

Deep Learning for Compilers

Hugh Leather
University of Edinburgh



Overview

- **Machine Learning for Compilers**
- **Generating Benchmarks**
- **Deep Learned Heuristics**
- **Deep Fuzzing Compiler Testing**
- **Future Work**





Chris Cummins

University of Edinburgh



Pavlos Petoumenos

University of Edinburgh



Zheng Wang

Lancaster University



Hugh Leather

University of Edinburgh



THE UNIVERSITY OF EDINBURGH
informatics

EPSRC Centre for Doctoral Training in
Pervasive Parallelism

EPSRC

Engineering and Physical Sciences
Research Council

Overview

- **Machine Learning for Compilers**

- Generating Benchmarks

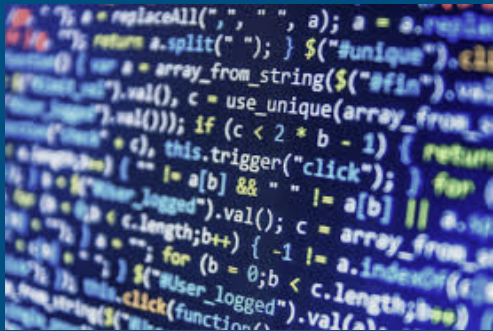
- Deep Learned Heuristics

- Deep Fuzzing Compiler Testing

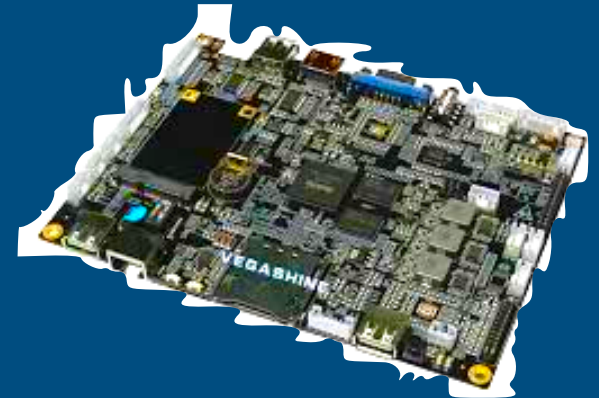
- Future Work



Compilers are *hard*



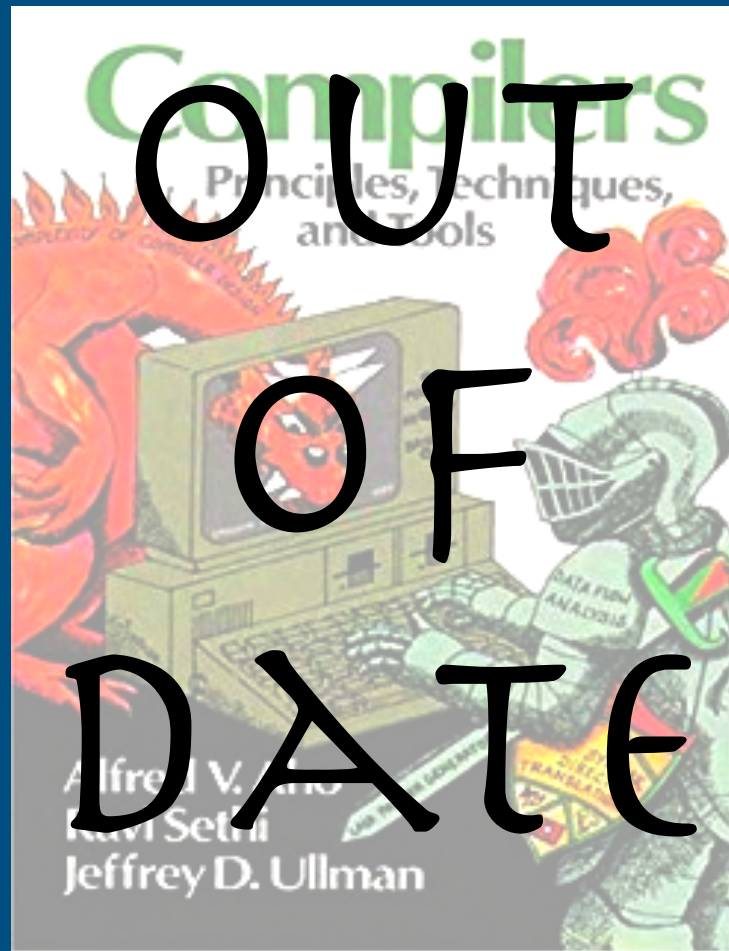
Huge number of variables
NP-hard or worse
Keep changing



Nondeterministic machines
Many components
Keep changing

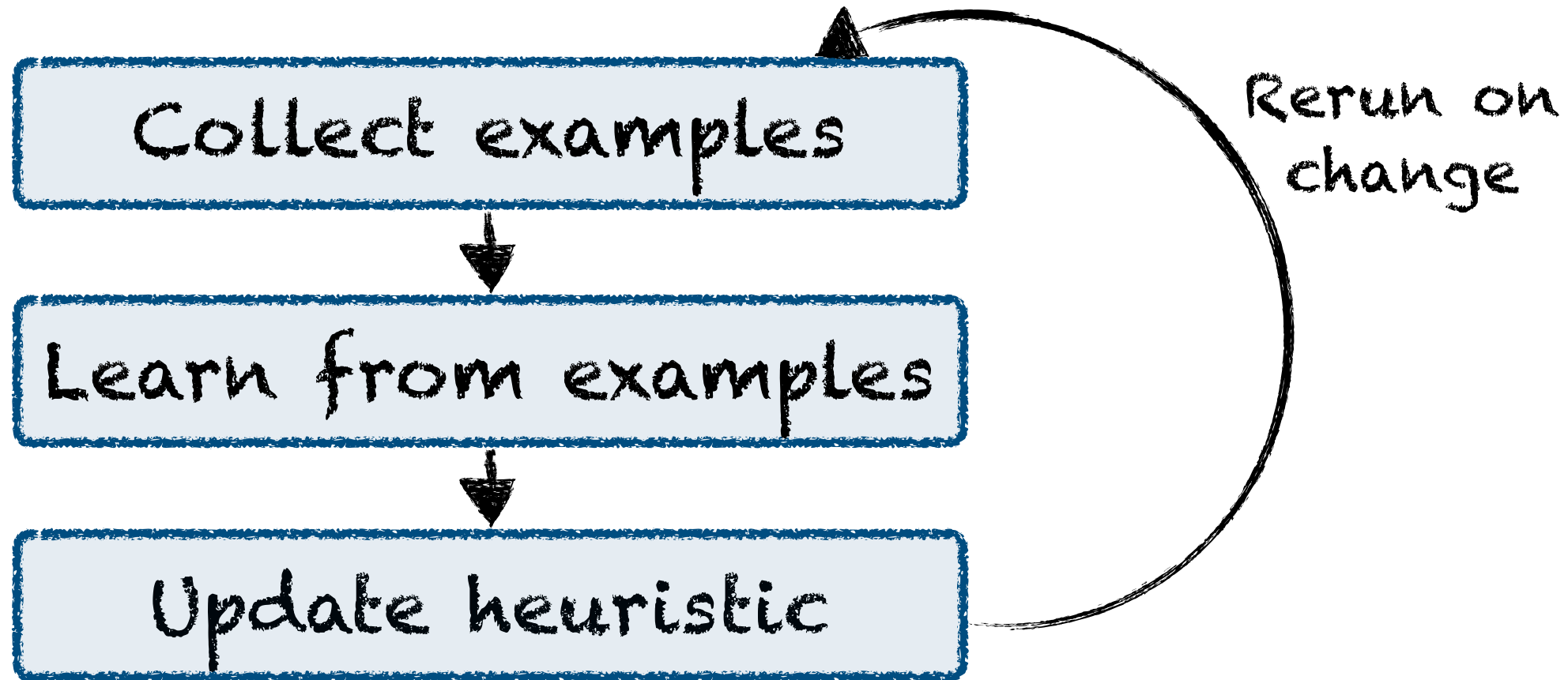
Compilers are *hard*

**Slow
programs**



**Energy
waste**

Machine Learning to the Rescue



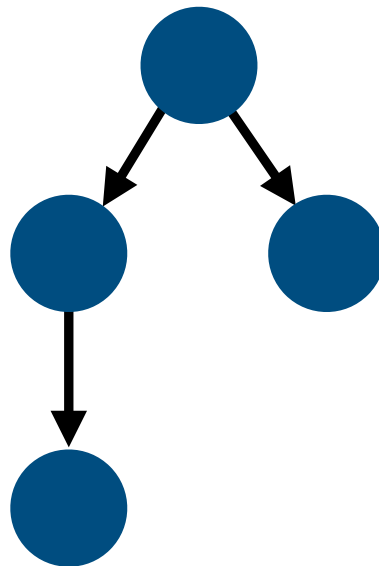
Summarise the Program

Program

```
int main(int argc, char** argv) {  
    printf("Hello, World!");  
    return 0;  
}
```

IR

(AST, CFG, DDG, etc.)



Features

Number of instructions

Mean dependency depth

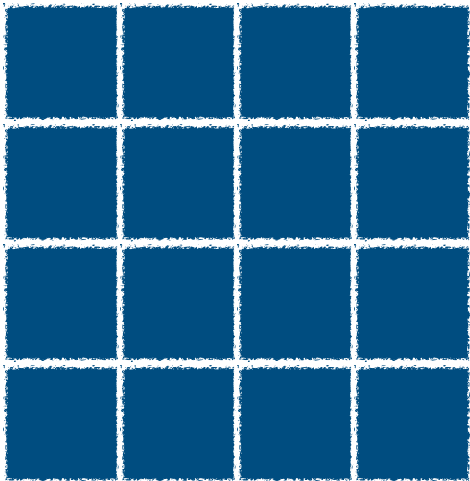
Trip count

Loop nest level



Gather Examples

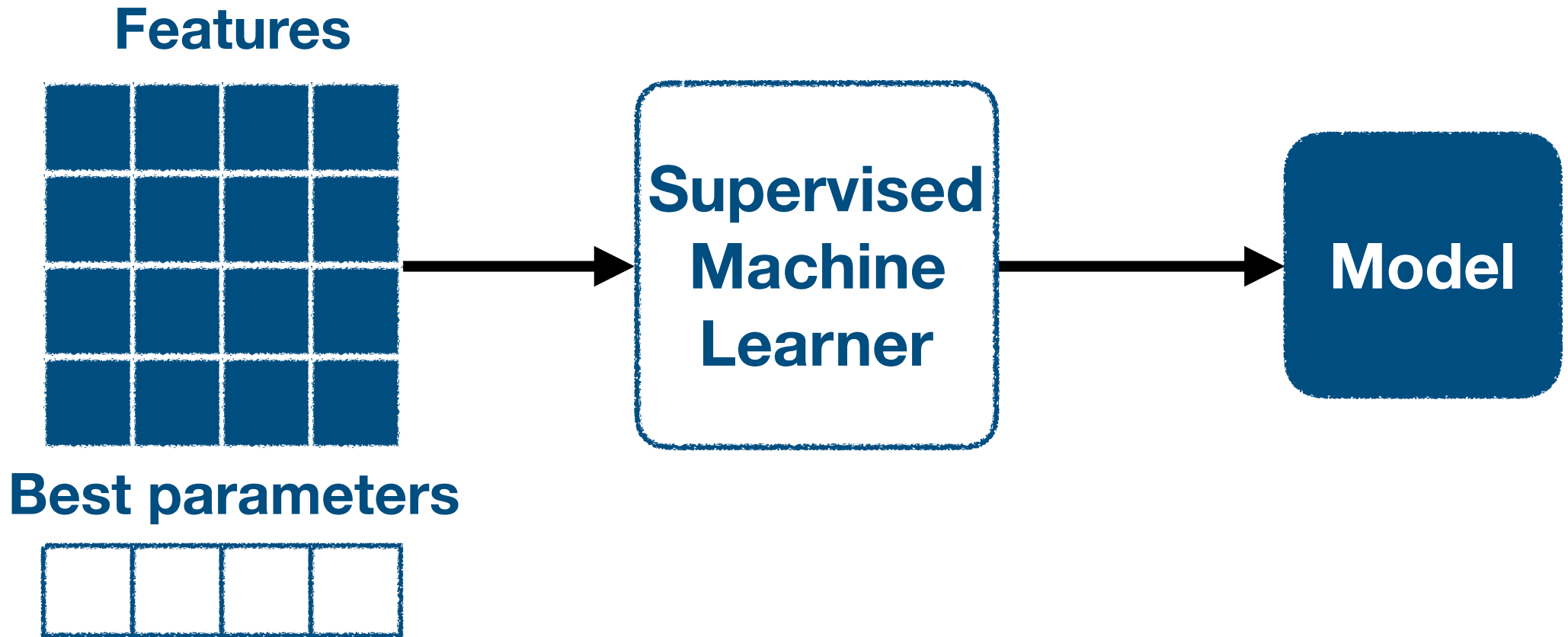
Features



Best parameters



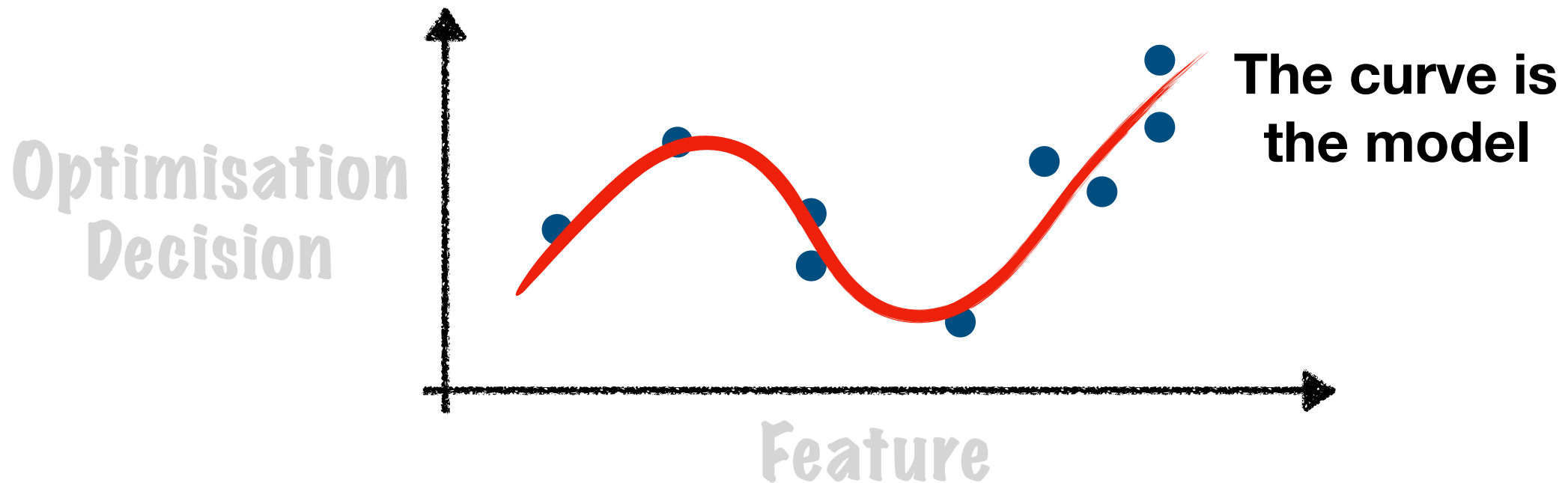
Learn a Model



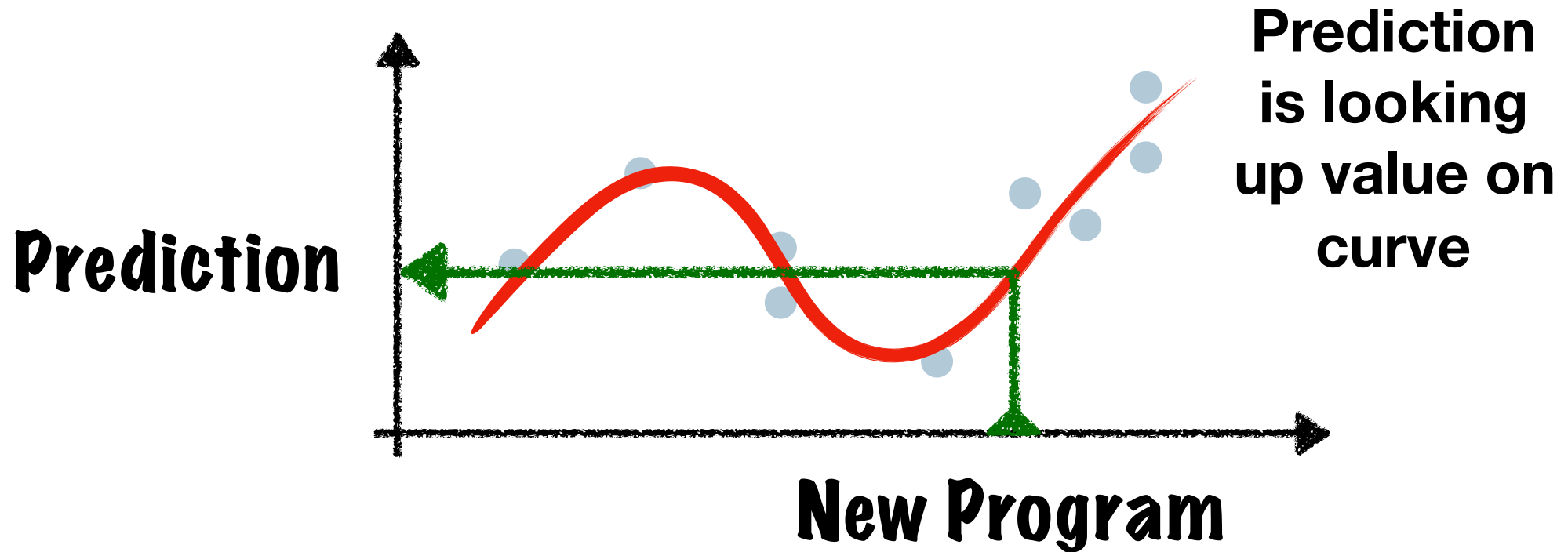
What is a Model?



What is a Model?



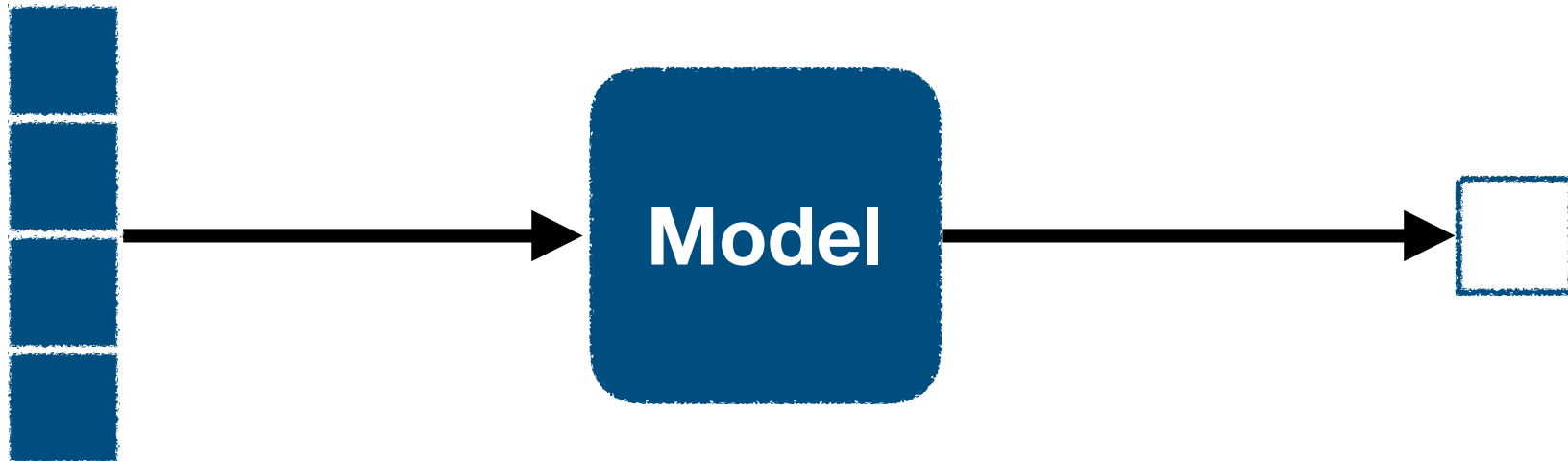
What is a Model?



Use the Model

New Program
Features

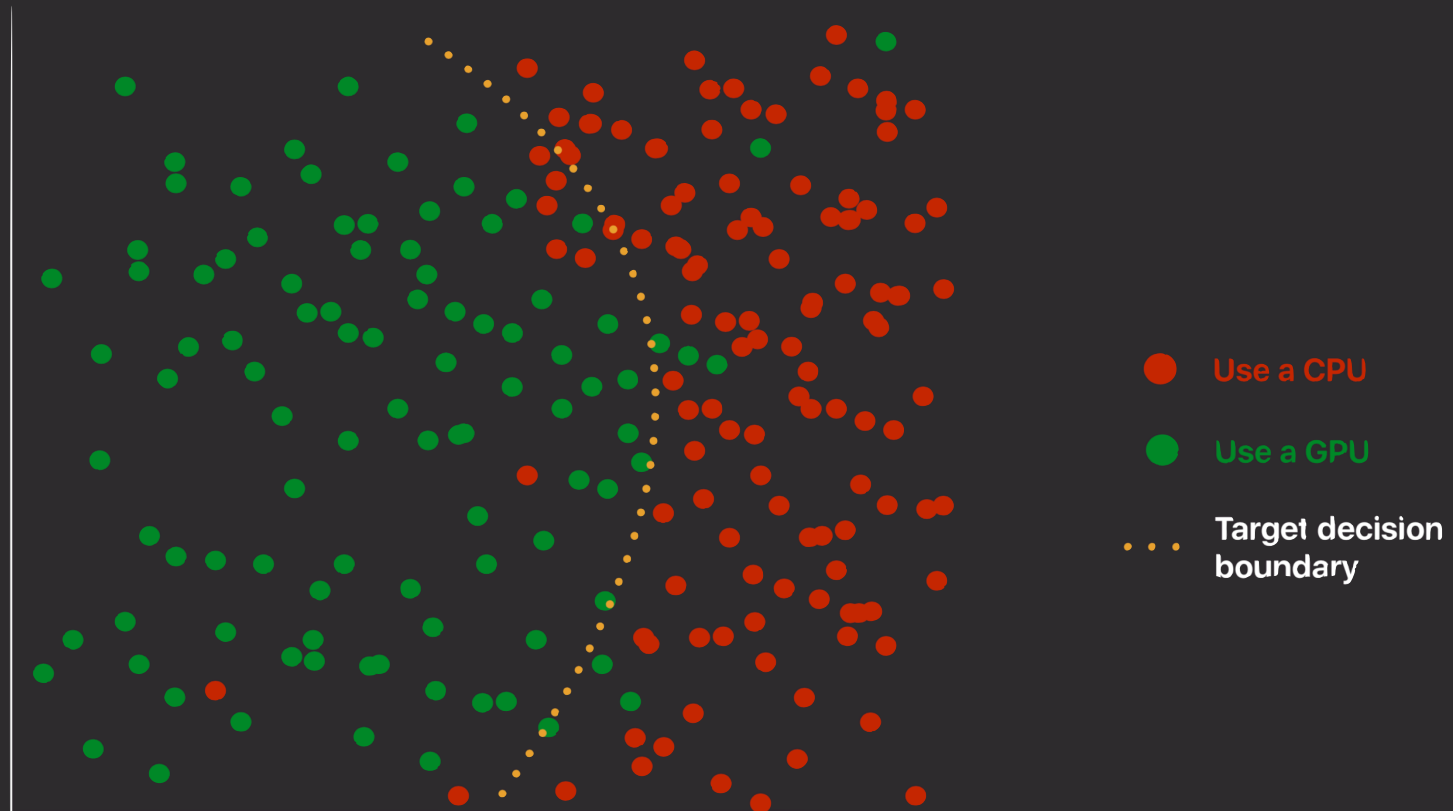
Predicted
parameters



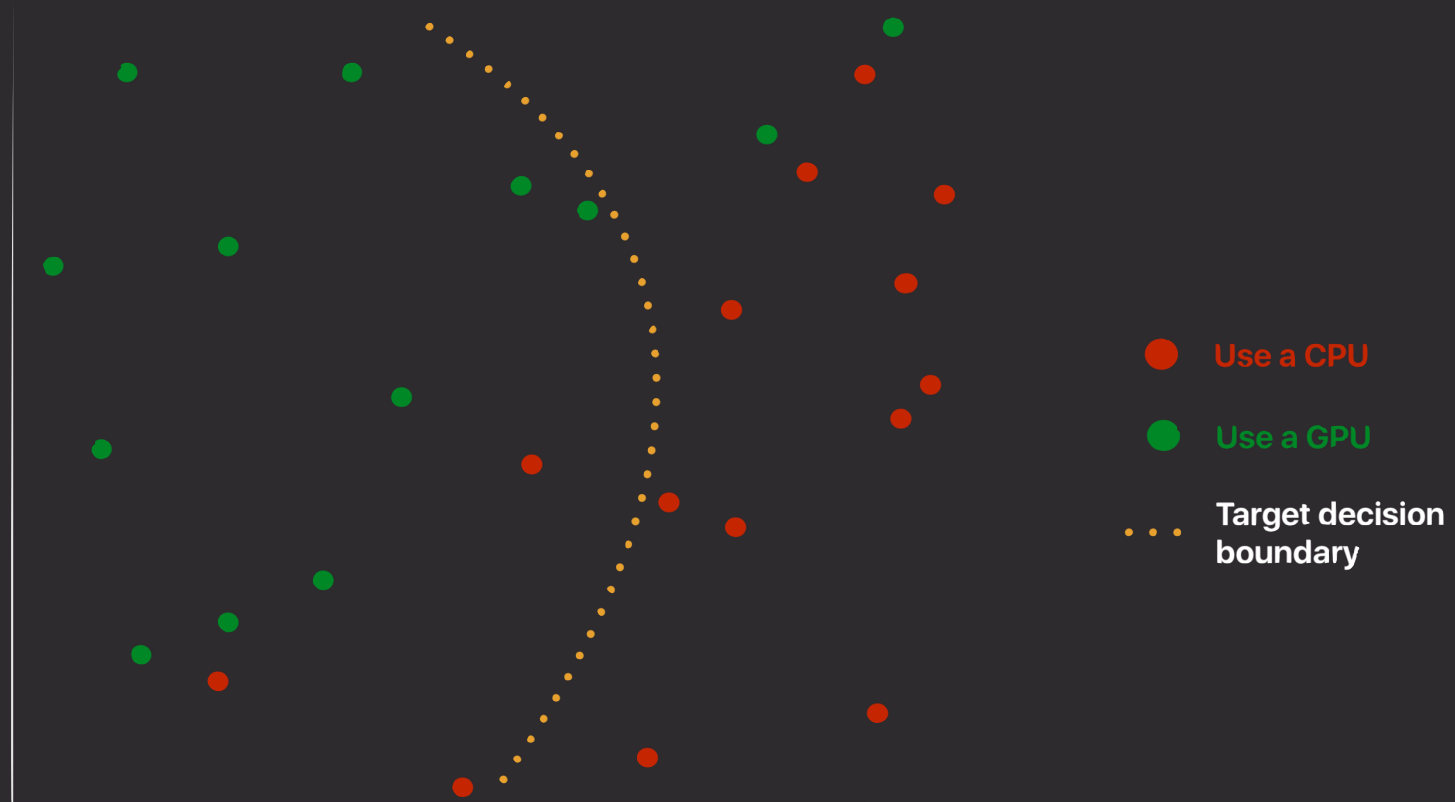
Overview

- Machine Learning for Compilers
- **Generating Benchmarks**
- Deep Learned Heuristics
- Deep Fuzzing Compiler Testing
- Future Work

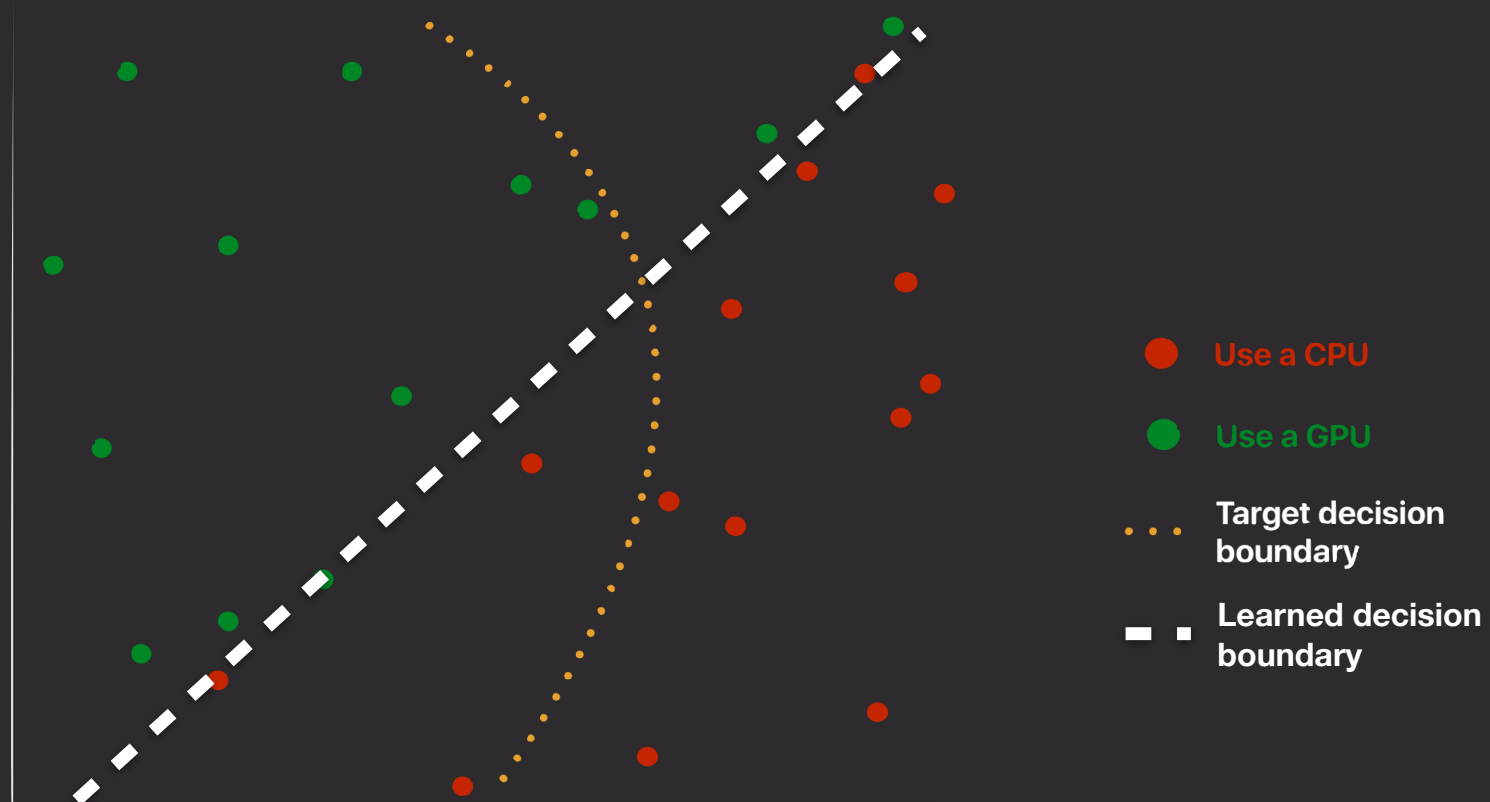
What we want.



What we get.

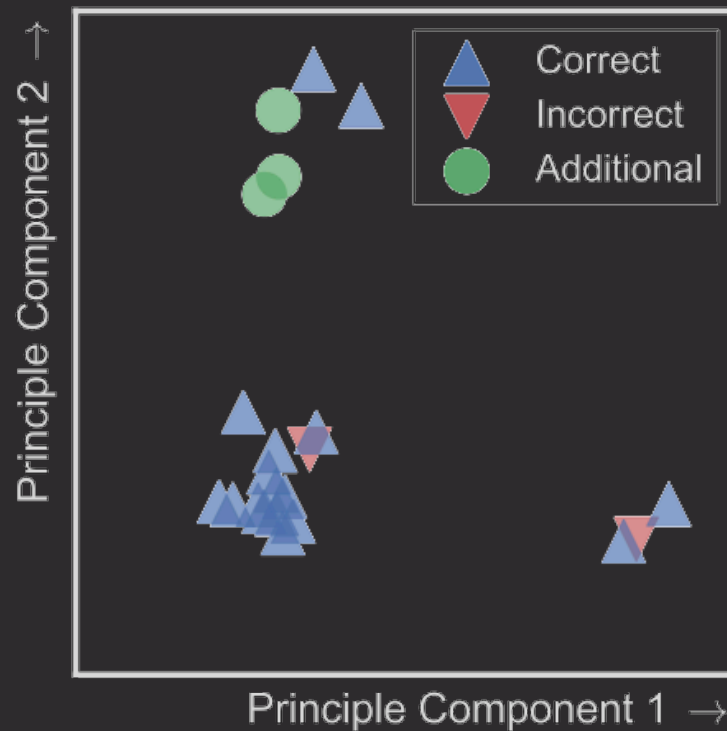
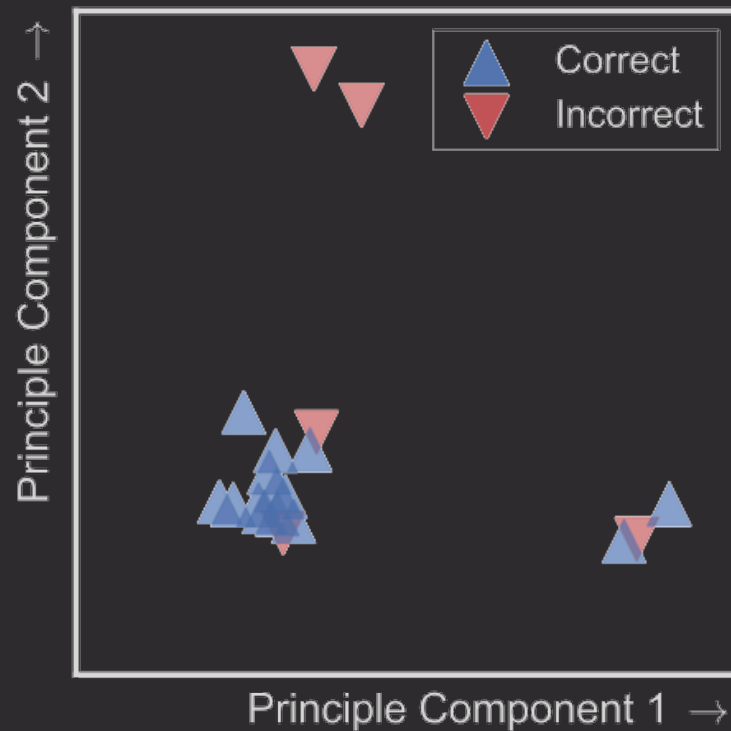


Learn the Wrong Thing!



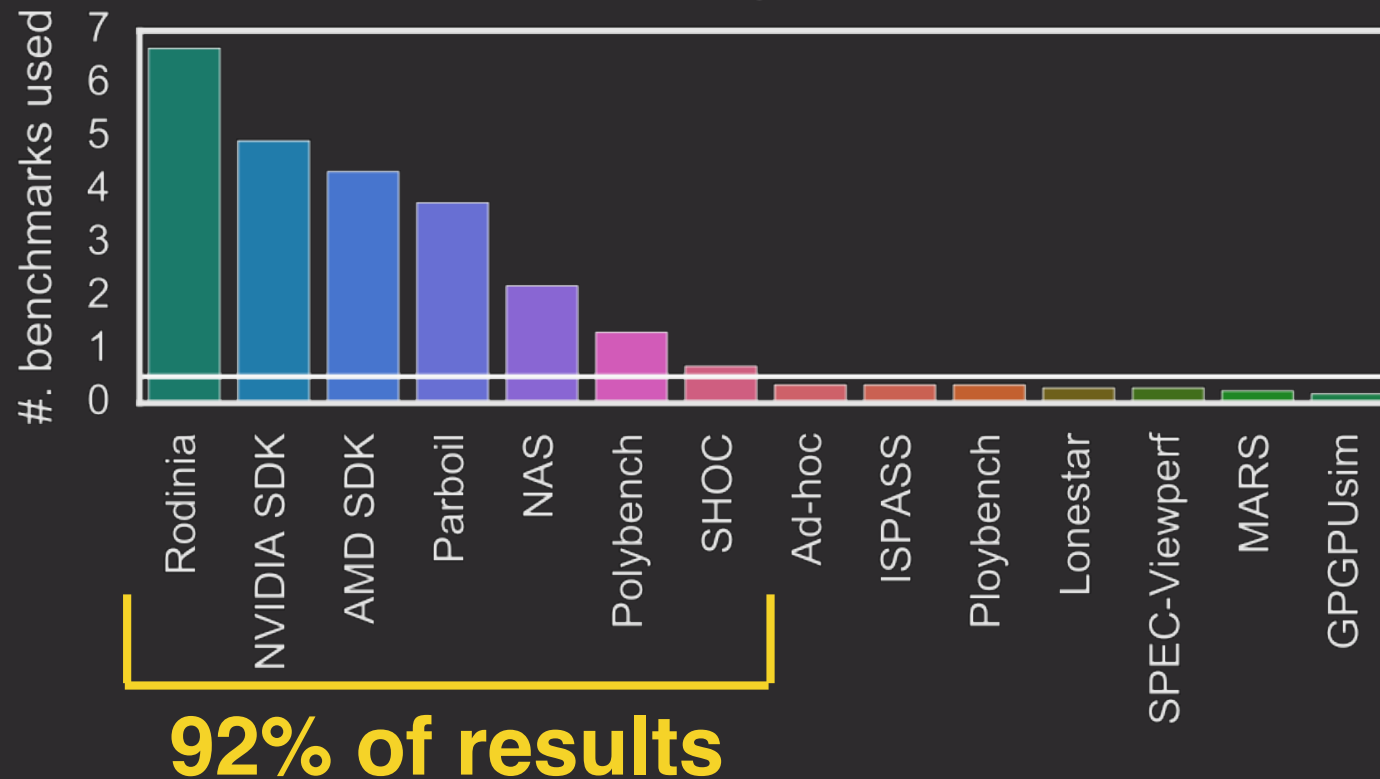
problem statement

1. more benchmarks = better models



problem statement

1. more benchmarks = better models
2. there aren't enough benchmarks



problem statement

1. more benchmarks = better models
2. there aren't enough benchmarks

avg compiler paper = 17

Iris dataset = 150

MNIST dataset = 60,000

ImageNet dataset = 10,000,000

problem statement

1. more benchmarks = better models
2. there aren't enough benchmarks
3. benchmarks must be diverse

	AMD	NPB	NVIDIA	Parboil	Polybench	Rodinia	SHOC
AMD	-	38.0%	74.5%	76.7%	21.7%	45.8%	35.9%
NPB	22.7%	-	45.3%	36.7%	13.4%	16.1%	23.7%
NVIDIA	29.9%	37.9%	-	21.8%	78.3%	18.1%	63.2%
Parboil	89.2%	28.2%	28.2%	-	41.3%	73.0%	33.8%
Polybench	58.6%	30.8%	45.3%	11.5%	-	43.9%	12.1%
Rodinia	39.8%	36.4%	29.7%	36.5%	46.1%	-	59.9%
SHOC	42.9%	71.5%	74.1%	41.4%	35.7%	81.0%	-

problem statement

- 1. more benchmarks = better models**
- 2. there aren't enough benchmarks**
- 3. benchmarks must be diverse**

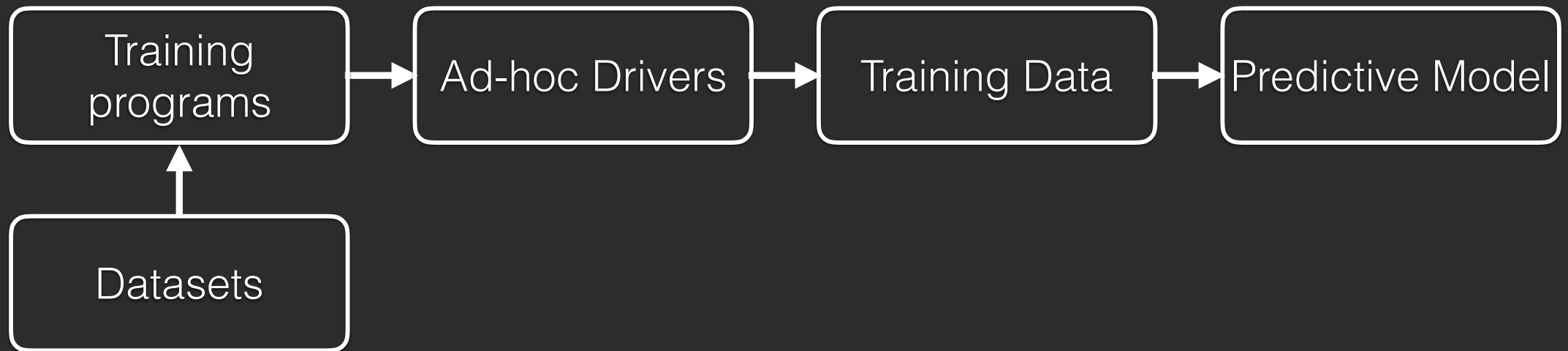


Contributions

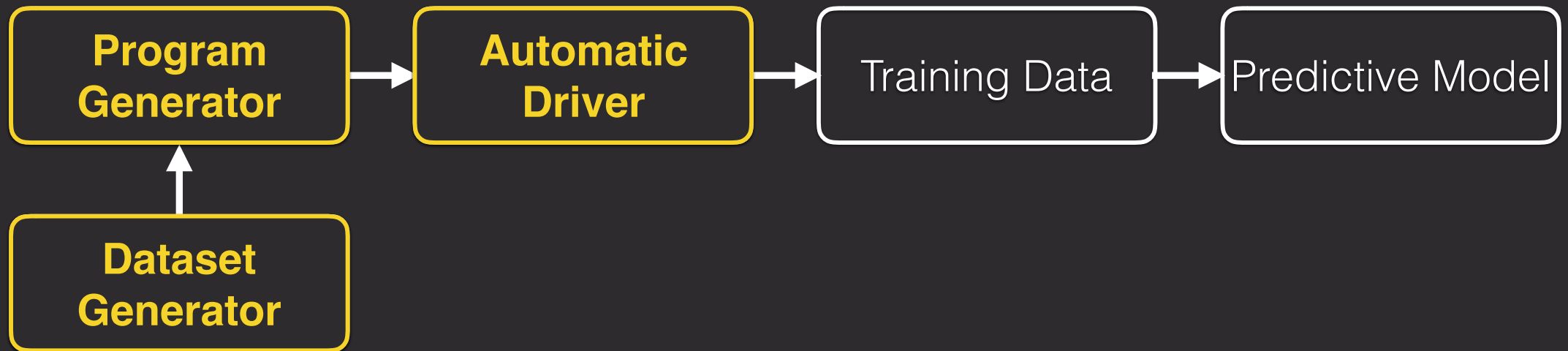
Human-like program generator

Model produces code
4.3x faster
than state of the art

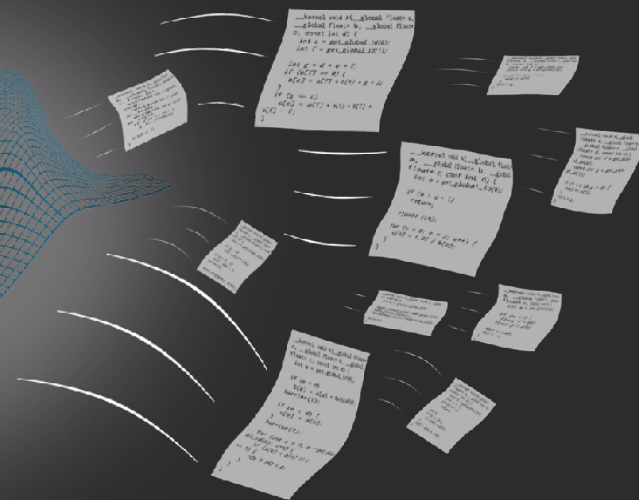
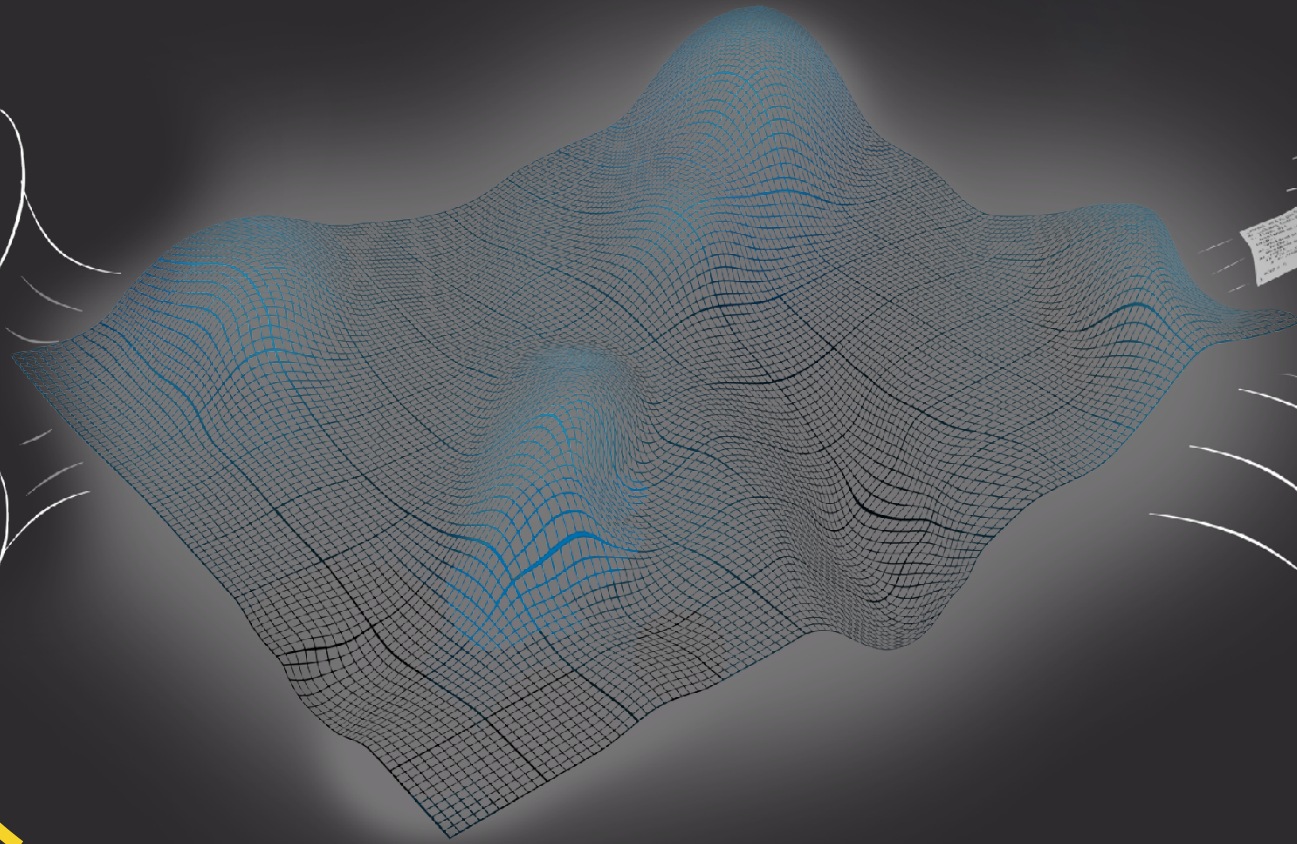
old approach



our approach

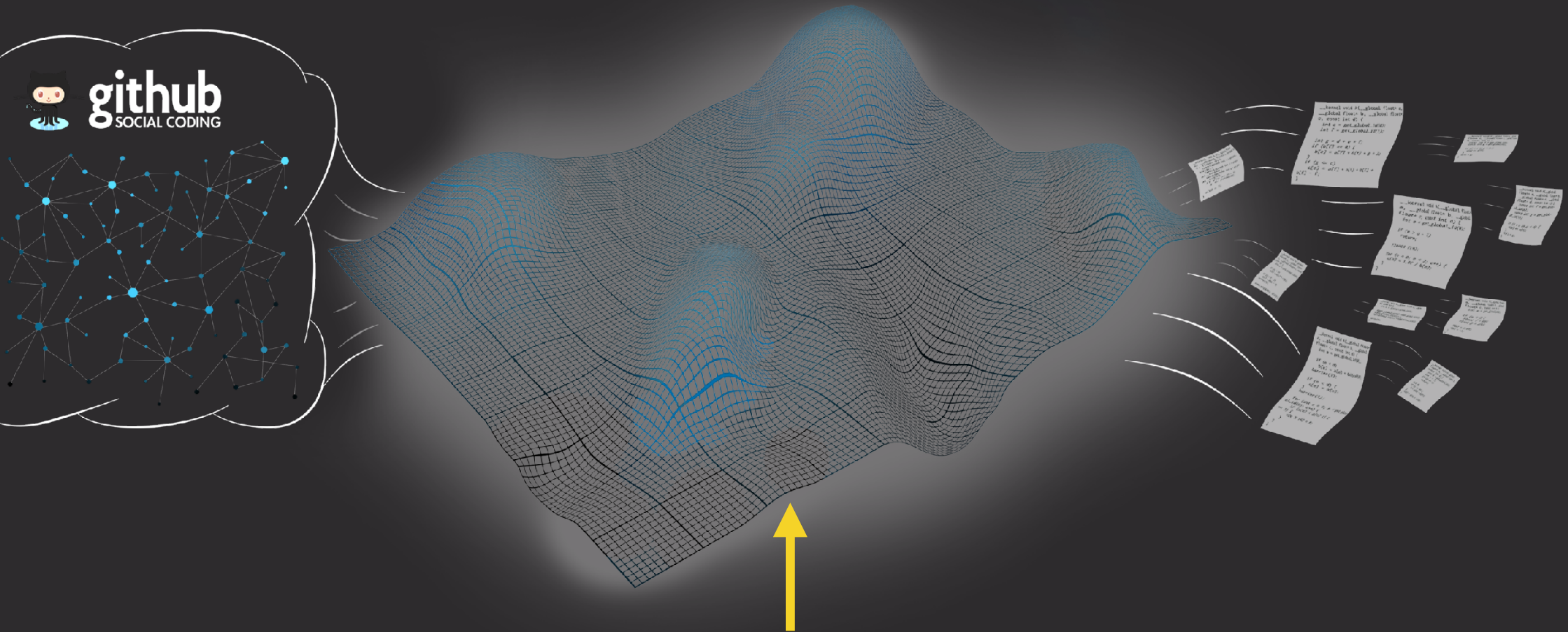


our approach



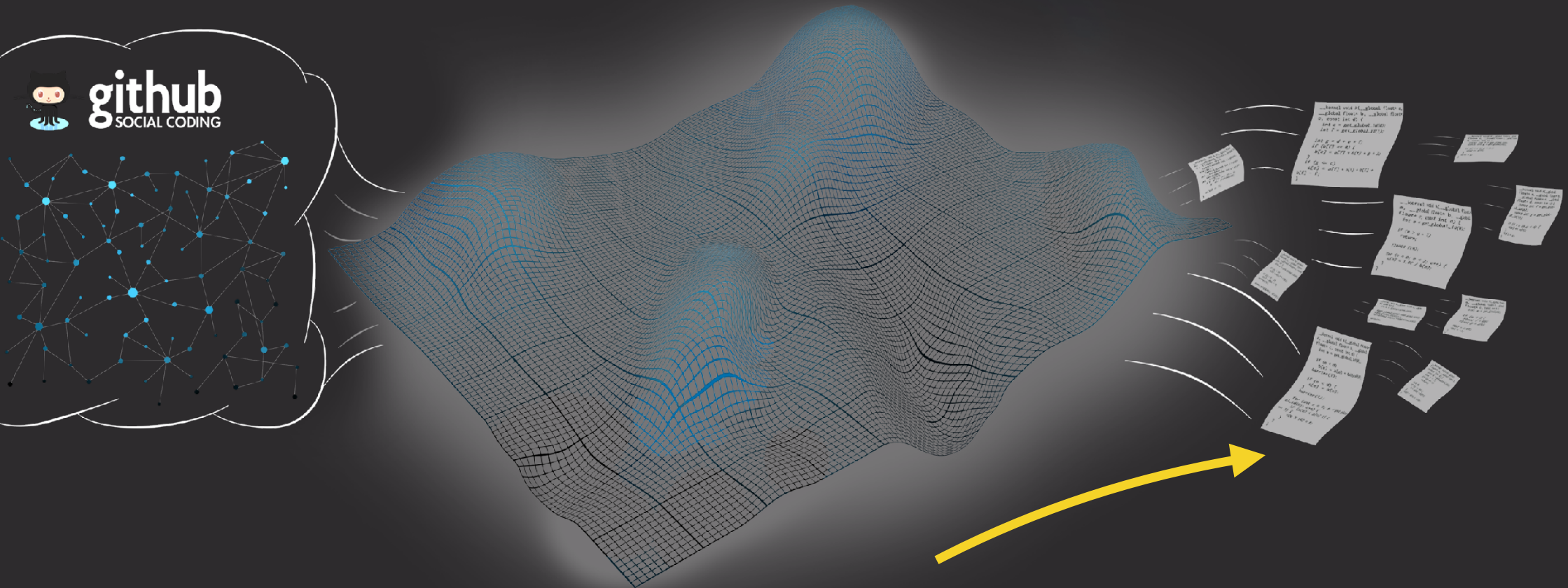
mine code from web

our approach

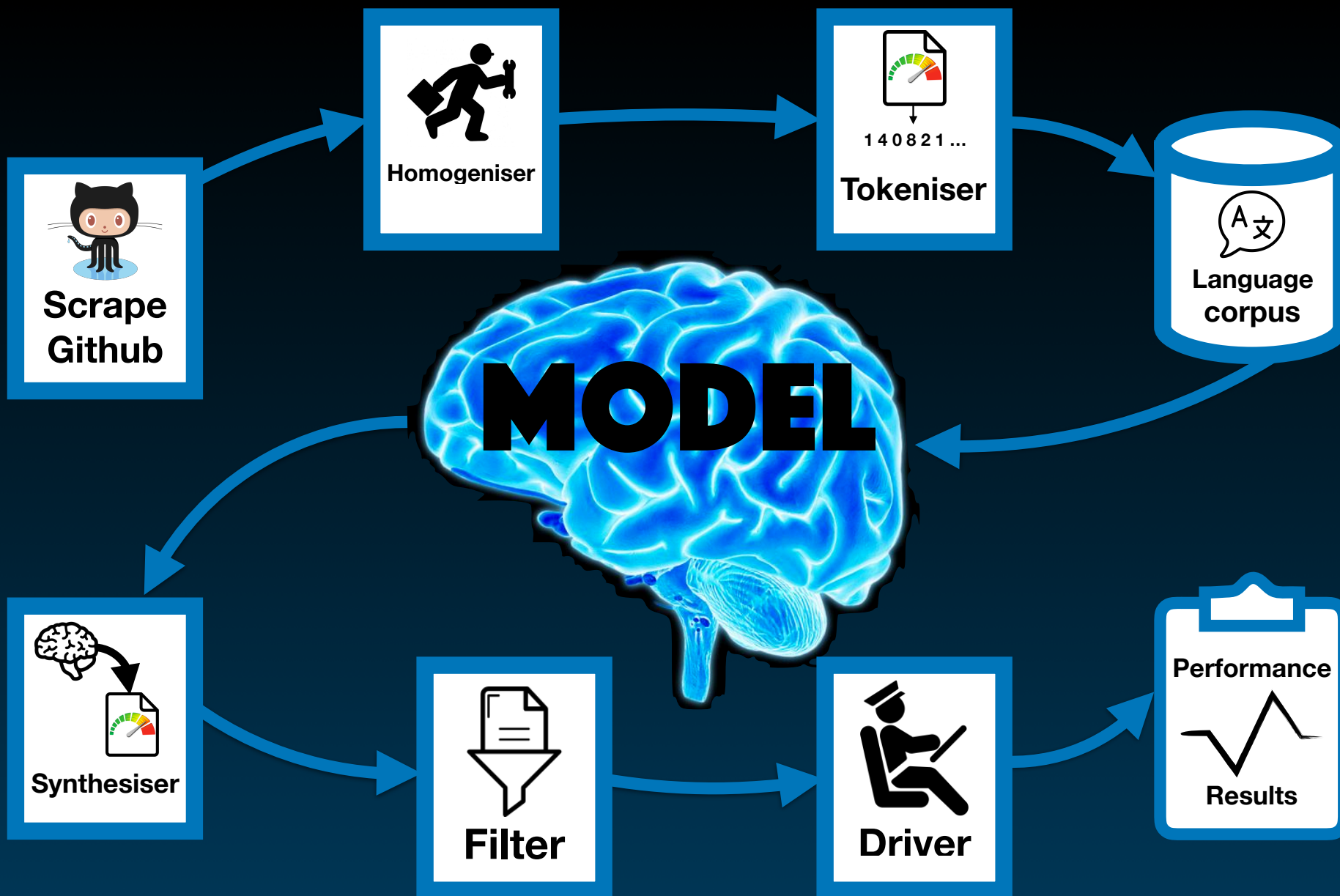


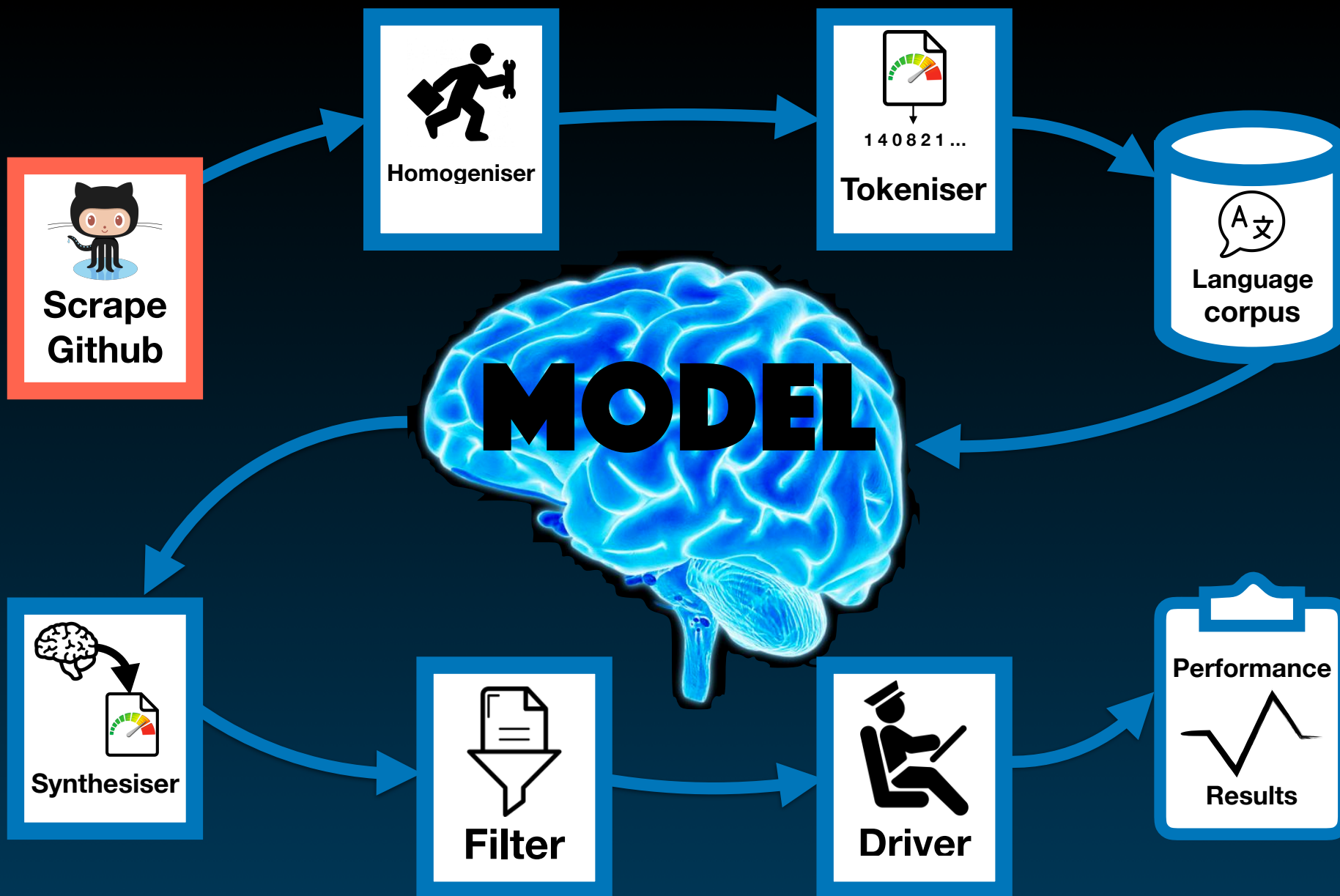
model source distr.

our approach



sample lang. model





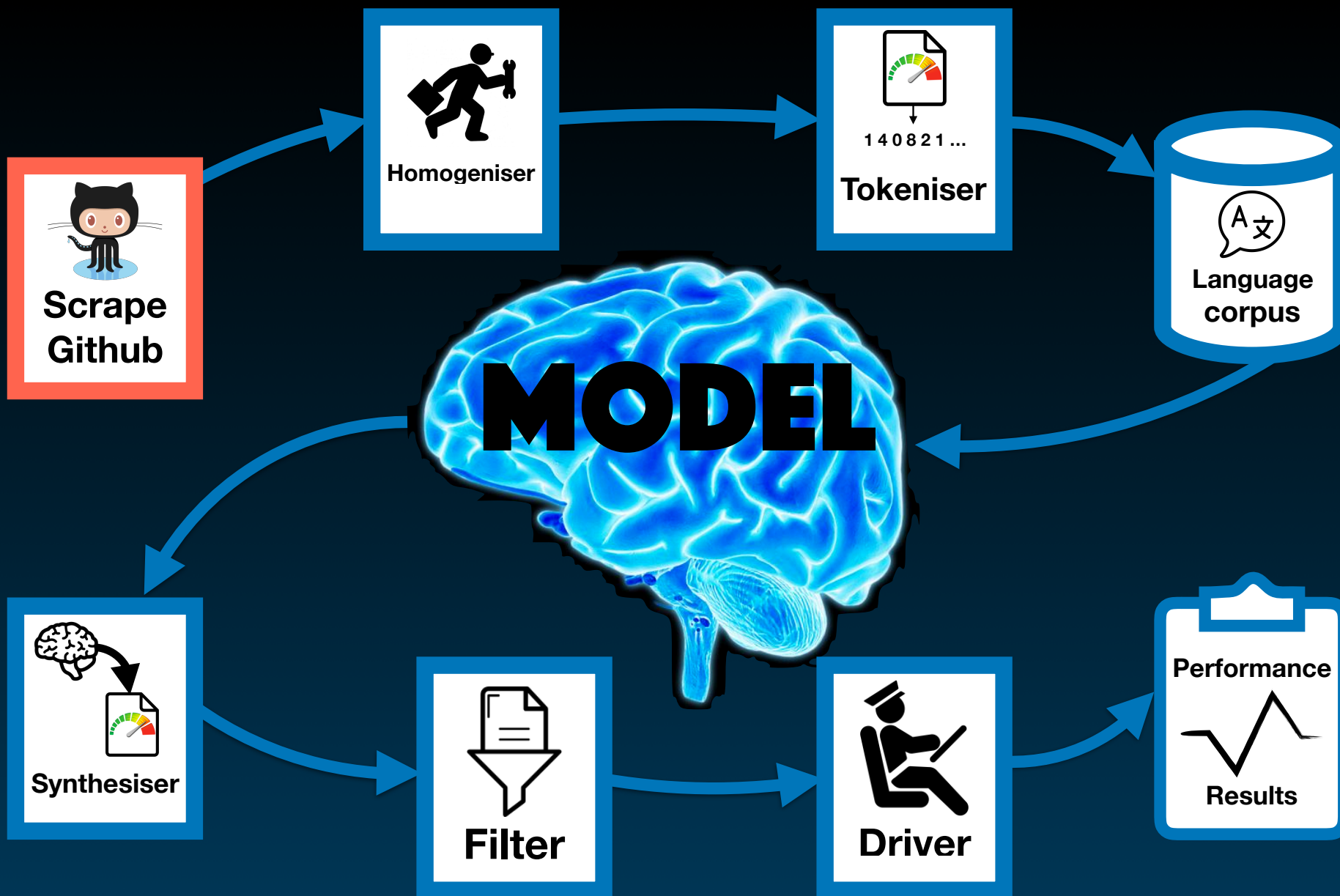
Infer the common usage of a PL from samples.

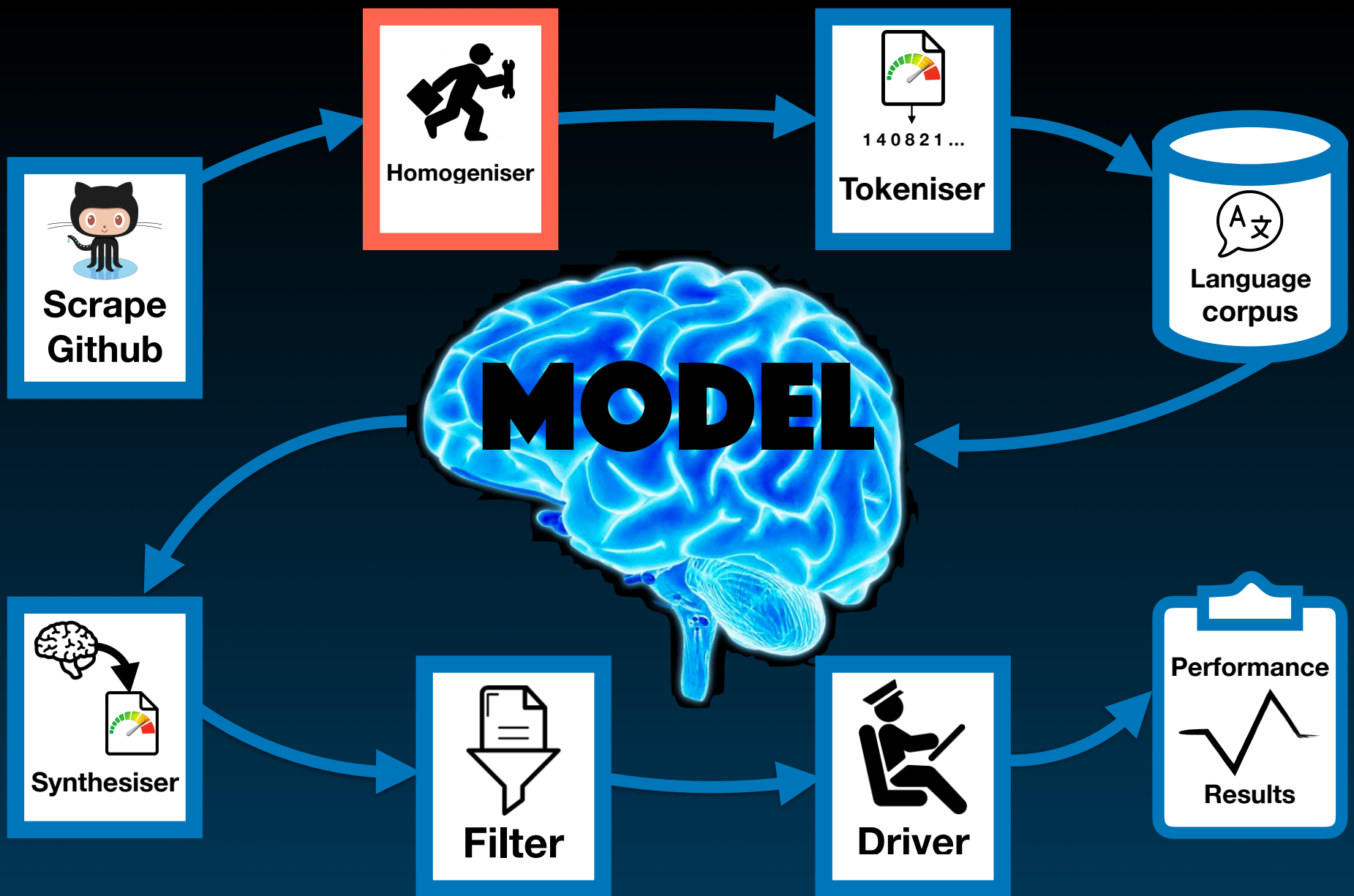
Huge repository of public knowledge: 

And they have an API :-)

2.8 million lines of OpenCL

```
$ curl https://api.github.com/search/repositories\?  
q=opencl&sort=stars&order=desc  
{  
  "total_count": 3155,  
  "incomplete_results": false,  
  "items": [  
    {  
      "id": 7296244,  
      "name": "lwjgl3",  
      "full_name": "LWJGL/lwjgl3",
```





```
/* Copyright (C) 2014, Joe Blogs. */
#define CLAMPING
#define THRESHOLD_MAX 1.0f
float myclamp(float in) {
#ifdef CLAMPING
    return in > THRESHOLD_MAX ? THRESHOLD_MAX : in < 0.0f ? 0.0f : in;
#else
    return in;
#endif // CLAMPING
}
__kernel void findAllNodesMergedAabb(__global float* in, __global float* out,
                                     int num_elems)
{
    // Do something really flipping cool
    int id = get_global_id(0);
    if (id < num_elems)
    {
        out[id] = myclamp(in[id]);
    }
}
```

```
/* Copyright (C) 2014, Joe Blogs. */
```

```
#define CLAMPING
```

```
#define THRESHOLD_MAX 1.0f
```

```
float myclamp(float in) {
```

```
#ifdef CLAMPING
```

```
    return in > THRESHOLD_MAX ? THRESHOLD_MAX : in < 0.0f ? 0.0f : in;
```

```
#else
```

```
    return in;
```

```
#endif // CLAMPING
```

```
}
```

```
__kernel void findAllNodesMergedAabb(__global float* in, __global float* out,  
                                     int num_elems)
```

```
{
```

```
    // Do something really flipping cool
```

```
    int id = get_global_id(0);
```

```
    if (id < num_elems)
```

```
    {
```

```
        out[id] = myclamp(in[id]);
```

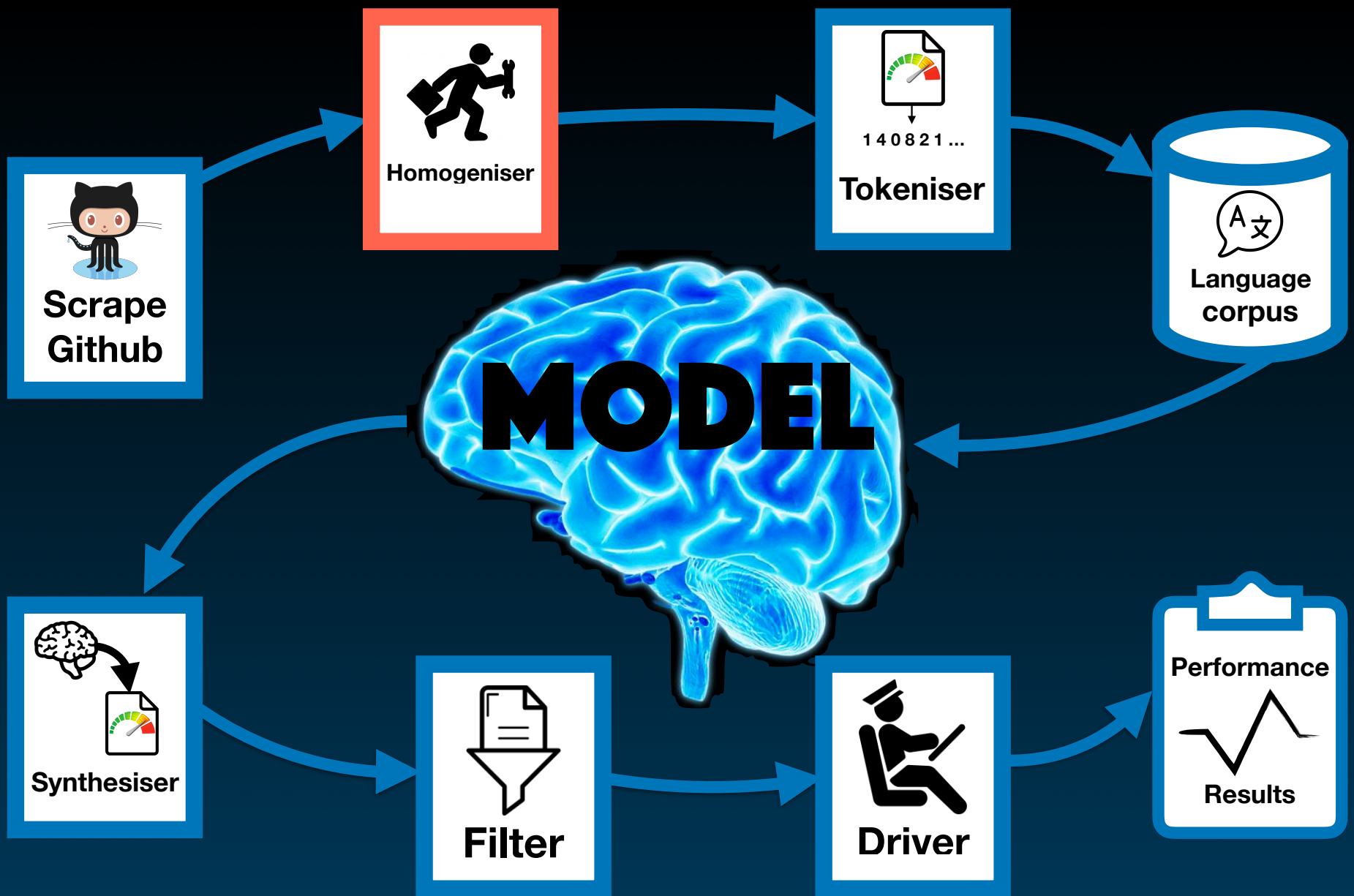
```
    }
```

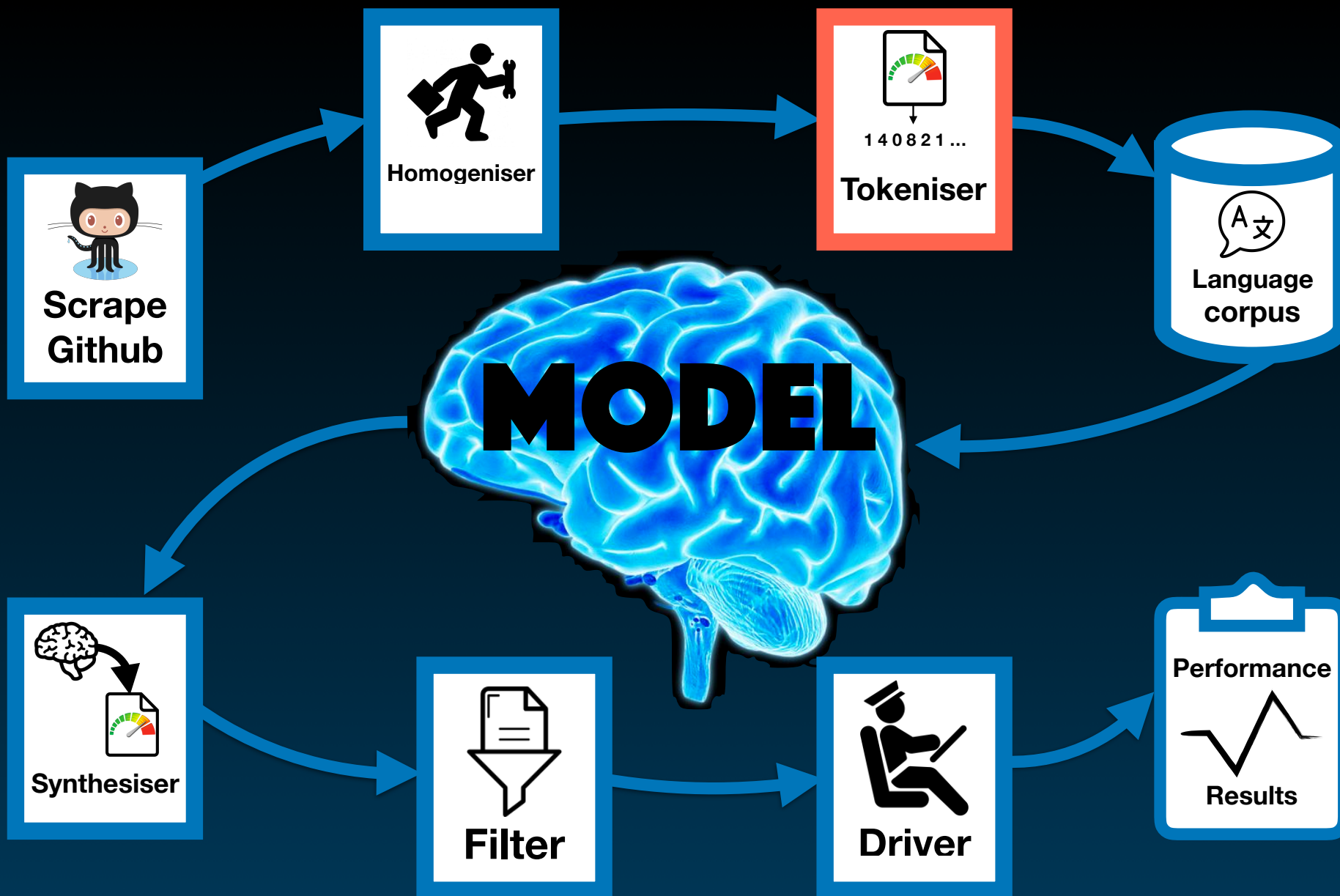
```
}
```

Strip comments
Preprocess
Rewrite names
Enforce code style

Strip comments
Preprocess
Rewrite names
Enforce code style

```
float A(float a) {  
    return a > 1.0f ? 1.0f : a < 0.0f ? 0.0f : a;  
}  
  
__kernel void B(__global float* b, __global float* c, int d) {  
    int e = get_global_id(0);  
    if (e < d) {  
        c[e] = A(b[e]);  
    }  
}
```





vocabulary encoding

```
kernel void A(global float* a, const float b) {  
    a[get_global_id(0)] *= 3.14 + b;  
}
```

Vocab:

Encoded:

vocabulary encoding

```
kernel void A(global float* a, const float b) {  
    a[get_global_id(0)] *= 3.14 + b;  
}
```

Vocab:

Token	Index
kernel	0

Encoded:

0

vocabulary encoding

```
kernel_void A(global float* a, const float b) {  
    a[get_global_id(0)] *= 3.14 + b;  
}
```

Vocab:

Token	Index
kernel	0
[space]	1

Encoded:

0 1

vocabulary encoding

```
kernel void A(global float* a, const float b) {  
    a[get_global_id(0)] *= 3.14 + b;  
}
```

Vocab:

Token	Index
kernel	0
[space]	1
void	2

Encoded:

0 1 2

vocabulary encoding

```
kernel void A(global float* a, const float b) {  
    a[get_global_id(0)] *= 3.14 + b;  
}
```

Vocab:

Token	Index
kernel	0
[space]	1
void	2

Encoded:

0 1 2 1

vocabulary encoding

```
kernel void A(global float* a, const float b) {  
    a[get_global_id(0)] *= 3.14 + b;  
}
```

Vocab:

Token	Index
kernel	0
[space]	1
void	2
A	3

Encoded:

0 1 2 1 3

vocabulary encoding

```
kernel void A(global float* a, const float b) {  
    a[get_global_id(0)] *= 3.14 + b;  
}
```

Vocab:

Token	Index
kernel	0
[space]	1
void	2
A	3
(4

Encoded:

0 1 2 1 3 4

vocabulary encoding

```
kernel void A(global float* a, const float b) {  
    a[get_global_id(0)] *= 3.14 + b;  
}
```

Vocab:

Token	Index
kernel	0
[space]	1
void	2
A	3
(4
global	5

Encoded:

0 1 2 1 3 4 5

vocabulary encoding

```
kernel void A(global_float* a, const float b) {  
  a[get_global_id(0)] *= 3.14 + b;  
}
```

Vocab:

Token	Index
kernel	0
[space]	1
void	2
A	3
(4
global	5

Encoded:

0 1 2 1 3 4 5 1

vocabulary encoding

```
kernel void A(global float* a, const float b) {  
    a[get_global_id(0)] *= 3.14 + b;  
}
```

Vocab:

Token	Index
kernel	0
[space]	1
void	2
A	3
(4
global	5
float	6

Encoded:

0 1 2 1 3 4 5 1 6

vocabulary encoding

```
kernel void A(global float* a, const float b) {  
    a[get_global_id(0)] *= 3.14 + b;  
}
```

Vocab:

Token	Index
kernel	0
[space]	1
void	2
A	3
(4
global	5
float	6
*	7

Encoded:

0 1 2 1 3 4 5 1 6 7

vocabulary encoding

```
kernel void A(global float*_a, const float b) {  
    a[get_global_id(0)] *= 3.14 + b;  
}
```

Vocab:

Token	Index
kernel	0
[space]	1
void	2
A	3
(4
global	5
float	6
*	7

Encoded:

0 1 2 1 3 4 5 1 6 7 1



vocabulary encoding

```
kernel void A(global float* a, const float b) {  
    a[get_global_id(0)] *= 3.14 + b;  
}
```

Vocab:

Token	Index
kernel	0
[space]	1
void	2
A	3
(4
global	5
float	6
*	7
a	8

Encoded:

0 1 2 1 3 4 5 1 6 7 1 8

vocabulary encoding

```
kernel void A(global float* a, const float b) {  
    a[get_global_id(0)] *= 3.14 + b;  
}
```

Vocab:

Token	Index
kernel	0
[space]	1
void	2
A	3
(4
global	5
float	6
*	7
a	8

Token	Index
,	9
const	10
b	11
)	12
{	13
\n	14
[15
get_global_id	16
0	17

Token	Index
]	18
=	19
3	20
.	21
1	22
4	23
+	24
;	25

Encoded:

0

1

2

1

3

4

5

1

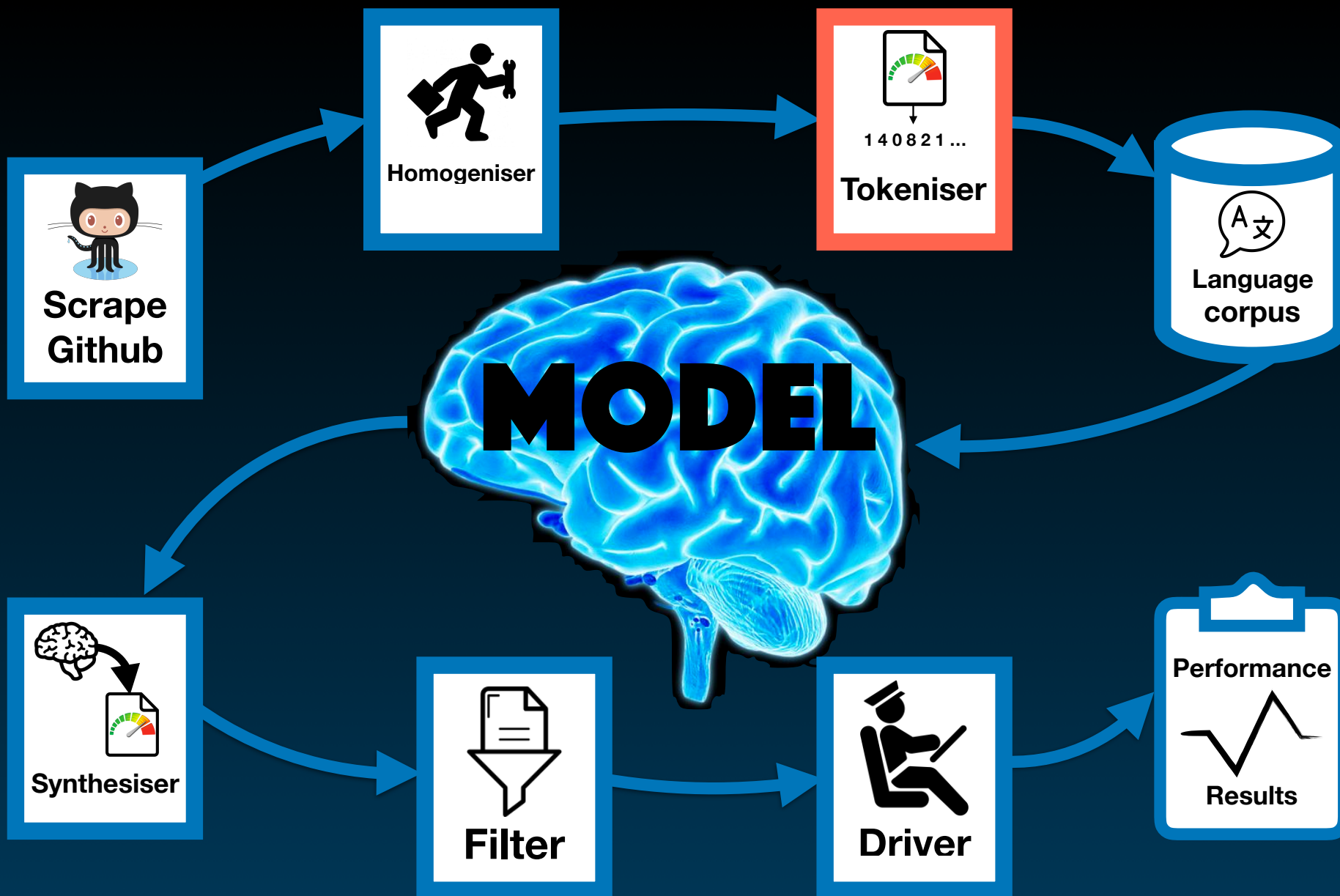
6

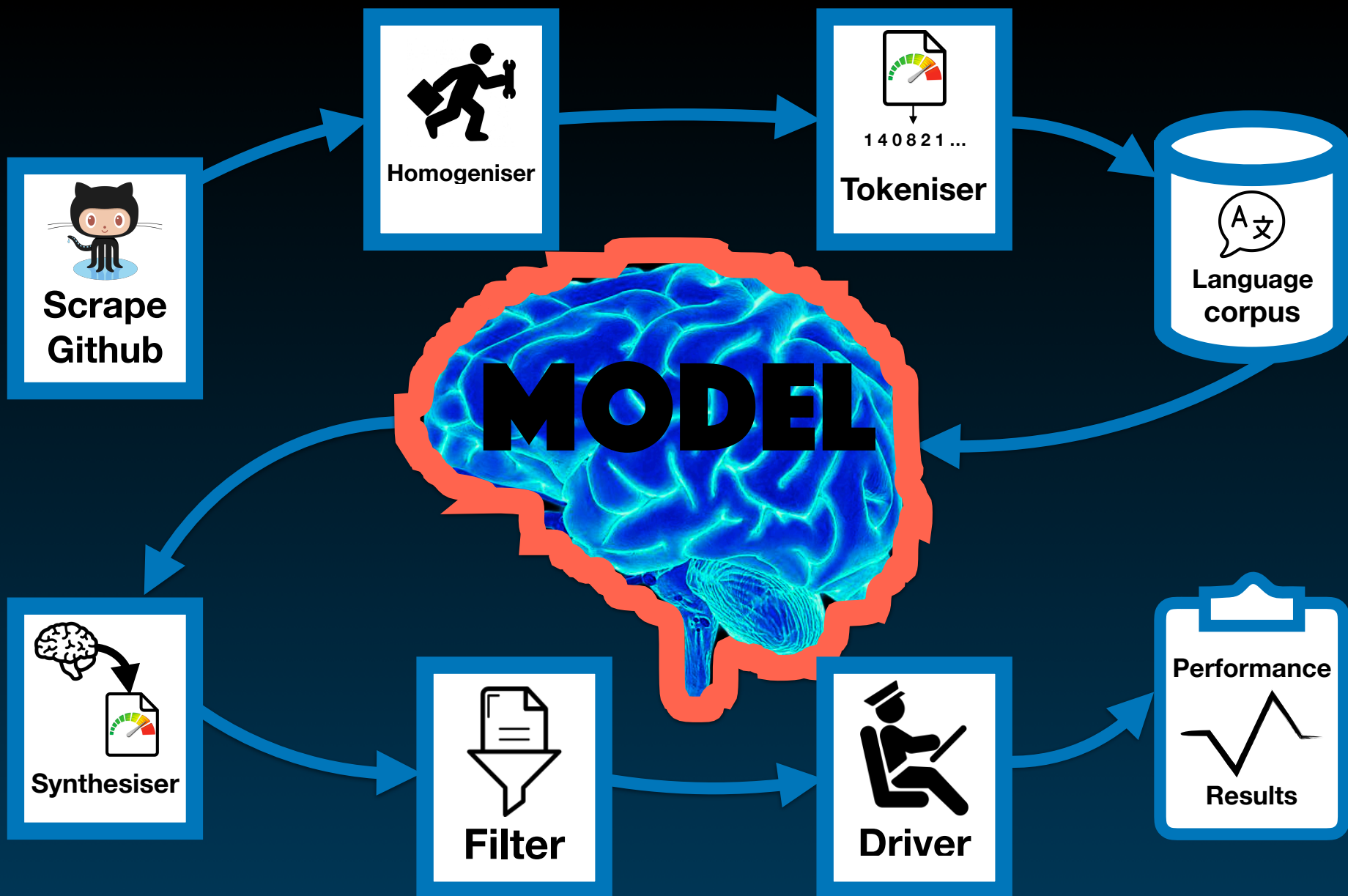
7

1

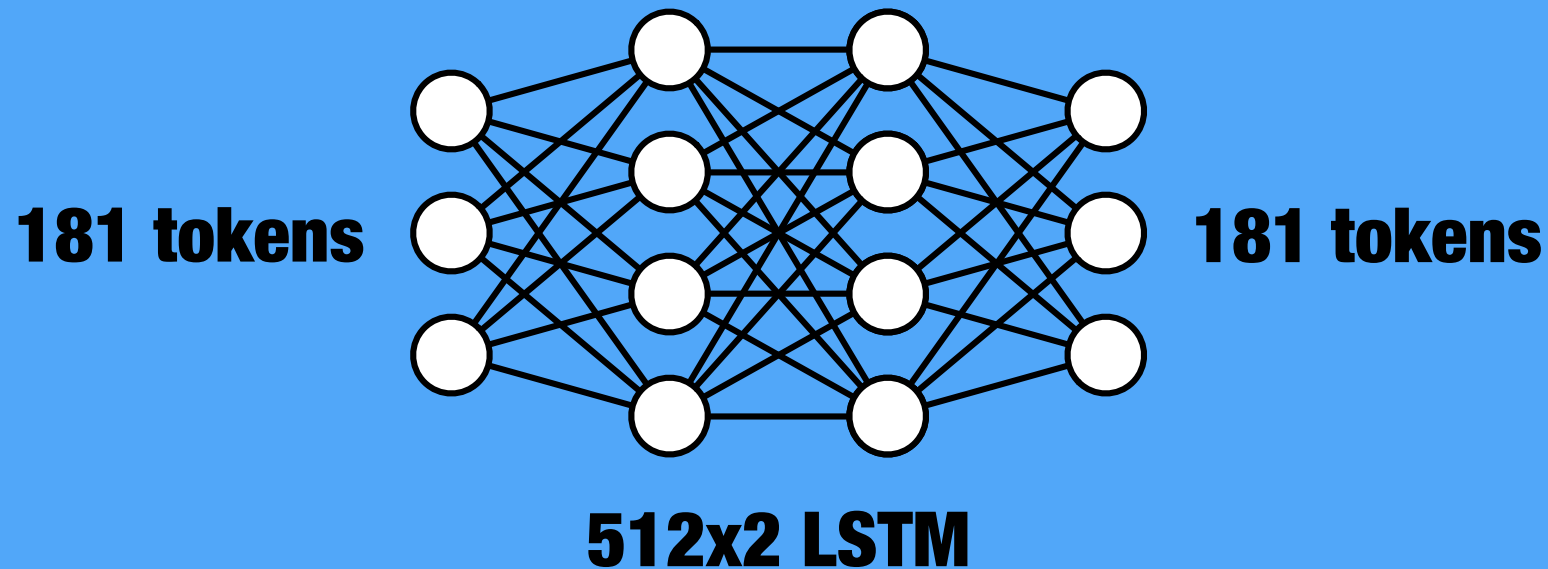
8

...





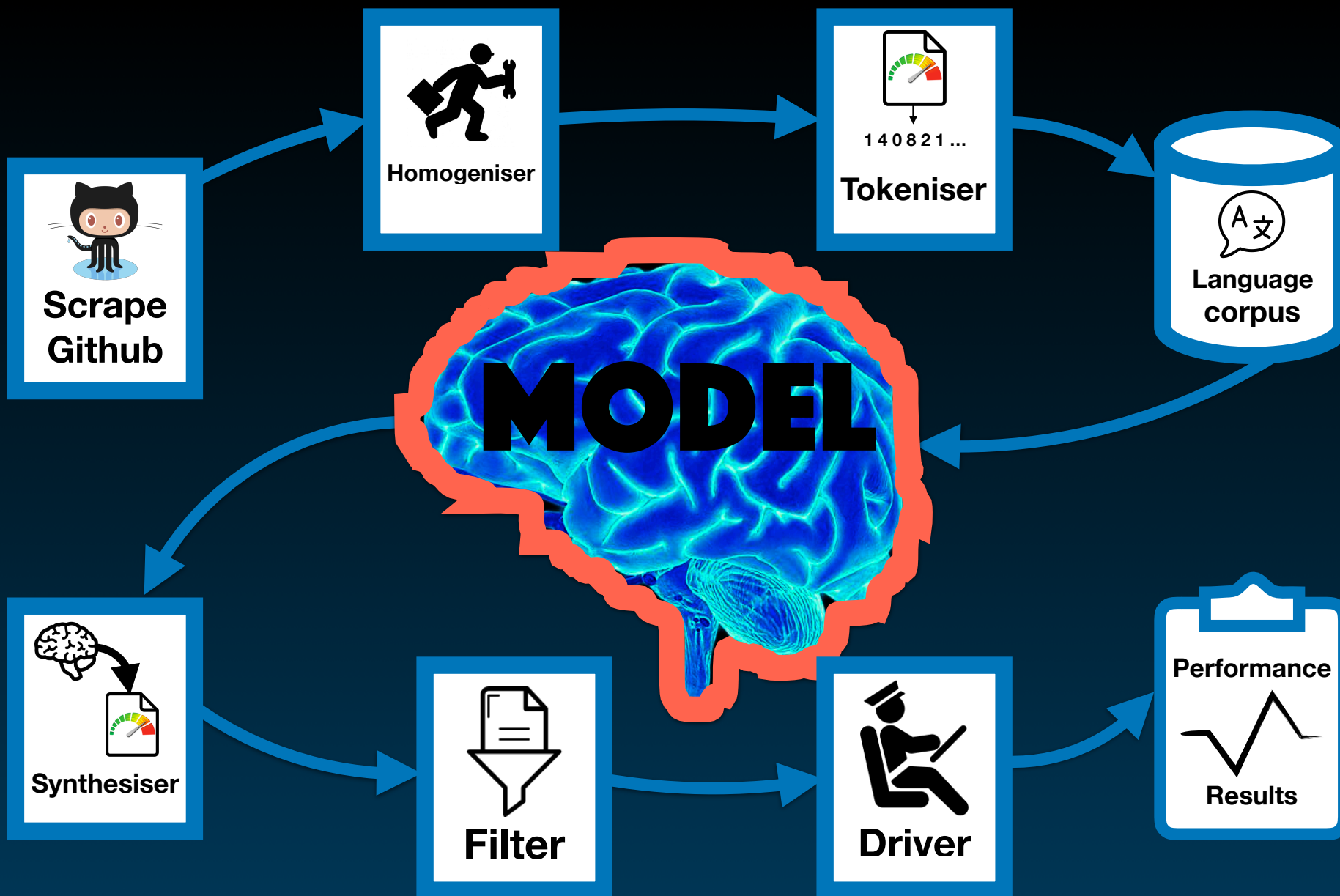
neural network

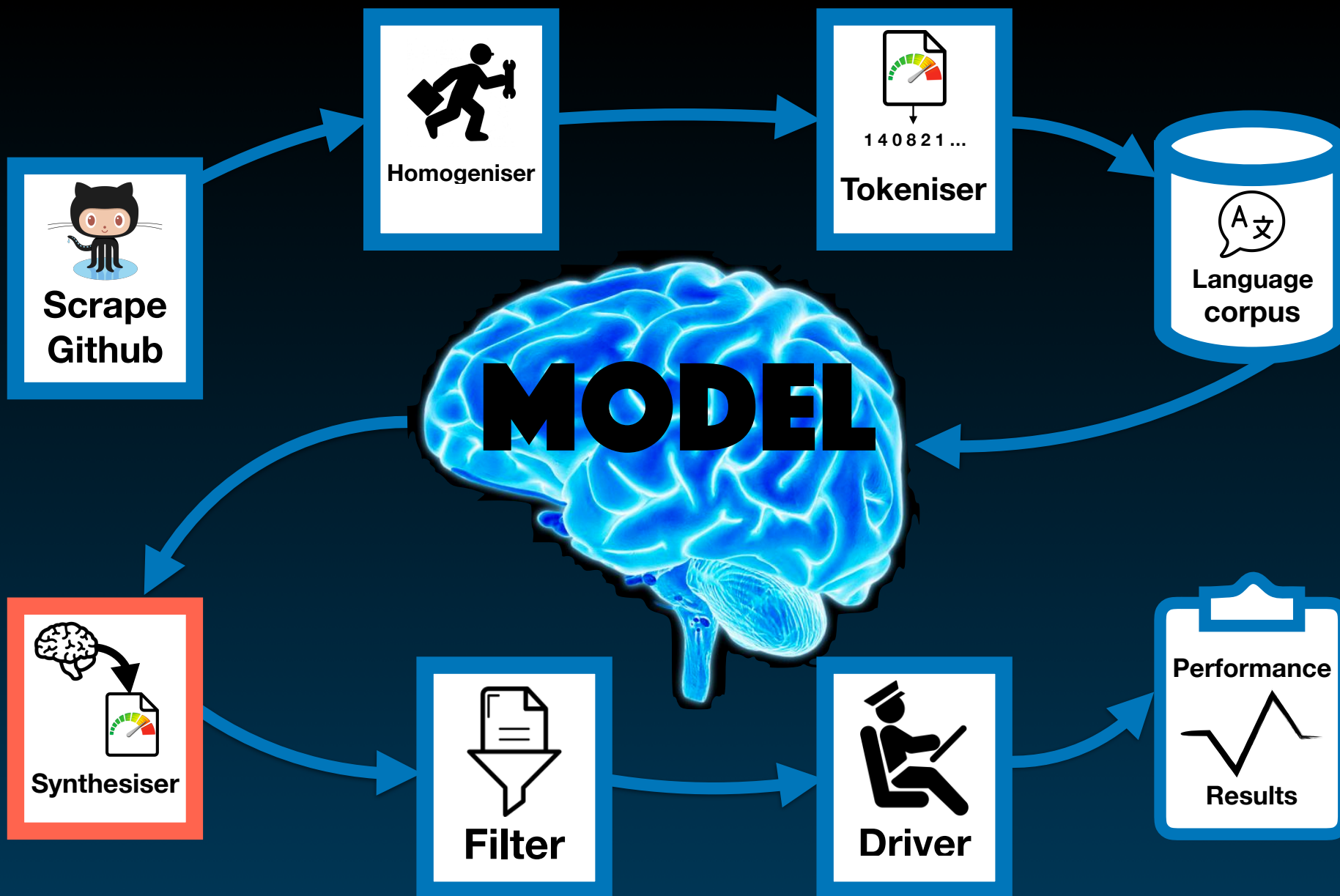


Input: 30M token corpus 0 1 2 ...

Learns probability distribution over corpus.

< 500 lines of code, 12 hours training on GPU.



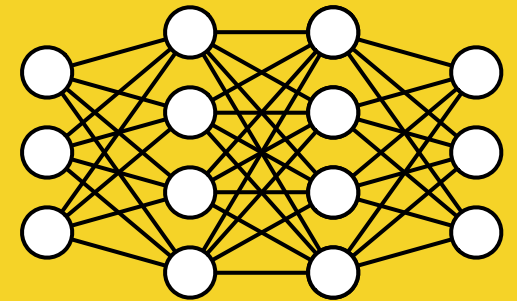


synthesizer + harness

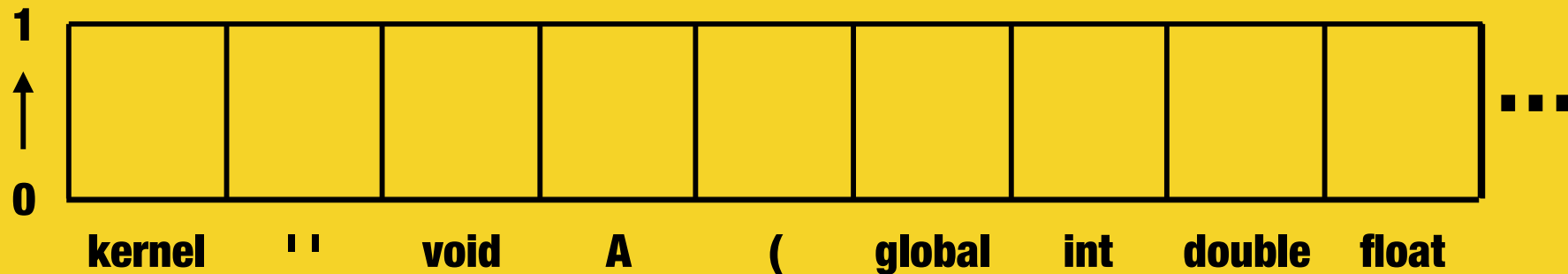
1. Seed the model with the start of a program.
2. Predict tokens until { } brackets balance.

Input:

0 1 2 1



Output:



Decoded:

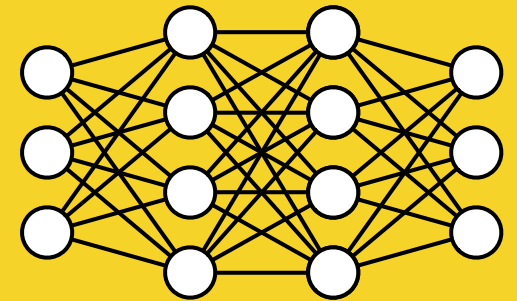
kernel void

synthesizer + harness

1. Seed the model with the start of a program.
2. Predict tokens until { } brackets balance.

Input:

0 1 2 1



Output:



Decoded:

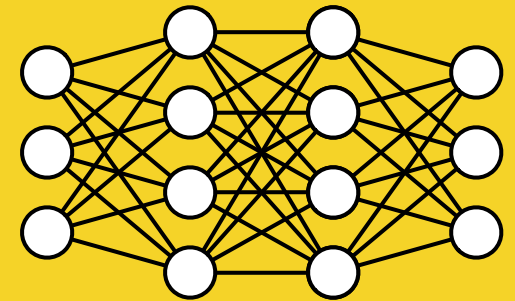
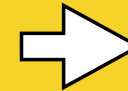
kernel void

synthesizer + harness

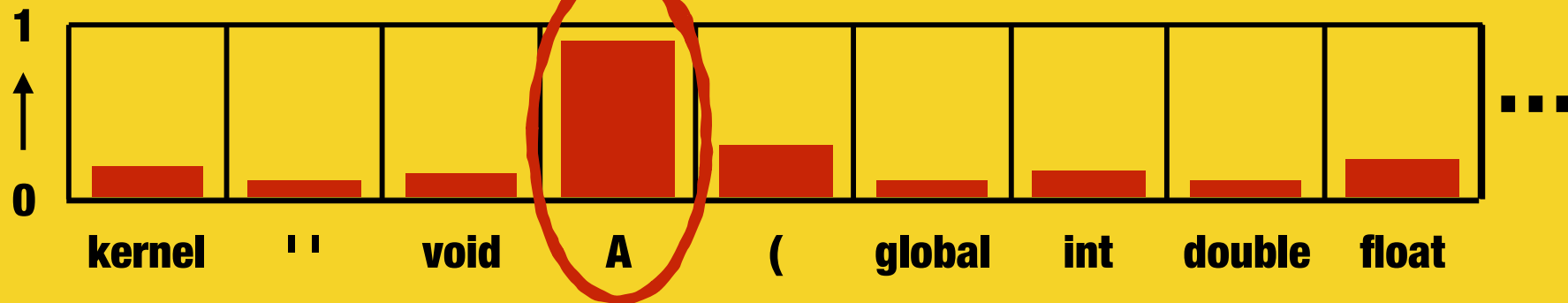
1. Seed the model with the start of a program.
2. Predict tokens until { } brackets balance.

Input:

0 1 2 1



Output:



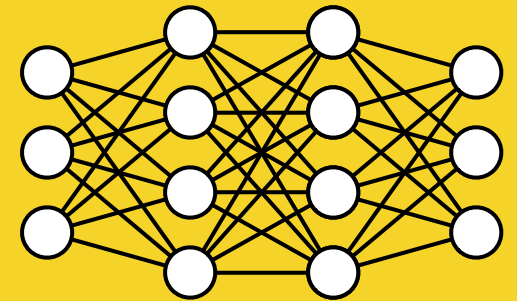
Decoded:

kernel void **A**

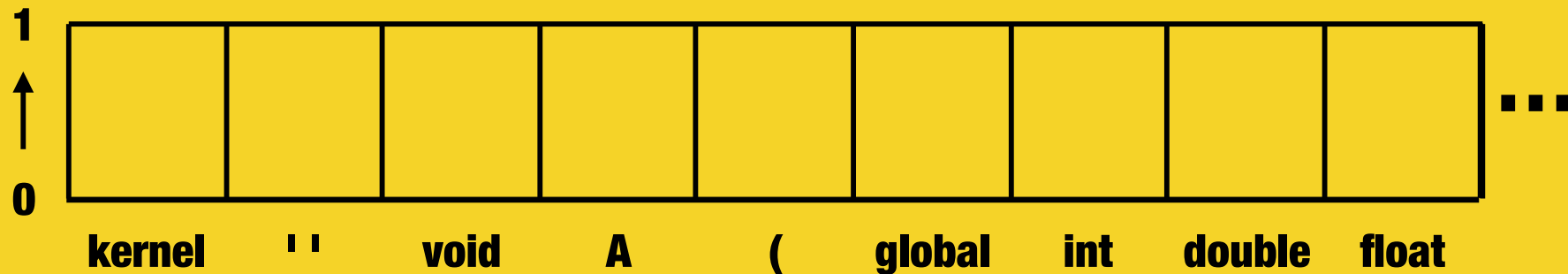
synthesizer + harness

1. Seed the model with the start of a program.
2. Predict tokens until { } brackets balance.

Input:



Output:



Decoded:

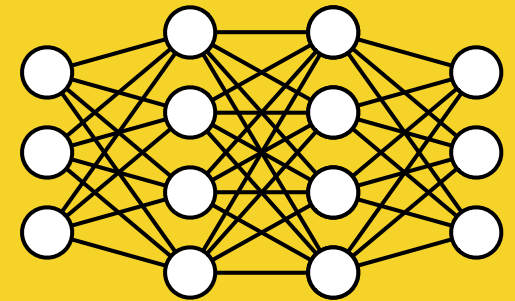
kernel void A

synthesizer + harness

1. Seed the model with the start of a program.
2. Predict tokens until { } brackets balance.

Input:

0 1 2 1 3



Output:



Decoded:

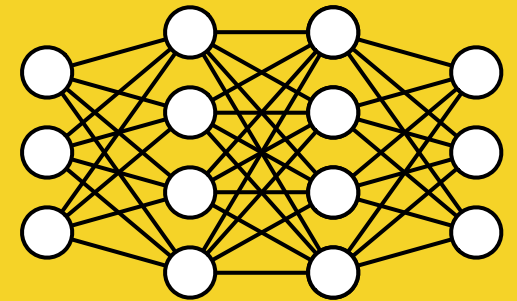
kernel void A

synthesizer + harness

1. Seed the model with the start of a program.
2. Predict tokens until { } brackets balance.

Input:

0 1 2 1 3



Output:



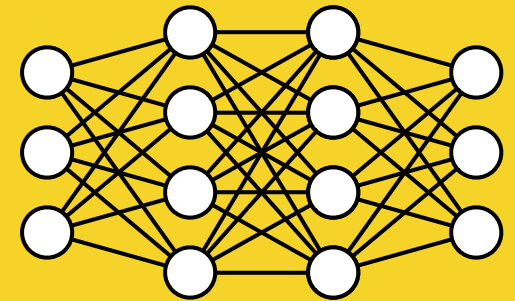
Decoded:

kernel void A(

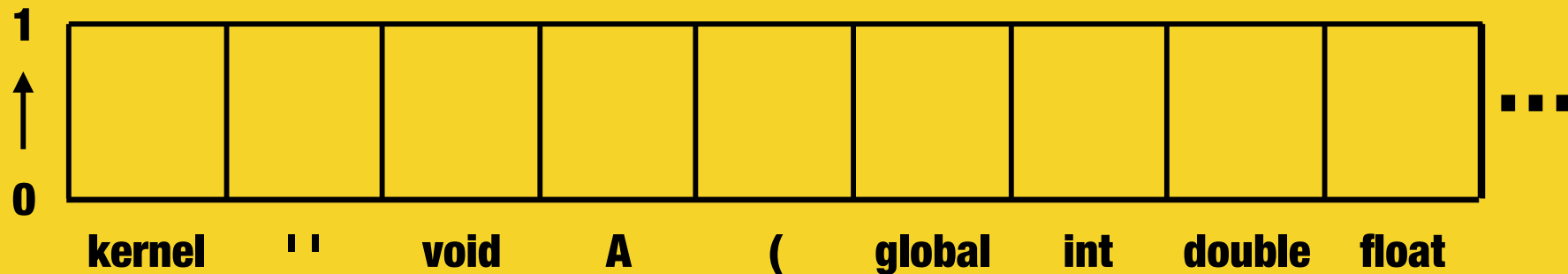
synthesizer + harness

1. Seed the model with the start of a program.
2. Predict tokens until { } brackets balance.

Input:



Output:



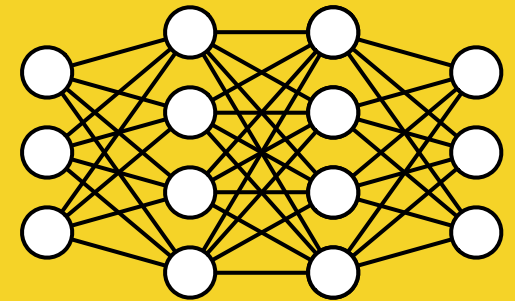
Decoded:

kernel void A(

synthesizer + harness

1. Seed the model with the start of a program.
2. Predict tokens until { } brackets balance.

Input:



Output:



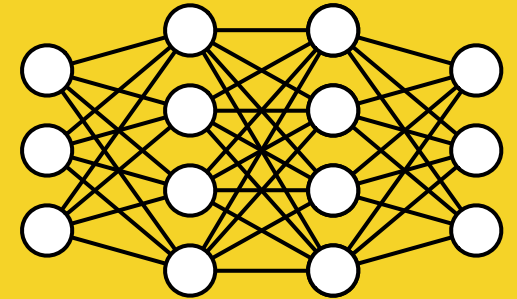
Decoded:

kernel void A(

synthesizer + harness

1. Seed the model with the start of a program.
2. Predict tokens until { } brackets balance.

Input:



Output:



Decoded:

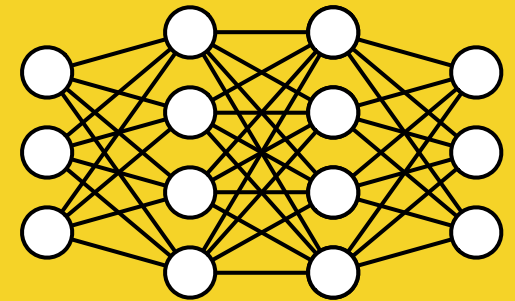
kernel void A(global

synthesizer + harness

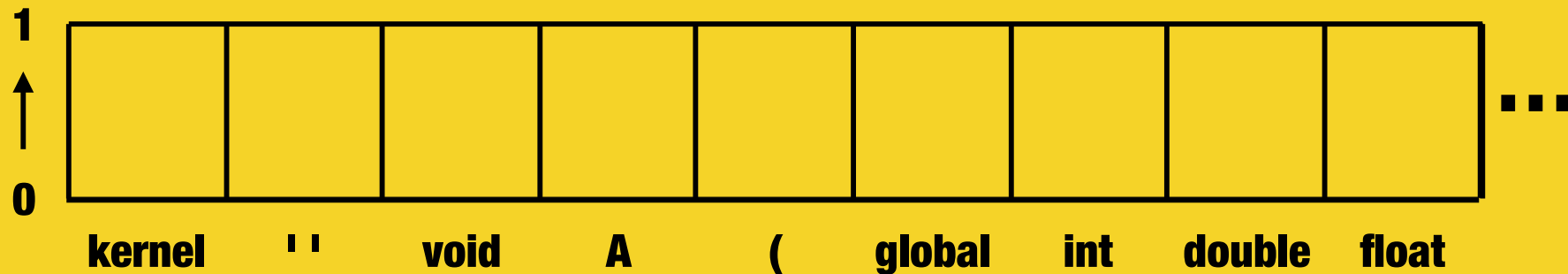
1. Seed the model with the start of a program.
2. Predict tokens until { } brackets balance.

Input:

... 2 1 3 4 5



Output:



Decoded:

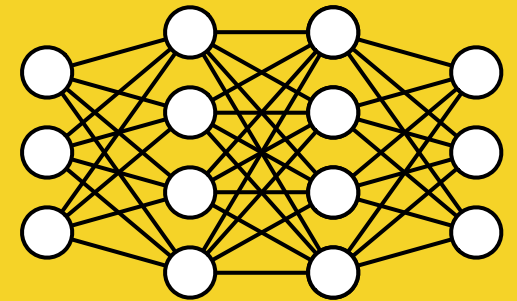
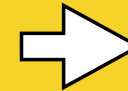
kernel void A(global

synthesizer + harness

1. Seed the model with the start of a program.
2. Predict tokens until { } brackets balance.

Input:

... 2 1 3 4 5



Output:



Decoded:

kernel void A(global

synthesizer + harness

1. Seed the model with the start of a program.
2. Predict tokens until { } brackets balance.
3. Can we parse signature?

Yes: Generate input data, compile and run it.

No: Compile it but don't run it.

Decoded:

```
kernel void A(global int* a) {
```

synthesizer + harness

1. Seed the model with the start of a program.
2. Predict tokens until { } brackets balance.
3. Can we parse signature?

Yes: Generate input data, compile and run it.

No: Compile it but don't run it.

Decoded:

```
kernel void A(global int* a) {
```



Examples

```
__kernel void A(__global float* a,
                __global float* b,
                __global float* c,
                const int d) {
    int e = get_global_id(0);
    float f = 0.0;
    for (int g = 0; g < d; g++) {
        c[g] = 0.0f;
    }
    barrier(1);

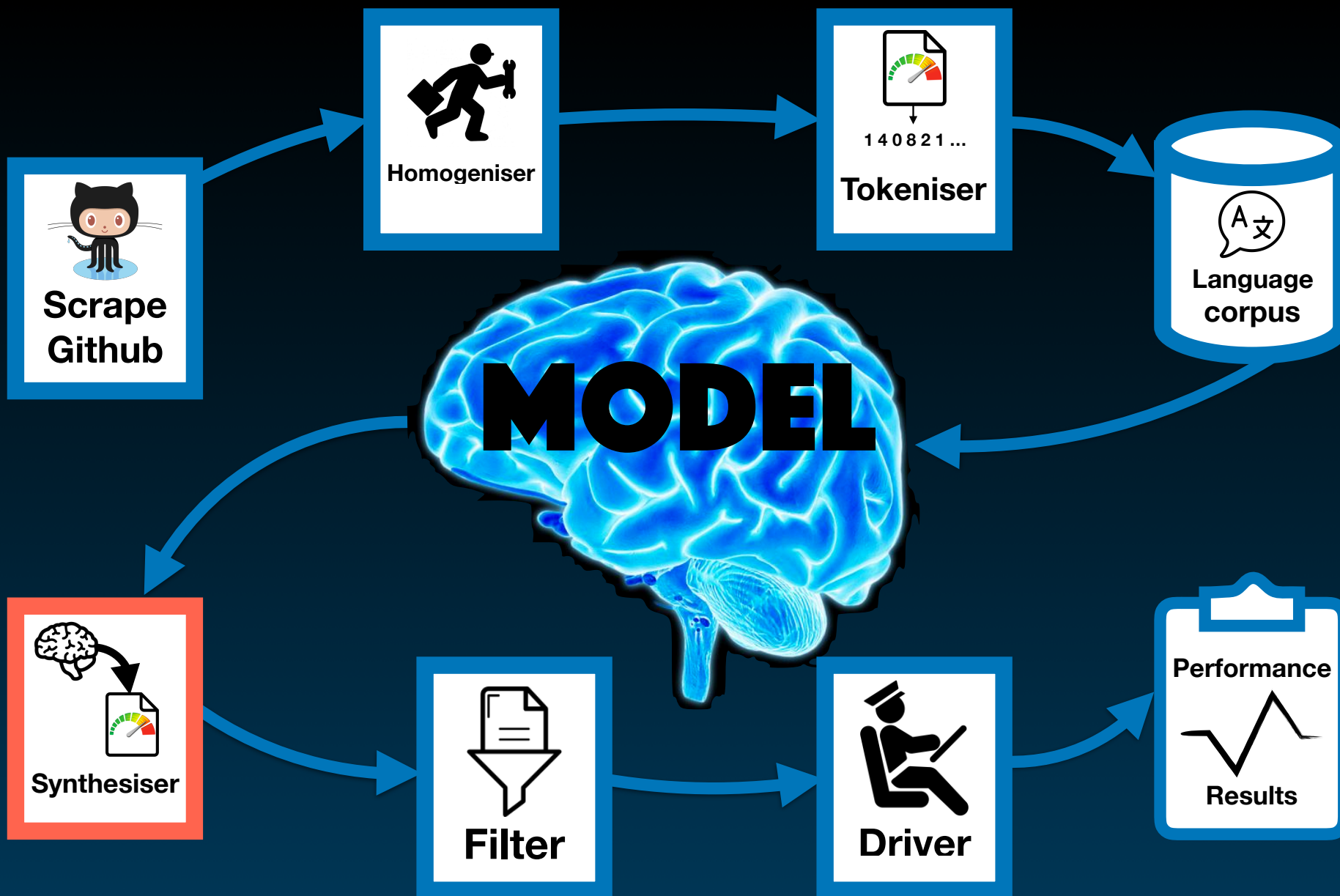
    a[get_global_id(0)] = 2*b[get_global_id(0)];
}
```

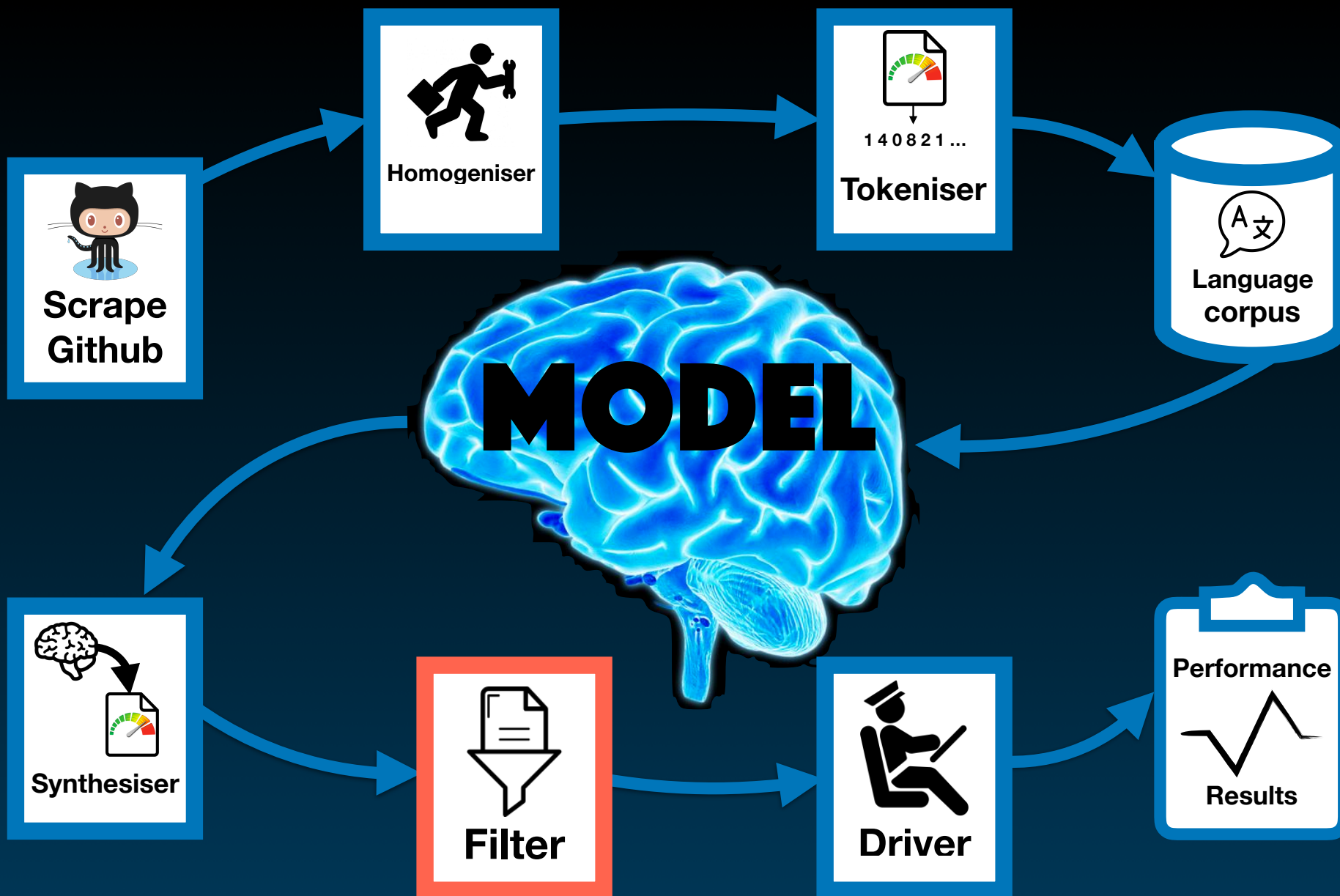
Examples

```
__kernel void A(__global float* a,  
                __global float* b,  
                __global float* c,  
                const int d) {  
    int e = get_global_id(0);  
    if (e >= d) {  
        return;  
    }  
    c[e] = a[e] + b[e] + 2 * a[e] + b[e] + 4;  
}
```

Examples

```
__kernel void A(__global float* a,
                __global float* b,
                __global float* c,
                const int d) {
    unsigned int e = get_global_id(0);
    float16 f = (float16)(0.0);
    for (unsigned int g = 0; g < d; g++) {
        float16 h = a[g];
        f.s0 += h.s0;
        f.s1 += h.s1;
        /* snip ... */
        f.sE += h.sE;
        f.sF += h.sF;
    }
    b[e] = f.s0 + f.s1 + f.s2 + f.s3 + f.s4 +
          f.s5 + f.s6 + f.s7 + f.s8 + f.s9 + f.sA +
          f.sB + f.sC + f.sD + f.sE + f.sF;
}
```





Does it compile?

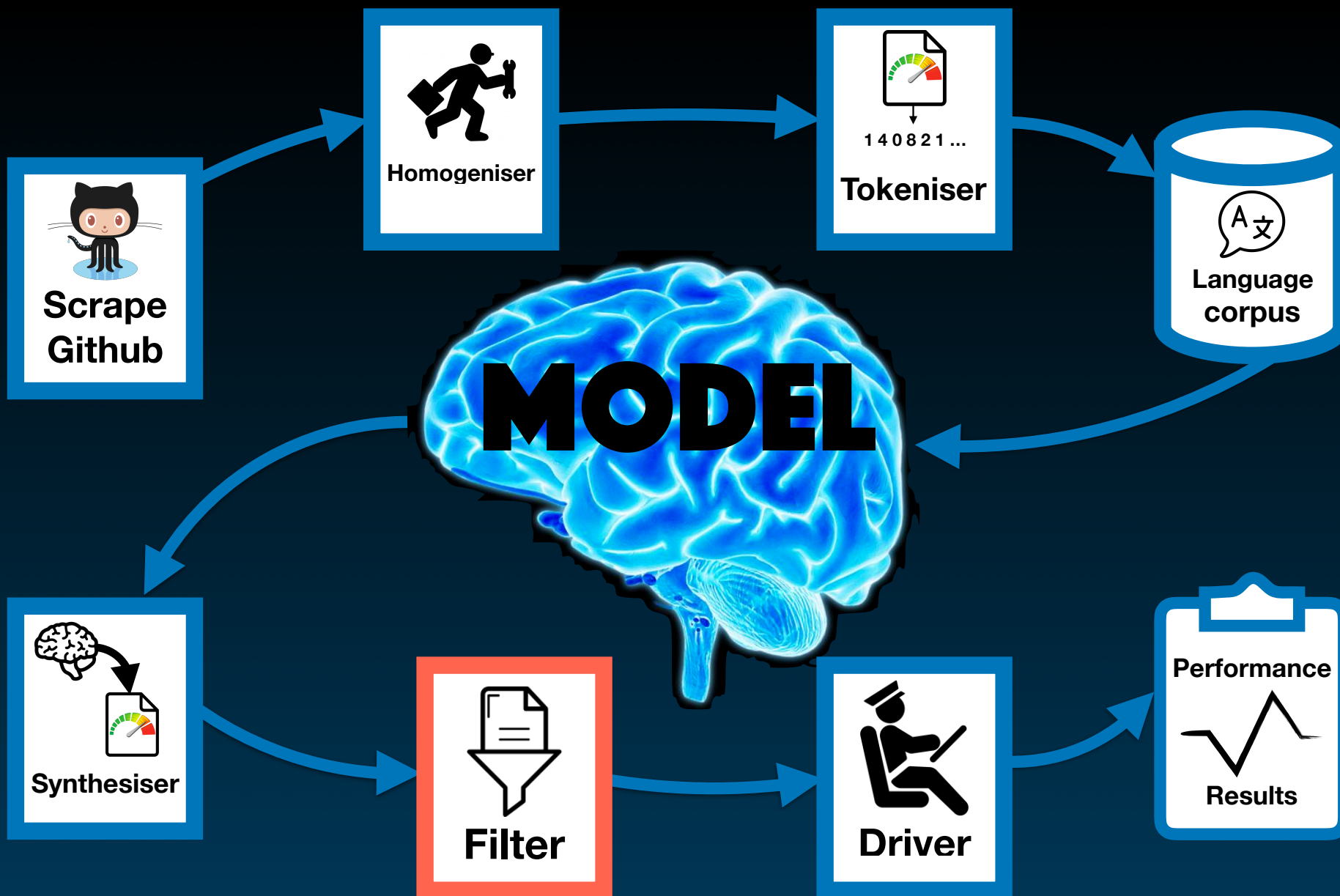
70% fail

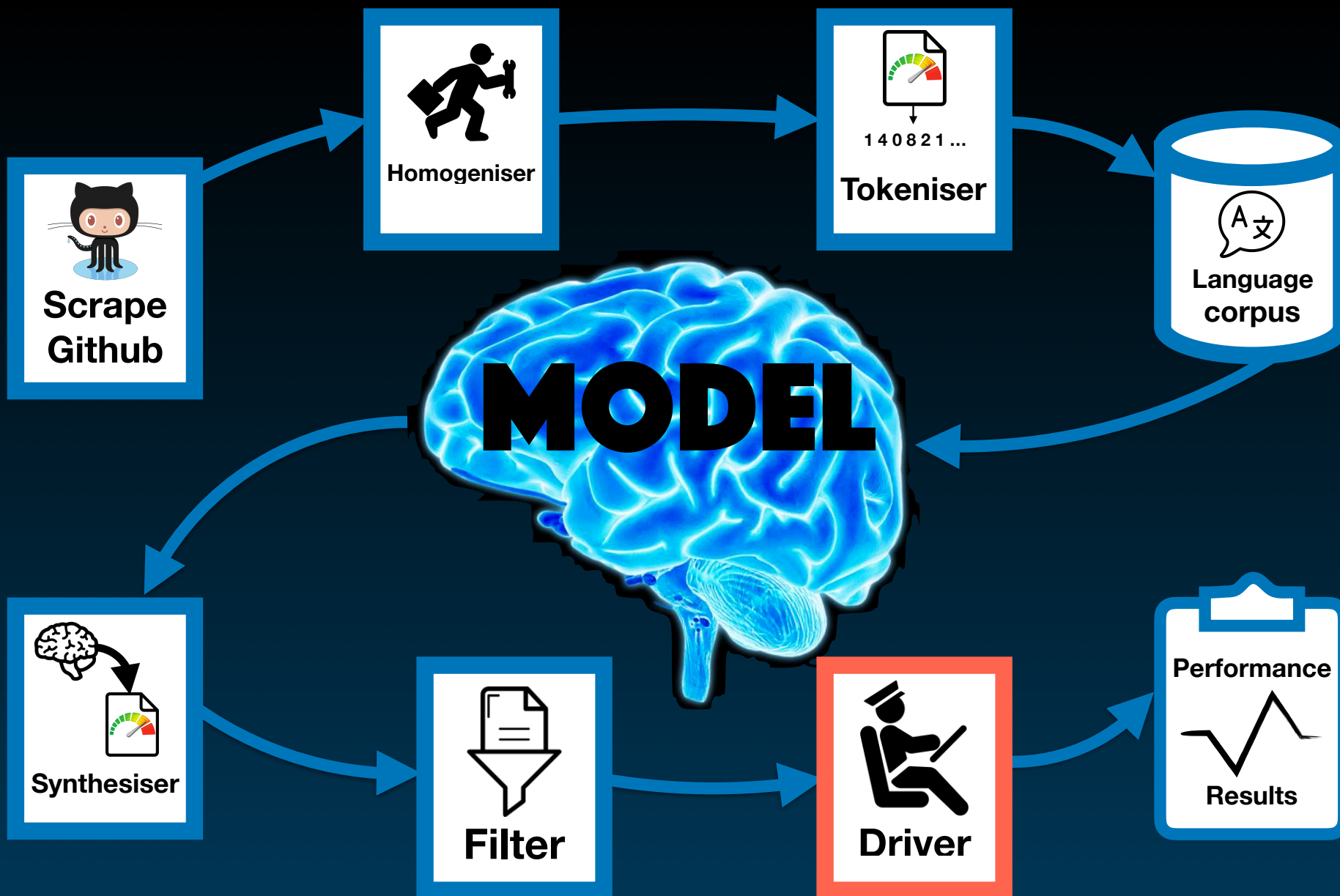
Does it do anything?

Dynamic checks

- *has output*
- *input dependent*
- *deterministic*

Yield 20-25%





```
__kernel void A(__global float* a,  
                __global float* b,  
                __global float* c,  
                const float d,  
                const int e) {  
    int f = get_global_id(0);  
    if (f >= e) {  
        return;  
    }  
    c[f] = a[f] + b[f] + 2 * c[f] + d + 4;  
}
```

```

__kernel void A(__global float* a,
                __global float* b,
                __global float* c,
                const float d,
                const int e) {
    int f = get_global_id(0);
    if (f >= e) {
        return;
    }
    c[f] = a[f] + b[f] + 2 * c[f] + d + 4;
}

```

Payload for size S:

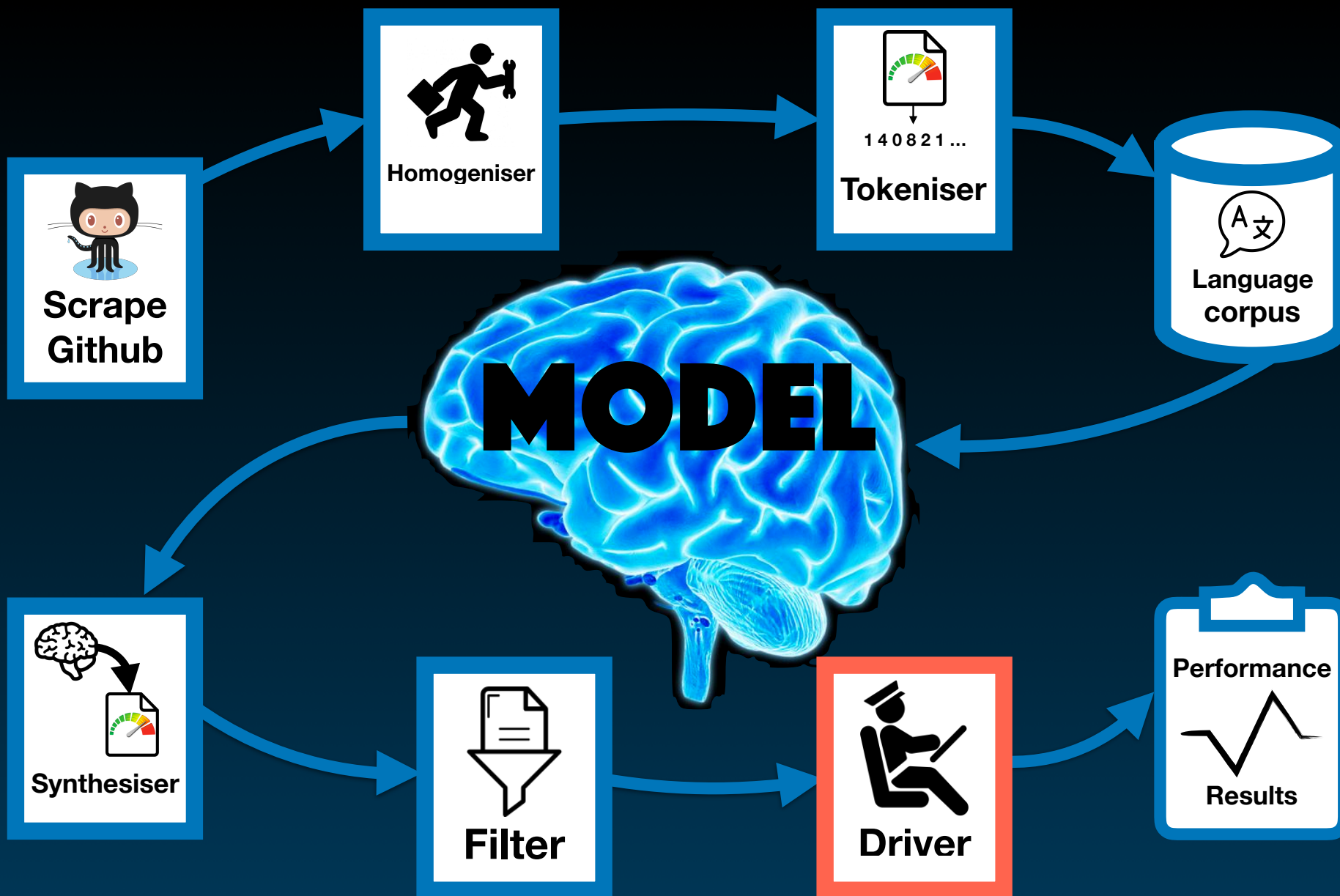
[rand()] * S

[rand()] * S

[rand()] * S

rand()

S



How well does it work?

Listing 3: Sample 3

```

1  __kernel void A(__global int* a, __global int* b, __global int* c,
    ↪ __global int* d, const uint e) {
2      const uint f = get_global_id(0);
3
4      if (e == 0 && f == 0)
5          *d = 0;
6      else if (f < e) {
7          int g = b[f];
8          uint h = c[f];
9          if (g > 0) {
10             a[h] = f;
11             h++;
12         }
13         if (f == e - 1)
14             *d = h;
15     }
16 }

```

Listing 4: Sample 4

```

1  __kernel void A(__global float* a, __global float* b, __global float* c,
    ↪ const int d) {
2      int e = get_global_id(0);
3
4      if (e < d) {
5          float f = b[e];
6          float g = a[e];
7          a[e] = f * 3.141592f / (f + 1.0f + e * 1024 - f) + (0.5f * g * 1.0f
    ↪ / 18.0f + e / 2.0f);
8      }
9
10     for (c = 0; c < 30; c++) {
11         c[e] = 0;
12     }
13 }

```

Listing 4: Sample 4

```

1  __kernel void A(int a, __global float* b, __global int* c, __global int*
    ↪ d, __local int* e, int f) {
2      int g = get_local_id(0);
3      e[get_local_id(0)] = 0;
4      barrier(1);
5      while (g < f) {
6          int h = c[g];
7
8          if (h != -1) {
9              __global float* i = b + g * a;
10             float i = 0;

```

Listing 7: Sample 7

```

1  __kernel void A(int a, int b, int c, __global const float* d, __global
    ↪ const float* e, __global float* f, float g) {
2      const int h = get_local_id(0);
3      const int i = get_group_id(0);
4
5      const int j = 4 * i + h;
6      const int k = 4 * i + h + a;
7      if (4 * i + h + a < c) {
8          float l = 0.0;
9          float m = 0.0;
10         float n = 0.0;
11         const float o = d[3 * (4 * i + h + a)];
12         const float p = d[3 * (4 * i + h + a) + 1];
13         const float q = d[3 * (4 * i + h + a) + 2];
14         for (int r = 0; r < c; r++) {
15             const float s = d[3 * r] - o;
16             const float t = d[3 * r + 1] - p;
17             const float u = d[3 * r + 2] - q;
18             const float v = (d[3 * r] - o) * (d[3 * r] - o) + (d[3 * r + 1] -
    ↪ p) * (d[3 * r + 1] - p) + (d[3 * r + 2] - q) * (d[3 * r +
    ↪ 2] - q) + g;
19             const float w = e[r] / (((d[3 * r] - o) * (d[3 * r] - o) + (d[3 *
    ↪ r + 1] - p) * (d[3 * r + 1] - p) + (d[3 * r + 2] - q) * (d[3 *
    ↪ r + 2] - q) + g) * sqrt((d[3 * r] - o) * (d[3 * r] - o)
    ↪ + (d[3 * r + 1] - p) * (d[3 * r + 1] - p) + (d[3 * r + 2]
    ↪ - q) * (d[3 * r + 2] - q) + g));
20             l = l + (d[3 * r] - o) * w;
21             m = m + (d[3 * r + 1] - p) * w;
22             n = n + (d[3 * r + 2] - q) * w;
23         }
24         f[j] = l;
25         f[k] = m;
26         f[i * 4 + h + 2] = n;
27     }
28 }

```

Listing 10: Sample 10

```

1  __kernel void A(__global ulong* a) {
2      int i, j;
3      struct S0 c_8;
4      struct S0* p_7 = &c_8;
5      struct S0 c_9 = {
6          {{0x43250E6DL, 2UL}, {0x43250E6DL, 2UL}, {0x43250E6DL, 2UL}},
7          {{0x43250E6DL, 2UL}, {0x43250E6DL, 2UL}, {0x43250E6DL, 2UL}},
8          {{0x43250E6DL, 2UL}, {0x43250E6DL, 2UL}},
9          0x4BF90EDCAD2086BDL,
10     };
11     c_8 = c_9;
12     barrier(0 | 1);
13 }

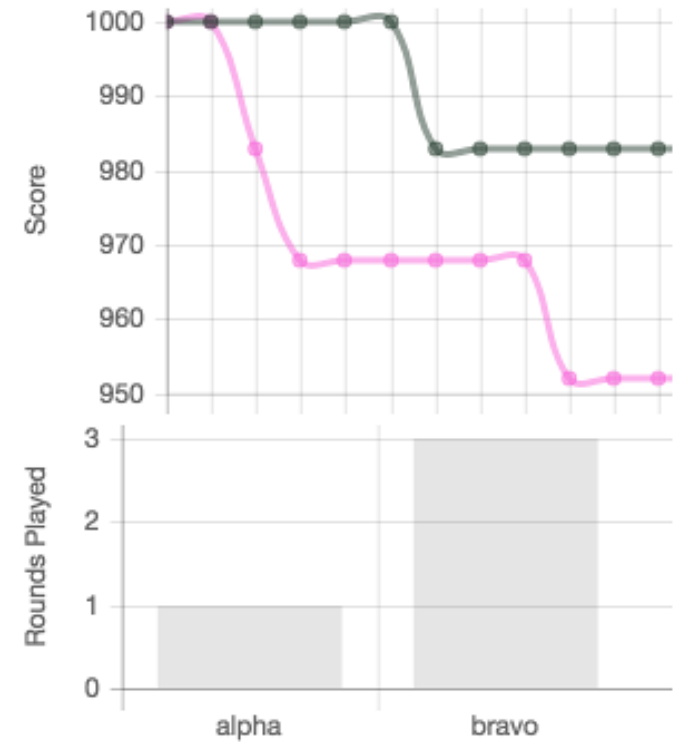
```

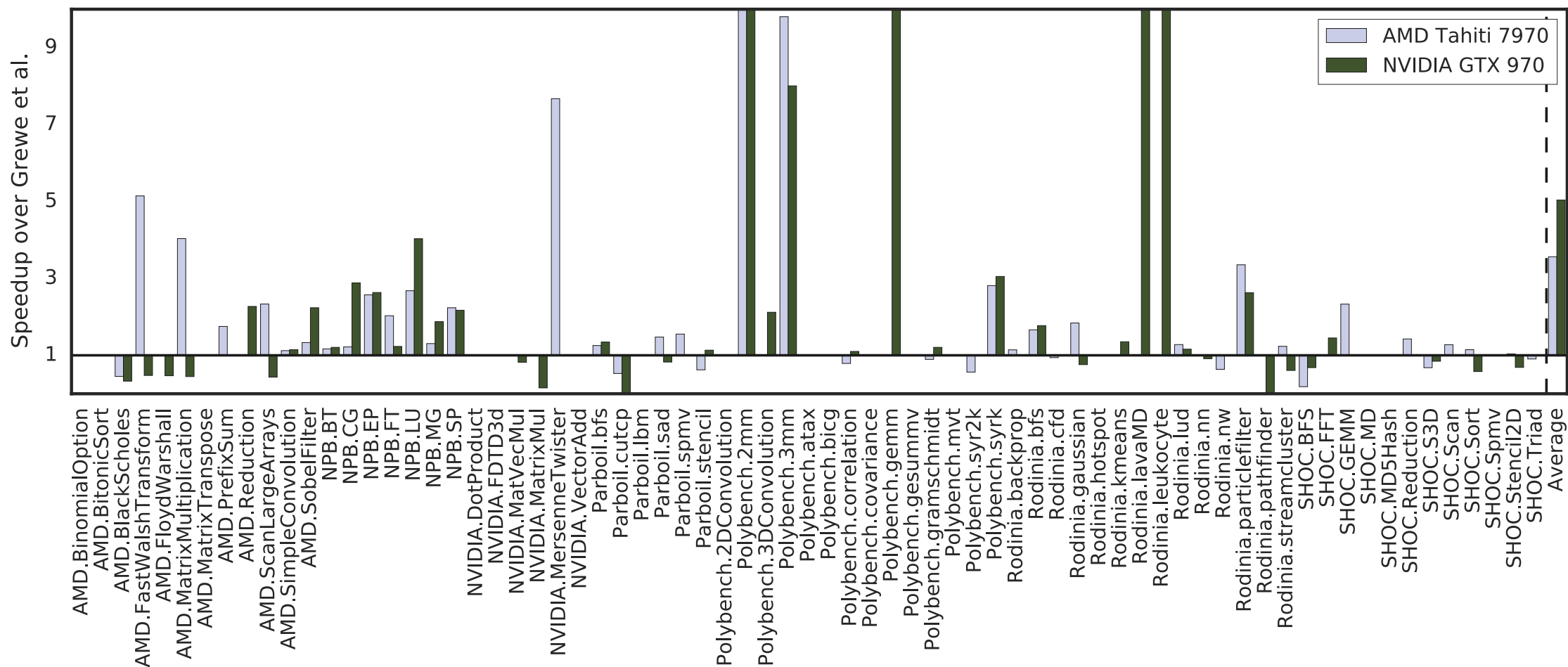
Round 1

Player: 1010, Robot: 938

```
__kernel void A(__global int* a, __global int* b, __global int* c, int d) {  
    int e = get_global_id(0);  
  
    if (e >= d) {  
        return;  
    } else {  
        a[e] = a[e];  
    }  
    b[d] = e;  
}
```

Try it

 Human Robot

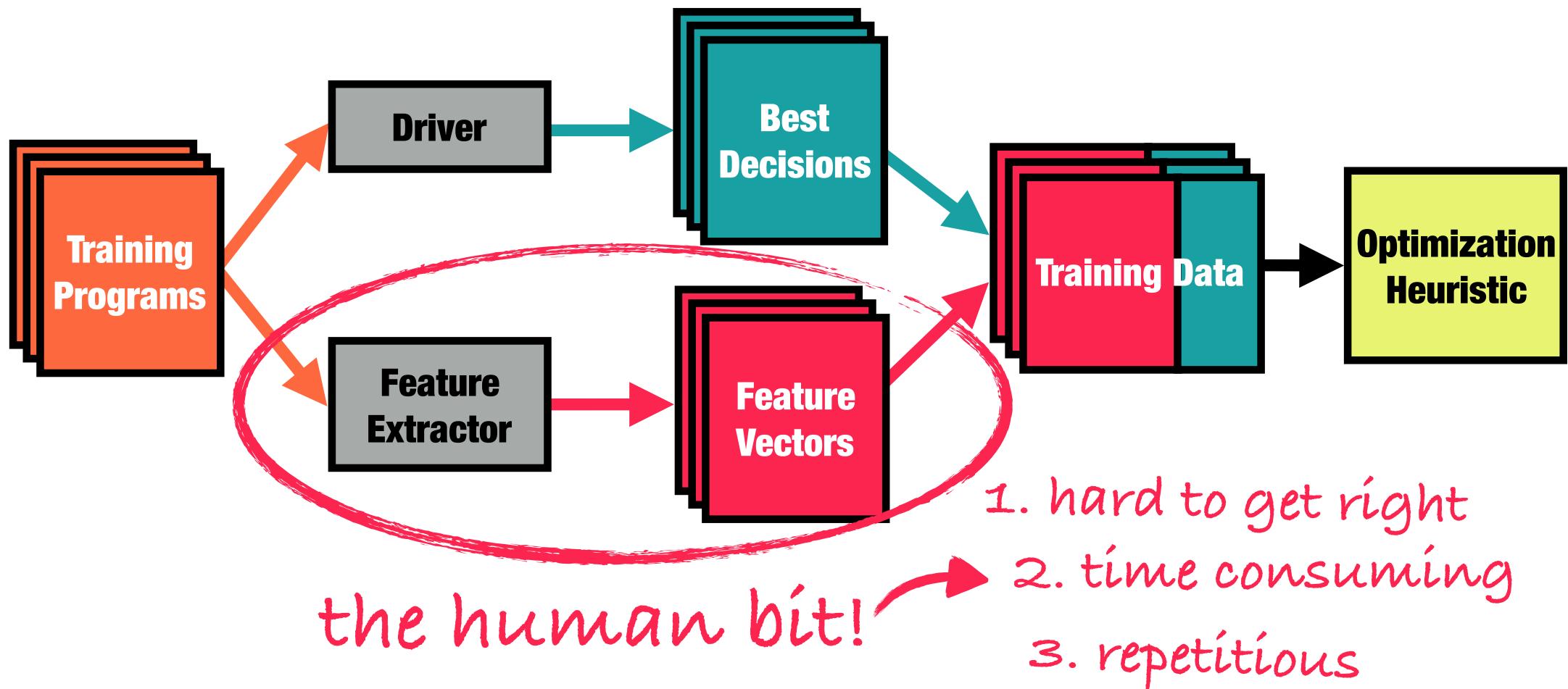


71 programs, 1,000 synthetic benchmarks. 4.30x faster

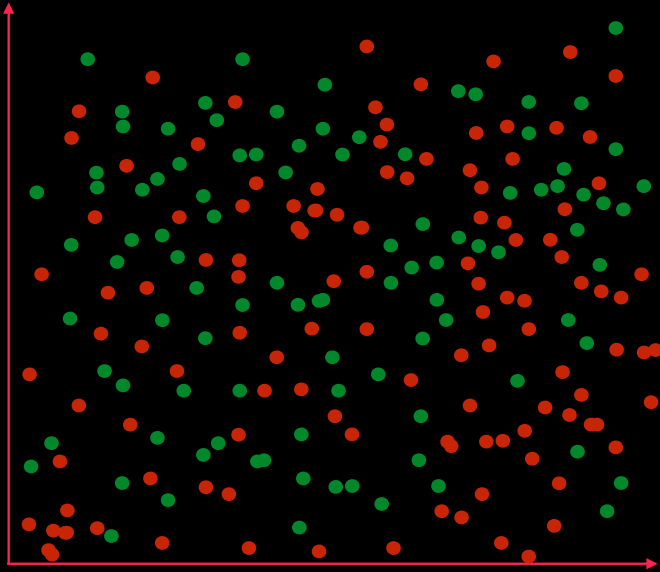
Overview

- Machine Learning for Compilers
- Generating Benchmarks
- **Deep Learned Heuristics**
- Deep Fuzzing Compiler Testing
- Future Work

Machine learning in compilers

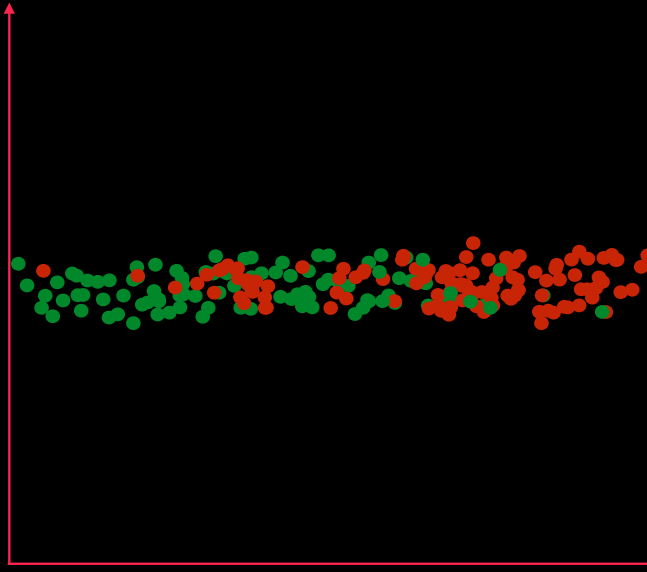


Ways to fail



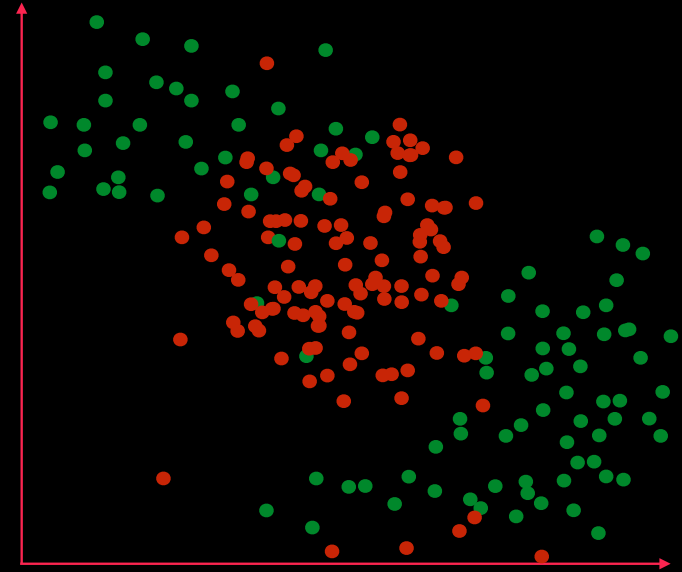
irrelevant

e.g. not capturing the right
information



incomplete

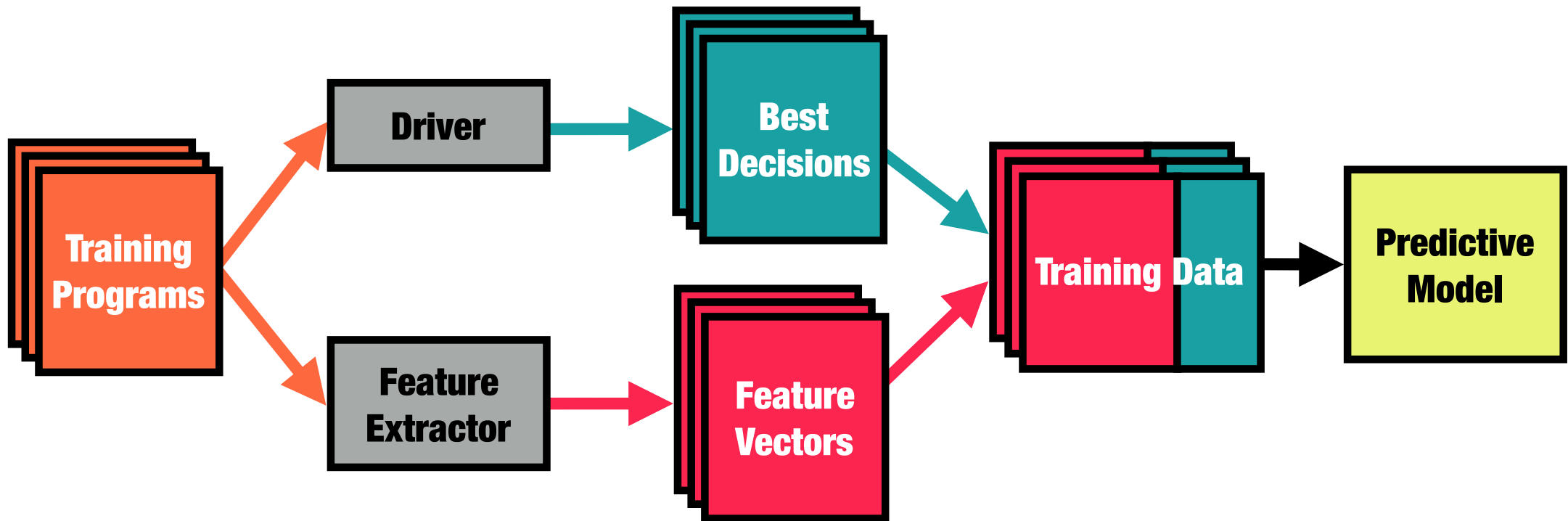
e.g. missing critical
information



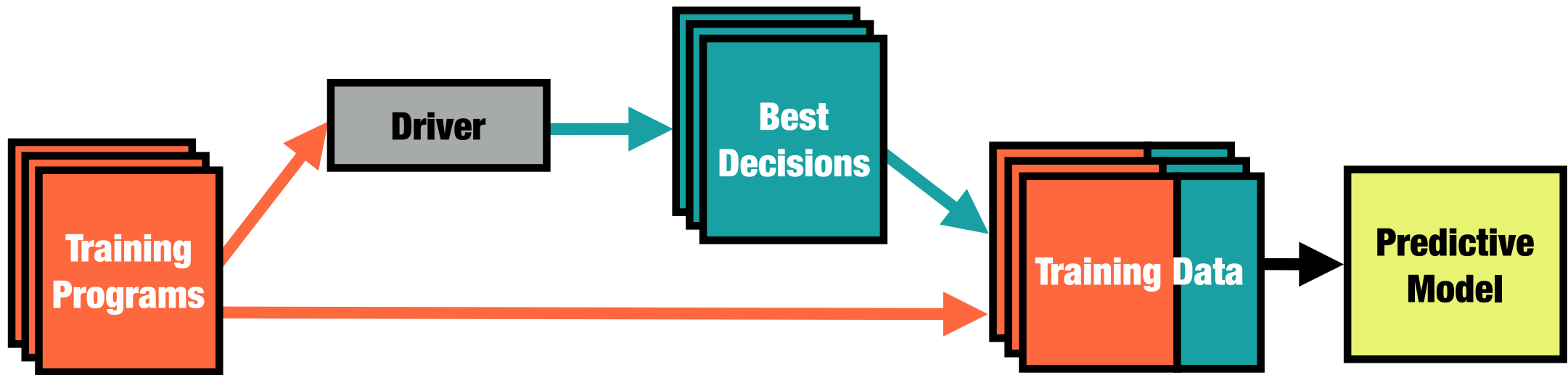
unsuitable

e.g. wrong combination of
features / model

What we have



What we need



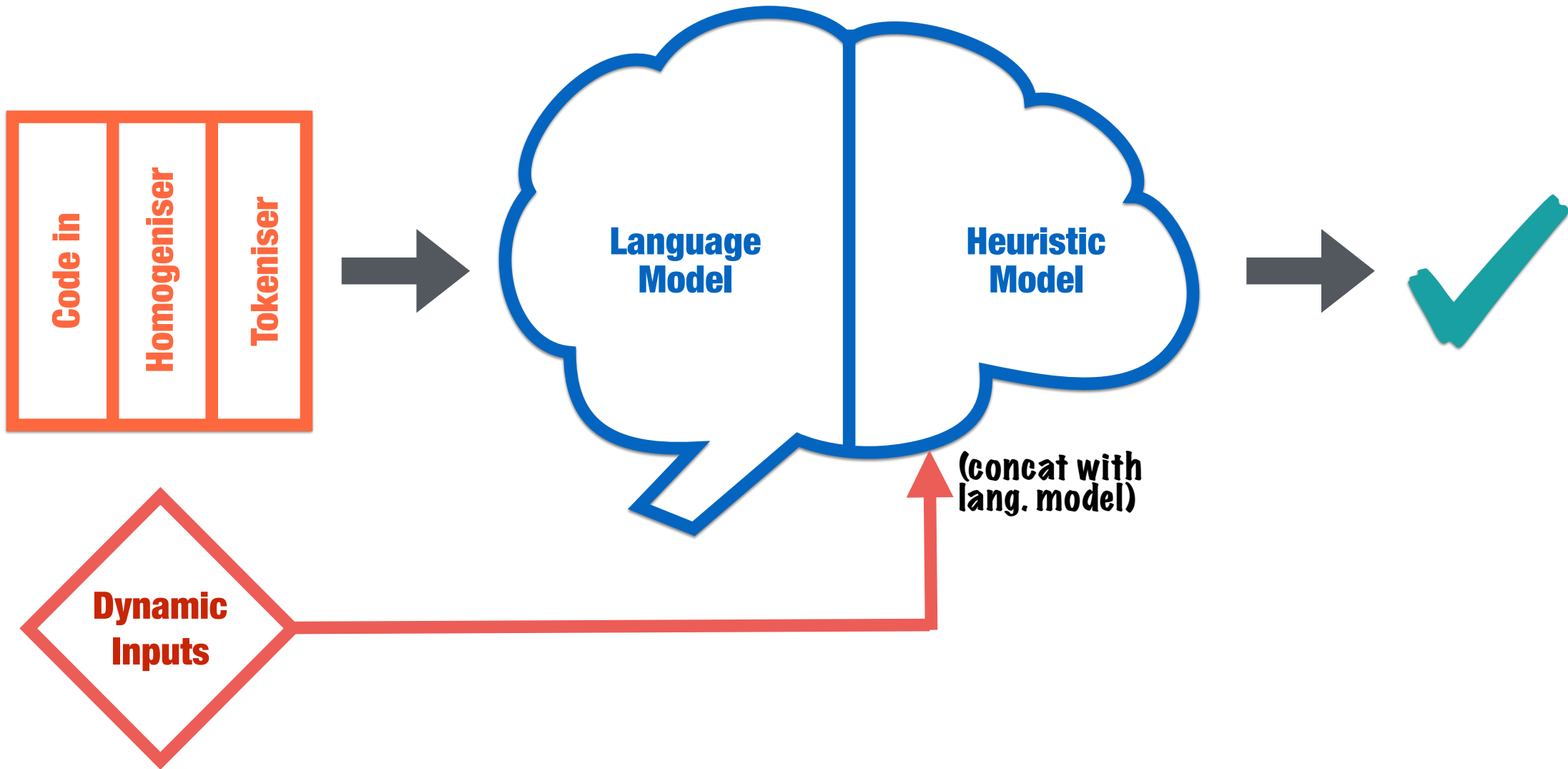
Contributions

Heuristics without features

Beats expert approach

Learning across heuristics

Dynamic Inputs



Prior Art

Heterogeneous Mapping

Portable Mapping of Data Parallel Programs to OpenCL for Heterogeneous Systems

Dominik Grewe Zheng Wang Michael F.P. O'Boyle
School of Informatics, University of Edinburgh
{dominik.grewe, zh.wang}@ed.ac.uk, mob@inf.ed.ac.uk

Abstract

General purpose GPU based systems are highly attractive as they give potentially massive performance at little cost. Realizing such potential is challenging due to the complexity of programming. This paper presents a compiler based approach to automatically generate optimized OpenCL code from data-parallel OpenMP programs for GPUs. Such an approach brings together the benefits of a clear high level language (OpenMP) and an emerging standard (OpenCL) for heterogeneous multi-cores. A key feature of our scheme is that it leverages existing transformations, especially data transformations, to improve performance on GPU architectures and uses predictive modeling to automatically determine if it is worthwhile running the OpenCL code on the GPU or OpenMP code on the multi-core host. We applied our approach to the entire NAS parallel benchmark suite and evaluated it on two distinct GPU based systems: Core i7/NVIDIA GeForce GTX 580 and Core i7/AMD Radeon 7970. We achieved average (up to) speedups of 4.51x and 4.20x (143x and 67x) respectively over a sequential baseline. This is, on average, a factor 1.63 and 1.56 times faster than a hand-coded, GPU-specific OpenCL implementation developed by independent expert programmers.

Categories and Subject Descriptors: D.3.4 [Programming Languages]: Processors—Compilers

General Terms: Experimentation, Languages, Measurement, Performance

Keywords: GPU, OpenCL, Machine-Learning Mapping

1. Introduction

Heterogeneous systems consisting of a host multi-core and GPU are highly attractive as they give potentially massive

performance at little cost. Realizing such potential, however, is challenging due to the complexity of programming. Users typically have to identify potential sections of their code suitable for SIMD style parallelization and rewrite them in an architecture-specific language. To achieve good performance, significant rewriting may be needed to fit the GPU programming model and to amortize the cost of communicating to a separate device with a distinct address space. Such programming complexity is a barrier to greater adoption of GPU based heterogeneous systems.

OpenCL is emerging as a standard for heterogeneous multi-core/GPU systems. It allows the same code to be executed across a variety of processors including multi-core CPUs and GPUs. While it provides functional portability it does not necessarily provide performance portability. In practice programs have to be rewritten and tuned to deliver performance when targeting new processors [16]. OpenCL thus does little to reduce the programming complexity barrier for users.

High level shared memory programming languages such as OpenMP are more attractive. They give a simple upgrade path to parallelism for existing programs using pragmas. Although OpenMP is mainly used for programming shared memory multi-cores, it is a high-level language with little hardware specific information and can be targeted to other platforms. What we would like is the ease of programming of OpenMP with the GPU availability of OpenCL that is then optimized for a particular platform and gracefully adapts to GPU evolution. We deliver this by developing a compiler based approach that automatically generates optimized OpenCL from a subset of OpenMP. This allows the user to continue to use the same programming language, with no modifications, while benefitting automatically from heterogeneous performance.

The first effort in this direction is [17]. Here, the OpenMPC compiler generates CUDA code from OpenMP programs. While promising, there are two significant shortcomings with this approach. Firstly, OpenMPC does not apply data transformations. As shown in this paper data transformation are crucial to achieve good performance on GPUs. Secondly, the programs are always executed on GPUs. While GPUs

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CGO '13, 23-27 February 2013, Shenzhen China.
978-1-4555-5222-0/13/05... ©2013 IEEE. \$15.00.

CGO'13
Grewe et. al

Prior Art

Thread Coarsening

Automatic Optimization of Thread-Coarsening for Graphics Processors

Alberto Magni
School of Informatics
University of Edinburgh
United Kingdom
a.magni@sms.ed.ac.uk

Christophe Dubach
School of Informatics
University of Edinburgh
United Kingdom
christophe.dubach@ed.ac.uk

Michael O'Boyle
School of Informatics
University of Edinburgh
United Kingdom
mob@inf.ed.ac.uk

ABSTRACT

OpenCL has been designed to achieve functional portability across multi-core devices from different vendors. However, the lack of a single cross-target optimizing compiler severely limits performance portability of OpenCL programs. Programmers need to manually tune applications for each specific device, preventing effective portability. We target a compiler transformation specific for data-parallel languages: *thread-coarsening* and show it can improve performance across different GPU devices. We then address the problem of selecting the best value for the coarsening factor parameter, i.e., deciding how many threads to merge together. We experimentally show that this is a hard problem to solve: good configurations are difficult to find and naive coarsening in fact leads to substantial slowdowns. We propose a solution based on a machine-learning model that predicts the best coarsening factor using kernel-function static features. The model automatically specializes to the different architectures considered. We evaluate our approach on 17 benchmarks on four devices: two Nvidia GPUs and two different generations of AMD GPUs. Using our technique, we achieve speedups between 1.11x and 1.33x on average.

1. INTRODUCTION

Graphical Processing Units (GPUs) are widely used for high performance computing. They provide cost-effective parallelism for a wide range of applications. The success of these devices has led to the introduction of a diverse range of architectures from many hardware manufacturers. This has created the need for a common programming language to harness the available parallelism in a portable way. OpenCL is an industry-standard language for GPUs that offers program portability across accelerators of different vendors: a single piece of OpenCL code is guaranteed to be executable on many diverse devices.

A uniform language specification, however, still requires programmers to manually optimize kernel code to improve performance on each target architecture. This is a tedious

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
PACT'14, August 24-27, 2014, Edinburgh, AB, Canada.
Copyright 2014 ACM 978-1-4503-2809-8/14/08...\$15.00.
<http://dx.doi.org/10.1145/2628071.2628087>.

process, which requires knowledge of hardware behavior, and must be repeated each time the hardware is updated. This problem is particularly acute for GPUs which undergo rapid hardware evolution.

The solution to this problem is a cross-architectural optimizer capable of achieving performance portability. Current proposals for cross-architectural compiler support [21, 34] all involve working on source-to-source transformations. Compiler intermediate representations [6] and ISAs [5] that span across devices of different vendors have still to reach full support.

This paper studies the issue of performance portability focusing on the optimization of the *thread-coarsening* compiler transformation. Thread coarsening [21, 30, 31] merges together two or more parallel threads, increasing the amount of work performed by a single thread, and reducing the total number of threads instantiated. Selecting the best coarsening factor, i.e., the number of threads to merge together, is a trade-off between exploiting thread-level parallelism and avoiding execution of redundant instructions. Making the correct choice leads to significant speedups on all our platforms. Our data show that picking the optimal coarsening factor is difficult since most configurations lead to performance downgrade and only careful selection of the coarsening factor gives improvements. Selecting the best parameter requires knowledge of the particular hardware platform, i.e., different GPUs have different optimal factors.

In this work we select the coarsening factor using an automated machine learning technique. We build our model based on a cascade of neural networks that decide whether it is beneficial to apply coarsening. The inputs to the model are static code features extracted from the parallel OpenCL code. These features include, among the others, branch divergence and instruction mix information. The technique is applied to four GPU architectures: *Fermi* and *Kepler* from Nvidia and *Cygnus* and *Tahiti* from AMD. While naive coarsening misses optimization opportunities, our approach gives an average performance improvement of 1.16x, 1.11x, 1.33x, 1.30x respectively.

In summary the paper makes the following contributions:

- We provide a characterization of the optimization space across four architectures.
- We develop a machine learning technique based on a neural network to predict coarsening.
- We show significant performance improvements across 17 benchmarks

PACT'14
Magni et. al

Prior Art

Heterogeneous Mapping

Thread Coarsening

Decision Space

**Binary
classification**
{CPU, GPU}

**One-of-six
classification**
{1, 2, 4, 8, 16, 32}

Model

Decision Tree

Cascading

Neural Networks

Prior Art

Heterogeneous Mapping

4 features

Combined from 7 raw values.

Instruction counts / ratios.

Thread Coarsening

Features

7 features

Principle Components of 34 raw values.

2 papers! 

Instruction counts / ratios / relative deltas.

Prior Art

Heterogeneous Mapping

Thread Coarsening

Hardware

**2x CPU-GPU
architectures**

**4x GPU
architectures**

Training Programs

7 Benchmark Suites

3 Benchmark Suites

Our Approach

Heterogeneous Mapping

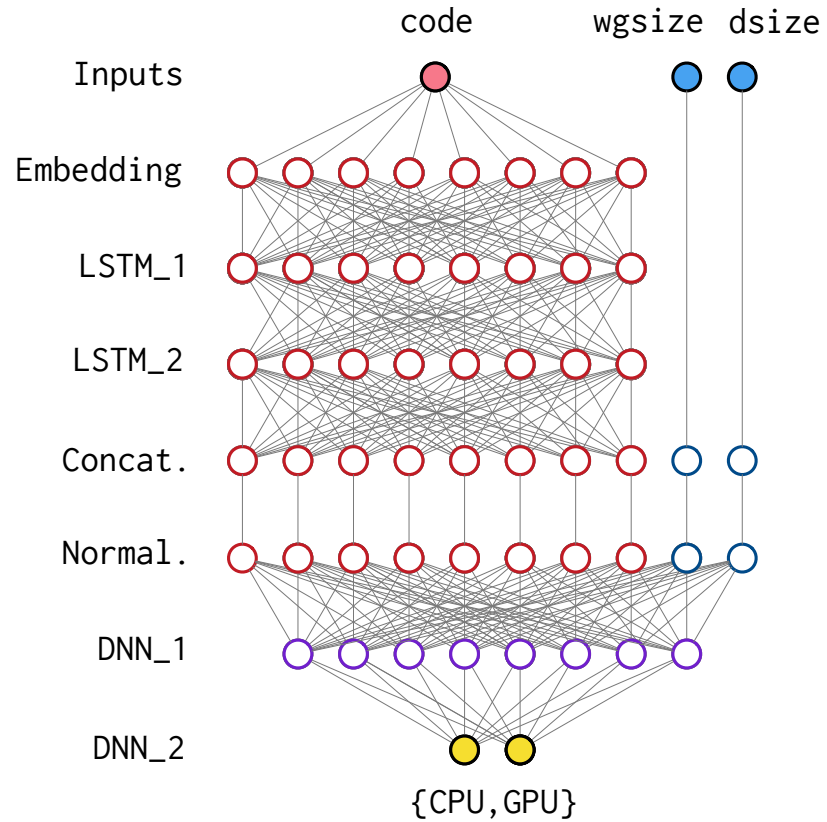
Thread Coarsening



- 1. Use the same model design for both**
- 2. No tweaking of parameters**
- 3. Minimum change - 3 line diff**

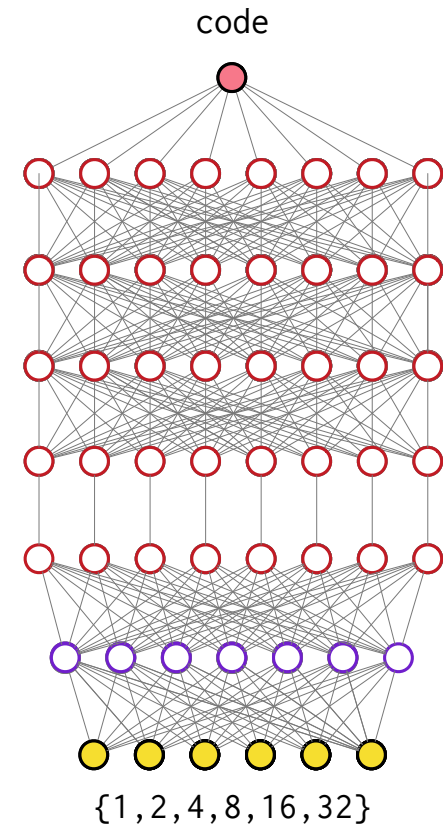
Neural Networks

Heterogeneous Mapping



(a)

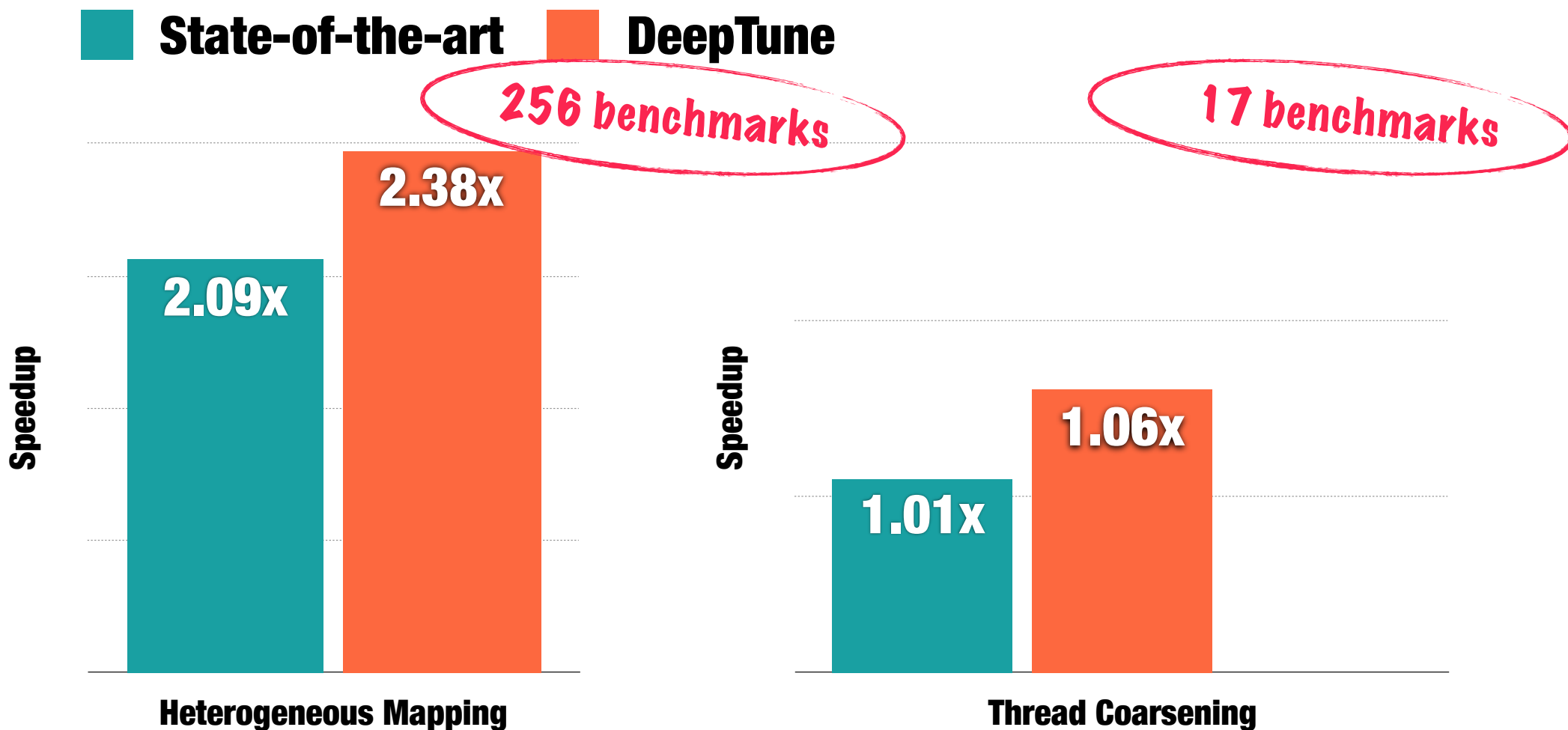
Thread Coarsening



(b)

How well does it work?

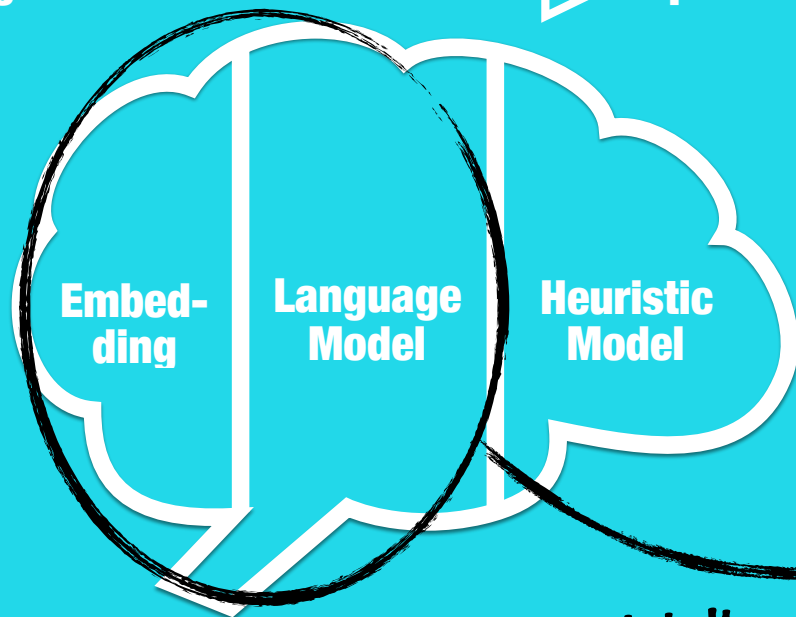
14% and 5% improvements over state-of-the-art



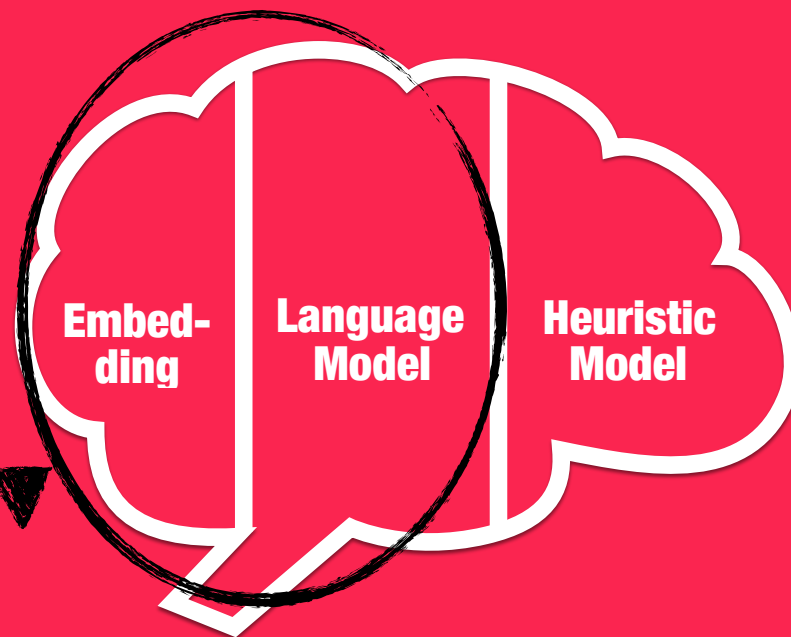
Transfer Learning

Heterogeneous Mapping

general  specialized



Thread Coarsening

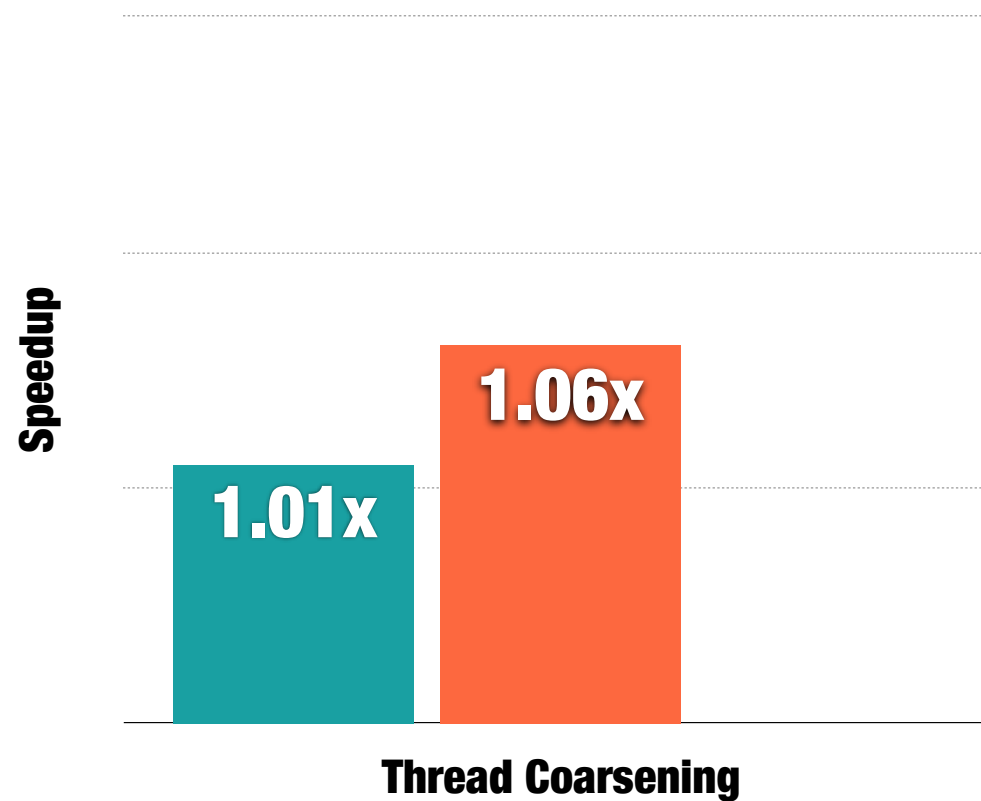
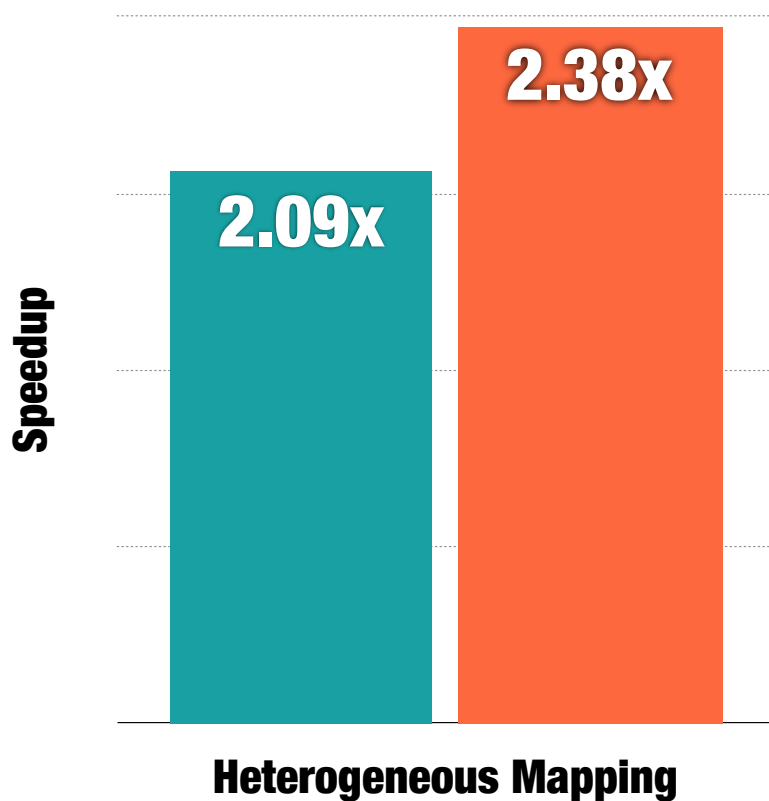


initialize with values



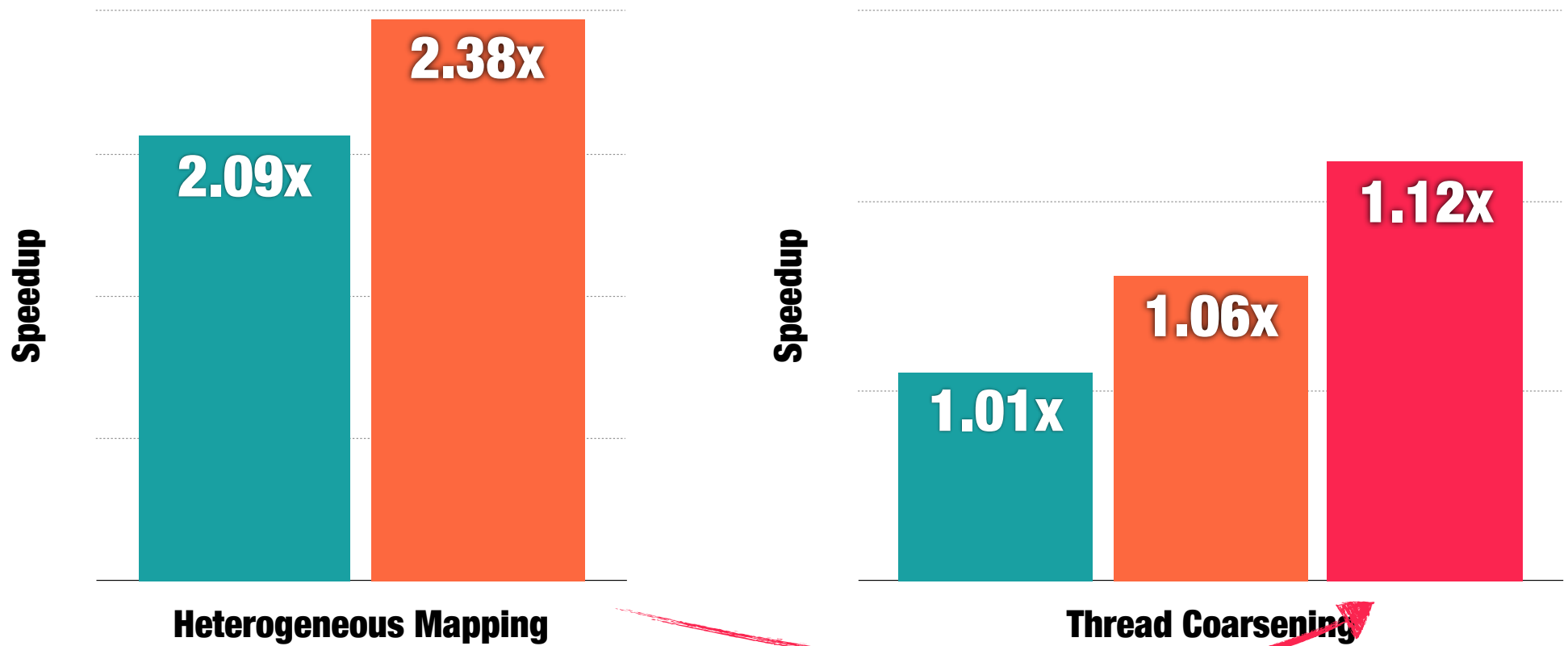
14% and 5% improvements over state-of-the-art

■ **State-of-the-art** ■ **DeepTune**



14% and 11% improvements over state-of-the-art

■ State-of-the-art ■ DeepTune ■ w. Transfer Learning



Overview

- Machine Learning for Compilers
- Generating Benchmarks
- Deep Learned Heuristics
- **Deep Fuzzing Compiler Testing**
- Future Work

compilers break

Compiler crash



Rewrite code around bug

Semantics change

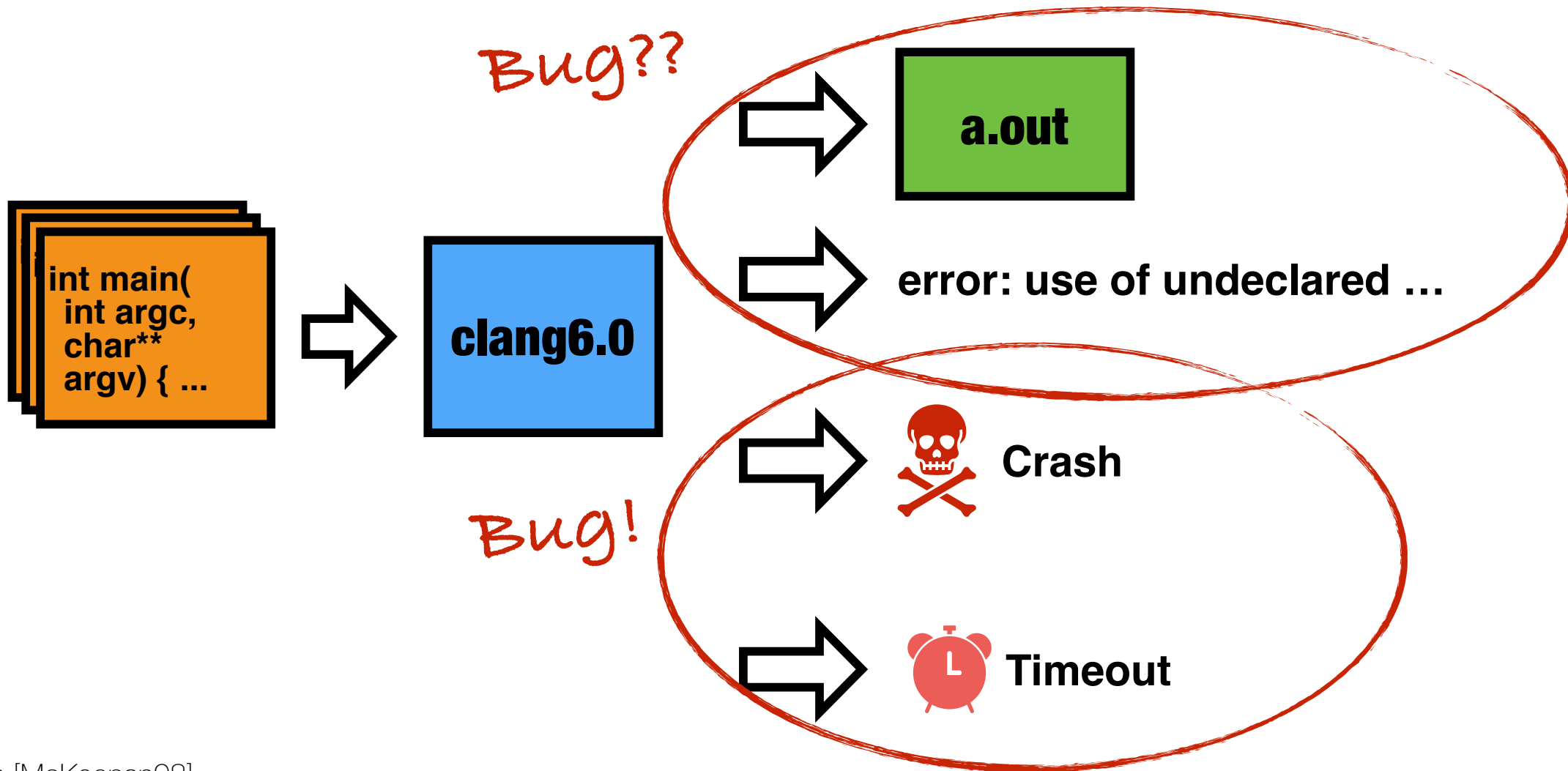


Security risk

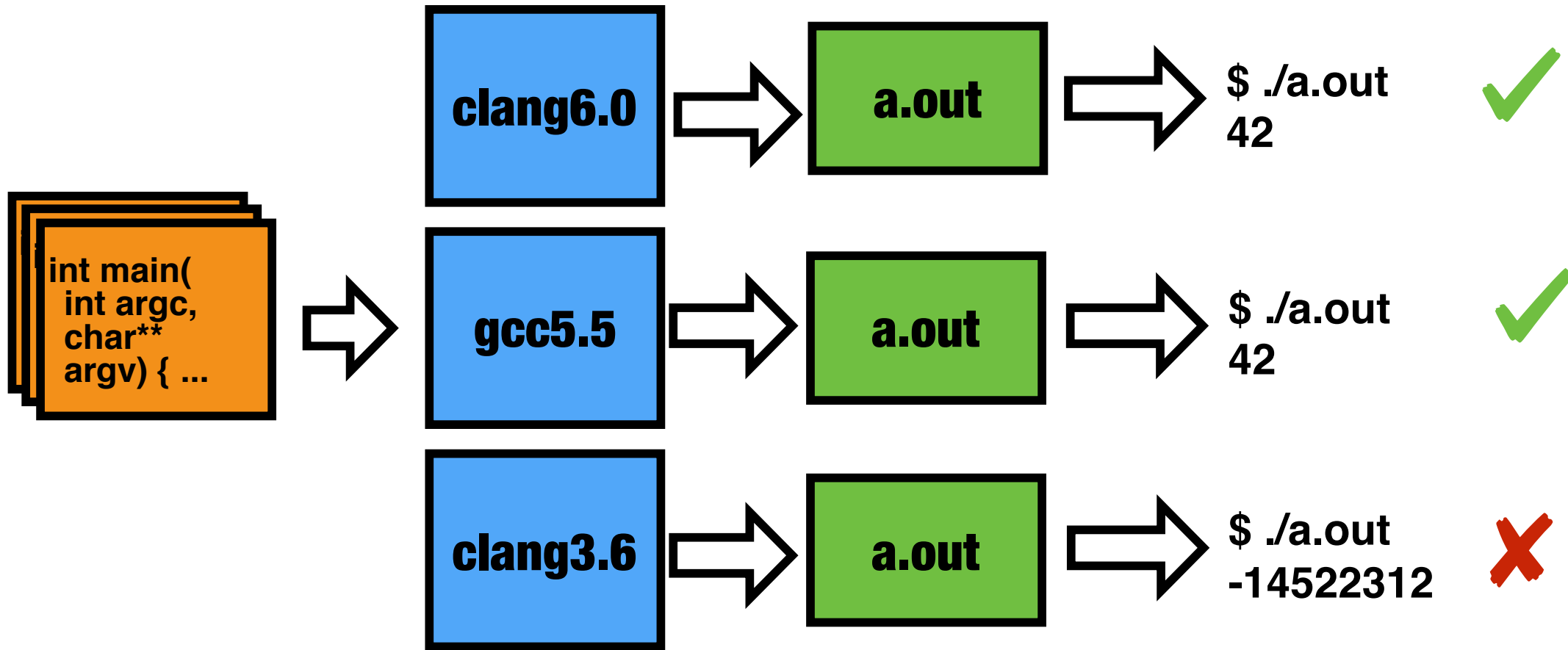
Regression suites

- **Slow**
- **Late**
- **Expensive**
- **Incomplete**

fuzzing a compiler

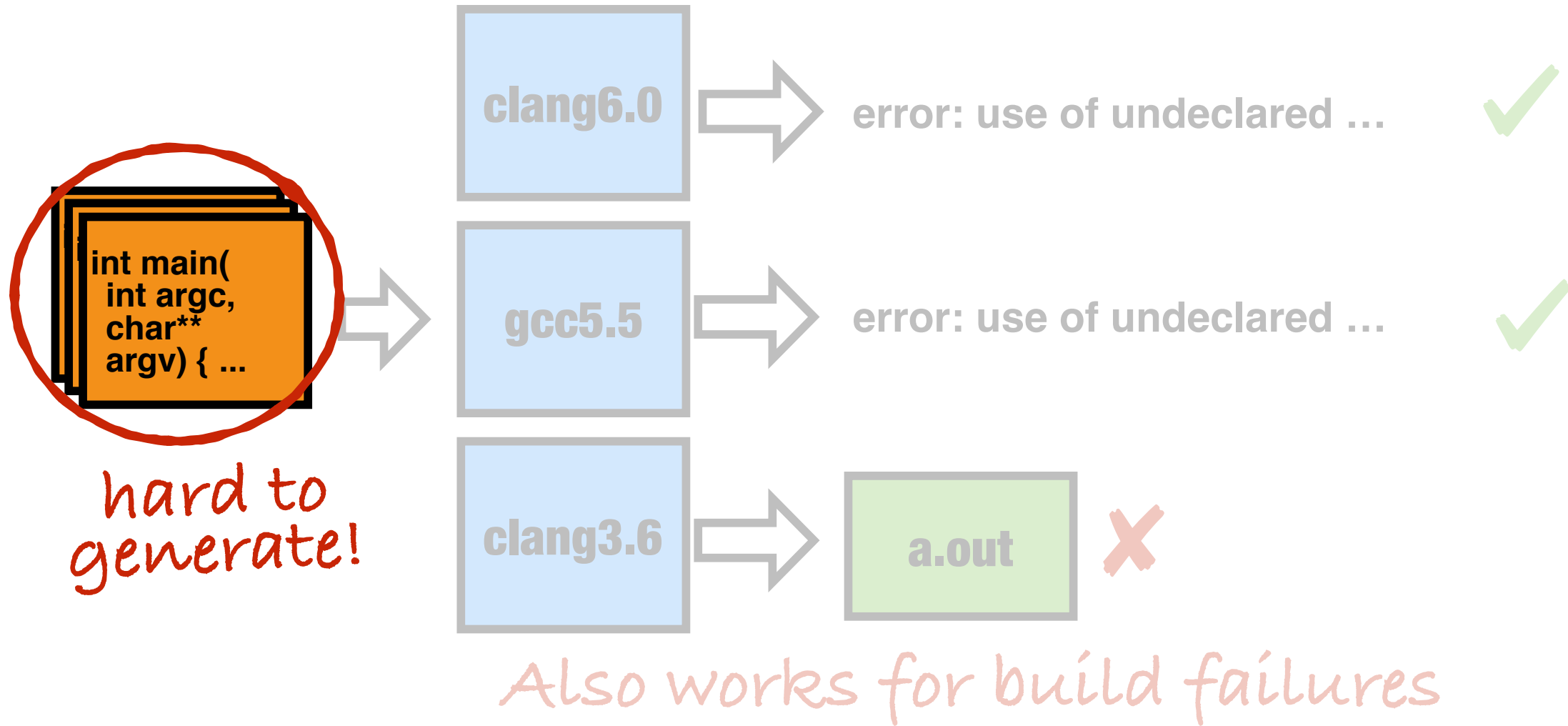


differential testing compilers



Majority rules

differential testing compilers



an ideal fuzzer

1. Cheap

Easy to implement and extend

(Languages and features grow quickly)

2. Interpretable Testcases

Necessary for triage

(i.e. 45 lines or less [Sun2016])

3. Plausible Output

Representative of handwritten code

(So that bugs gets fixed)

state-of-the-art: CLSmith

<https://github.com/ChrisLidbury/CLSmith>

```
#include "CLSmith.h"

struct S0 {
    int32_t g_4[4][10];
    ...
};

kernel void A(global ulong *r) {
    int i, j, k;
    struct S0 c_1856;
    struct S0* p_1855 = &c_1856;
    c_1856 = c_1857;
    func_1(p_1855);
    barrier(CLK_LOCAL_MEM_FENCE
| CLK_GLOBAL_MEM_FENCE);
    for (i = 0; i < 4; i++)
        for (j = 0; j < 10; j++)
            ...>g_4[i][j], "p_1855->g_4[i][j]",
            print_hash_value);
    result[get_linear_global_id()] =
    crc64_context ^
```

Random grammar enumeration.

Extensive static analyses support subset of OpenCL features.

Targets compiler middle ends.

Incredibly effective!

100s of bugs to date.

state-of-the-art: CLSmith

<https://github.com/ChrisLidbury/CLSmith>

1. **Cheap ✖ nope!**

Years to develop! 50k lines of C++.
Each PL feature engineered by hand.

2. **Interpretable Testcases ✖ nope!**

Avg. 1200 lines (excluding headers).
Requires reduction: ~4 hours / test.

3. **Plausible Output ✖ nope!**

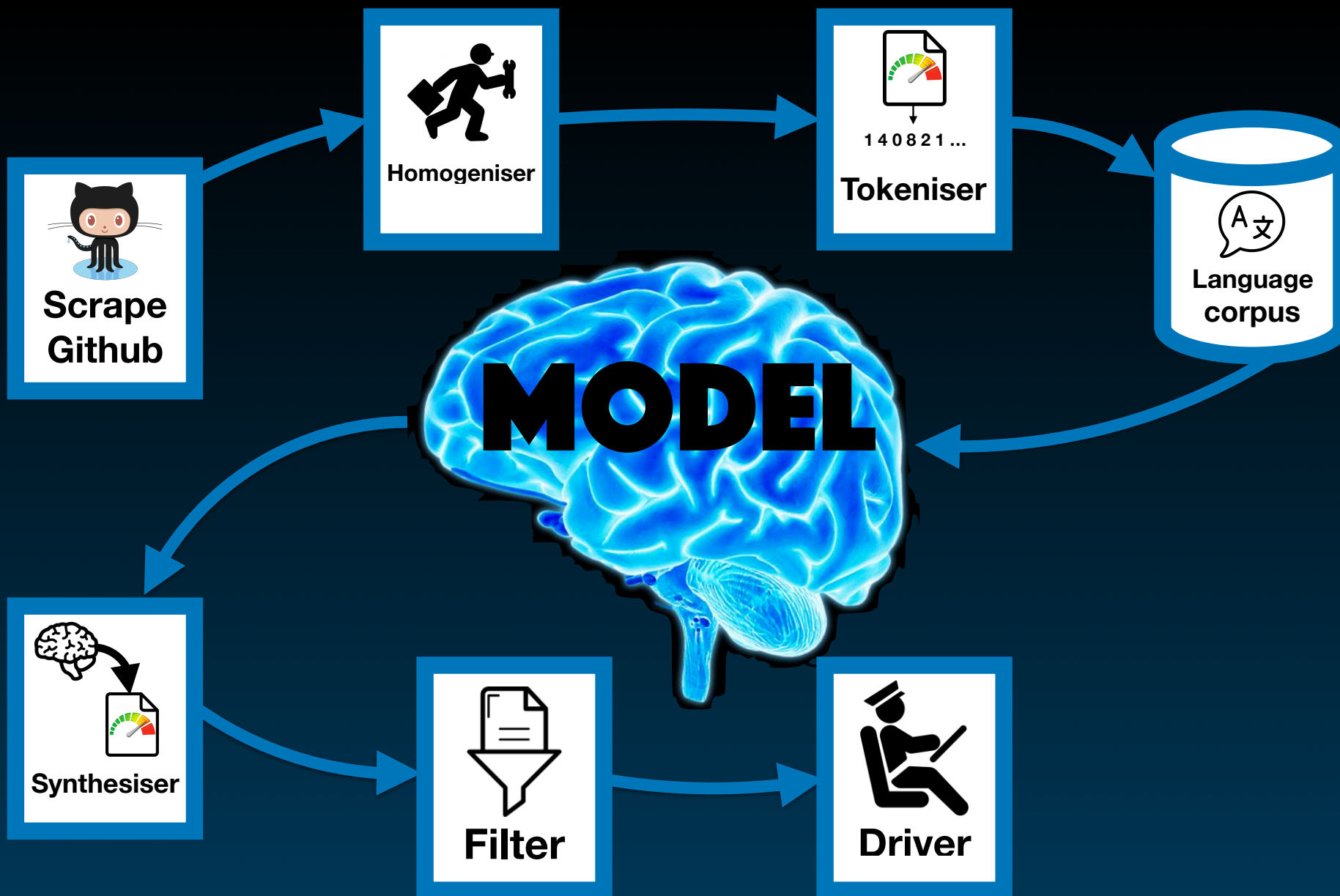
Unusual and restricted combinations of PL features.
87 dials control “shape” of output - hand tuned.

contributions

**Automatic inference of fuzzers
from examples.**

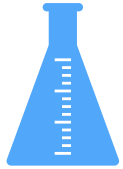
102x less code than state-of-art.

**Similar bug finding power, simpler
test cases.**



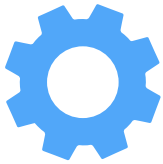
how well does it work?

testing campaign



10 OpenCL compilers

3 GPUs, 5 CPUs, Xeon Phi, Emulator



Test with optimizations on / off

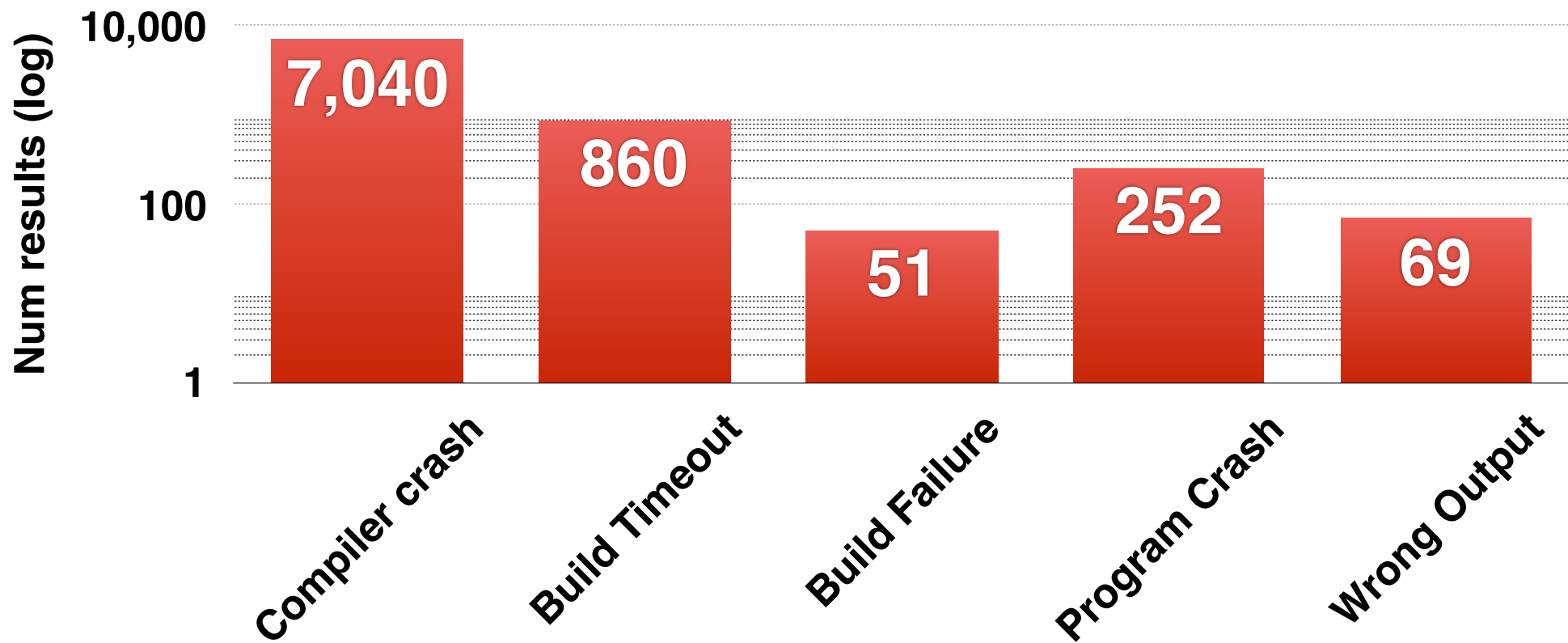
Treat as separate testbeds



48 hours per testbed

results overview

Errors in every compiler!



67 bug reports to date...

... crashes during parsing / compilation

```
void A() {void* a; uint4 b=0; b=(b>b)?a:a }
```

Affects: Intel OpenCL SDK 1.2.0.25

```
kernel void A(global int* a) {  
    int b = get_global_id(0);  
    a[b] = (6 * 32) + 4 * (32 / 32) + a;  
}
```

Affects: Beignet 1.3

“Bad code” finds bugs in error handling

67 bug reports to date...

... crashes during type checking

```
kernel void A() {  
    __builtin_astype(d, uint4);  
}
```

Affects: 6 / 10 compilers we tested

**Unexpected outcome: Learning from
handwritten code leads to bugs found in
compiler builtins!**

67 bug reports to date...

... errors in optimizers

```
kernel void A(global double* a, global double* b,  
              global double* c, int d, int e) {  
    double f;  
    int g = get_global_id(0);  
    if (g < e - d - 1)  
        c[g] = (((e) / d) % 5) % (e + d);  
}
```

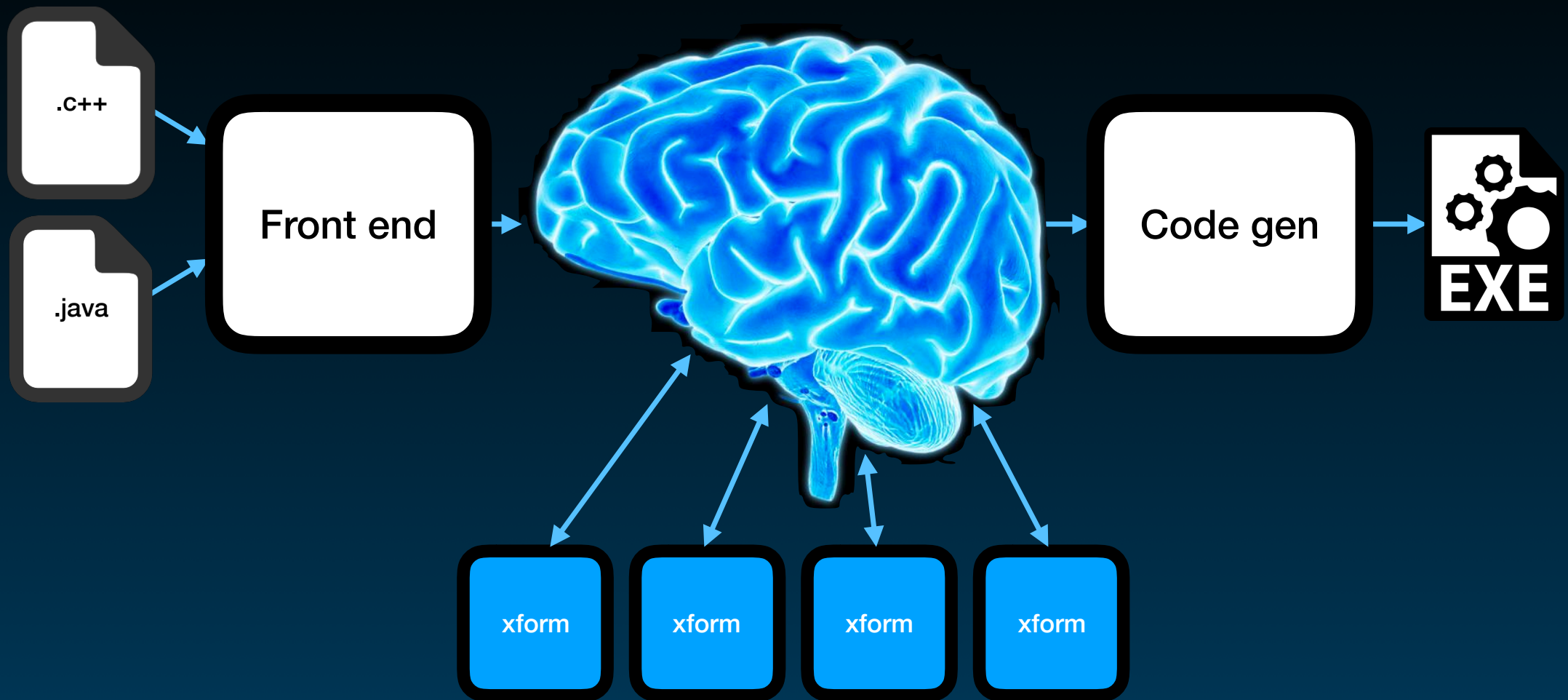
Affects: Intel OpenCL SDK 1.2.0.25

**CLSmith doesn't allow
thread-dependent control flow.**

Overview

- Machine Learning for Compilers
- Generating Benchmarks
- Deep Learned Heuristics
- Deep Fuzzing Compiler Testing
- **Future Work**

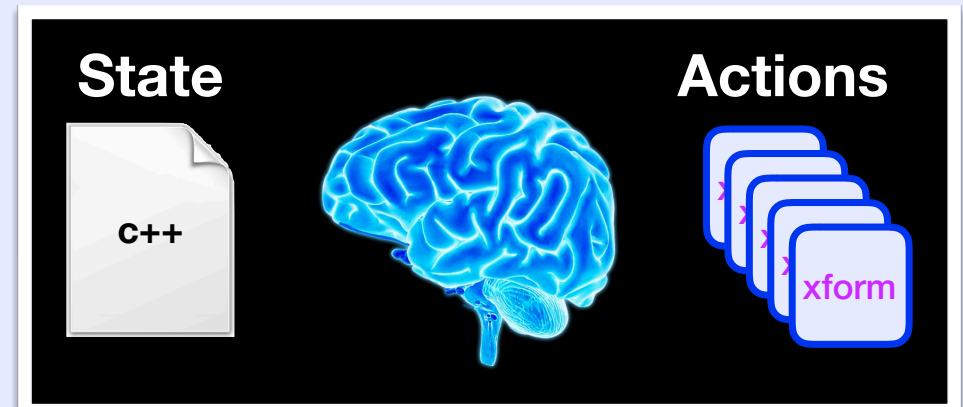
Deep Compilation



Deep Reinforced Super Optimisation

Super optimisation

- Brute force search for optimal code
- Excellent results
- **Slow**
- Need smart search



- Use reinforcement learning
- DNN chooses actions
- Actions are xform or change focus
- Stop when predicts no improvement

Deep Data Flow

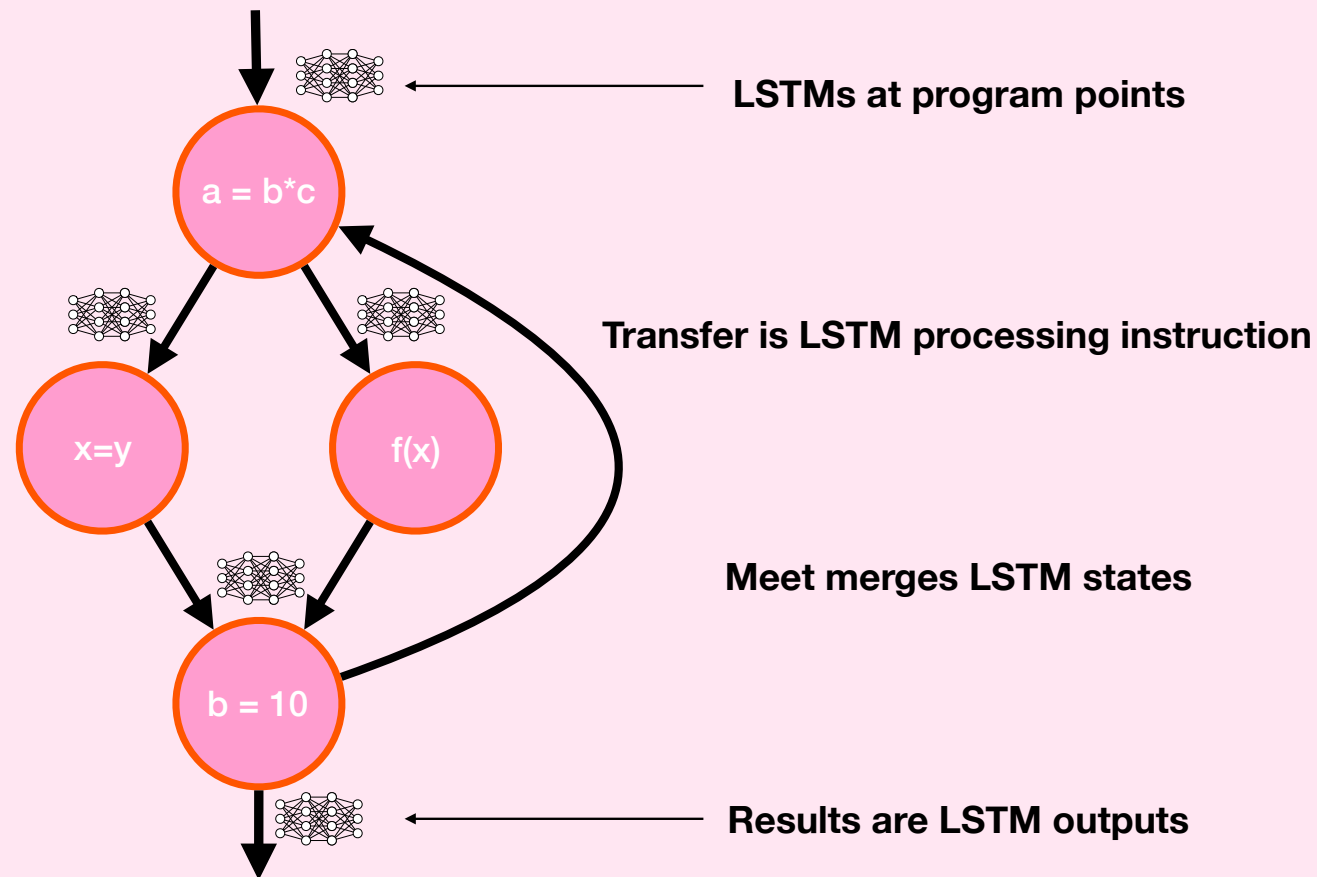
Learn analyses for heuristics not correctness

DNN struggle with data flow

LSTM cannot analyse even reachability on CFG

But can learn if given traces

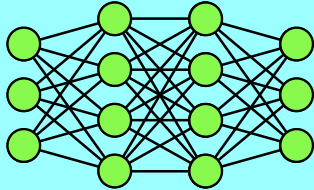
Can we extend to abstract interpretation?



Automatic Bug Triage

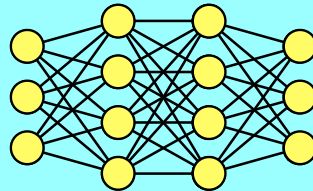
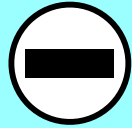
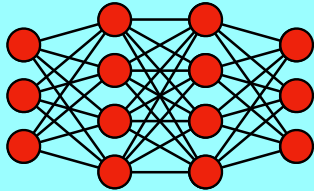
Fuzzers make thousands of bug cases too quickly

Non buggy
programs

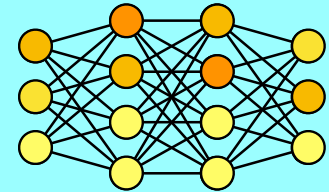
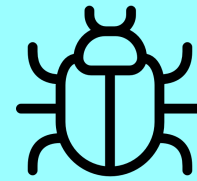


Learn language models
to discriminate

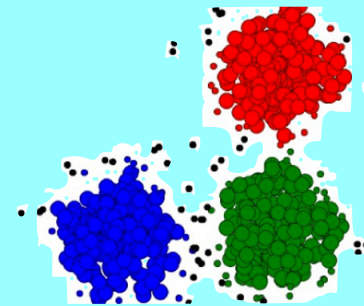
Buggy
programs



DNN Difference
encodes
bug recognition



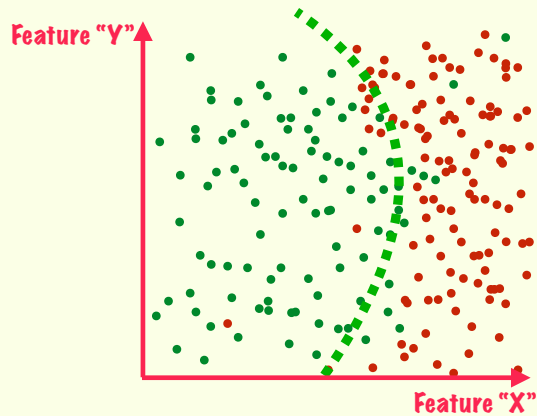
Record
activation paths



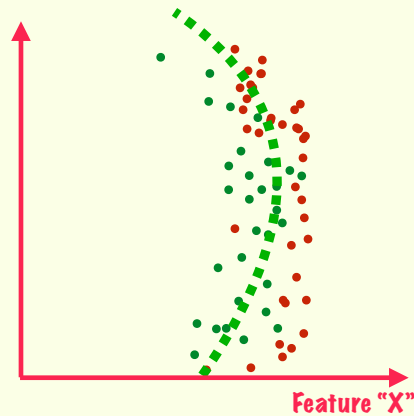
Cluster by
activation paths

Deep Active Learning

Most points uninteresting



Good ones do just as well

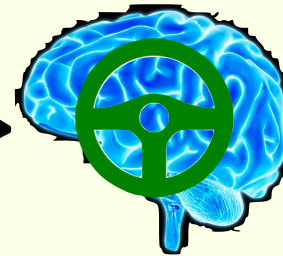


**Active learning
directly selects
useful points**

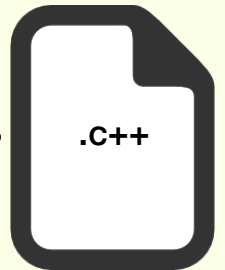
Desired
features



Drivable
language
model



Matching
program



Conclusion

- **Deep learning = better compilers**
- **Deep learning = lower cost**
- **Fun stuff still to do**