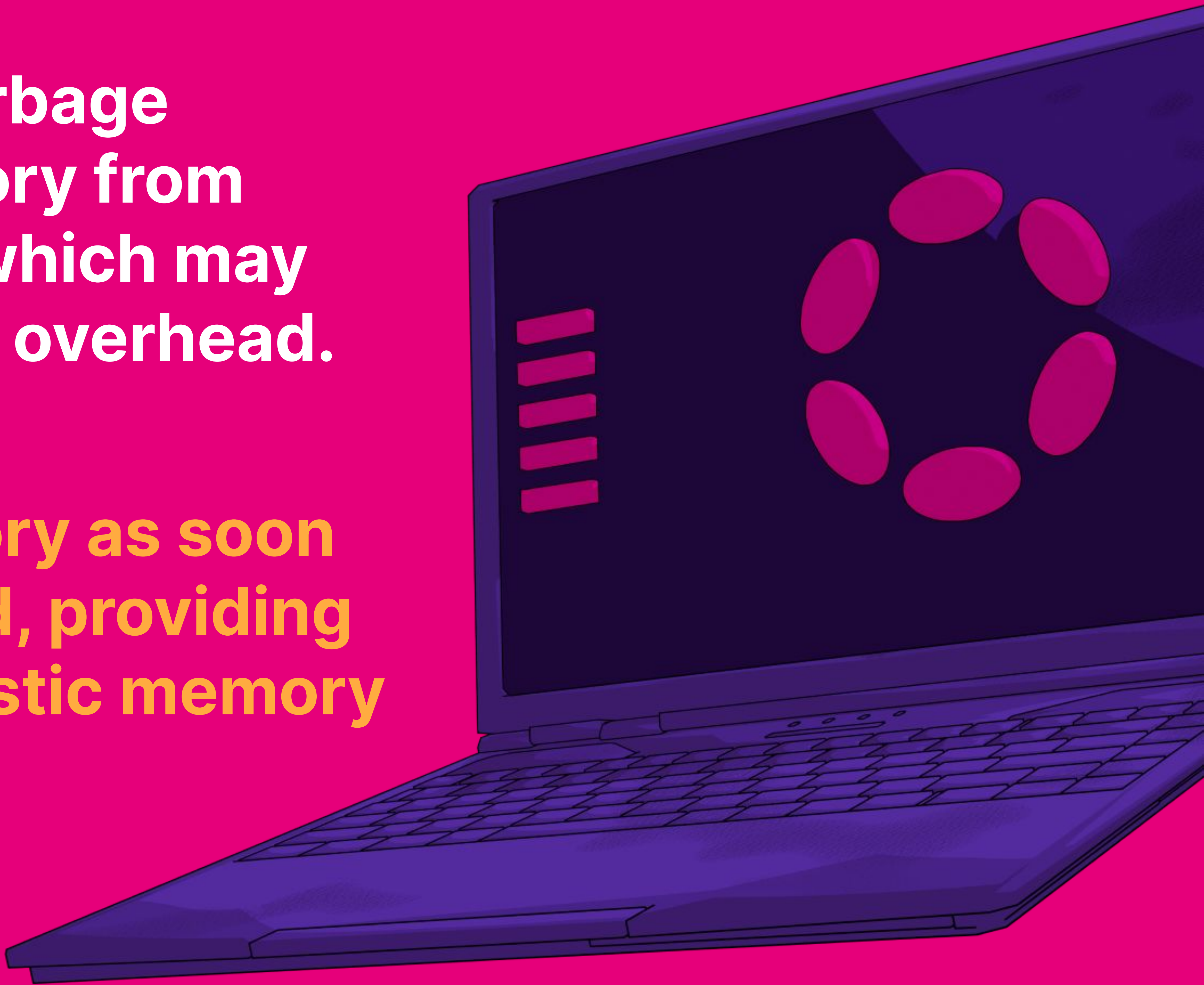
```rust
fn create_large_vec() → Vec<i32> {
    let large_vec = vec![42; 1_000_000];
    large_vec
}

fn main() {
    for i in 0..5 {
        let large_vec = create_large_vec();
        println!("Iteration {}: Created large vec",
i + 1);
    }
}
```

@HummusOnRails

Polkadot

JavaScript relies on garbage collection to free memory from previous large arrays, which may introduce performance overhead.

Rust deallocates memory as soon as it's no longer needed, providing efficient and deterministic memory management.



@HummusOnRails

Polkadot

Ownership

@HummusOnRails

Polkadot

```python
def modify_list(input_list):
    input_list.append("Python")

def main():
    original_list = ["Hello", "from"]
    print("Before:", original_list)

    modify_list(original_list)
    print("After:", original_list)

main()
```

@HummusOnRails

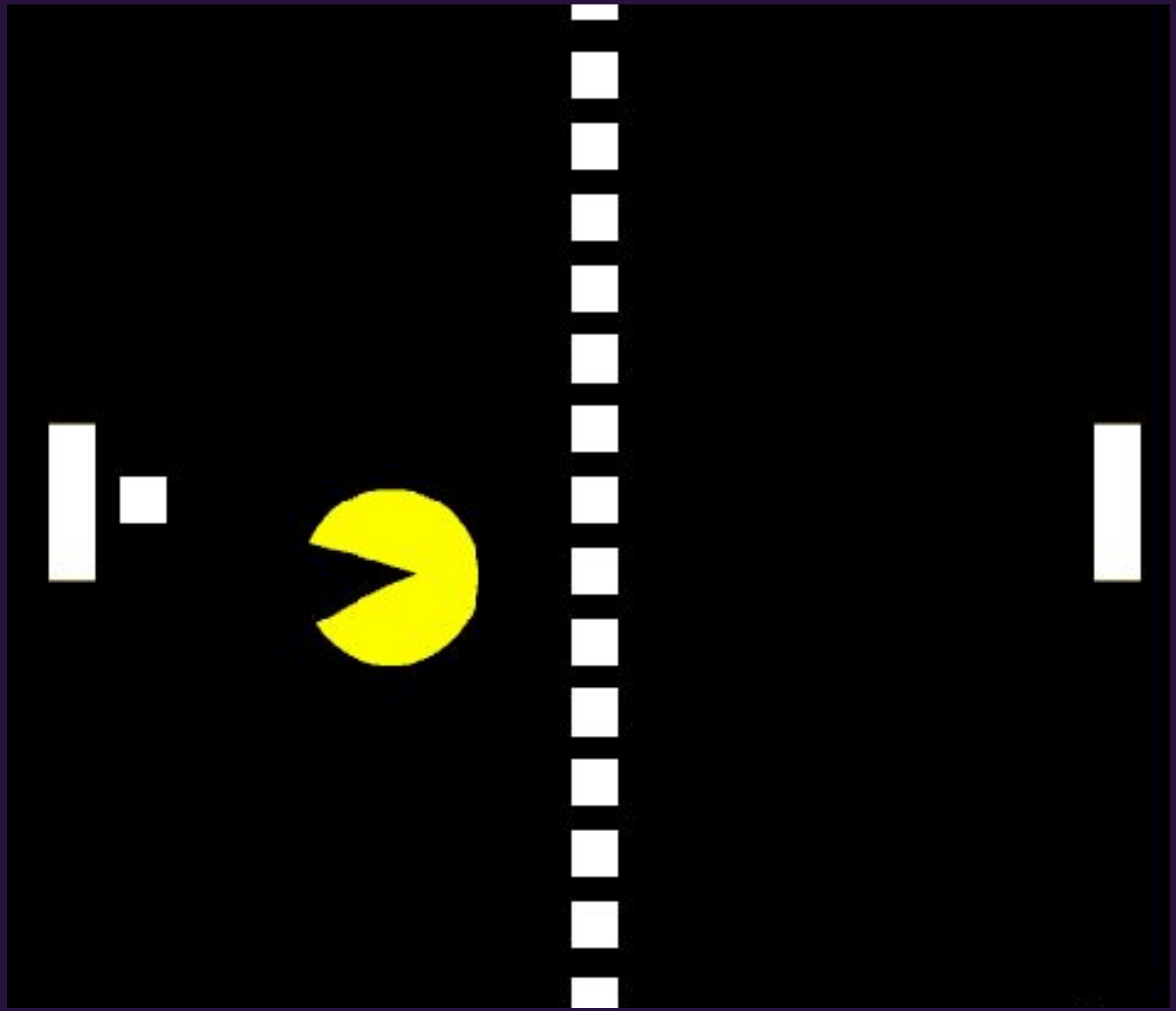
Polkadot

```rust
fn modify_vec(mut input_vec: Vec<&str>) -> Vec<&str> {
    input_vec.push("Rust");
    input_vec
}

fn main() {
    let mut original_vec = vec!["Hello", "from"];
    println!("Before: {:?}", original_vec);

    original_vec = modify_vec(original_vec);
    println!("After: {:?}", original_vec);
}
```

```rust
fn consume_and_return(input: String) -> String {
    println!("Inside function: {}", input);
    input
}

fn main() {
    let original_string = String::from("Hello, Rust!");
    println!("Before: {}", original_string);

    let returned_string = consume_and_return(original_string);
    // The line below will cause a compilation error if uncommented
    // println!("After: {}", original_string);

    println!("Returned: {}", returned_string);
}
```

@HummusOnRails

Polkadot

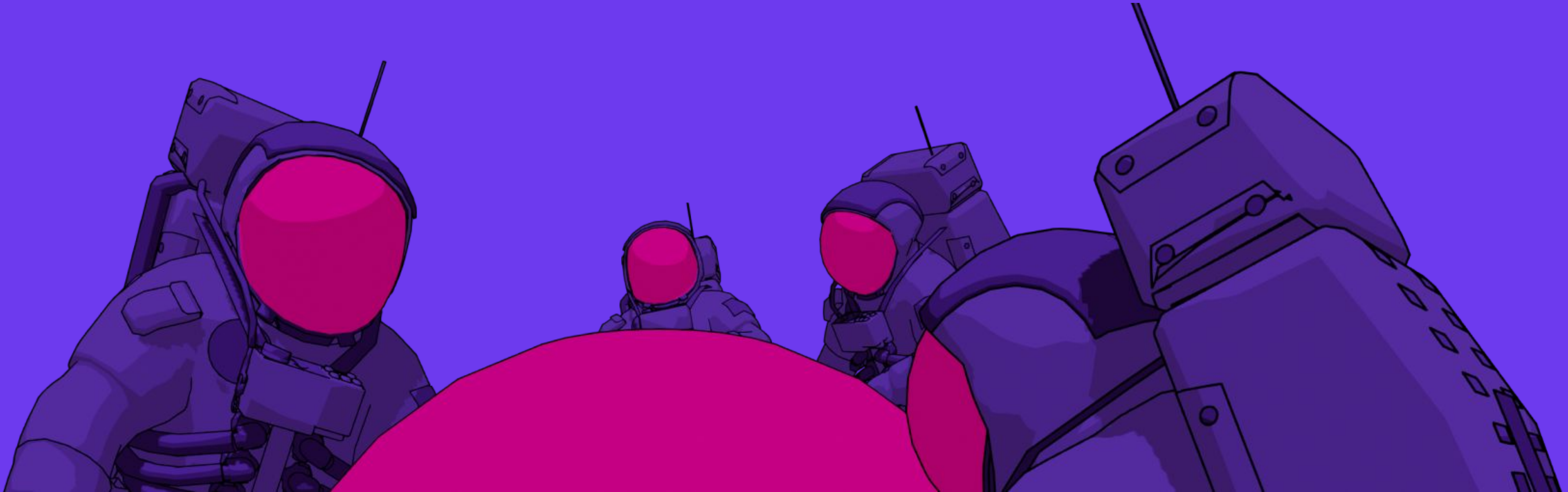


Once Pac-man (i.e. *Rust*) consumes (i.e. *owns*) the ball, it's no longer available

@HummusOnRails

Polkadot

Generics



Polkadot

What are generics?

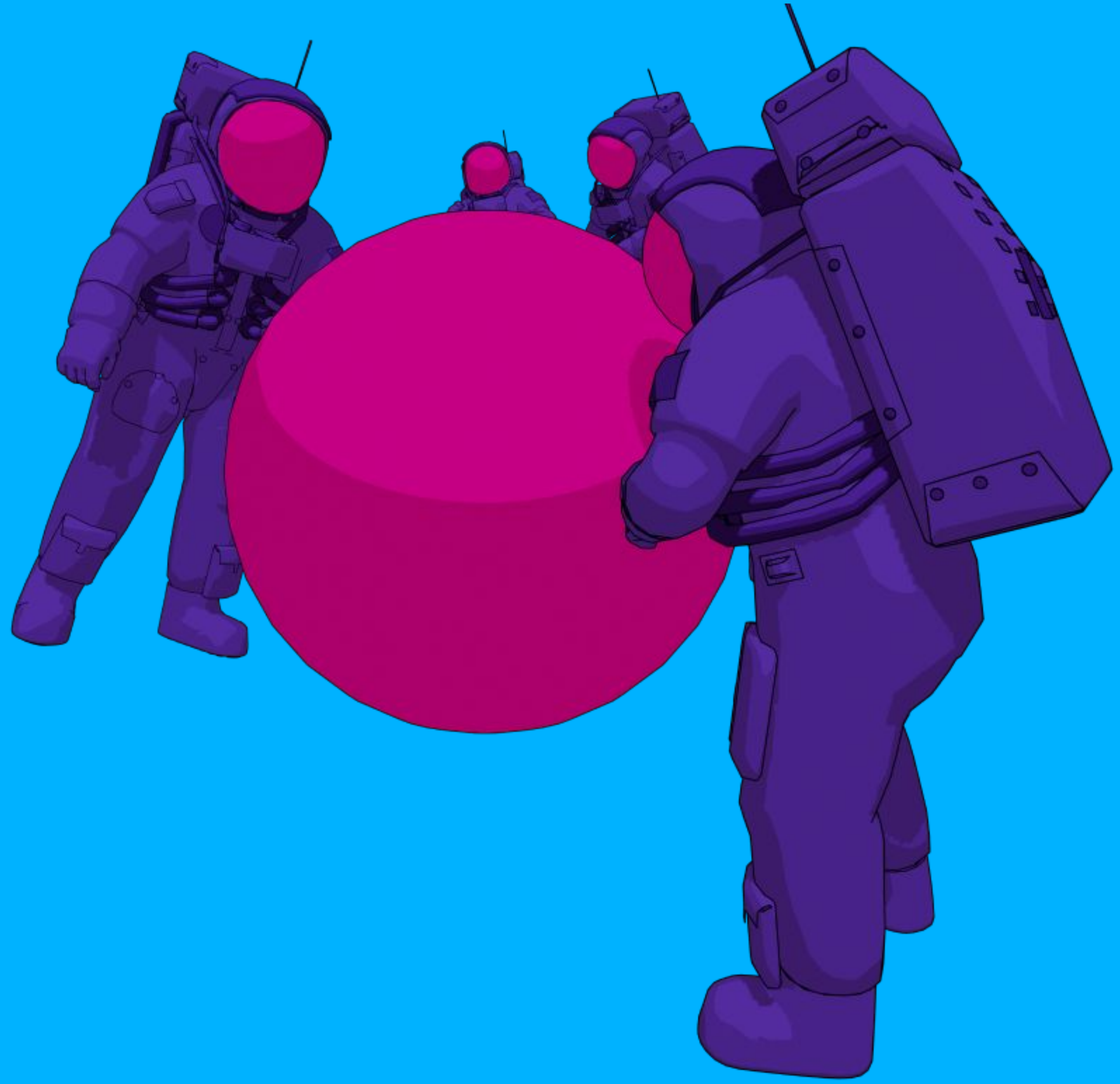
@HummusOnRails

Polkadot

Generics allow writing flexible, reusable, and type-safe code without specifying concrete data types

@HummusOnRails

Create functions, structs, enums, and traits that work with multiple data types



@HummusOnRails

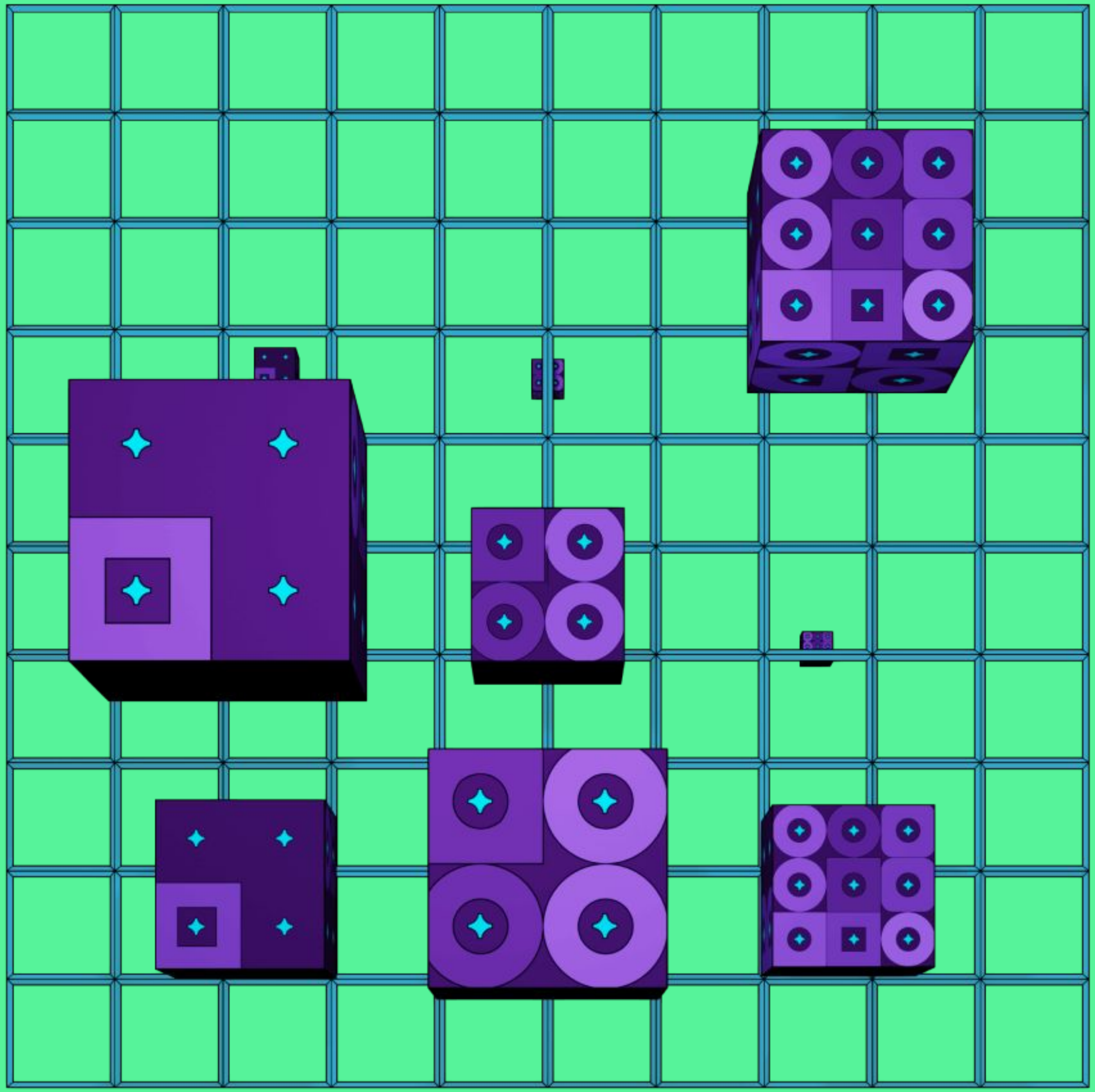
Polkadot

Why use generics?

@HummusOnRails

Polkadot

Generics provide a way to achieve flexibility similar to dynamically typed languages while maintaining type safety benefits of static typing



@HummusOnRails

Polkadot

Generics Syntax

Polkadot

```rust
# Generics function example

fn print_elements<T>(elements: &[T]) {
    for element in elements {
        println!("{:?}", element);
    }
}

fn main() {
    let numbers = vec![1, 2, 3];
    let words = vec!["Hello", "Rust", "Generics"];

    print_elements(&numbers);
    print_elements(&words);
}
```

@HummusOnRails

```rust
// Generics struct example

struct Pair<T> {
    first: T,
    second: T,
}

fn main() {
    let integer_pair = Pair { first: 1, second: 2 };
    let string_pair = Pair { first: "Hello", second: "World" };

    println!("Int                              pair.first, integer_pair.second);
    println!("Strin                            ir.first, string_pair.second);
}
```

It’s Reusable!

Polkadot

@HummusOnRails

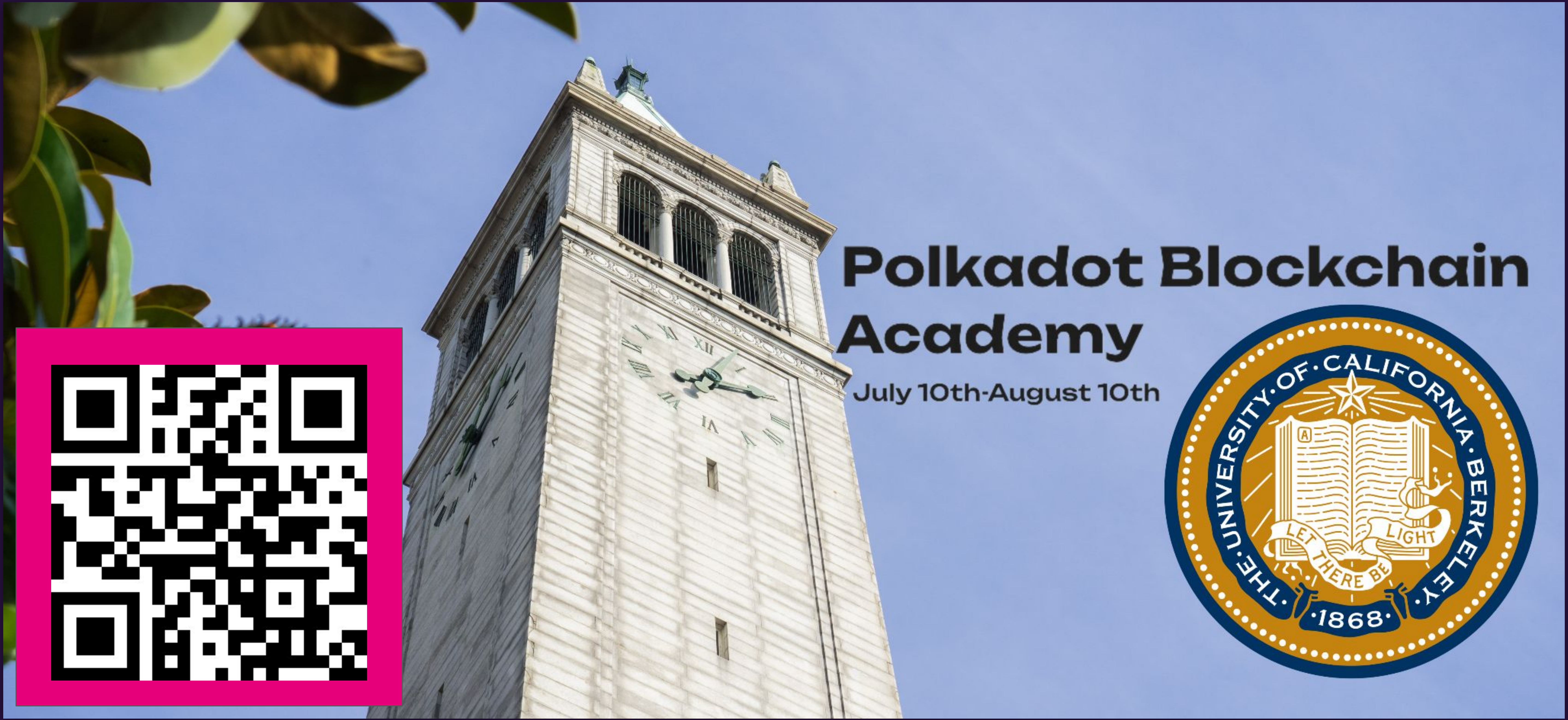
Polkadot

substrate_

```
pub struct Module<T: Config<I>, I: Instance = DefaultInstance>(PhantomData<(T, I)>);

impl<T: Config<I>, I: Instance> Module<T, I> {
    // ...
}

pub trait Config<I: Instance = DefaultInstance>: frame_system::Config {
    type Event: From<Event<Self, I>> + Into<<Self as frame_system::Config>::Event>;
    // ...
}

decl_module! {
    pub struct Module<T: Config<I>, I: Instance = DefaultInstance> for enum Call where origin: T::Origin
{
        // ...
    }
}
```

@HummusOnRails

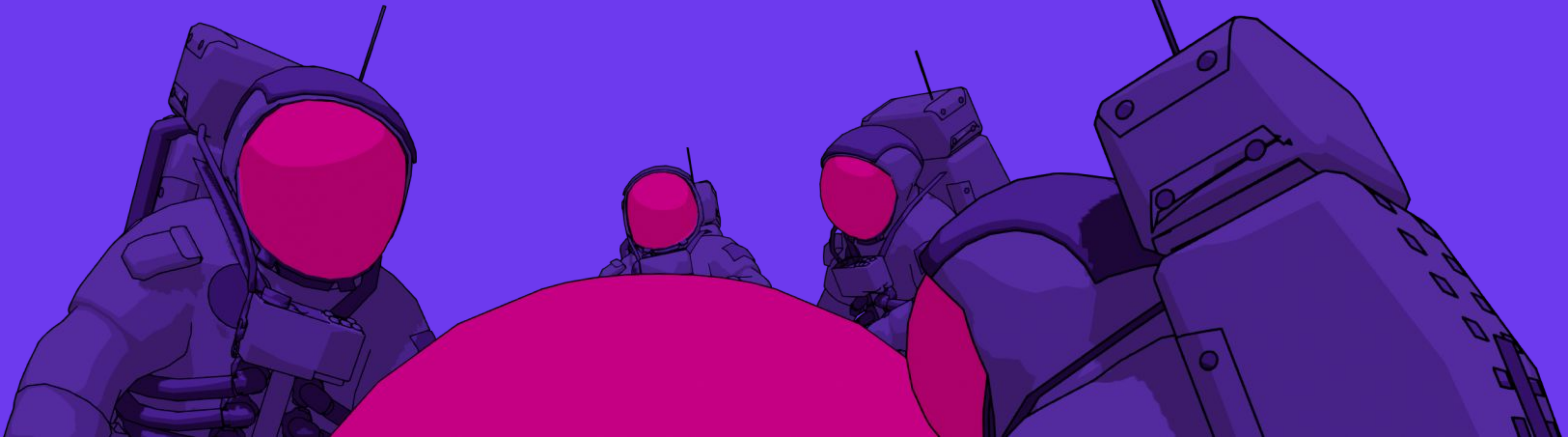
Polkadot



@HummusOnRails

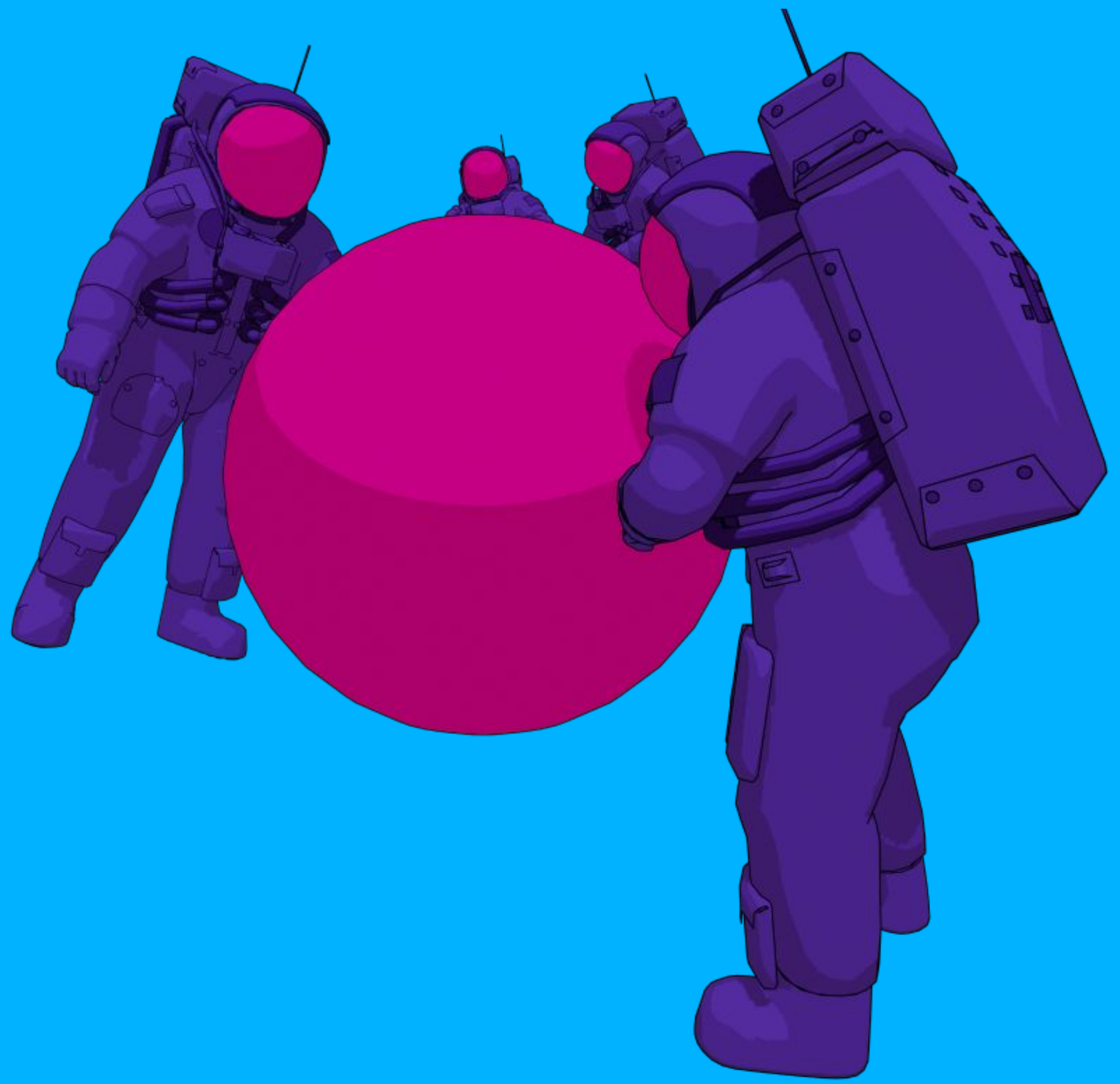
Polkadot

Continue Your Learning



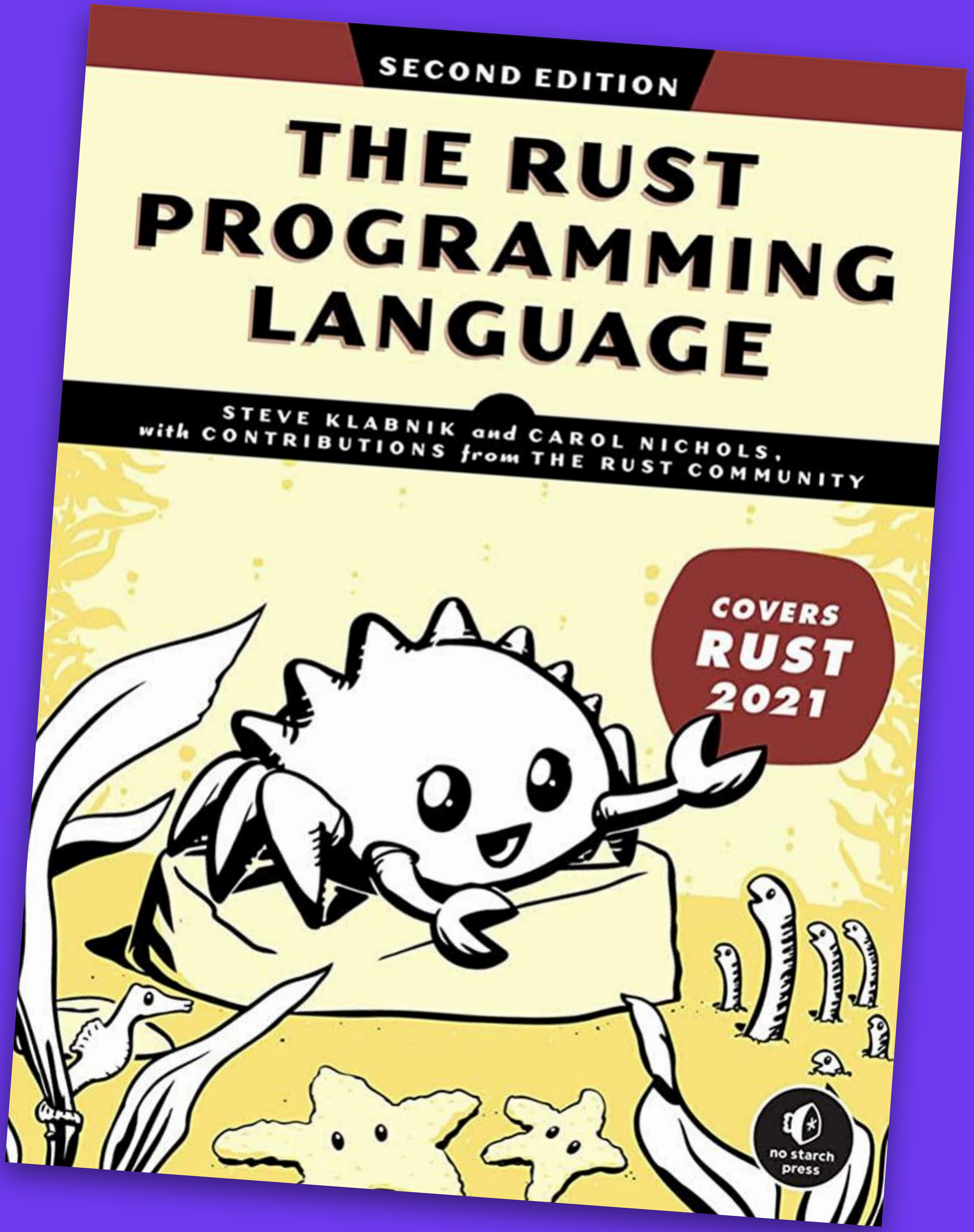
Polkadot

Maybe you heard about the Rust book before...



@HummusOnRails

Polkadot



@HummusOnRails

Polkadot

Have you heard about the Brown University version?

@HummusOnRails

Polkadot

Experiment: Improving the Rust Book

What is this?

This website is an experiment by Brown University researchers Will Crichton and Shriram Krishnamurthi. The goal of this experiment is to evaluate and improve the content of the Rust Book to help people learn Rust more effectively.

BROWN

How does it work?

This website has the same structure as the Rust Book, but modified in two ways:

1. **Interactive quizzes are added in each section.** These quizzes help you test your understanding of Rust. The quizzes also help us determine which sections need improvement.
2. **Some explanations will be changed.** For instance, we will experiment with modifying some of the text, including replacing it with visualizations.

Quiz Question 2 / 3

Question 2

If `x : u8 = 0`, what will happen when computing `x - 1`?

Response

- It will always panic.
- It will always return 255.
- It depends on the compiler mode.

Submit

https://rust-book.cs.brown.edu/

@HummusOnRails

Polkadot

Rustlings

github.com/rust-lang/rustlings

Polkadot

web3
foundation

Intro to Rust Course

@HummusOnRails

Polkadot

Module 1 - Why Learn Rust?

- What is Rust?
- Why Rust is the future
- WASM TLDR & its relation to Rust

Module 2 - Rust 101 - The Basics

- Variables & Mutability
- Data Types
- Functions & Comments
- Loops & Logic Flows

Module 3 - Intro to Intermediate Rust: Ownership, Borrowing, & Slices

- Rust's Ownership Model
- Rust's Borrowing Model
- Slices in Rust

Module 4 - Intro to Intermediate Rust: Enums & Matching Patterns

- Enums
- Panic! in Rust
- Error handling with Result & Option

Module 5 - Intro to Intermediate Rust - Data Structs & Collections

- Structs
- Defining Methods for Structs
- Vectors, Strings & Hashmaps
- Vectors vs Strings - what's the difference?

Module 6 - Intro to Advanced Rust - Traits, Generics, & Lifetimes

- Defining behavior with Traits
- Reducing Duplication with Generics
- Associated Types vs Generics
- Lifetimes in Rust

Module 7 - Intro to Advanced Rust - Iterators & Closures

- Using Closures for Ultimate Code Reuse
- Using Iterators with Vectors
- Loops or Iterators?

Module 8 - Learning Cargo, Rust's Package Management System & Unit Testing

- Defining & Reading cargo.toml
- Installing a crate
- Defining features for a crate
- Unit Tests in Rust

Variable Scope

Let's run through an example to demonstrate the importance of scope and how it relates to ownership:

```
// You can define a new scope within main using this syntax.
// Everything within this scope is valid until the closing curly bracket.
// For example, you cannot use `x` outside of this scope - as the variable is dropped from memory

{
    // Declare an integer x
    let x = 10;
    // We can do whatever we want in the context of this scope,
    // but after it ends, we cannot use x.
}
// We cannot use `x` here!
```

In terms of ownership, the *variable* `x` is owned by this scope, defined by curly brackets, until the end of the scope. From here, the Rust compiler handles the de-allocation of memory in a safe manner.

Ownership gets more complex when dealing with values not defined at compile-time. For more information on the complexities of ownership, read the Rust book's explanation.

Try it out!

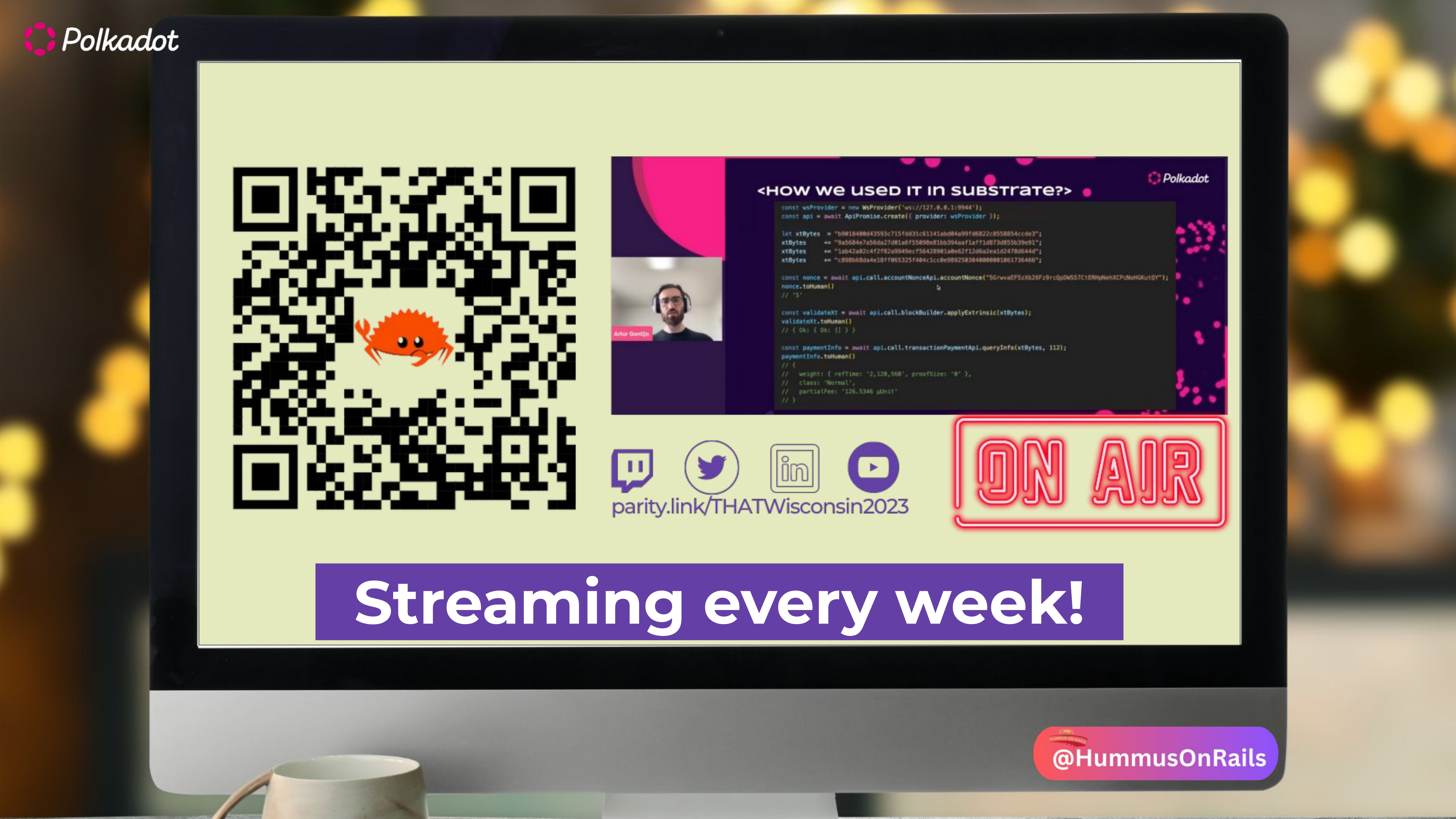
RUN ··· DEBUG STABLE ··· SHARE TOOLS CONFIG ?

```
fn main() {
    let y = 11;
     // You can define a new scope within main using this syntax.
     // Everything within this scope is valid until the closing curly bracket.
     // For example, you cannot use `x` outside of this scope - as the variable is dropped from memor
     {
         // Declare an integer x
         let x = 10;
         // We can do whatever we want in the context of this scope,
         // but after it ends, we cannot use x.
         println!("x: {x}, y: {y}");
     }
     // We cannot use `x` here! It was owned by the inner scope defined previously.
     // println!("{x}"); - this wouldn't work.
     // Printing `y` is perfectly fine, as it it's ownership and scope is above that
     // of x's.
     println!("only y: {y}");
}
```

What's happening here?

We define two variables here: `x` and `y`. Both are fixed size, and known at compile time. The difference between the two is that `y` is owned by the scope of the `main` function, while `x` is owned by another inner scope. The code illustrates that `x` is dropped after the inner scope ends, while `y` can still be used until the end of the `main` function's scope.

https://w3f.github.io/w3f-education/docs/introrust


Polkadot
<HOW WE USED IT IN SUBSTRATE?>
Polkadot
Artur Gontijo
parity.link/THATWisconsin2023
ON AIR
Streaming every week!
@HummusOnRails

Polkadot

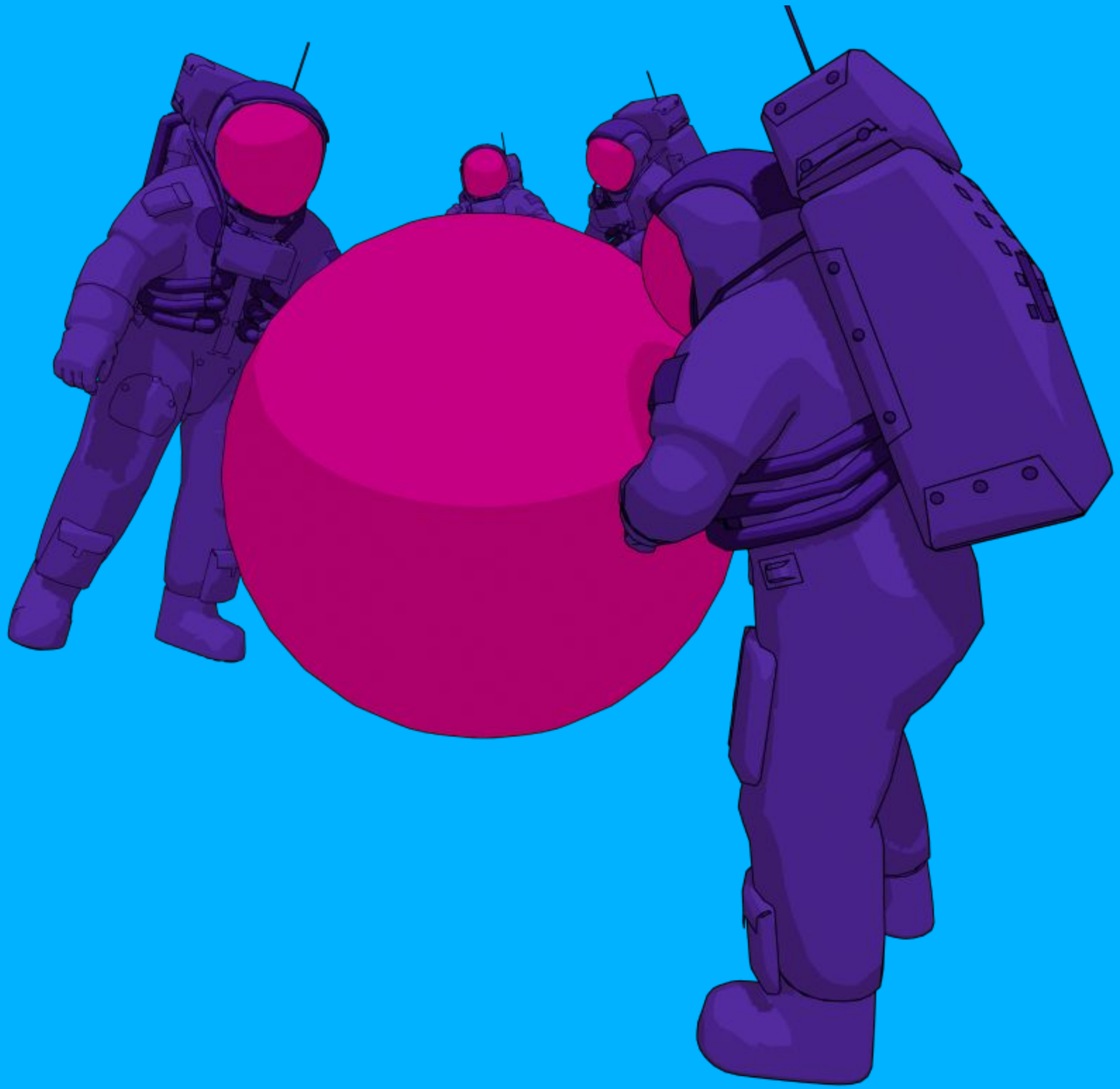
Learning Rust from

isn’t always easy

@HummusOnRails

Polkadot

But, with the right resources and community it is possible!



@HummusOnRails

Polkadot

Let’s keep learning Rust together!

@HummusOnRails

Polkadot

Session Feedback

@HummusOnRails