Scientific Software Design

Anshu Dubey (she/her)
Argonne National Laboratory

Better Scientific Software tutorial @ ISC23

Contributors: Anshu Dubey (ANL), Mark C. Miller (LLNL), David Bernholdt (ORNL)

License, Citation and Acknowledgements

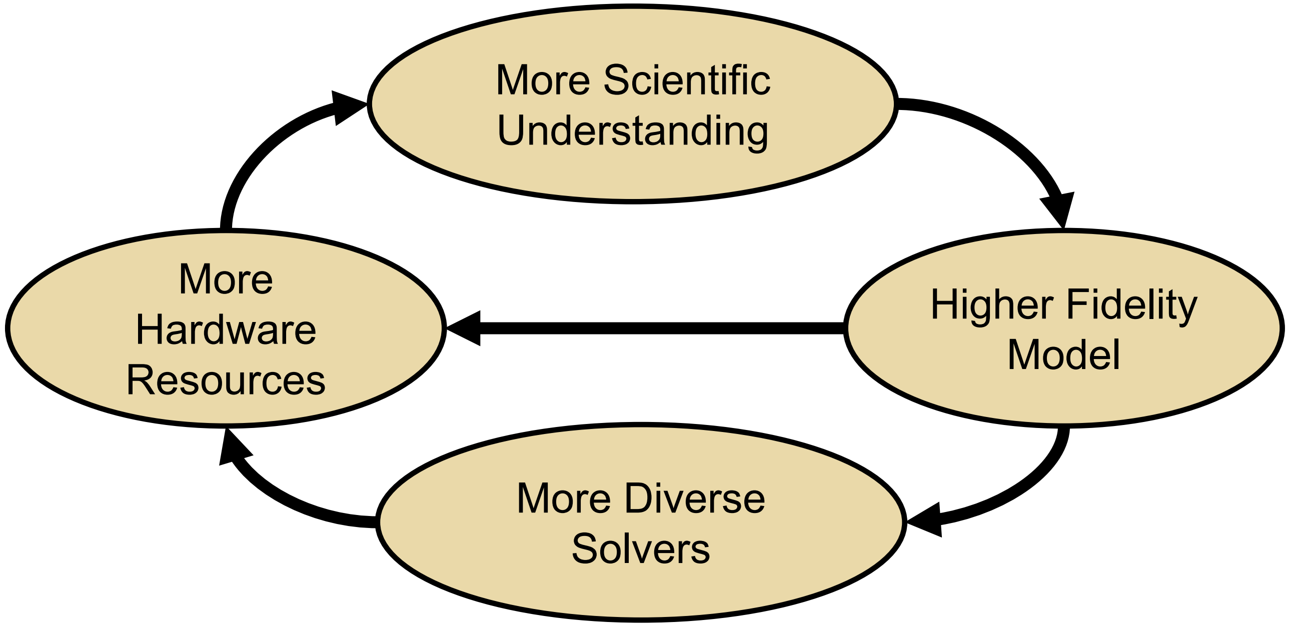
License and Citation

- **The requested citation the overall tutorial is:** Anshu Dubey and David M. Rogers, Better Scientific Software tutorial, in ISC High Performance, Hamburg, Germany and online, 2023. DOI: 10.6084/m9.figshare.22790762.
- Individual modules may be cited as *Speaker, Module Title*, in *Tutorial Title*, …

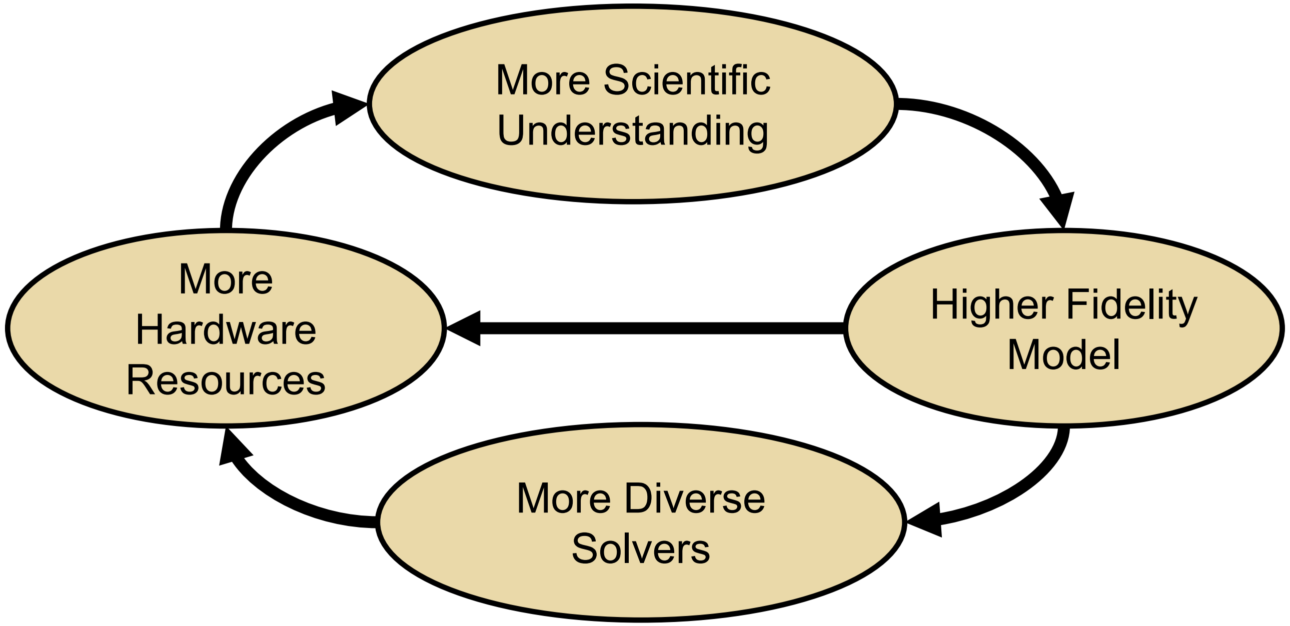
Acknowledgements

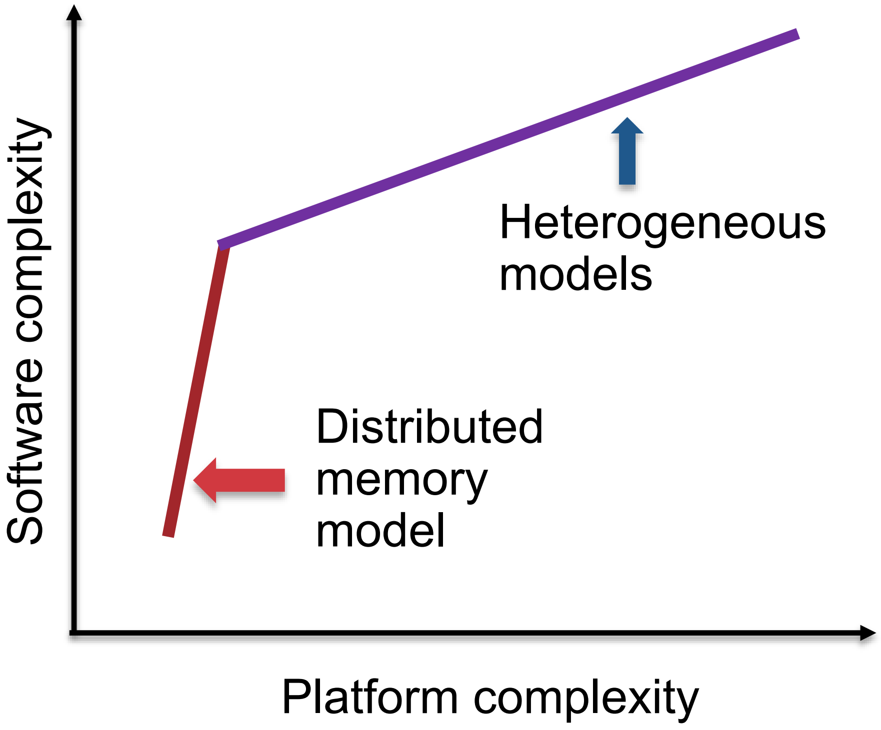
- This work was supported by the U.S. Department of Energy Office of Science, Office of Advanced Scientific Computing Research (ASCR), and by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.
- This work was performed in part at the Argonne National Laboratory, which is managed by UChicago Argonne, LLC for the U.S. Department of Energy under Contract No. DE-AC02-06CH11357.
- This work was performed in part at the Lawrence Livermore National Laboratory, which is managed by Lawrence Livermore National Security, LLC for the U.S. Department of Energy under Contract No. DE-AC52-07NA27344.
- This work was performed in part at the Los Alamos National Laboratory, which is managed by Triad National Security, LLC for the U.S. Department of Energy under Contract No.89233218CNA000001
- This work was performed in part at the Oak Ridge National Laboratory, which is managed by UT-Battelle, LLC for the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.
- This work was performed in part at Sandia National Laboratories. Sandia National Laboratories is a multi-mission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

HPC Computational Science Use-case

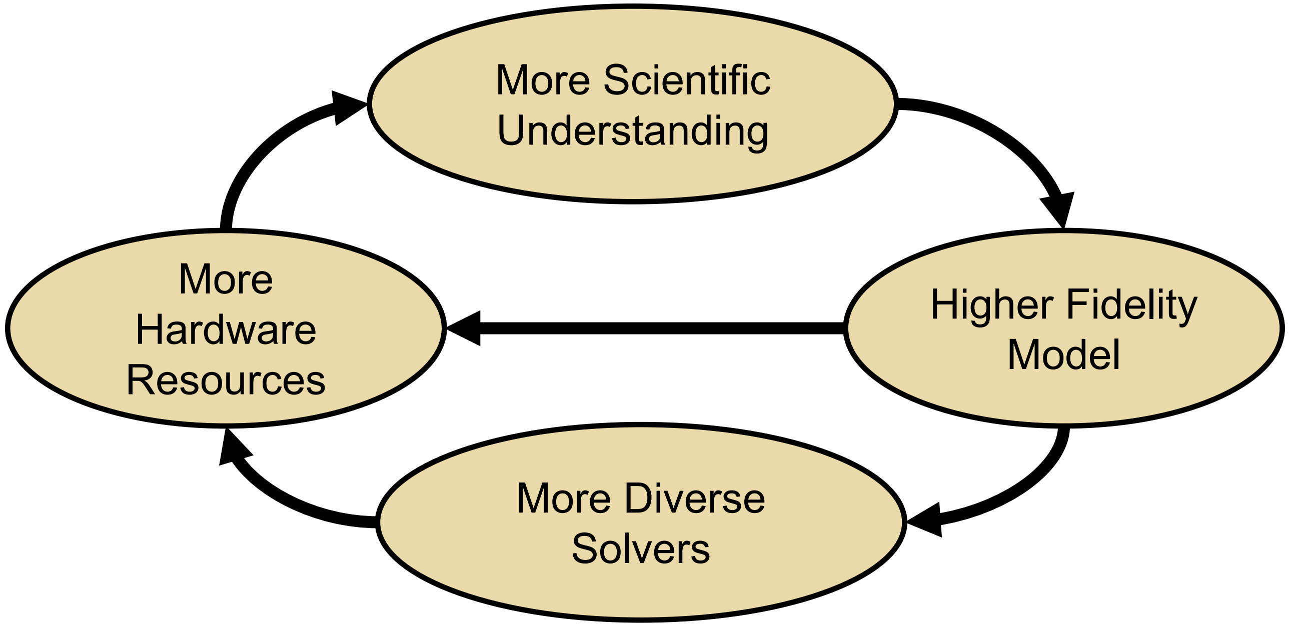


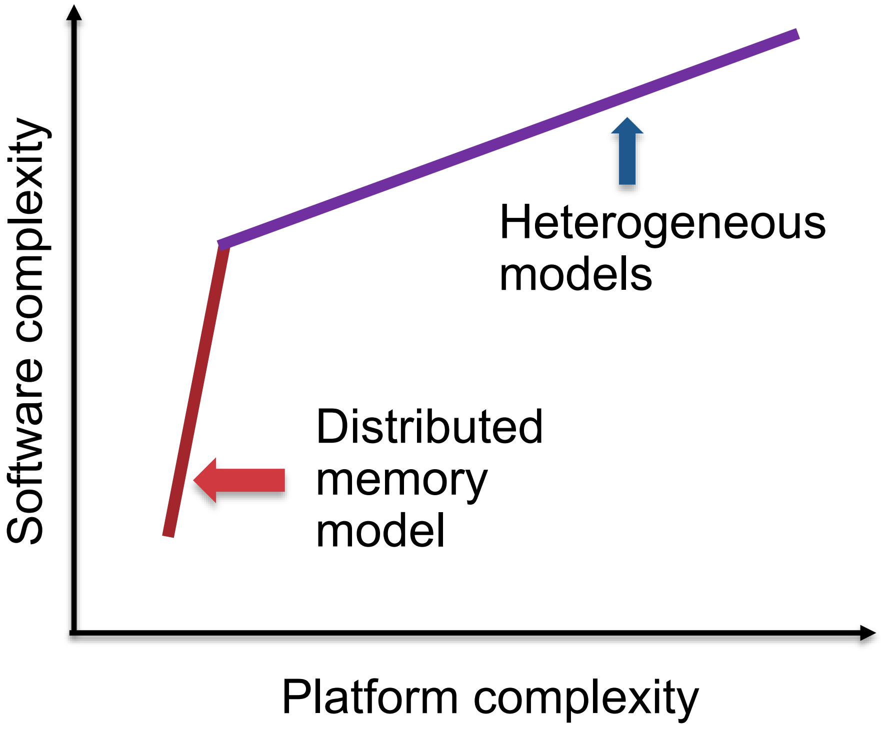
HPC Computational Science Use-case





HPC Computational Science Use-case





- ❑ Many components may be under research
- ❑ Software continuously evolves
- ❑ All use cases are different and unique

General Design Principles for HPC Scientific Software

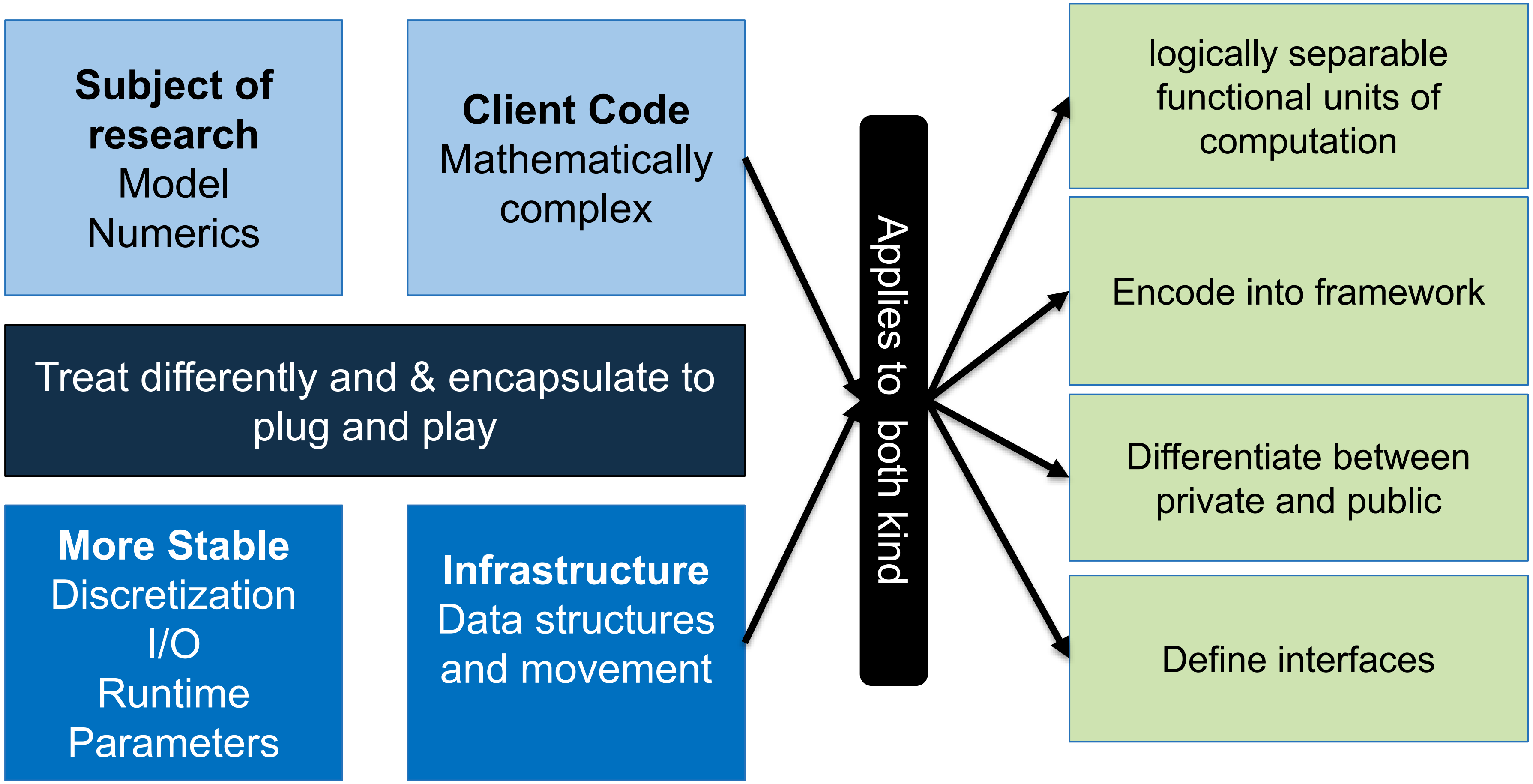
Considerations

- Multidisciplinary teams
  - Many facets of knowledge
  - To know everything is not feasible
- Two types of code components
  - Infrastructure (mesh/IO/runtime …)
  - Science models (numerical methods)
- Codes grow
  - New ideas => new features
  - Code reuse by others

Design Implications

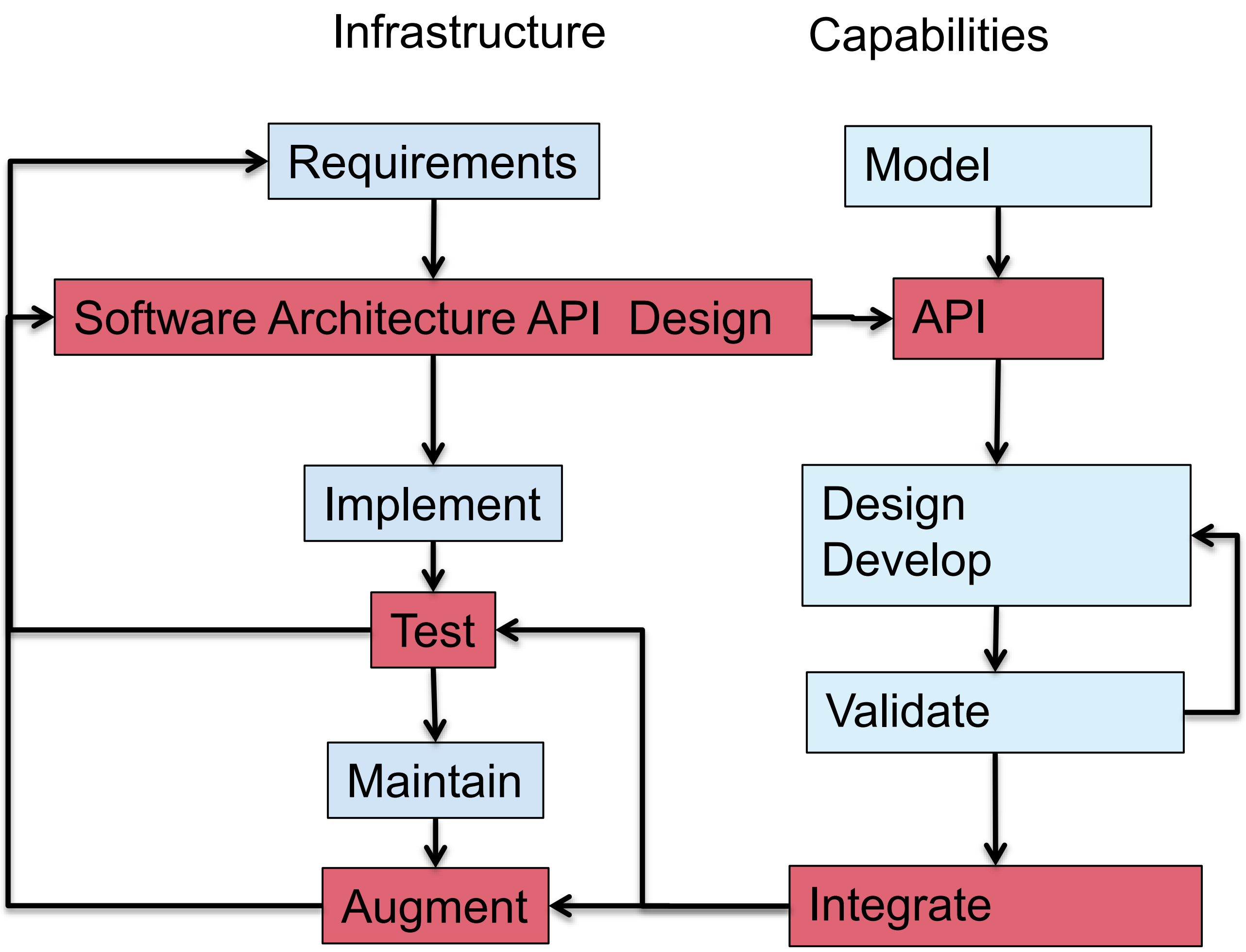
- Separation of Concerns
  - Shield developers from unnecessary complexities
- Work with different lifecycles
  - Long-lasting vs quick changing
  - Logically vs mathematically complex
- Extensibility built in
  - Ease of adding new capabilities
  - Customizing existing capabilities

General Design Principles for HPC Scientific Software



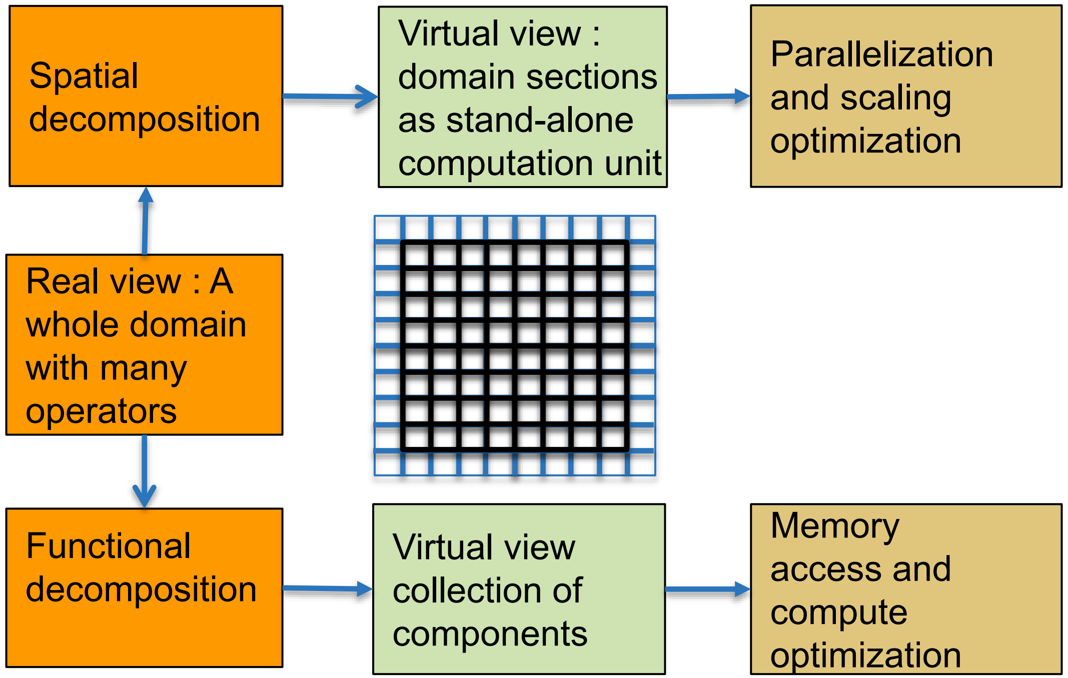
Design first, then apply programming model to the design instead of taking a programming model and fitting your design to it.

A Design Model for Separation of Concerns



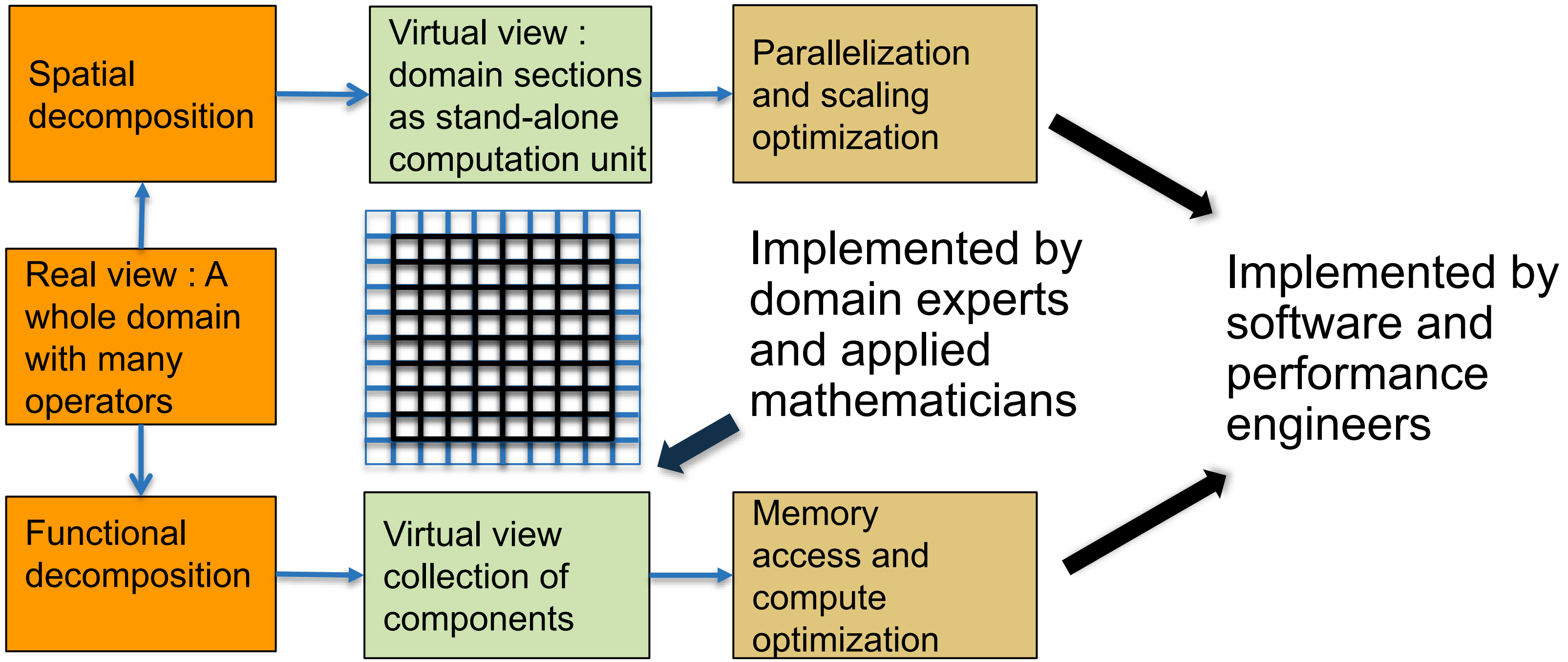
Example: Architecting Multiphysics PDEs

- Virtual view of functionalities
- Decomposition into units and definition of interfaces

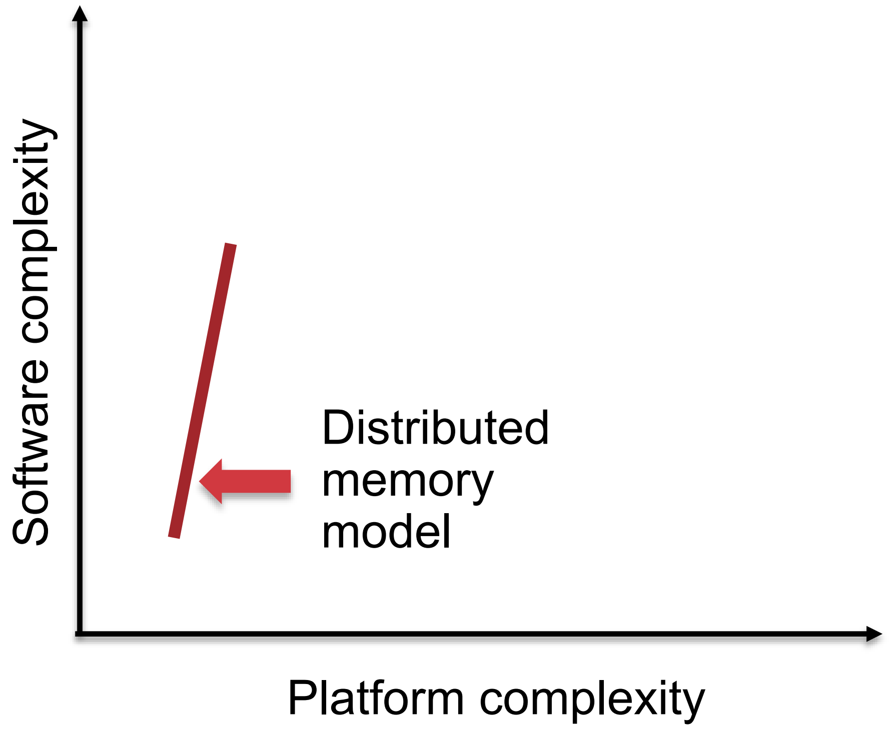


Example: Multiphysics PDEs for Distributed Memory Parallelism

- Virtual view of functionalities
- Decomposition into units and definition of interfaces

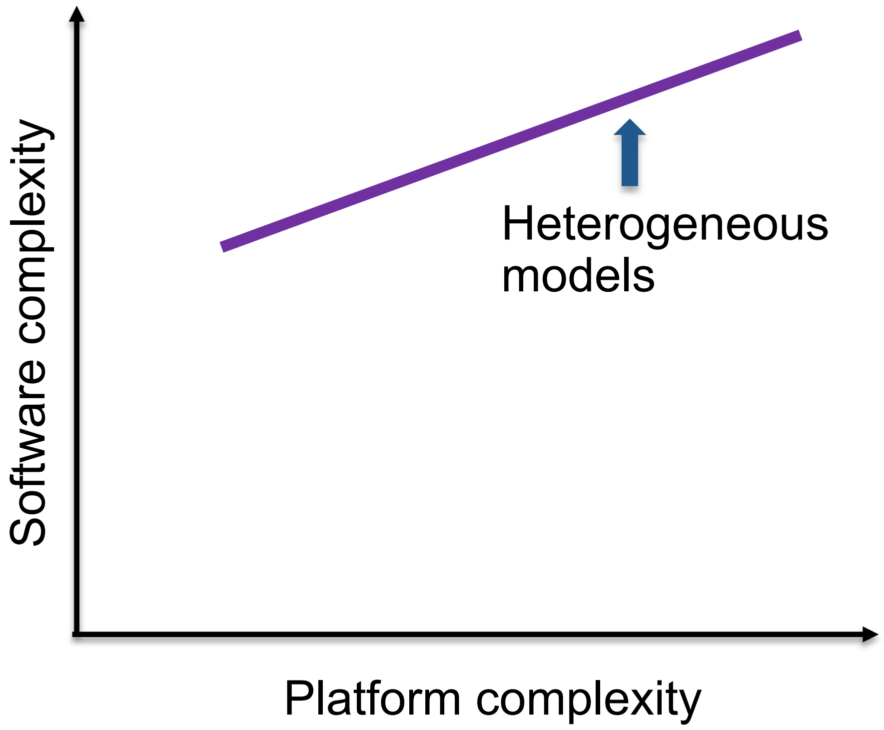


Takeaways Until Now



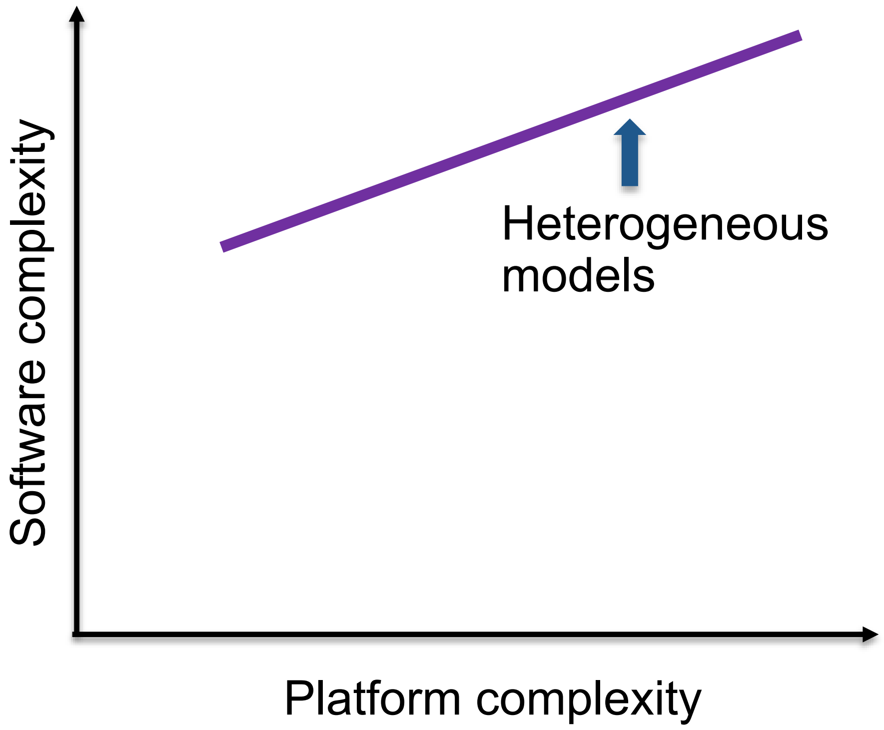
- Differentiate between slow changing and fast changing components of your code
- Understand the requirements of your infrastructure
- Implement separation of concerns
- Design with portability, extensibility, reproducibility and maintainability in mind
- Do not design with a specific programming model in mind

A New Paradigm Because of Platform Heterogeneity



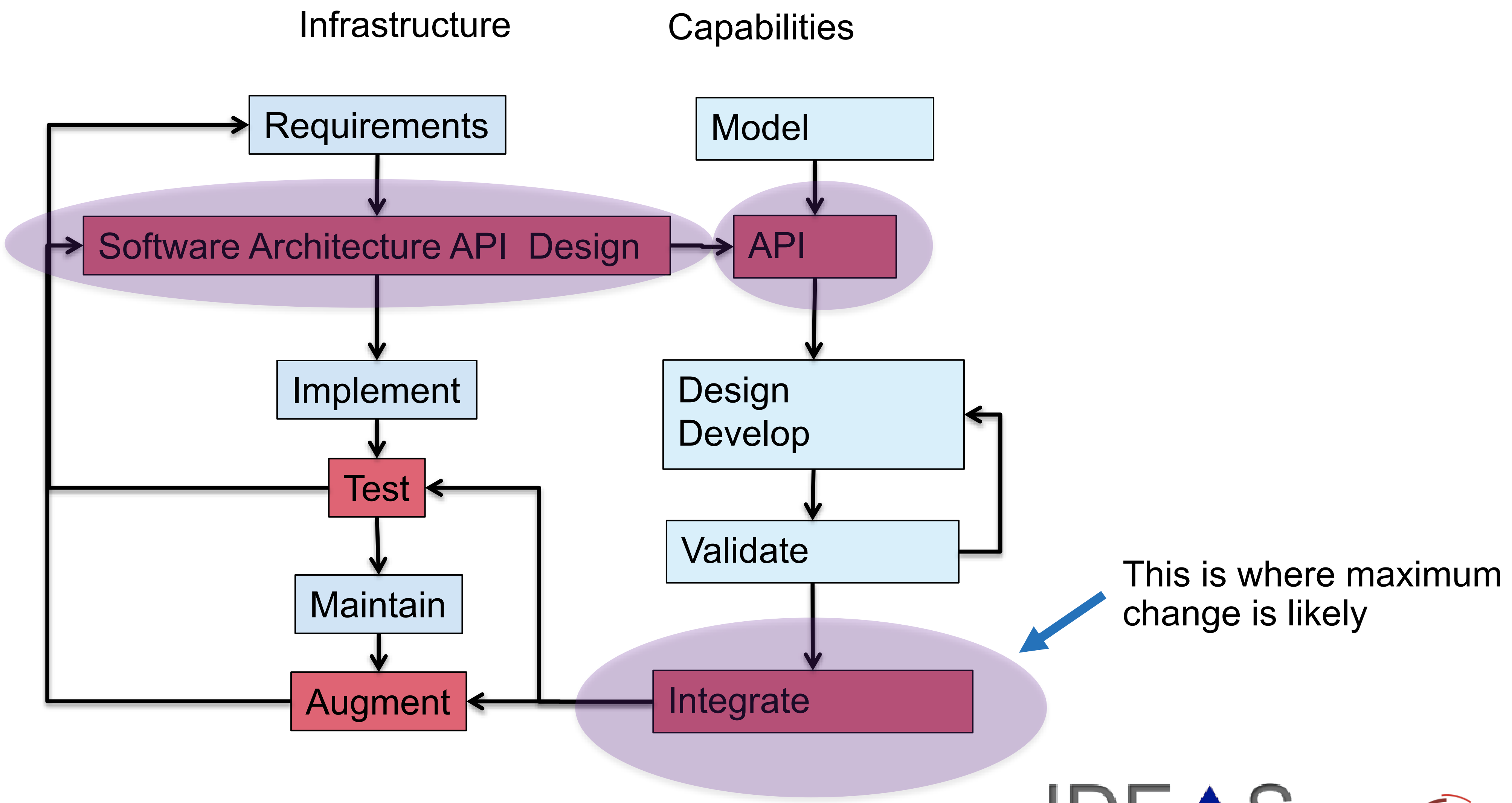
- Question - do the design principles change?

A New Paradigm Because of Platform Heterogeneity



- Question - do the design principles change?
- The answer is – not really
- The details get more involved

Handling Heterogeneity – Hardware and Software



Platform Heterogeneity

Computation

Memory

Network

Platform Heterogeneity

Computation

Memory

Network

CPU

GPU

Other acceler-ators

Other devices

Platform Heterogeneity

Computation

CPU

GPU

Other accelero -ators

Other devices

Memory

Cache hierar- chy

Device memory

NVram

Other types

Network

Platform Heterogeneity

Computation
- CPU
- GPU
- Other acceler-ators
- Other devices

Memory
- Cache hierar-chy
- Device memory
- NVram
- Other types

Network
- Between nodes
- Within node
- With I/O
- Other types

Platform Heterogeneity

Computation		Memory		Network	
CPU	GPU	Cache hierar-chy	Device memory	Between nodes	Within node
Other acceler-ators	Other devices	NVram	Other types	With I/O	Other types

And memory access models: unified memory / gpu-direct / explicit transfer

Mechanisms Needed by the Code

Mechanisms to unify expression of computation

- Minimize maintained variants of source suitable for all computational devices
- Reconcile differences in data structures

Mechanisms Needed by the Code

Mechanisms to unify expression of computation

- Minimize maintained variants of source suitable for all computational devices
- Reconcile differences in data structures

Mechanisms to move work and data to computational targets

- Moving between devices
  - Launching work at the destination
  - Hiding latency of movement
- Moving data off node

Mechanisms Needed by the Code

Mechanisms to unify expression of computation
- Minimize maintained variants of source suitable for all computational devices
- Reconcile differences in data structures

Mechanisms to move work and data to computational targets
- Moving between devices
  - Launching work at the destination
  - Hiding latency of movement
- Moving data off node

Mechanisms to map work to computational targets
- Figuring out the map
  - Expression of dependencies
  - Cost models
- Expressing the map

Mechanisms Needed by the Code

Mechanisms to unify expression of computation

- Minimize maintained variants of source suitable for all computational devices
- Reconcile differences in data structures

Mechanisms to move work and data to computational targets

- Moving between devices
  - Launching work at the destination
  - Hiding latency of movement
- Moving data off node

Mechanisms to map work to computational targets

- Figuring out the map
  - Expression of dependencies
  - Cost models
- Expressing the map

So, what do we need?

- Abstractions layers
- Code transformation tools
- Data movement orchestrators

Underlying Ideas: Unification of Computational Expressions

Make the same code work on different devices

Same algorithm different data layouts or operation sequence:

- A way to let compiler know that ”this” expression can be specialized in many ways
- Definition of specializations
- Often done with template meta-programming

More challenging if algorithms need to be fundamentally different

- Support for alternatives

Underlying Ideas: Moving Work and Data to the Target

Parallelization Models

Hierarchy in domain decomposition

- Distributed memory model at node level – still very prevalent, likely to remain so for a while
- Also done with PGAS models – shared with locality being important

Assigning work within the node

- "Parallel For" or directives with unified memory
- Directives or specific programming model for explicit data movement

More complex data orchestration system for asynchronous computation

- Task based work distribution

Underlying Ideas: Mapping Work to Targets

This is how many abstraction layers work

- Infer the structure of the code
- Infer the map between algorithms and devices
- Infer the data movements
- Map computations to devices
- These are specified either through constructs or pragmas

It can also be the end user who figures out the mapping
In either case performance depends upon how well the mapping is done

Mechanisms Needed by the Code : Example Flash-X

Mechanisms to unify expression of computation

Macros with inheritance

Mechanisms Needed by the Code : Example Flash-X

Mechanisms to unify expression of computation

Macros with inheritance

Mechanisms to move work and data to computational targets

Domain specific runtime

Mechanisms Needed by the Code : Example Flash-X

Mechanisms to unify expression of computation

Macros with inheritance

Mechanisms to move work and data to computational targets

Domain specific runtime

Mechanisms to map work to computational targets

DSL for recipes with code generator

Mechanisms Needed by the Code : Example Flash-X

Mechanisms to unify expression of computation

Macros with inheritance

Mechanisms to move work and data to computational targets

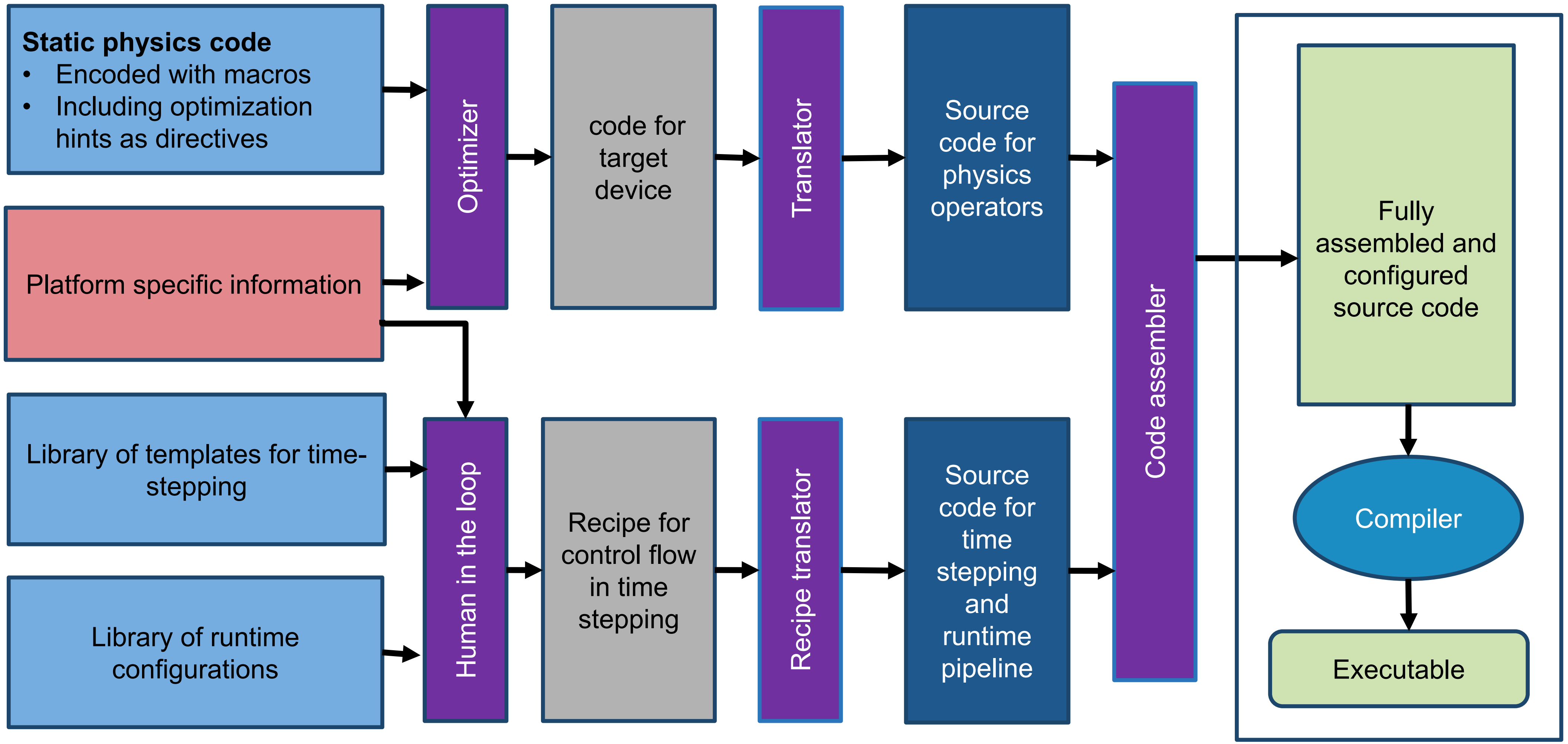
Domain specific runtime

Mechanisms to map work to computational targets

DSL for recipes with code generator

Composability in the source
A toolset of each mechanism
Independent tool sets

Overview of Flash-X Design Approach with Separation of Concerns in tools



Final takeaways

- The key to both performance portability and longevity is careful software design
- Extensibility should be built into the design
- Design should be independent of any specific programming model
- Composability and flexibility help with performance portability
- Resources:
  - https://www.exascaleproject.org/
  - https://doi.org/10.6084/m9.figshare.13283714.v1
  - https://bssw.io/blog_posts/performance-portability-and-the-exascale-computing-project
  - https://www.exascaleproject.org/event/kokkos-class-series
  - A Design Proposal for a Next Generation Scientific Software Framework
  - Software Design for Longevity with Performance Portability