NOVEMBER 5, 2025

HARNESSING THE POWER OF JULIA ON ALCF'S POLARIS AND AURORA SYSTEMS

MICHEL SCHANEN

U.S. DEPARTMENT of ENERGY

Argonne National Laboratory is a U.S. Department of Energy laboratory managed by UChicago Argonne, LLC.

Argonne NATIONAL LABORATORY

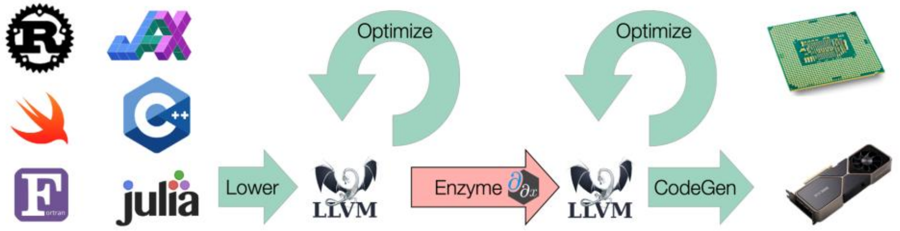
WHY JULIA?

WHY JULIA

- 1959 Fortran and ALGOL: **Mathematical algorithms** on computers
- 1969 C: **Flexibility and performance** for operating systems
- 1980 C++: Structure support for **large code bases**
- 1991 Python: **Productivity and ease of us**
- **?: Productivity, Flexibility/Portability, and Performance**

ANOTHER TRY

- Developed by Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral Shah
- Released in 2012
- **Designed for numerical and scientific computing**



1. **Performance**
   — Based on the LLVM compiler
2. **Flexibility**
   — LLVM IR or similar languages are used for a wide variety of hardware and architectures (x86, CUDA, ROCm, SPIRV…)
   — Just-in-time compilation, only code that runs is compiled
   — Metaprogramming and reflections (macros)
3. **Productivity**
   — Ease of use coupled with integrated continuous integration, package management, and version management (Manifest)
   — Interactive

U.S. DEPARTMENT of ENERGY
Argonne National Laboratory is a U.S. Department of Energy laboratory managed by UChicago Argonne, LLC.
Argonne NATIONAL LABORATORY

PRODUCTIVITY

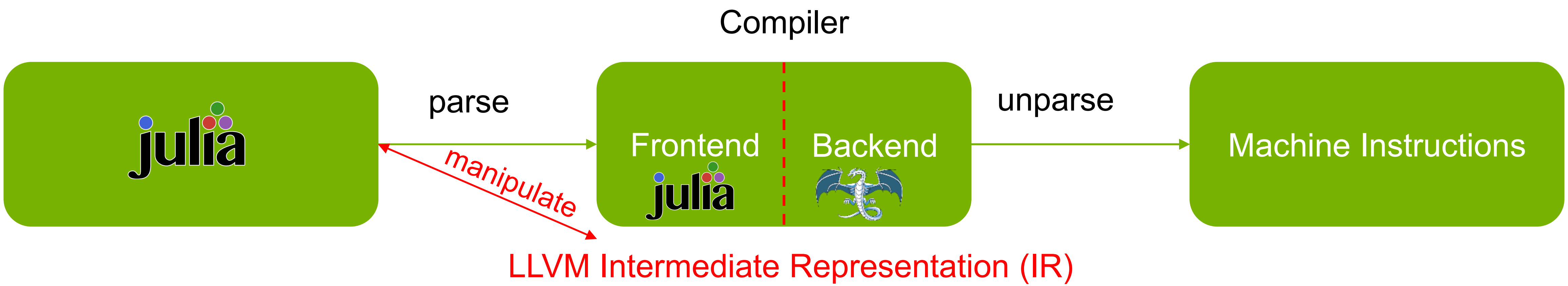
Code development and maintenance cost far exceed research.

Interior Point Method Optimization Solver for Nonlinear Optimization

Solver	Language	Lines of Code
Ipopt	C++	100,000
MadNLP	julia	6,000

MadNLP has GPU APIs

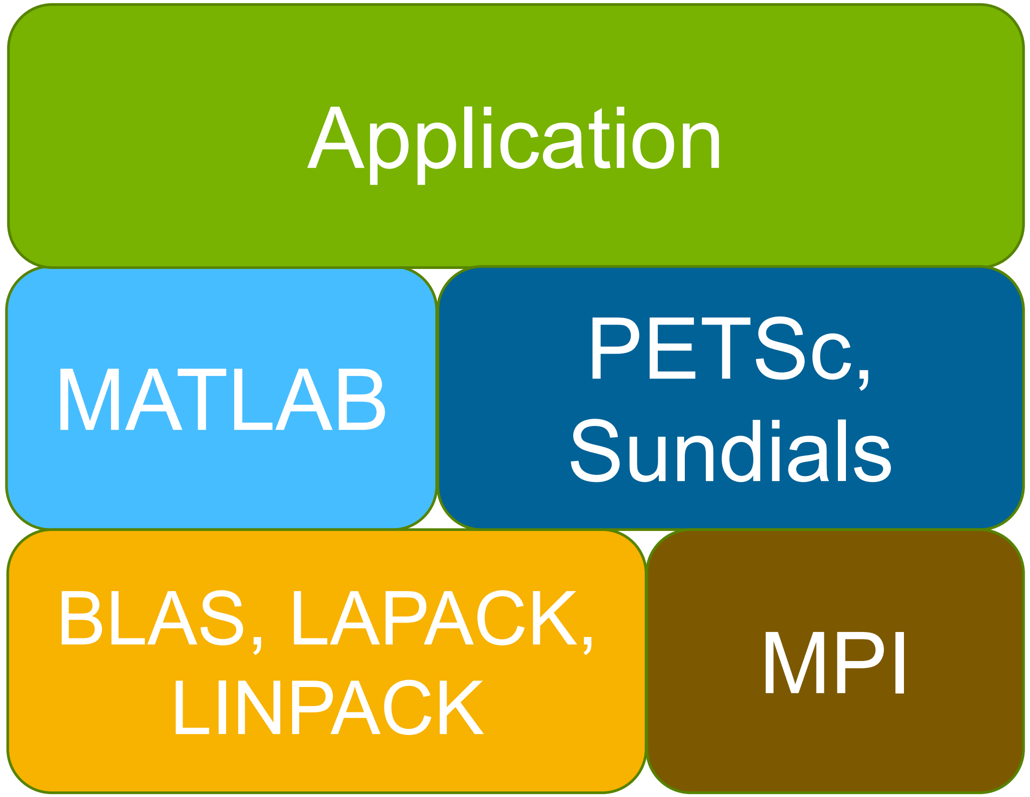
JULIA AND LLVM



- Julia leverages LLVM to implement a just-in-time compiled language with native metaprogramming and code reflection. Code is compiled at runtime.
- Language support for IR and expression tree manipulation
- Advantage: Highly flexible (JIT) C/C++-like performance (LLVM backend)
  - Ada, C, C++, D, Delphi, Fortran, Haskell, Julia, Objective-C, Rust, and Swift
- Disadvantage: LLVM was not created with JIT in mind
- See interpreted languages: Trying to reduce compilation (Python)

JULIA AND HPC FOUNDATIONS

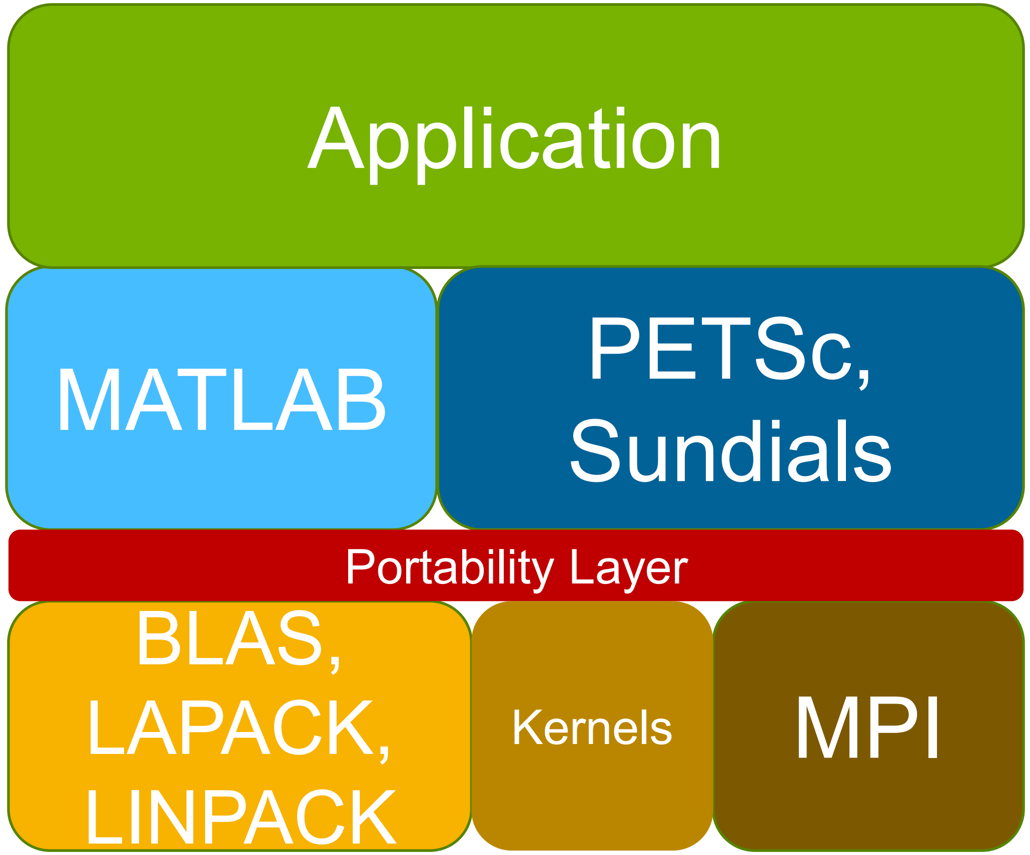
- Early HPC stack for linear algebra



- Abstraction for high-performance linear algebra (BLAS, LAPACK)
- Abstraction for distributed computing (MPI)
- Linear algebra for distributed linear algebra (e.g., PETSc)
- Abstraction for high-level algorithms

JULIA AND HPC FOUNDATIONS

- GPU HPC stack for linear algebra



- Abstraction for high-performance linear algebra (BLAS, LAPACK)
- Abstraction for distributed computing (MPI)
- Linear algebra for distributed linear algebra (e.g., PETSc)
- Abstraction for high-level algorithms
- Kernel programming
- Portability Layer
- All in one language with no compilation hell

U.S. DEPARTMENT of ENERGY
Argonne National Laboratory is a U.S. Department of Energy laboratory managed by UChicago Argonne, LLC.
Argonne NATIONAL LABORATORY

JULIA AND HPC SOFTWARE

- Interfaces to C and Python are straightforward

```julia
using Libdl

const libc_path = Libdl.find_library(["libc.so.6", "libc"])
println("Found libc at: $libc_path")

const libc_handle = Libdl.dlopen(libc_path)
const puts_fptr = Libdl.dlsym(libc_handle, :puts)


function julia_puts(s::String)
    ccall(
        puts_fptr,           # Function "puts" from library "libc"
        Cint,                # The C function 'puts' returns an int
        (Cstring,),          # It takes one argument, a C string (char*)
        "Hello, C from Julia!" # The string we want to print
    )
end

julia_puts("This is from the Julia wrapper function.")
```

```
"libc.so.6"

Found libc at: libc.so.6

Ptr{Nothing}(0x00007fa0c173f320)

Ptr{Nothing}(0x00007fa0c1551e50)

julia_puts (generic function with 1 method)

Hello, C from Julia!
21
```

JULIA AND HPC SOFTWARE

```
╭─michel@intel-G501 ~/git/julia_alcf/aurora/example ‹main●›
╰─$ julia --project=.
               _
   _       _ _(_)_     |  Documentation: https://docs.julialang.org
  (_)     | (_) (_)    |
   _ _   _| |_  __ _   |  Type "?" for help, "]?" for Pkg help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 1.12.1 (2025-10-17)
 _/ |\__'_|_|_|\__'_|  |  Official https://julialang.org release
|__/                   |

(example) pkg> add oneAPI
   Resolving package versions...


julia> using oneAPI

julia> oneAPI.versioninfo()
Binary dependencies:
- NEO: 25.35.35096+0
- libigc: 2.18.5+0
- gmmlib: 22.8.1+0
- SPIRV_LLVM_Translator: 21.1.1+0
- SPIRV_Tools: 2025.4.0+0
- oneAPI_Support: 0.9.2+0 (oneMKL v2025.2.0)

Toolchain:
- Julia: 1.12.1
- LLVM: 18.1.7

1 driver:
- 00000000-0000-0000-1874-85ae01038918 (v1.3.35096, API v1.6.0)

1 device:
- Intel(R) Arc(TM) A750 Graphics

julia> █
```

- Julia provides an intricate cross-compilation system for providing binary artifacts linking only against local libc
- oneAPI Intel dependencies are provided precompiled for any Linux system

PORTABILITY IN JULIA

U.S. DEPARTMENT of ENERGY

Argonne National Laboratory is a U.S. Department of Energy laboratory managed by UChicago Argonne, LLC.

Argonne NATIONAL LABORATORY

PORTABILITY IN JULIA

Layers	**Julia Code**			
Abstraction	GPUArrays.jl + KernelAbstractions.jl			
Generation	CUDA.jl	AMDGPU.jl	oneAPI.jl	Host/CPU
IR / C API	CUDA	ROCm	Intel Compute Runtime	LLVM

- All vendor languages support LLVM IR or SPIRV intermedia representations (IR)
- Only high-level language with support for GPUs
- BLAS, LAPACK etc is **dispatched** to vendor libraries **based on type**
- Native performance using vendor toolchain
- However: simplified API for synchronization
  - Julia uses one CUDA stream. Kernel executions are ordered and synchronous
  - If you really want very high-performance -> go down to C/C++
  - Trade-off between user friendliness and performance

PORTABILITY IN JULIA: KERNELABSTRACTIONS

- Architecture abstraction through backends
- Generic DSL for SIMD kernels using Julia macros (e.g., `@index`, `@kernel`)
- Uses `GPUCompiler.jl` infrastructure to transform lowered Julia LLVM IR for each backend

```julia
using oneAPI #, CUDA, AMDGPU
using KernelAbstractions
using Adapt

backend = oneAPIBackend()
a = adapt(backend, ones(10))
b = adapt(backend, ones(10))
c = similar(a)

# KernelAbstractions.jl kernel definition
@kernel function myadd!(c, a, b)
    i = @index(Global)
    c[i] = a[i] + b[i]
end
# Instantiate kernel
kernel! = myadd!(backend)
# Launch the kernel
kernel!(c, a, b; ndrange=length(c))
```

PORTABILITY IN JULIA: GPUARRAYS

- Supports the broadcast operator '.'
- Supplies common methods for GPU arrays using `KernelAbstractions.jl` (e.g., `sort!, accumulate`)

```julia
using oneAPI #, CUDA, AMDGPU
using Adapt
using LinearAlgebra

# Other backends: CUDABackend(), AMDGPUBackend()
backend = oneAPIBackend()

a = adapt(backend, ones(10))
b = adapt(backend, ones(10))
c = similar(a)

# Generates a kernel using the broadcast operator
c .= a .+ b
# Sorts array c in place
sort!(c)

A = adapt(backend, rand(10,10))
x = adapt(backend, rand(10))
# Matrix-vector multiplication
y = mul!(similar(x), A, x)
```

PORTABILITY IN JULIA: MPI

- No complex derived type handling
- Support of GPU-aware MPI
- Even functions can be sent

```julia
using MPI
using oneAPI

MPI.Init()
comm = MPI.COMM_WORLD
rank = MPI.Comm_rank(comm)
size = MPI.Comm_size(comm)

println("Hello from rank $rank of $size")

struct DerivedType{VT <: AbstractVector}
    v::VT
    a::Int
end


a = DerivedType(adapt(backend, ones(10)), 42)
# Send and receive the derived type round-robin
req = MPI.isend(a, comm; dest=(rank + 1) % size)
b = MPI.recv(comm; source=(rank - 1 + size) % size)
MPI.Wait(req)

MPI.Finalize()
```

PORTABILITY IN JULIA: LIBRARIES

Example: Krylov.jl

- A portable Krylov solver library with support for NVIDIA, AMD, and Intel GPUs
- Comes with all Krylov algorithms under the sun
- Transparent interfaces to preconditioners through `KrylovPreconditioners.jl`
- Works with custom operators (e.g., written in `KernelAbstractions.jl`)
- Example: Minimizing a function f, `@allowscalar` 🤔

```julia
using ForwardDiff, Krylov, CUDA
using GPUArrays

xk = -ones(4)
f(x) = (x[1] - 1)^2 + (x[2] - 2)^2 + (x[3] - 3)^2 + (x[4] - 4)^2
g(x) = ForwardDiff.gradient(f, x)
H(x) = ForwardDiff.hessian(f, x)
@allowscalar d, stats = cg(H(xk), -g(xk))
xk += d # [1,2,3,4]
```

WHY SO FAR ONLY IN JULIA?

- Compute gradient of `f(x) = exp(dot(x,x))`
- CUDA, linear algebra, and forward automatic differentiation are provided by different packages (**separation of concerns**) and are composed at runtime (**just-in-time compilation**).
- The code for the gradient of $f(x)$ in CUDA is created and only exists at runtime!

```
1 using CUDA
2 using LinearAlgebra
3 using ForwardDiff
4 f(x) = exp(dot(x,x))
5 x = Array([2.0,3.0])
6 f(x)
7 @code_llvm(f(x)) # Look at the LLVM IR
8 f(CuArray(x))
9 @show g = ForwardDiff.gradient(f, CuArray(x))
```

- Compact code with three off the shelf packages from the official Julia package registry that leverage 1-4 on the left

NON-PORTABLE VENDOR LIBRARIES

Example: Sparse Direct Linear Solver CUDSS

- CUDSS is the state-of-the-art sparse direct linear solver for GPUs
- No other GPU vendor has a performant sparse direct solver
- Julia cannot avoid vendor specifics, but tries to unify API as much as possible
- CUDSS Julia interface is maintained by Alexis Montoison (MCS)
- Provides a generic Julia interface for linear solvers/factorizations
- For best performance CUDSS API

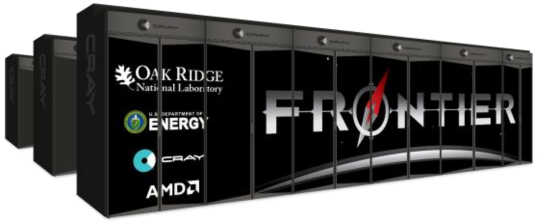
ARCHITECTURE OVERVIEW

PAST

Many core with GPUs

CURRENT

Back to vector processors (GPUs)



Future

NVIDIA and Oracle to Build US Department of Energy's Largest AI Supercomputer for Scientific Discovery

Bold US Investment of 100,000 NVIDIA Blackwell GPUs Kickstarts Era of Agentic AI-Powered Science at Argonne National Laboratory for Public Researchers

October 28, 2025

FPGAs, wafer computer (LLVM based compilers)

JULIA AT ALCF: POLARIS, AURORA, AND BEYOND

JULIA AT ALCF: GOAL

Goal

- module load Julia
- Support for GPUs, MPI, filesystems, and large-scale submissions

Julia-specific challenges

- Compiled at runtime requires file access by every process
- Offline compilation of binaries is quirky (Support Julia 1.12)
- Julia's package manager
- Artifacts and state in Manifest.toml

CURRENT STATUS: EXPERIMENTAL

- Part of official documentation: https://docs.alcf.anl.gov/polaris/programming-models/julia/
- No system deployment
- Module files and setup are found in https://github.com/anlsys/julia_alcf

README MIT license

Julia Environment Setup for Polaris and Aurora at ALCF

This repository contains scripts and module files to set up a Julia programming environment on the Polaris and Aurora supercomputers at the Argonne Leadership Computing Facility (ALCF). The setup includes configuration for Julia's depot path, module files for easy loading of the Julia environment, and necessary dependencies.

Quick Start

1. Clone this repository to your local machine or directly on the ALCF system.

```
git clone https://github.com/anlsys/julia_alcf.git
```

U.S. DEPARTMENT of ENERGY
Argonne National Laboratory is a U.S. Department of Energy laboratory managed by UChicago Argonne, LLC.
Argonne NATIONAL LABORATORY

SETUP

- Do not install in your home directory!
- `git clone https://github.com/anlsys/julia_alcf.git`
- `cd julia_alcf/Polaris` or `cd julia_alcf/aurora`
- `./setup.sh`
- It will prompt for a `JULIA_DEPOT_PATH`. Select a folder on a fast filesystem and accessible by the compute nodes
- It will install the latest Julia (1.12.1)
- It will install module files and a `LocalPreferences.toml` into `JULIA_DEPOT_PATH`

LOAD JULIA

- `module use $JULIA_DEPOT_PATH/modulefiles && module load julia`
- Add one line to your shell initialization scripts (e.g., `~/.bashrc`)
  - `export JULIA_DEPOT_PATH=YOUR_DEPOT_PATH`
- All done!

USING JULIA ON AN ALCF CLUSTER

- Clone your project with the Project.toml and do the usual setup on the login nodes
  - `julia –project=.`
  - `] update`
- If your code uses CUDA.jl you cannot run it using GPUs on the login nodes!
- Using MPI would precompile your code on every process (potential call from filesystem team)
- Run the code on 1 node to precompile as much as possible
  - `qsub –I –l select=1,walltime=1:00:00,filesystems=home:eagle –A [PROJECT] –q debug`
  - `julia –project simulation.jl`
- Extreme large-scale runs (not recommended right now, contact us)
  - Compress your `JULIA_DEPOT_PATH` and move it to the local SSDs with the submission script

FUTURE

- We will support Intel GPUs the best we can through maintenance of oneAPI.jl
  - https://github.com/JuliaGPU/oneAPI.jl
- Support for CUDA is far more mature
- Post-Aurora system will be NVIDIA based
  - Great support more unified memory and other CUDA features
  - Support for NVIDIA libraries
- Rollout of a system-wide module in the coming months

Thank You

TALK STRUCTURE

- Why Julia
- What is Julia?
  - Multiple dispatch, JIT in LLVM, functional programming
- Julia and HPC Foundations
  - BLAS
  - GPUCompiler.jl, CUDA.jl, oneAPI.jl, KernelAbstractions.jl
  - MPI
  - Sparse Linear Solver
- Julia on Supercomputers
  - Challenges
    - Artifacts, local libraries, (pre)compilation, filesystems, and ABIs
  - Current Solution
    - Julia and system depot_path are provided on local SSDs, user depot_path self-managed on project filesystem
  - Best practices
    - How to submit job and precompile
- Status and Outlook
  - Module load Julia: Adds Julia to path, provides a system depot_path with precompiled oneAPI.jl/CUDA.jl, MPI.jl, and HDF5.jl
  - Agenda with rollout, links to current setup on Github
  - Disclaimer: Experimental, we need feedback