# Design-Driven Assurance in Wyvern

**Jonathan Aldrich** and **Alex Potanin**

Workshop on New Object-Oriented Languages (NOOL) at SPLASH 2017
October 2017

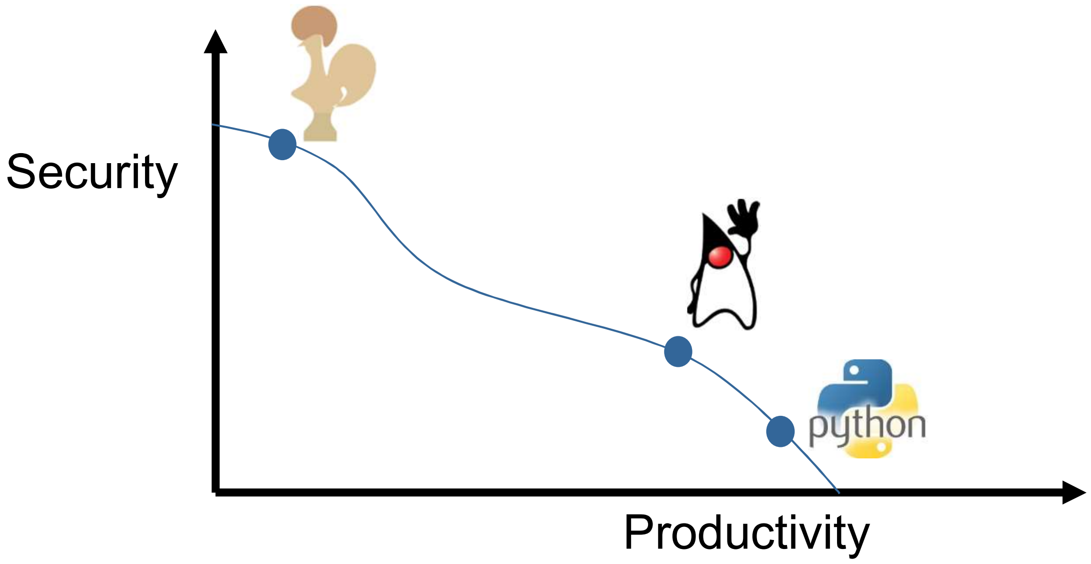
Carnegie Mellon University
School of Computer Science

TE WHARE WĀNANGA O TE ŪPOKO O TE IKA A MĀUI
VICTORIA
UNIVERSITY OF WELLINGTON

# The Wyvern Programming Language

- Designed for security and productivity from the ground up
- General purpose, but emphasizing web/mobile/IoT apps

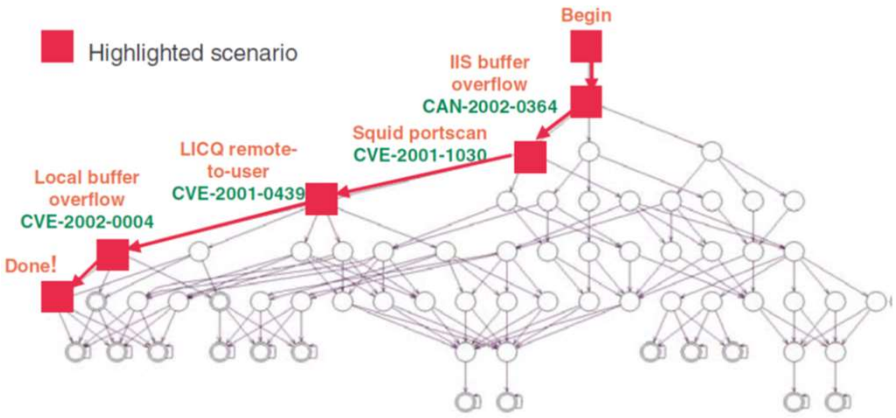
- But you might ask:
  - Isn't there a **tradeoff** between security and productivity?



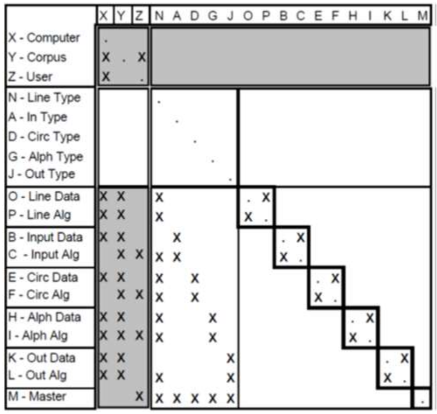
  - What is Wyvern's **secret sauce**?

# Insight: Engineering Impact of Design

- Design constraints drive program properties [Bass et al.]

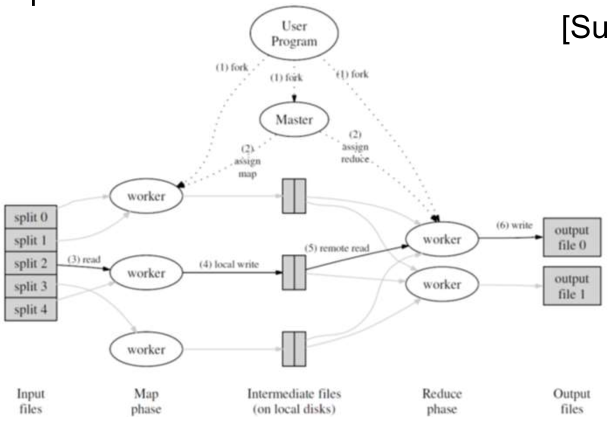


**Security** analysis with attack graphs
[Sheyner et al. ’02]



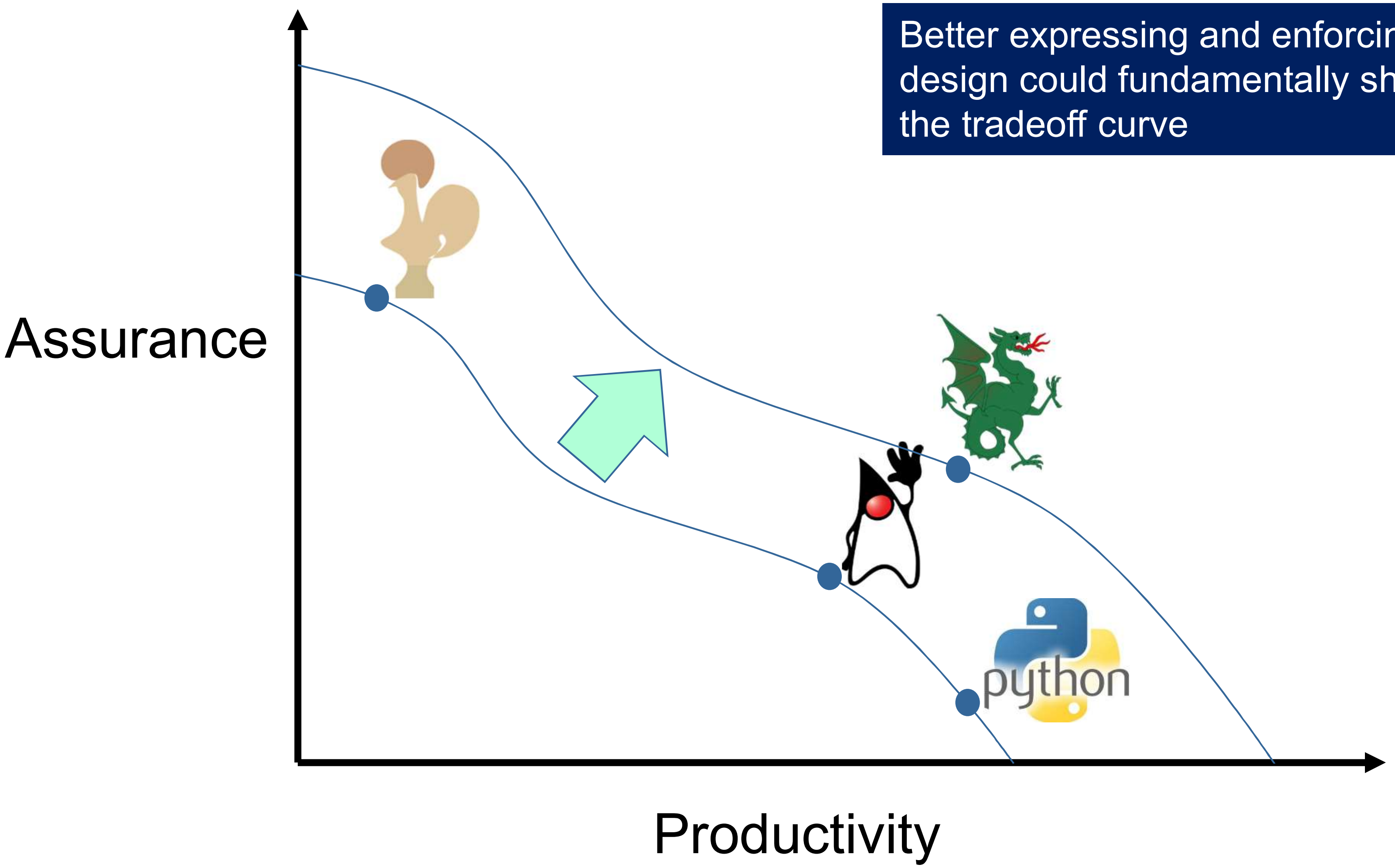
|  | X | Y | Z | N | A | D | G | J | O | P | B | C | E | F | H | I | K | L | M |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| X - Computer | . |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Y - Corpus | X | . | X |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Z - User | X |  | . |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| N - Line Type |  |  |  | . |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| A - In Type |  |  |  |  | . |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| D - Circ Type |  |  |  |  |  | . |  |  |  |  |  |  |  |  |  |  |  |  |  |
| G - Alph Type |  |  |  |  |  |  | . |  |  |  |  |  |  |  |  |  |  |  |  |
| J - Out Type |  |  |  |  |  |  |  | . |  |  |  |  |  |  |  |  |  |  |  |
| O - Line Data | X | X |  | X |  |  |  |  | . | X |  |  |  |  |  |  |  |  |  |
| P - Line Alg | X | X |  | X |  |  |  |  | X | . |  |  |  |  |  |  |  |  |  |
| B - Input Data | X | X |  |  | X |  |  |  |  |  | . | X |  |  |  |  |  |  |  |
| C - Input Alg |  | X | X | X | X |  |  |  |  |  | X | . |  |  |  |  |  |  |  |
| E - Circ Data | X | X |  | X |  | X |  |  |  |  |  |  | . | X |  |  |  |  |  |
| F - Circ Alg |  | X | X | X |  | X |  |  |  |  |  |  | X | . |  |  |  |  |  |
| H - Alph Data | X | X |  | X |  |  | X |  |  |  |  |  |  |  | . | X |  |  |  |
| I - Alph Alg | X | X | X | X |  |  | X |  |  |  |  |  |  |  | X | . |  |  |  |
| K - Out Data | X | X |  |  |  |  |  | X |  |  |  |  |  |  |  |  | . | X |  |
| L - Out Alg | X | X |  | X |  |  |  | X |  |  |  |  |  |  |  |  | X | . |  |
| M - Master |  |  | X | X | X | X | X | X |  |  |  |  |  |  |  |  |  |  | . |

Module dependencies and **evolution**
[Sullivan et al. ’01]



MapReduce: **Scalable** big data
[Dean and Ghemawat ’04]

# Shifting the Tradeoff Curve



Better expressing and enforcing design could fundamentally shift the tradeoff curve

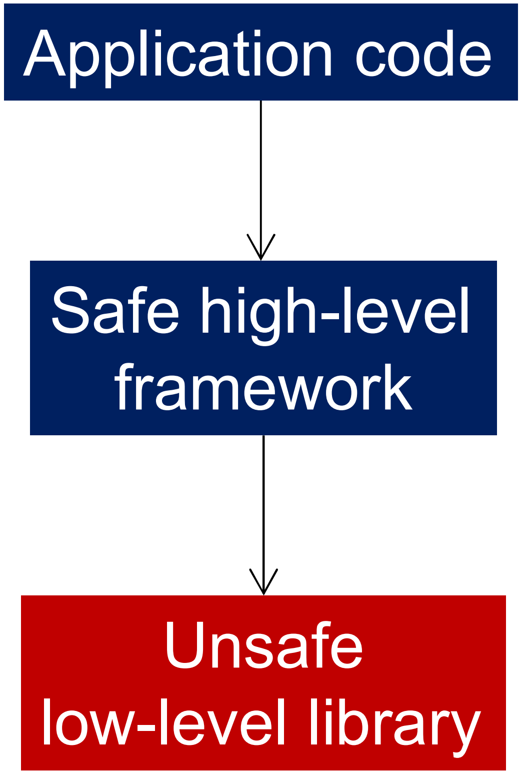
# Design-Driven Assurance in Wyvern

- The Wyvern Approach: ***Usable Design-Driven Assurance***
  - Usable mechanisms to express and enforce large-scale design
  - Support for built-in assurance of critical properties, esp. security

- Key mechanisms for expressing and enforcing design
  - Modules and architecture express ***high-level design***
  - Extensible notation express ***code-level design***
  - Types, capabilities, and effects to ***enforce design***

# An Old Idea: Layered Architectures [Dijkstra 1968]

- Lowest layer: an unsafe, low-level library
  - provides basic access to resources
- Middle layer: a higher-level framework
  - enforces safety invariants over resources
- Top layer: the application
- Code must obey strict layering
  - Application must only use the safe framework
- Many variants
  - Secure networking framework
  - Safe SQL-access library
  - Replicated storage library
  - Map-reduce library, …



- RQ: Can we use ***capabilities*** to enforce layered resource access?
  - Capability: an unforgeable token controlling access to a resource [Dennis & Van Horn 1966]

# Module Linking as Architecture

```
require db.stringSQL
```

To access external resources like a database, main requires a **capability** from the run-time system. A capability is an unforgeable token controlling access to a resource

```
application.run()
```

stringSQL

# Module Linking as Architecture

We can import code modules, but they have no *ambient authority* to access resources (cf. NewSpeak). sqlApplication cannot access the database by itself.

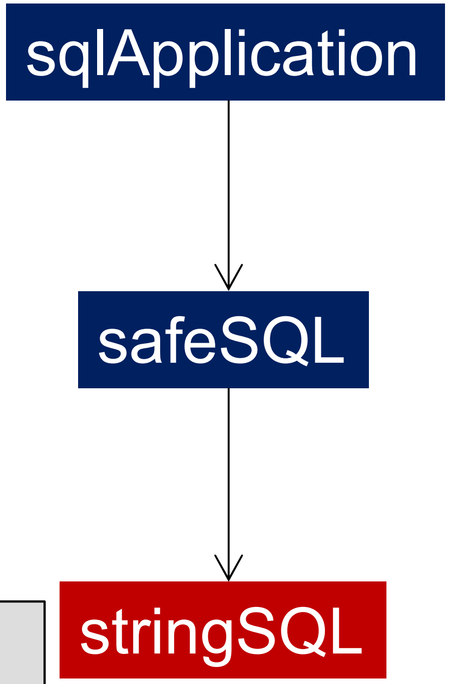
```
require db.stringSQL

import db.safeSQL
import app.sqlApplication

val sql = safeSQL(stringSQL)
val application = sqlApplication(sql)

application.run()
```

We must instantiate a sqlApplication object, passing it the resources it needs. We pass only a capability to the safe library.



sqlApplication

safeSQL

stringSQL

# Module Linking as Architecture

```
module def sqlApplication(safeSQL : db.SafeSQL)

def run() : Int
    // application code
```

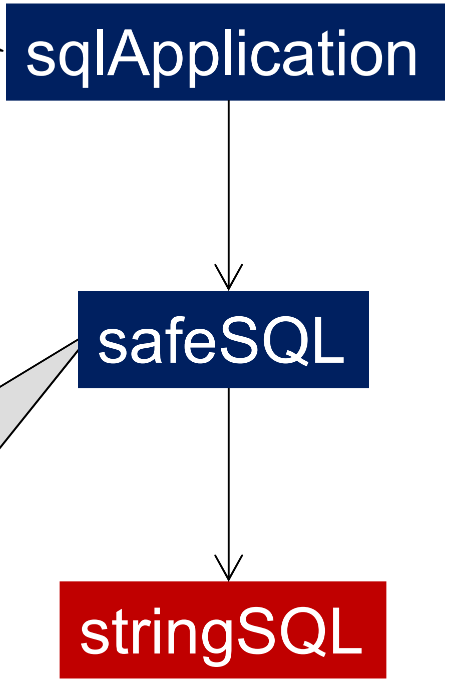
```
require db.stringSQL

import db.safeSQL
import app.sqlApplication

val sql = safeSQL(stringSQL)
val application = sqlApplication(sql)

application.run()
```

```
module def safeSQL(strSQL : db.StringSQL)
// implement ADT
// in terms of strings
```



# But won’t it be a pain to link everything?

- Most Wyvern modules don’t have state, can be freely imported
- Statically tracked: stateful modules/objects are or **resource** types

```
type SetM
    resource type Set
        def add(v:Int)
        def isMember(v:Int):Bool
    def makeSet():Set
```

Type of module is pure; no static state. Objects created by module may be stateful resources, though.

```
resource type File
    def write(s:String)
```

Provides access to OS resource

```
module setM : SetM …
```

```
module def client(aFile:File)
import setM …
```

Resources must be passed in; pure modules can just be imported

- **resource** types capture state or system access; other types do not
  - Useful design documentation; e.g. MapReduce tasks should be stateless
  - Supports powerful equational reasoning, safe concurrency, etc.

# But I *like* my insecure SQL library!

- Pasting strings is convenient:

```
connection.executeQuery(
        "SELECT * FROM Students WHERE name = '" + studentName + "';");
```

- A fully secure library might not be nearly as nice:

```
connection.executeQuery(select(star, new String[] { “Students” },
                            equals(column(“name”), studentName)));
```

- Prepared queries are also not great (and not fully secure):

```
PreparedStatement s = connection.prepareStatement(
        "SELECT * FROM Students WHERE name = ?;");
s.setString(1, userName);
s.executeQuery();
```

# Wyvern: *Usable* Secure Programming

- A SQL query in Wyvern

```
import metadata sqlLang

connection.executeQuery(~)
        SELECT * FROM Students WHERE name = {studentName}
```

Imports a DSL for SQL queries, including metadata for parsing

~ triggers parser for SQL DSL on indented lines

Can provide IDE support, e.g. syntax highlighting, autocomplete, …

Safely incorporates dynamic data—as data, not a command

- Compare the (insecure) alternative

```
connection.executeQuery(
        "SELECT * FROM Students WHERE name = '" + studentName + "';");
```

- Claim: the secure version **more natural** *and* **more usable**
  - We hope to evaluate this empirically in the near future

# Run-Time Architecture (ongoing work)

```
import lang architecture

architecture clientServer
        component c:Client
        component s:Server

        connector link:HTTPSCtr

        connect c.getInfo and s.sendInfo with link
```

Imports the architecture DSL

DSL impl uses capabiliites internally to ensure components only communicate via connections

Architecture specifies use of connector library with desired security characteristics

Connector implemented using metaprogramming that generates boilerplate, enhancing usability

# Reasoning about Authority with Types

- How do we reasoning about the ***authority*** of an object?
  - i.e. what effects (writes, system operations) can an object have? [Miller 2006]
  - Prior work: semantic defn. of ***eventual authority*** [Drossopoulou et al., 2016]
  - Prior work: **topological bound** on authority [Miller 2006; Maffeis et al. 2010]
- Approximate authority informally using types [Melicher et al., 2017]

```
type HttpRequestor
    // HTTP get request on a URL
    def get(url:String):String
```

If we trust the HttpRequestor implementation, we can (informally) reason about the authority of MyADT: to do HTTP get requests.
More precise than topological bound.

```
// defined in a pure module
type MyADT
    def operation(x:Int):String
def makeADT(req:HttpRequestor):MyADT
```

MyADT is born with permission to an HttpRequestor. The type proves it can't get additional permissions

# Reasoning about Authority with Effects

- How do we reasoning about the ***authority*** of an object?
  - i.e. what effects (writes, system operations) can an object have? [Miller 2006]
  - Prior work: semantic defn. of ***eventual authority*** [Drossopoulou et al., 2016]
  - Prior work: **topological bound** on authority [Miller 2006;Maffeis et al. 2010]
- Current work: reason ***formally, precisely*** about authority using effects

```
effect getRequest

type Requestor // untrusted code
    def get(url:String):String { getRequest }

type MyADT
    def operation(x:Int):String { getRequest }
def makeADT(req:HttpRequestor):MyADT
```

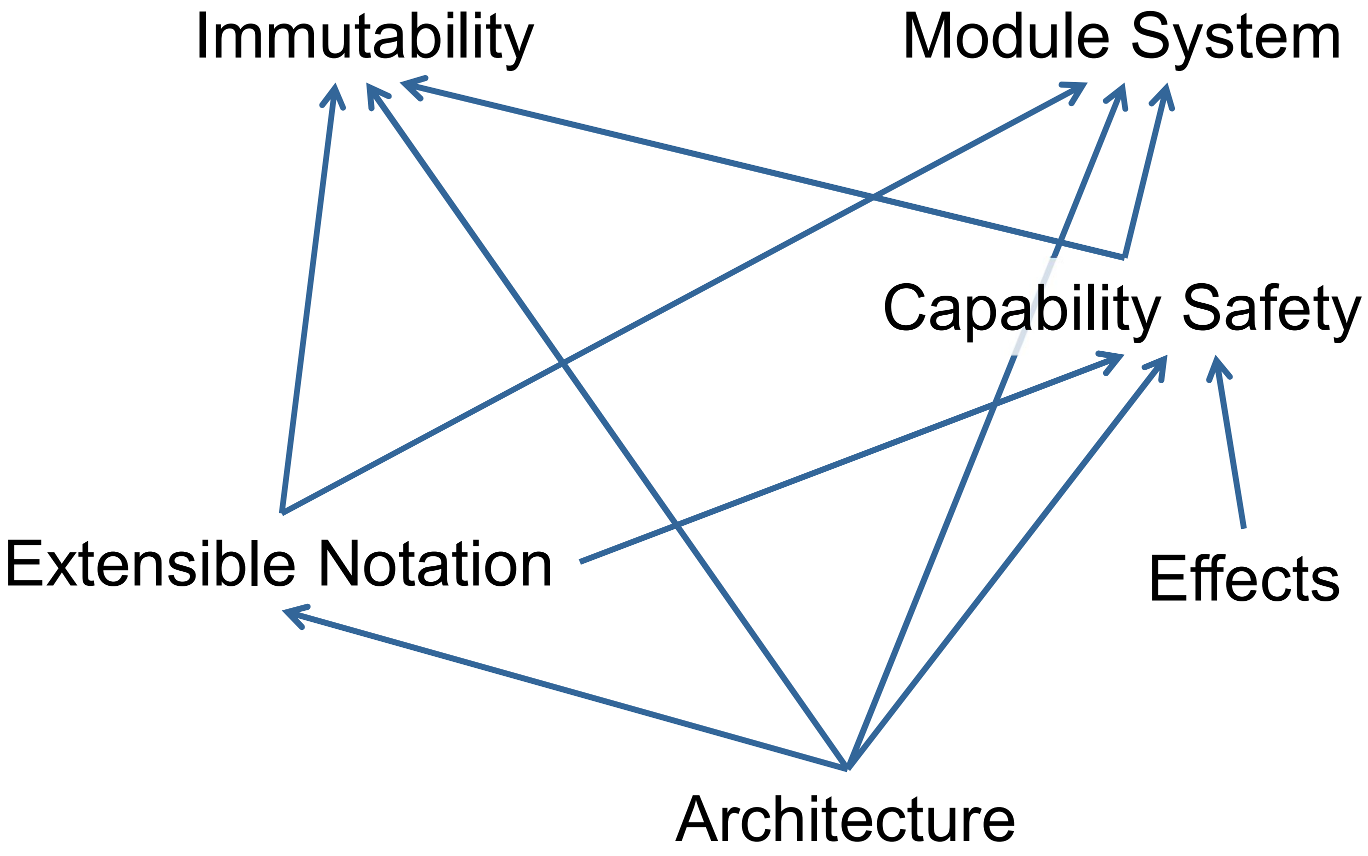
Trusted HTTP library implements getRequest functionality using network

We don't know/trust the Requestor implementation, but the effect bounds its authority (and the authority of clients)

# Wyvern Design Principles From 3 Fields

- SE: Express design that impacts engineering at scale
  - Enforcing **system organization**: both code and run-time structure
  - **Immutability** constraints play architectural role
  - **Effects** for reasoning about authority in the large
- PL: Formal properties that are deep and widely applicable
  - **Composability** of language extension [Omar et al. 2014]
  - **Immutability** is used widely and provides high reasoning leverage
  - **Capability safety** can be leveraged to enforce design properties
- HCI: Empirical focus on usability and user tasks
  - SQL arguably a **natural notation** [Myers et al. 2004] for queries
  - **IDE support** for languages has high impact on tasks
  - Empirical study on **usability of immutability** [Coblenz et al. 2017]

# Synergies in Language Design



# Wyvern: Design-Driven Assurance

- Novel approach to achieve high usability and assurance

- Leverage new mechanisms for capturing design constraints
  - Foundational: Immutability, capabilities, extensible notation
  - Scaling up: Modules, architecture, effects

- Drivers
  - SE: Design constraints that impact engineering at scale
  - PL: Formal properties that are deep and widely applicable
  - HCI: Empirical focus on usability and user tasks

- Follow on work: extensible checking, gradual verification