The Good, The Magical and The Insane:

Scala 3 design patterns

VIRTUS LAB

About me

Łukasz Biały

Scala Developer Advocate
@ VirtusLab

Scala, FP, Distributed Systems

VIRTUSLAB

About this talk

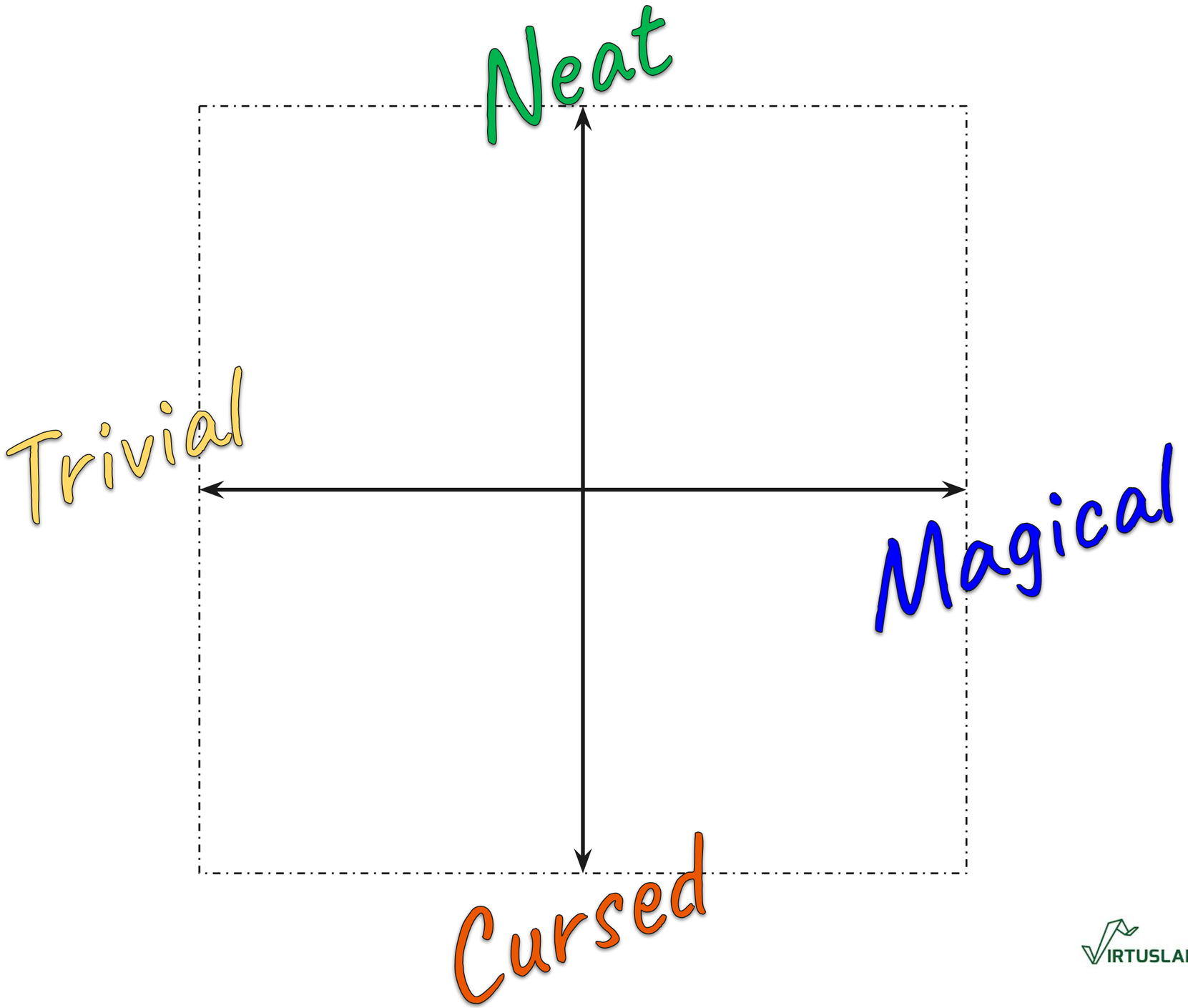
- Scala 3 enables new API design patterns for library builders
- We are building stuff using Scala 3
- We have several *Scala Wizards* in our *Ivory Basement*
- Some of their ideas are *explorative* in nature…

The compass

Two axes, four extremes:

- Neat - useful & helpful, nice DX
- Cursed - weird, not that useful, high WTF/minute factor, bad DX
- Trivial™ - easy to grok, doesn't abuse the language (much)
- Magical - quite contrived, might give headaches

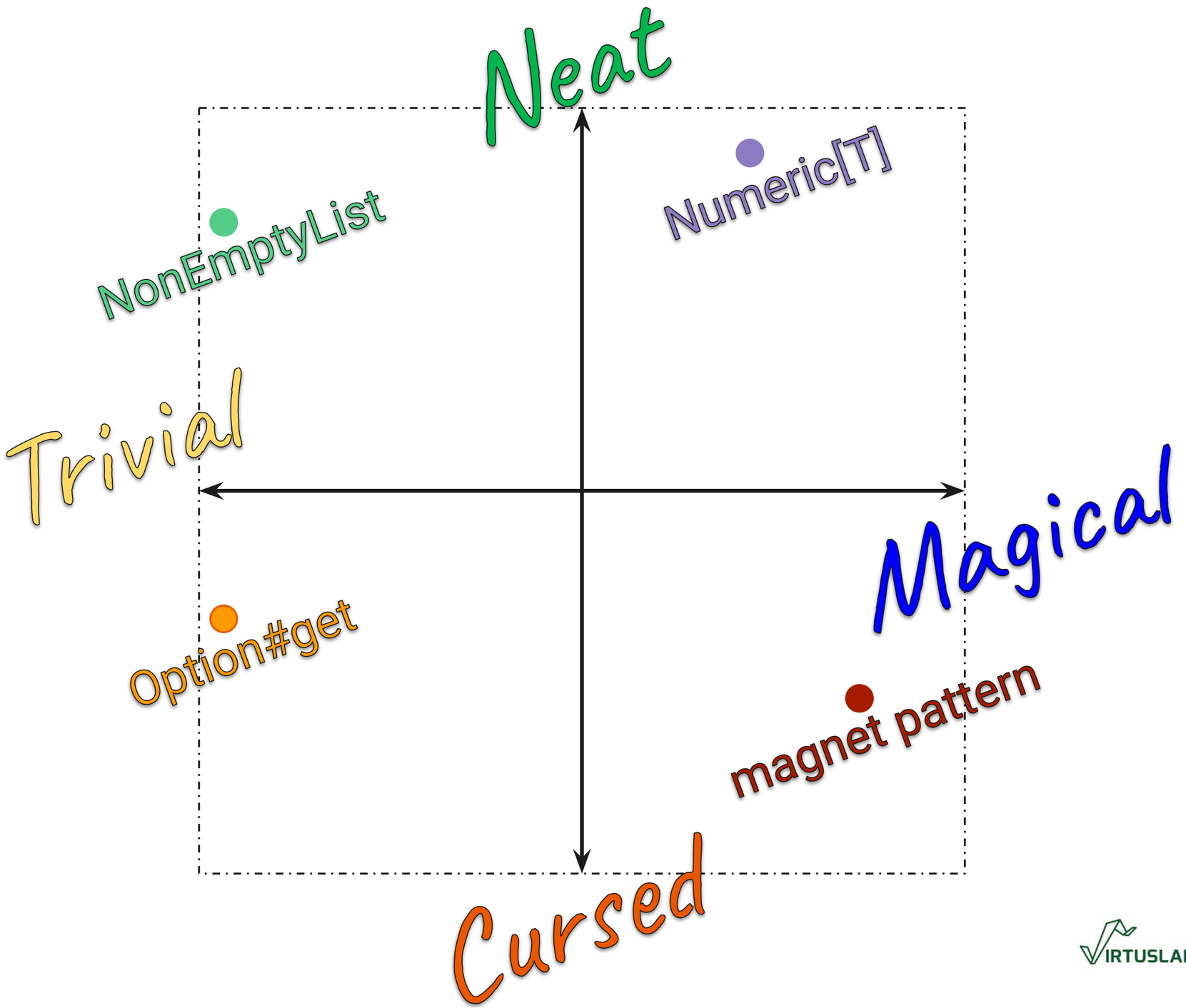
The compass



Examples

- Option#get, Seq#head
  - trivial but quite cursed
- NonEmptyList
  - trivial but quite useful
- Numeric[T] and .max, .min
  - a bit magical but very useful
- magnet pattern with nested implicits
  - very magical, debatable usefulness

The compass



The Good

Custom refined types

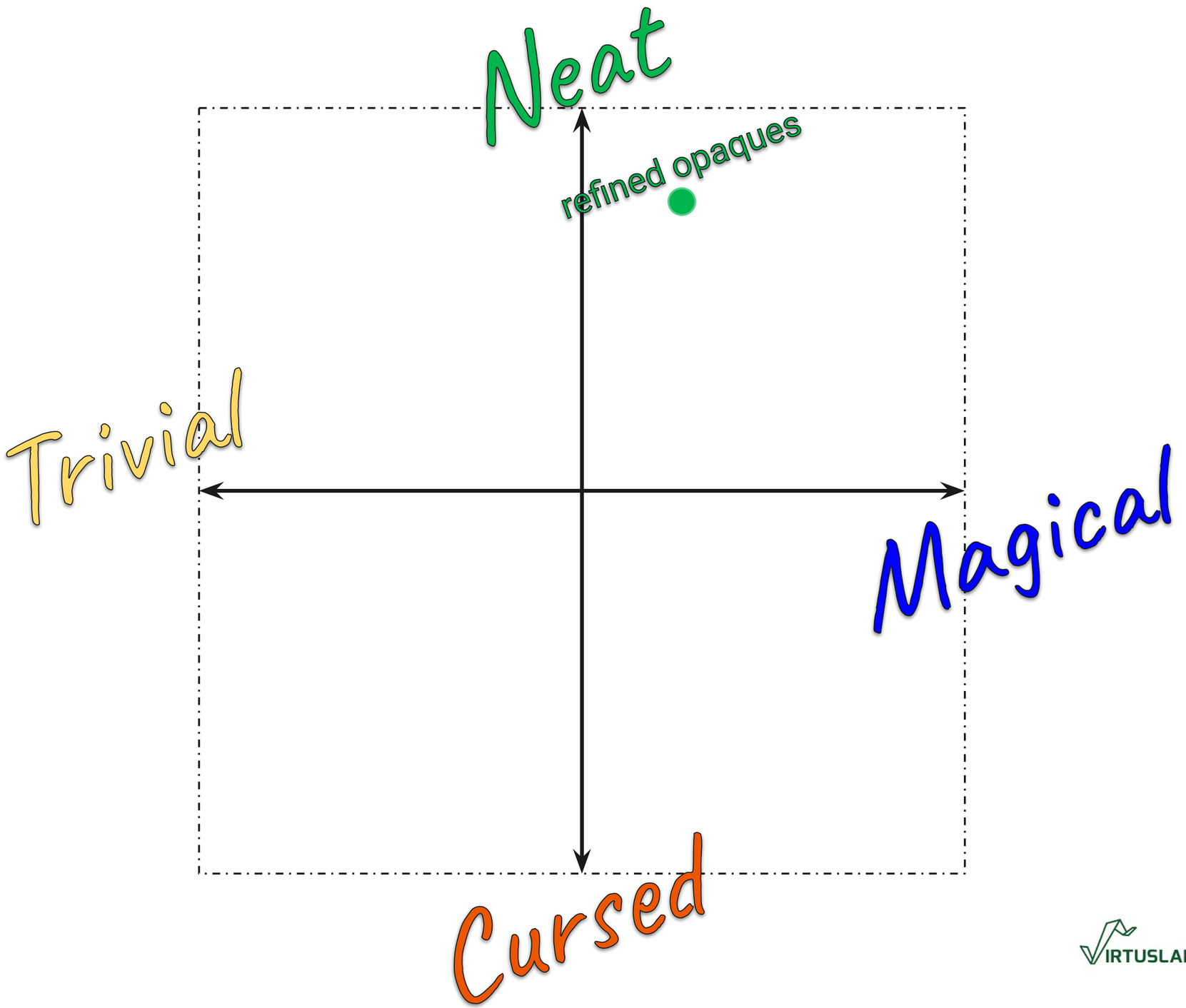
Mission:

You need to constraint values that users of your Scala API can provide but inputs are otherwise quite basic types like String or Int

Custom refined types

```
object types:
  import scala.compiletime.*
  import scala.compiletime.ops.string.*

  opaque type ConstrainedStr <: String = String

  object ConstrainedStr:
    inline def from(s: String): ConstrainedStr =
      requireConst(s)
      inline if !constValue[Matches[s.type, "some-prefix:.+"]] then
        error("this string doesn't have the necessary prefix of `some-prefix:`")
      else s
  end ConstrainedStr

  inline implicit def str2ConstrainedStr(inline s: String): ConstrainedStr =
    ConstrainedStr.from(s)

import types.*

@main def testConstrainedStr(): Unit =
  val ok: ConstrainedStr = "some-prefix:hello"
  val fail: ConstrainedStr = "hello" // does not compile
```

The compass



VIRTUSLAB

Fun with Postel's law

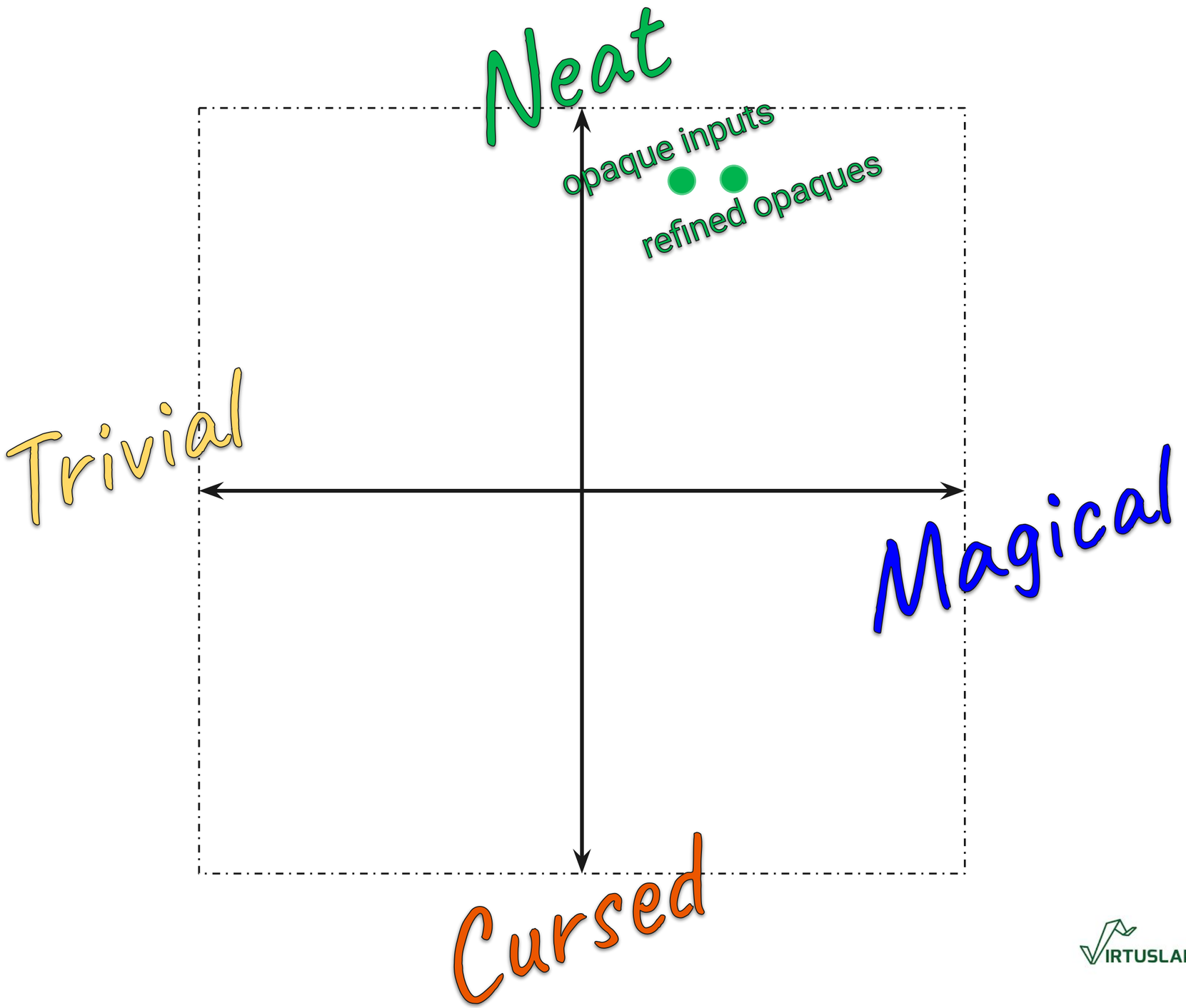
Mission:

You want your API to accept multiple types without much fuss because you only care about their common denominator in the end

Fun with Postel's law

```scala
object inputs:
  opaque type Optional[+A] >: A | Option[A] = A | Option[A]

  given {} with
    extension [A](x: Optional[A])
      def toOption: Option[A] = x match
        case x: Option[A] @unchecked => x
        case x: A @unchecked         => Some(x)

import inputs.*

def myApi(x: Optional[Int] = None): Unit =
  println(x.toOption)

@main def testInputs(): Unit =
  myApi(42)       // prints Some(42)
  myApi(Some(42)) // prints Some(42)
  myApi()         // prints None
  myApi(None)     // prints None
```

The compass


Neat
opaque inputs
refined opaques
Trivial
Magical
Cursed

Context proxies

Mission:

Your api has methods with different shapes taking different objects as parameters and you'd like to streamline DX and cut down the amount of imports

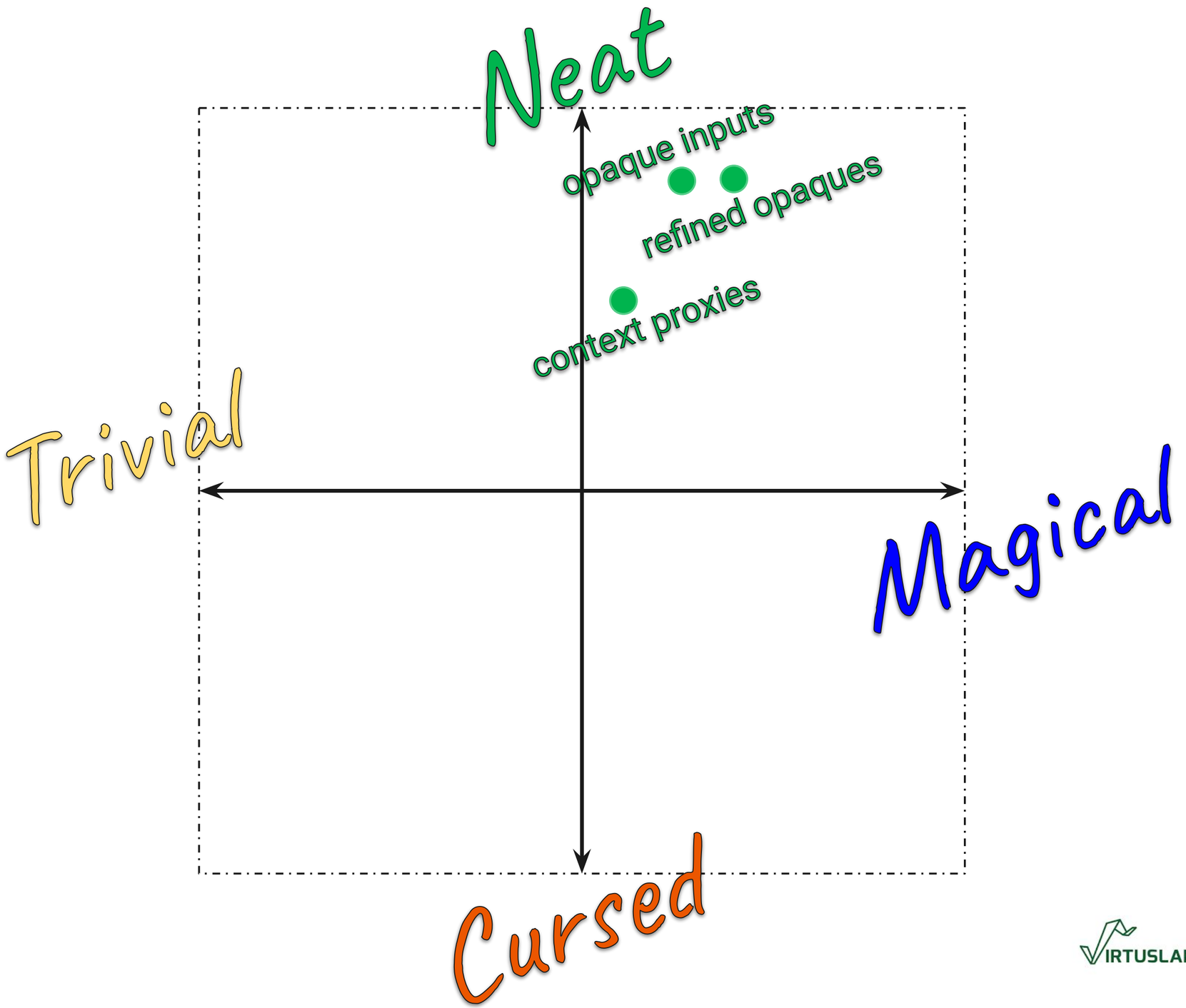
Context proxies

```
object contextproxies:
  case class A()
  case class B()

  case class OptionsForA(a: Int)
  case class OptionsForB(b: String)

  sealed trait Variant:
    type Constructor
    val constructor: Constructor

  object Variant:
    trait A extends Variant:
      type Constructor = OptionsForA.type
      val constructor = OptionsForA
    object A extends A

    trait B extends Variant:
      type Constructor = OptionsForB.type
      val constructor = OptionsForB
    object B

  def opts(using v: Variant): v.Constructor = v.constructor
```

Context proxies

```
def apiA(arg: String, options: OptionsForA): Unit
def apiB(arg: Int, options: OptionsForB): Unit

def smartA(arg: String, options: Variant.A ?=> OptionsForA): Unit
def smartB(arg: Int, options: Variant.B ?=> OptionsForB): Unit

import contextproxies.*

@main def testContextProxies(): Unit =
  apiA("ok", OptionsForA(42))
  apiB(23, OptionsForB("uhhhh"))

  smartA("nice", opts(42))
  smartB(5, opts("hello"))
```

The compass


Neat
opaque inputs
refined opaques
context proxies
Trivial
Magical
Cursed

Opaque API wrappers

Mission:

Your api has methods with different shapes taking different objects as parameters and you'd like to streamline DX and cut down the amount of imports

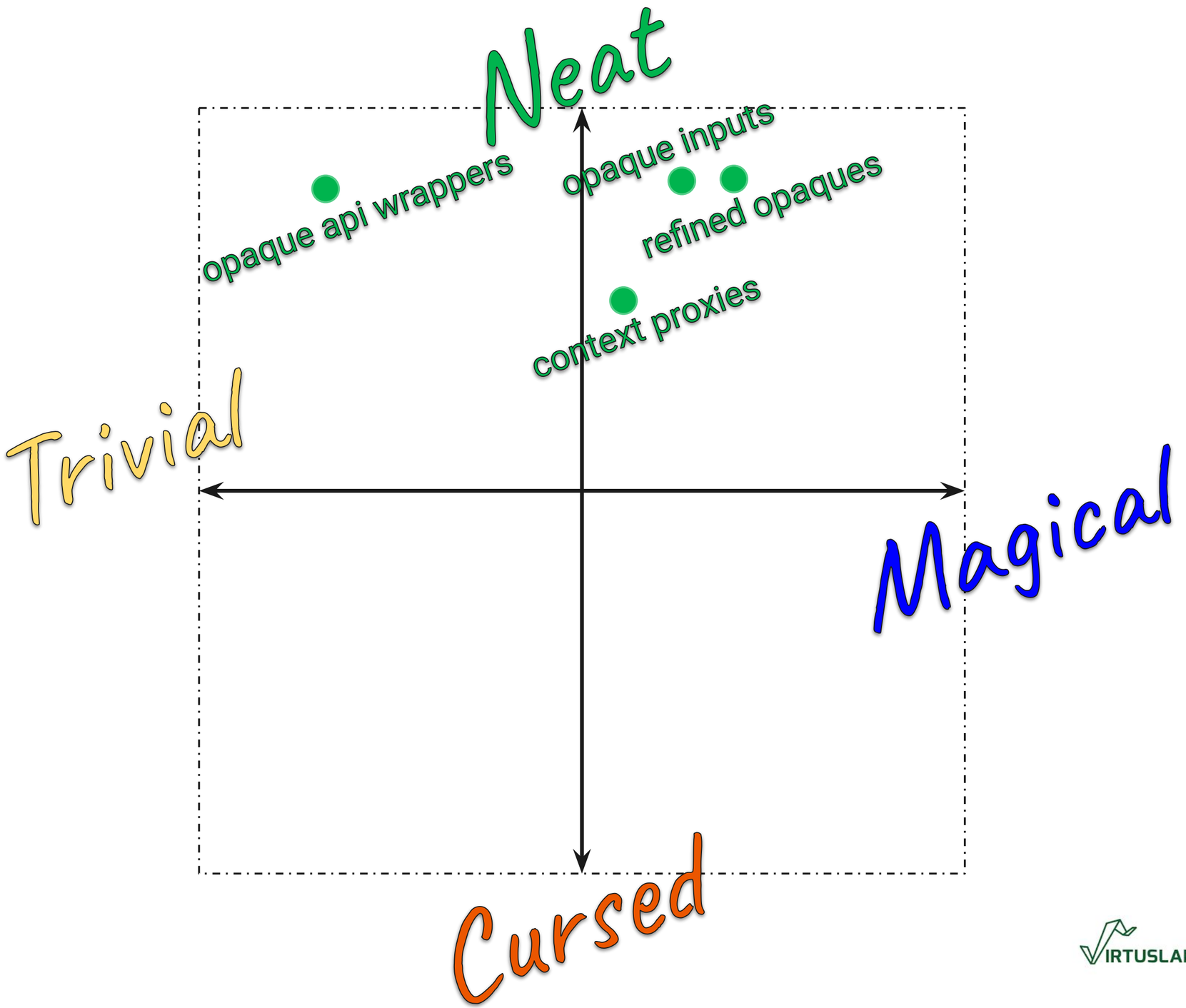
Opaque API wrappers

```scala
object opaqueapiwrappers:
  import io.github.iltotore.iron.*
  import io.github.iltotore.iron.constraint.numeric.*

  // only compiler knows this!
  opaque type File = java.io.File

  extension (f: File)
    def createNewFile(): Either[Exception, Boolean] =
      try Right(f.createNewFile())
      catch case e: Exception => Left(e)

    def setLastModified(time: Long :| Positive): Boolean =
      f.setLastModified(time)

  object File:
    def apply(path: String): File = java.io.File(path)
```

Opaque API wrappers

```scala
import opaqueapiwrappers.*
import io.github.iltotore.iron.*

@main def testSafeFile(): Unit =
  val file = File("test.txt")
  file.createNewFile() match
    case Right(true)  => println("File created")
    case Right(false) => println("File already exists")
    case Left(e)      => println(s"Path malformed: ${e.getMessage}")

  file.setLastModified(0)
  // -- Constraint Error ----------------------------------------------
  // Could not satisfy a constraint for type scala.Long.

  // Value: 0L
  // Message: Should be strictly positive
  // ------------------------------------------------------------------
```

The compass



VIRTUSLAB

The Magical

Macro based interpolators

Mission:

You need to allow your user to interpolate weird stuff and want to do it in a typesafe way because it’s still static code, right?

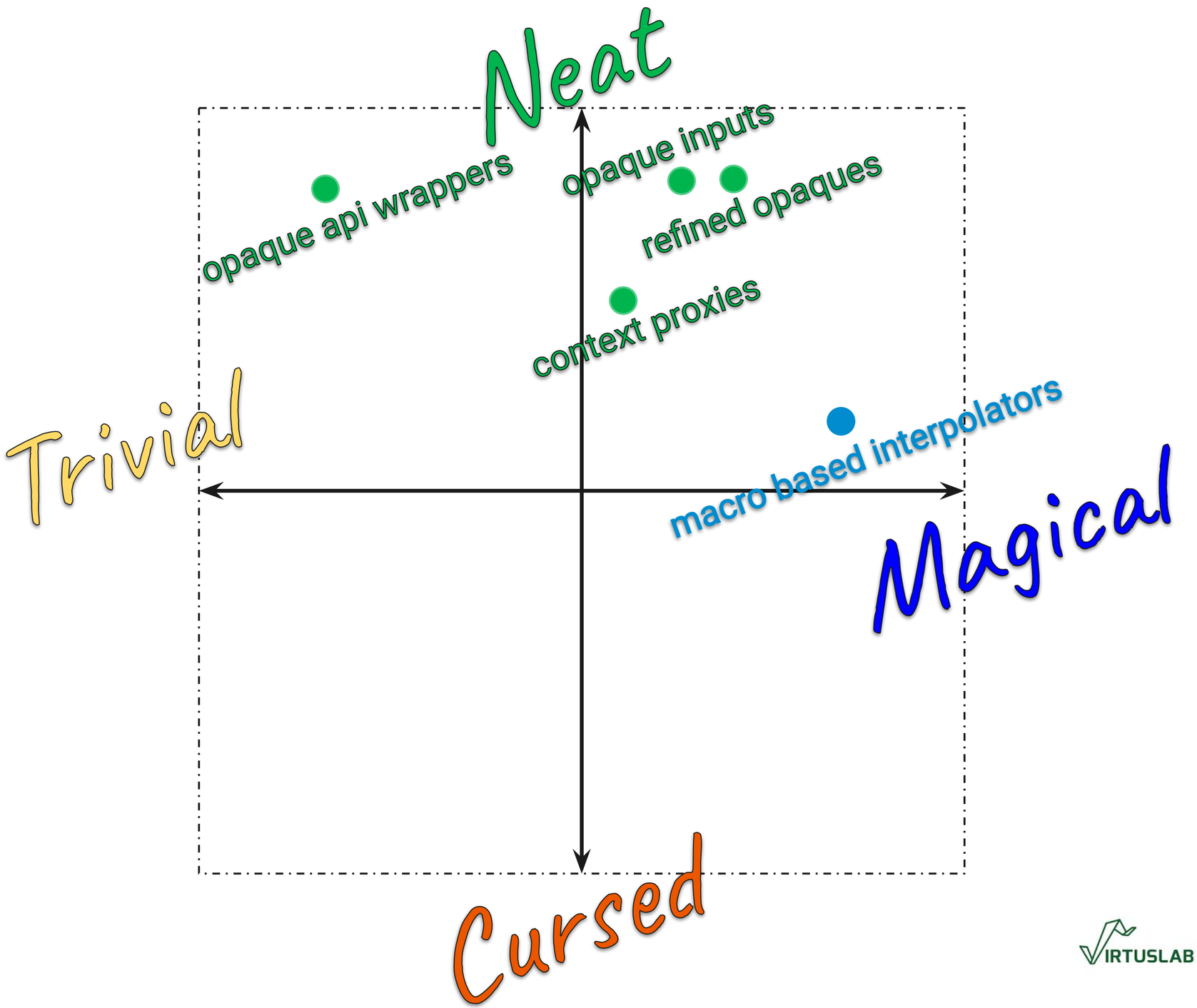
Macro based interpolators

Jump to VsCode

Macro based interpolators

```scala
import interpolators.*
import io.circe.*

@main def testJsonInterpolation(): Unit =
  val a = 23
  val b = JsonObject("c" -> Json.fromString("hello")).toJson

  val json: Json = json"""{"a": 23}"""
  println(json) // prints {"a":23}

  val json2: Json = json"""{"a": $a}"""
  println(json2) // prints {"a":23}

  val json3: Json = json"""{"a": $a, "b": $b}"""
  println(json3) // prints {"a":23,"b":{"c":"hello"}}
```

The compass



Type providers

Mission:

You need to allow your user to interpolate weird stuff and you want to type stuff based on the contents of whatever was interpolated

Type providers

Jump to Github

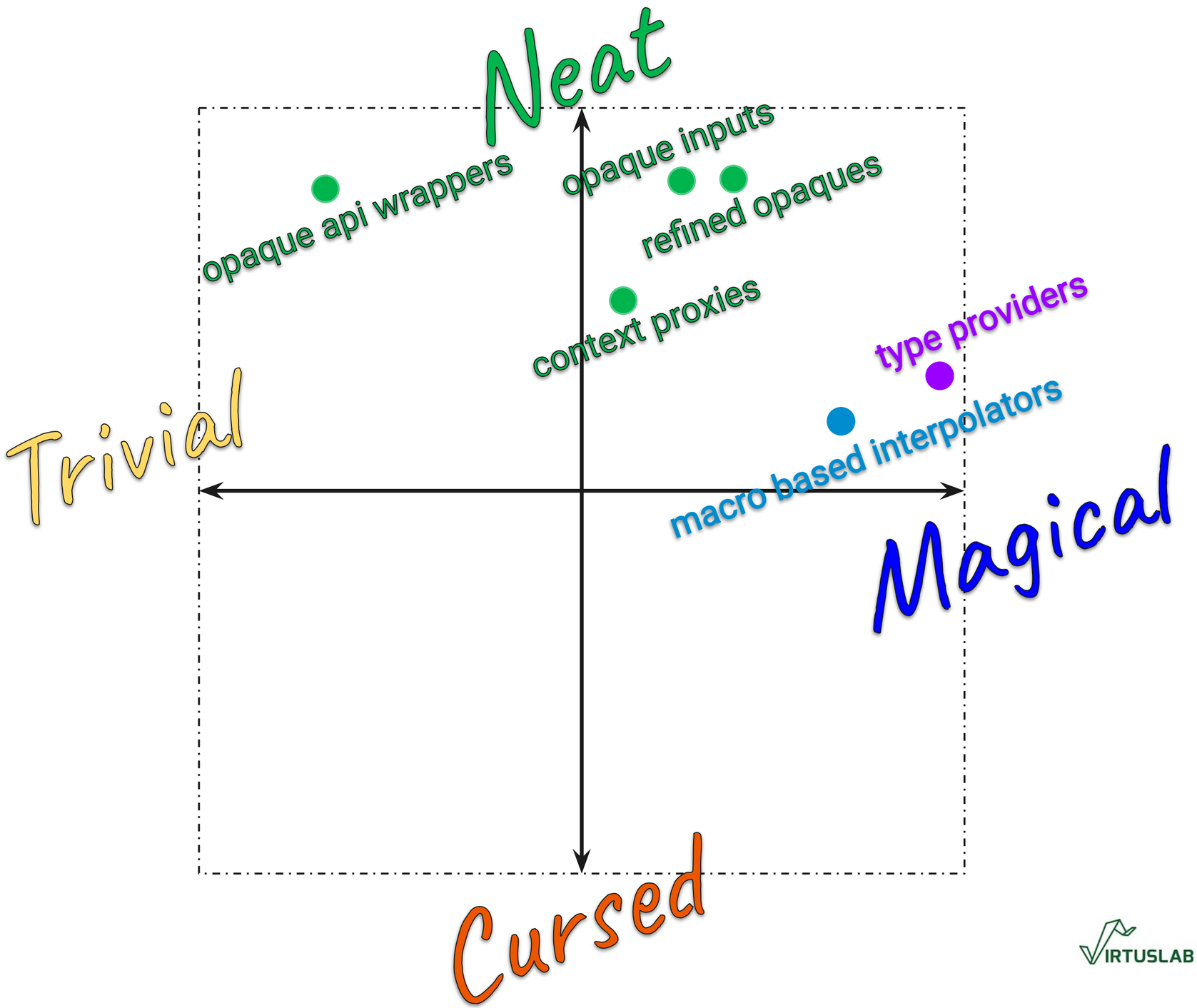
Type providers

```
@main def playground =
  val number = 10
  val repoName = "dotty"

  val myQuery = query"""
    |query {
    |  organization(login: "lampepfl") {
    |    projectV2(number: 6) {
    |        items(first: 10) {
    |        pageInfo {
    |          hasNextPage
    |          endCursor
    |        }
    |        nodes {
    |          id
    |          content {
    |            __typename
    |            ... on PullRequest {
    |              number
    |              mergedAt
    |            }
    |          }
    |        }
    |      }
    |    }
    |  }
    |}
  """
```

Type providers

```
val res = myQuery.send(
  uri"https://api.github.com/graphql",
  "Kordyjan",
  os.read(os.pwd / os.up / "key.txt")
)

val x = res.organization.projectV2.items.stream
  .take(30)
  .map(_.content.asPullRequest)
  .collect:
    case Some(pr) =>
      s"#${pr.number.toDouble.toInt}: ${pr.mergedAt}"
  .zipWithIndex
  .map: (str, i) =>
    s"${i + 1}. $str"
  .foreach(println)
```

The compass



Lifted syntax operators

Mission:

Your monadic DSL would be easier to use if properties of datatypes would be available **on the monadic values themselves**

Lifted syntax operators

Jump to Github

Lifted syntax operators

```
class Foo:
  val value = Some("abc")
  def methodNoParams = Some(true)
  def methodEmptyParens() = Some(0)
  def methodSingleArg(i: Int) = Some(i + 1)
  def methodTwoArgs(i: Int, j: Int) = Some(i + j)
```

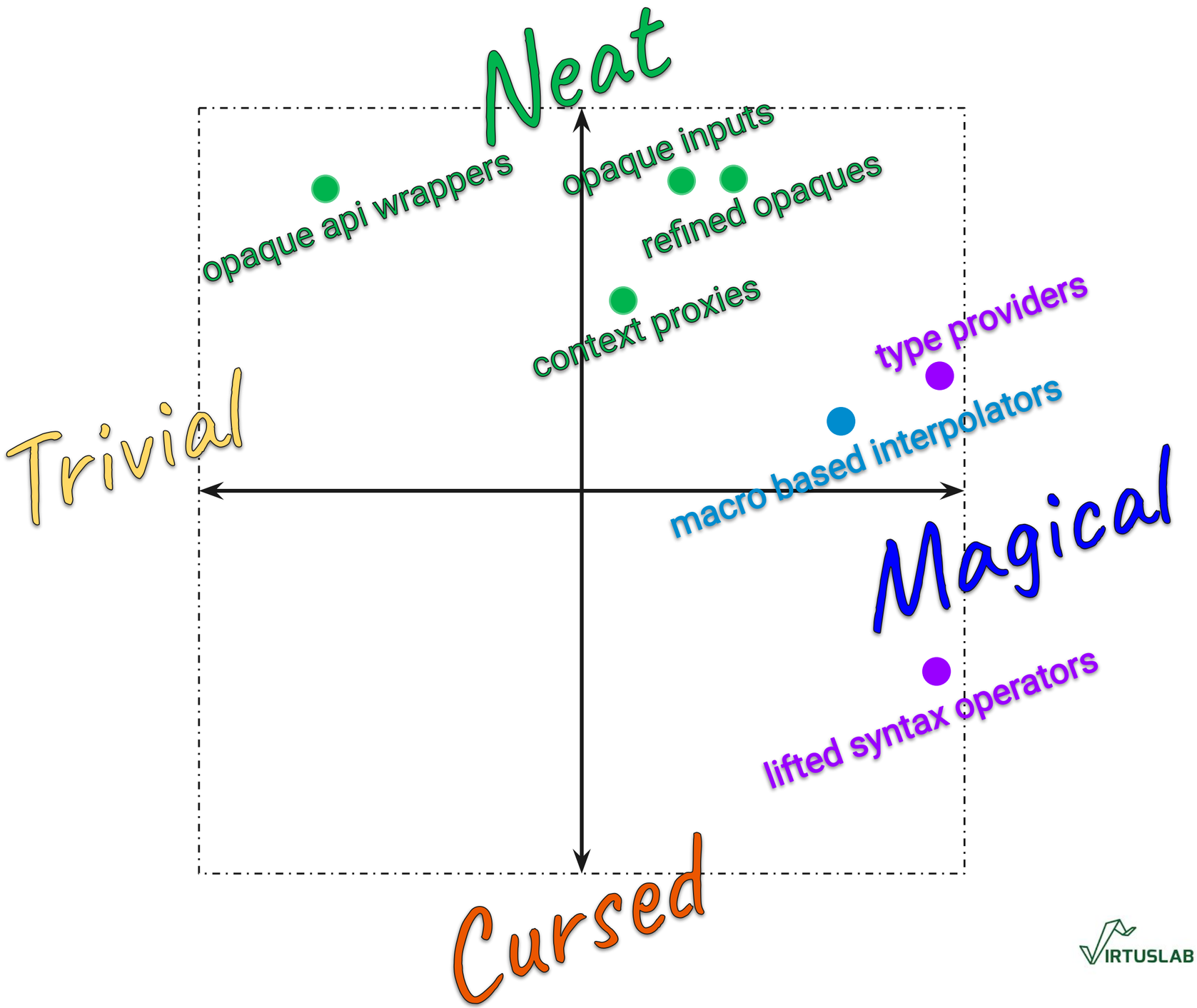
Lifted syntax operators

```
val opt: Option[Foo] = Option(new Foo)

@main def run() = {
  println(opt.?.value) // Some(Some(abc))
  println(opt.??.value) // Some(abc)

  println(opt.?.methodNoParams) // Some(Some(true))
  println(opt.??.methodNoParams) // Some(true)

  println(opt.?.methodEmptyParens()) // Some(Some(0))
  println(opt.??.methodEmptyParens()) // Some(0)

  println(opt.?.methodSingleArg(0)) // Some(Some(1))
  println(opt.??.methodSingleArg(0)) // Some(1)

  println(opt.?.methodTwoArgs(1, 2)) // Some(Some(3))
  println(opt.??.methodTwoArgs(1, 2)) // Some(3)
```

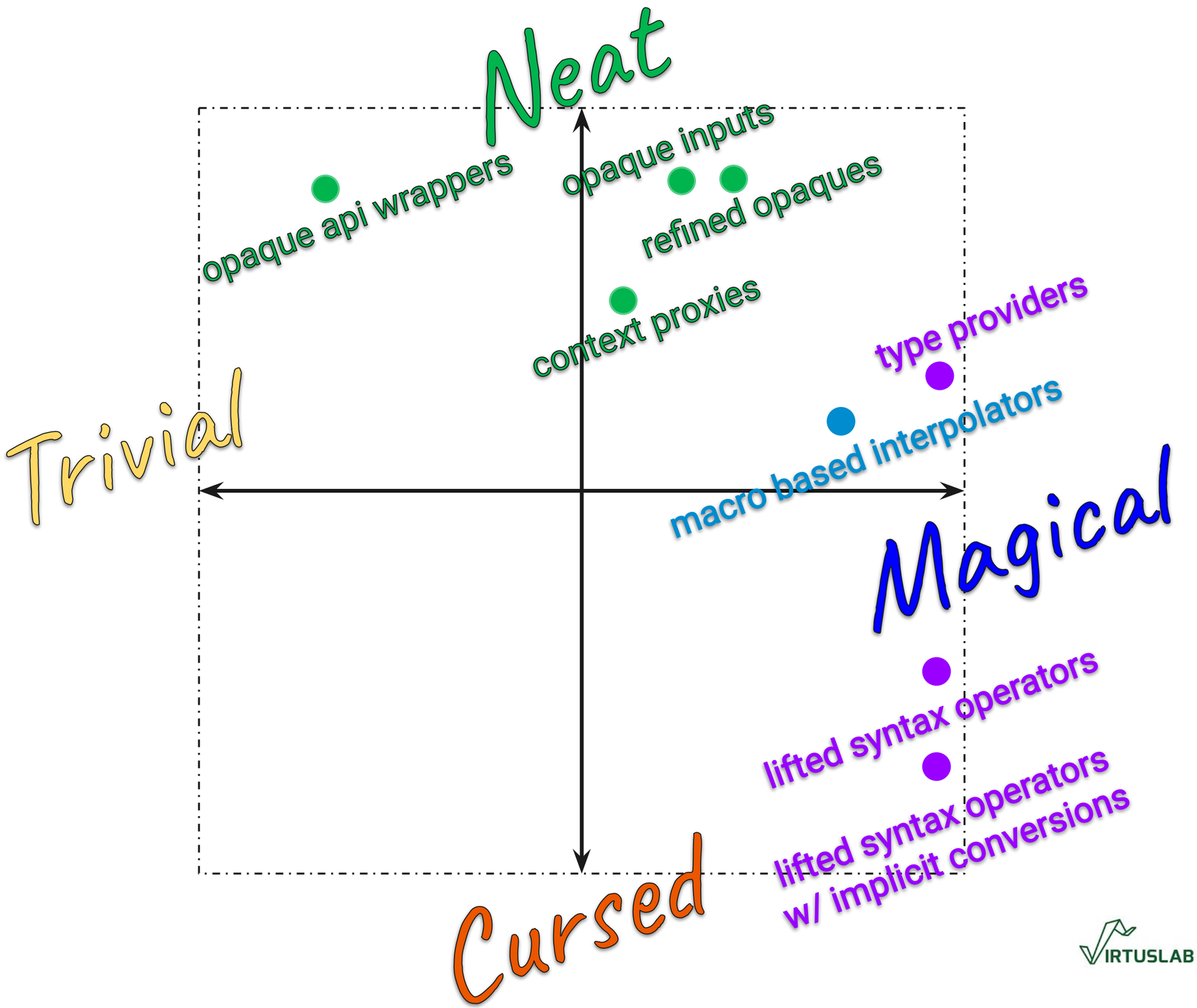
Lifted syntax operators

```
val res = myQuery.send(
  uri"https://api.github.com/graphql",
  "Kordyjan",
  os.read(os.pwd / os.up / "key.txt")
)

val x = res.organization.projectV2.items.stream
  .take(30)
  .map(_.content.asPullRequest)
  .collect:
    case Some(pr) =>
      s"#${pr.number.toDouble.toInt}: ${pr.mergedAt}"
  .zipWithIndex
  .map: (str, i) =>
    s"${i + 1}. $str"
  .foreach(println)
```

The compass


Neat
Trivial
Magical
Cursed
opaque api wrappers
opaque inputs
refined opaques
context proxies
type providers
macro based interpolators
lifted syntax operators

The compass



The Insane

Fever syntax

Mission:

You need to build a structured DSL that takes a lot of properties...

Fever syntax

```
trait Builder
def *(using b: Builder): b.type = b

object cursedsdkasterisksyntax:

  case class Foo private (
      name: String,
      arg1: Option[Int],
      arg2: Option[String],
      arg3: Seq[Bar.BarArgsBuilder]
  )
```

Fever syntax

```
case class FooArgsBuilder private[Foo] (
    private[Foo] val _arg1: Option[Int],
    private[Foo] val _arg2: Option[String],
    private[Foo] val _arg3: Seq[Bar.BarArgsBuilder]
) extends Builder:
  def arg1(arg: Int) = copy(_arg1 = Some(arg))
  def arg2(arg: String) = copy(_arg2 = Some(arg))
  def arg3(args: Bar.BarArgsBuilder ?=> Bar.BarArgsBuilder*) =
    copy(_arg3 = args.map(_(using Bar.BarArgsBuilder())))

object FooArgsBuilder:
  def apply(): FooArgsBuilder =
    FooArgsBuilder(_arg1 = None, _arg2 = None, _arg3 = Seq.empty)
```

Fever syntax

```
object Bar:
  case class BarArgsBuilder private[Bar] (
      private val _arg1: Option[String],
      private val _arg2: Option[Boolean]
  ) extends Builder:
    def arg1(arg: String) = copy(_arg1 = Some(arg))
    def arg2(arg: Boolean) = copy(_arg2 = Some(arg))

  object BarArgsBuilder:
    def apply(): BarArgsBuilder =
      BarArgsBuilder(_arg1 = None, _arg2 = None)
```

Fever syntax

```
import cursedsdkasterisksyntax.*

@main def testCursedAsteriskSyntax(): Unit =
  val foo = Foo.make("foo1")(
    * arg1 1
      arg2 "qwe"
      arg3 (
        * arg1 "xxx"
          arg2 true,
        * arg1 "xxx"
          arg2 false
      )
  )

  println(foo)
```

Fever syntax

Another

Fever syntax

```
trait ArgsBuilder

// There will never be an instance of this type
class Unreachable private ()

object cursednewsdk:
  object Service:
    type Args = ServiceArgsBuilder

  class ServiceArgsBuilder extends ArgsBuilder:
    // stub to make the setter work
    def spec(using Unreachable) = ???
    private var spec_ : ServiceSpecArgsBuilder = null
    def spec_=(arg: ServiceSpecArgsBuilder) =
      spec_ = arg
```

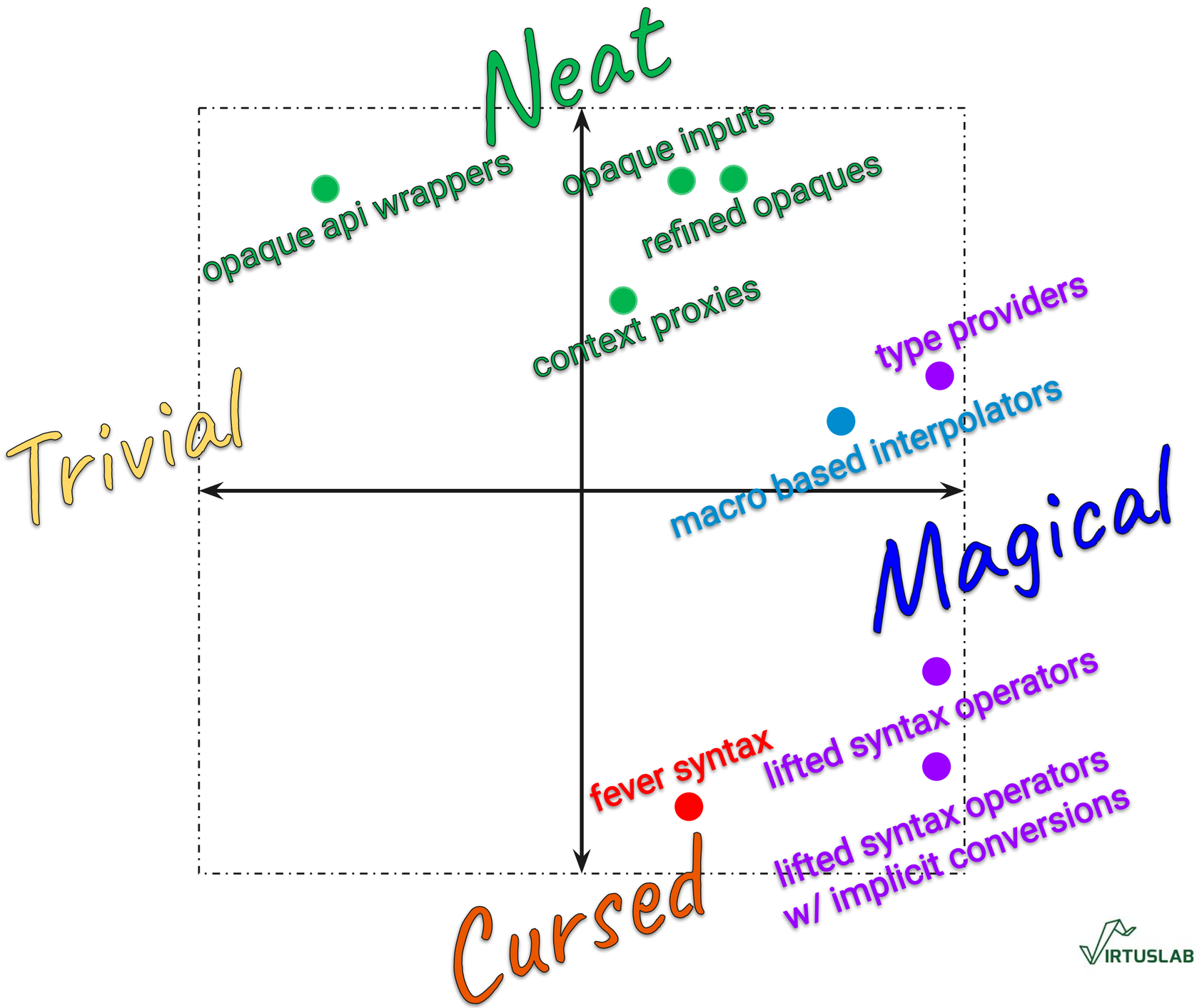
Fever syntax

```
object Port:
  type Args = PortArgsBuilder

class PortArgsBuilder extends ArgsBuilder:
  // stub to make the setter work
  def name(using Unreachable) = ???
  private var name_ : String = null
  def name_=(arg: String): Unit =
    name_ = arg

  // stub to make the setter work
  def port(using Unreachable) = ???
  private var port_ : Int = 0
  def port_=(arg: Int): Unit =
    port_ = arg

def service(name: String, args: ServiceArgsBuilder) = ???
```

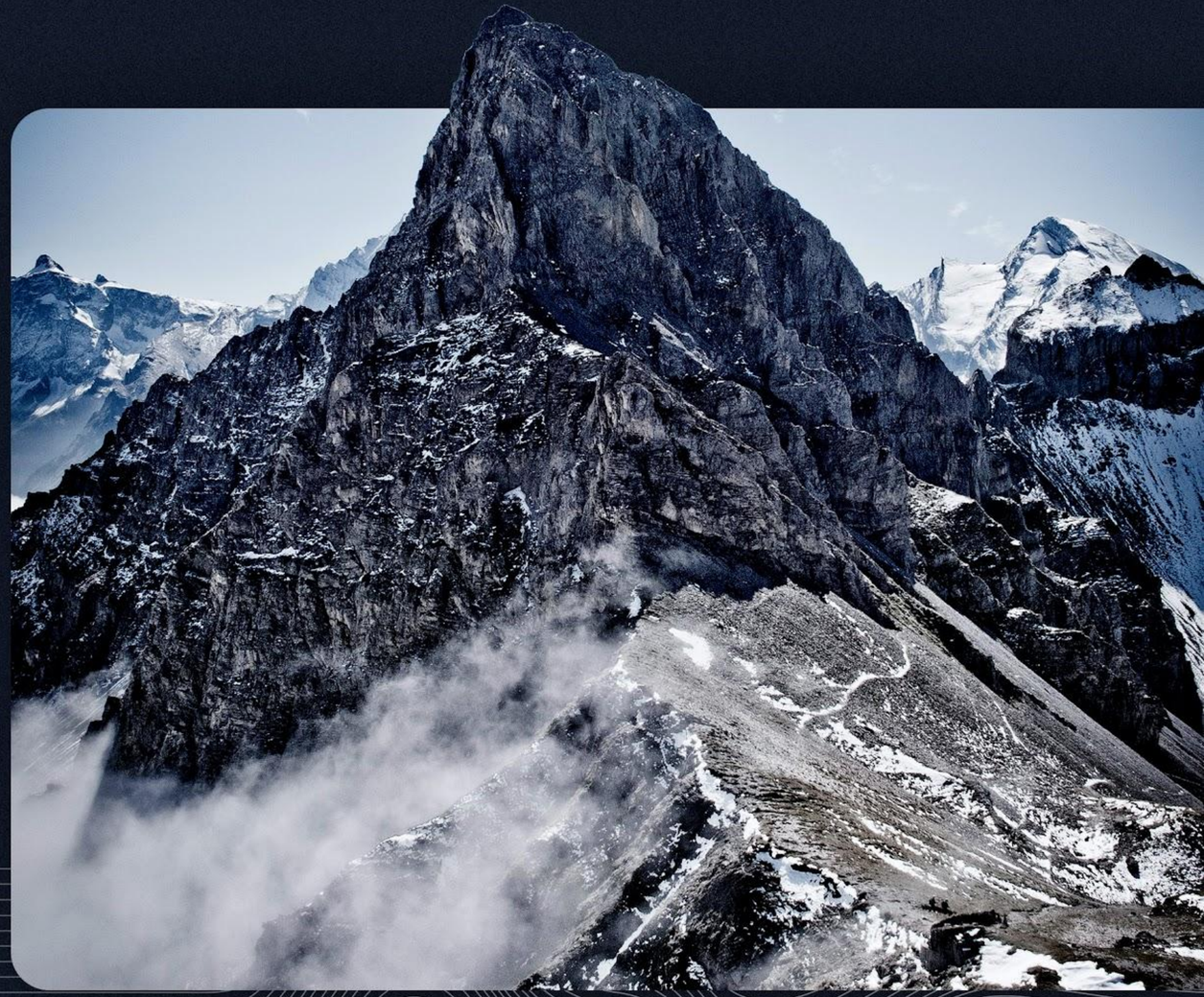
Fever syntax

```
import cursednewsdk.*

@main def testCursedFeverBracesSyntax(): Unit =
  val labels = List("label1", "label2")

  val port2 = new Port.Args:
    name = "port2"
    port = 456

  val nginxService = service(
    "nginx",
    args = new:
      spec = new:
        selector = labels
        ports = List(
          new {
            name = "port1"
            port = 123
          },
          port2
        )
  )
```

The compass



Thank you



VIRTUS LAB