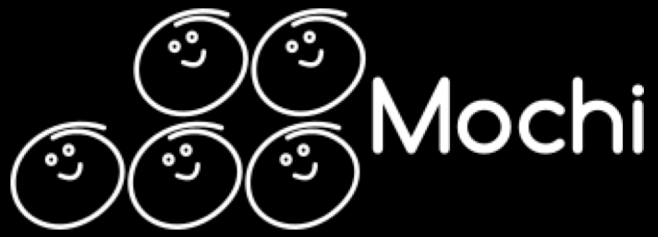


Methodology for the Rapid Development of Scalable HPC Data Services

Matthieu Dorier, Philip Carns, Kevin Harms, Robert Latham, Robert Ross, Shane Snyder,
Justin Wozniak, Samuel K. Gituérrez, Bob Robey, Brad Settlemyer, Galen Shipman,
Jerome Soumagne, James Kowalkowski, Marc Paterno, Saba Sehrish

PDSW-DISCS 2018
Dallas, TX

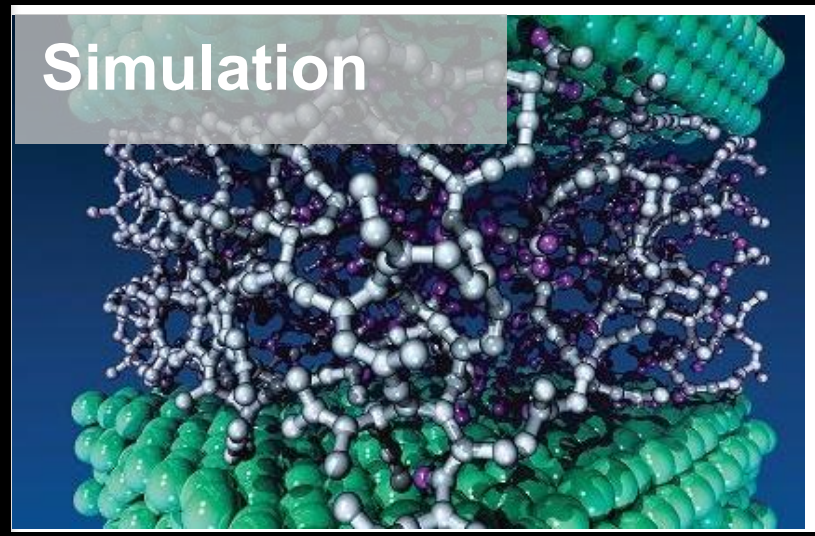
Argonne NATIONAL LABORATORY

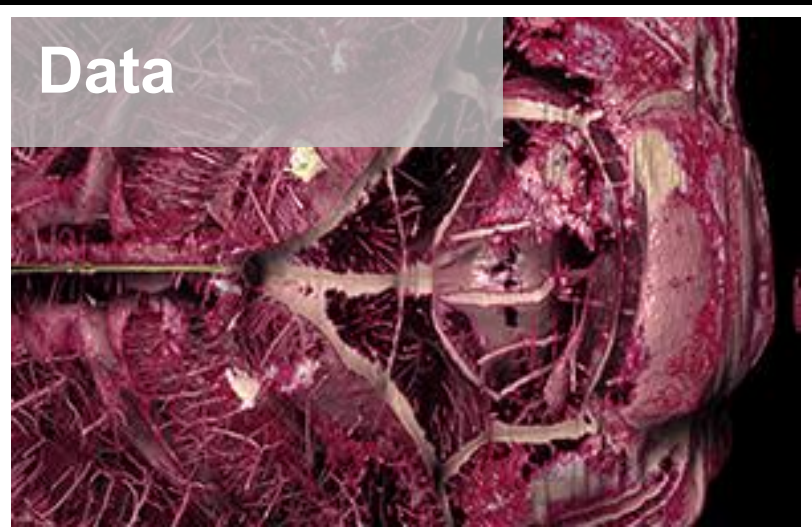
Los Alamos NATIONAL LABORATORY EST. 1943

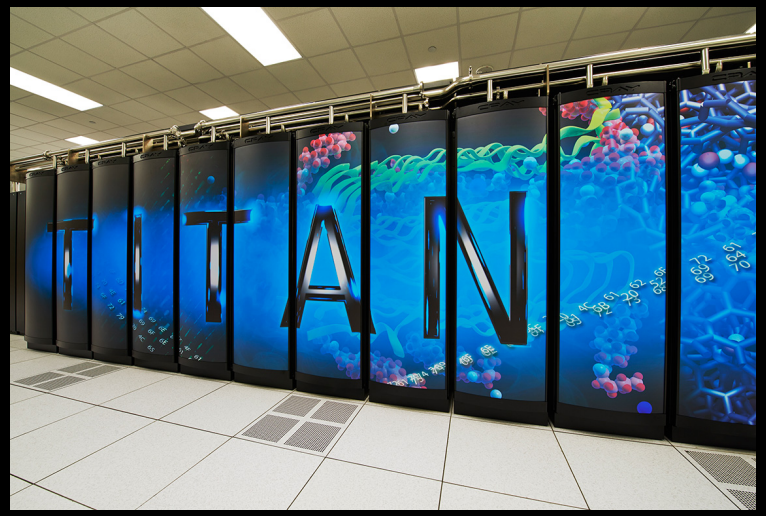
HDF

Fermilab

New Applications and Systems: Demand for New Services







Top image credit B. Helland (ASCR). Bottom left and right images credit ALCF. Bottom center image credit OLCF.

New Applications and Systems: Demand for New Services

- Different application use cases have different data needs
- "One size fits all" doesn't work: need customized data services for each to meet mission goals
- This poses a significant technical challenge: how to enable rapid development of such services (*agility*) while still preserving performance (*efficiency*) and production quality (*maintainability*)

Key idea: address this challenge via composable data services.

Top image credit B. Helland (ASCR). Bottom left and right images credit ALCF. Bottom center image credit OLCF.

Towards reusable components for data services

Advantages
- Well established
- Standard interface

Drawbacks
- Single consistency model
- Complex to maintain and tune
- Files often inappropriate

Towards reusable components for data services

Parallel File Systems

Hand-crafted Data Services

Advantages
- Well established
- Standard interface

Drawbacks
- Single consistency model
- Complex to maintain and tune
- Files often inappropriate

Advantages
- Tuned for this application
- Appropriate consistency model
- Appropriate data model

Drawbacks
- Difficult to maintain
- Not reusable
- Scare users

Towards reusable components for data services

Parallel File Systems

Hand-crafted Data Services

Advantages
- Well established
- Standard interface

Drawbacks
- Single consistency model
- Complex to maintain and tune
- Files often inappropriate

Composable micro-services

- Reusable across services
- Easy to maintain
- Not so scary to users
- Adaptable, configurable
- Can use latest tech

Advantages
- Tuned for this application
- Appropriate consistency model
- Appropriate data model

Drawbacks
- Difficult to maintain
- Not reusable
- Scare users

Common capabilities need by data services

- Runtime substrate
  - RPC, RDMA
  - Threading/Tasking
- Core components
  - Bulk storage management
  - Key/Value storage
  - Group membership
  - Diagnostics and monitoring
- Programmability/Expressiveness
  - Embedded interpreters
  - Wrappers (Python, C++, etc.)

Common capabilities need by data services

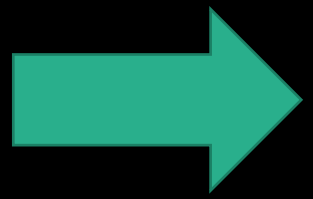
- Runtime substrate
  - RPC, RDMA
  - Threading/Tasking
- Core components
  - Bulk storage management
  - Key/Value storage
  - Group membership
  - Diagnostics and monitoring
- Programmability/Expressiveness
  - Embedded interpreters
  - Wrappers (Python, C++, etc.)

Common capabilities need by data services

- Runtime substrate
  - RPC, RDMA
  - Threading/Tasking
- Core components
  - Bulk storage management
  - Key/Value storage
  - Group membership
  - Diagnostics and monitoring
- Programmability/Expressiveness
  - Embedded interpreters
  - Wrappers (Python, C++, etc.)

Common capabilities need by data services

- Runtime substrate
  - RPC, RDMA
  - Threading/Tasking
- Core components
  - Bulk storage management
  - Key/Value storage
  - Group membership
  - Diagnostics and monitoring
- Programmability/Expressiveness
  - Embedded interpreters
  - Wrappers (Python, C++, etc.)



- Composed services
  - FlameStore
  - HEPnOS
  - SDSDKV

Challenges in Composing HPC Microservices

- Formalize composition
- Unify single-process, multi-process, single-node, and multi-node designs
- Maximize efficient use of resources (network, storage)

Mochi

http://www.mcs.anl.gov/research/projects/mochi/

Vision

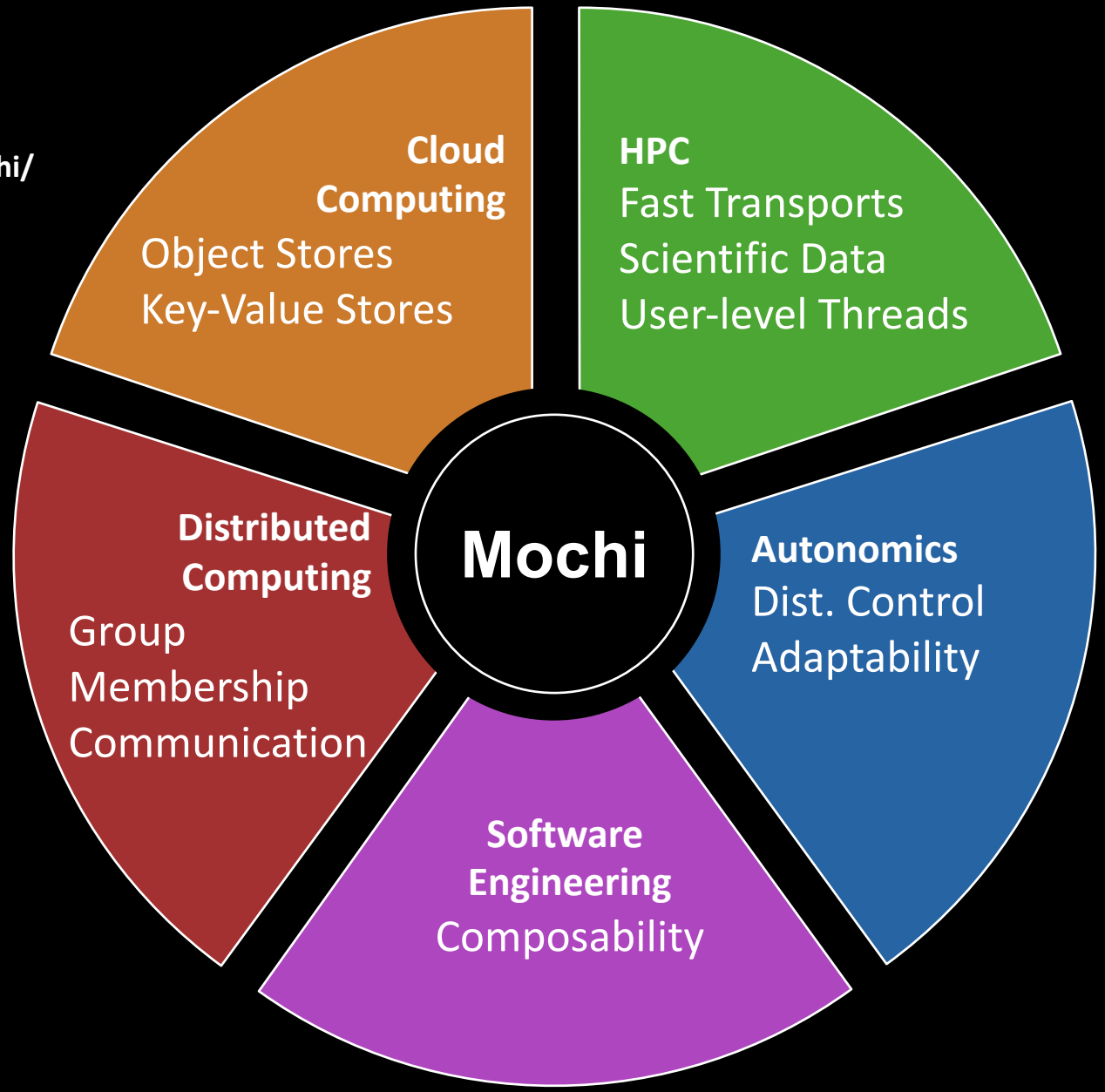
Lowering the barriers to distributed services in computational science.

Approach

- Familiar models (key/value, object, file)
- Easy to build, adapt, and deploy
- Lightweight, user-space components
- Modern hardware support

Impact

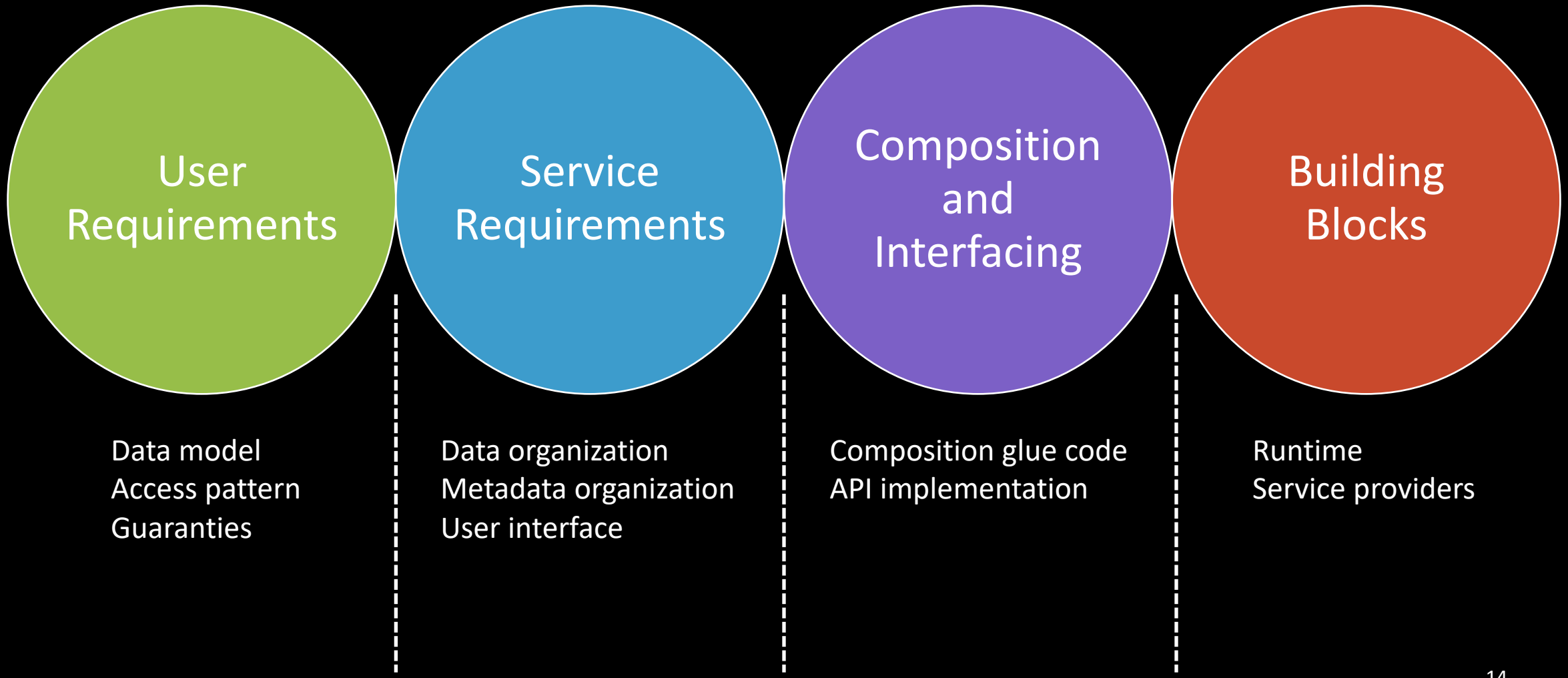
- Better, more capable services for specific use cases on high-end platforms
- Significant code reuse
- Ecosystem for service development



Carnegie Mellon University
Argonne NATIONAL LABORATORY
Los Alamos NATIONAL LABORATORY EST. 1943
HDF

Let’s dive into the methodology

Matching building blocks to user requirements



Identifying application needs

- Which data model?
  - Arrays, meshes, objects
  - Namespace, metadata
- Which access pattern?
  - Characteristics (e.g. access sizes)
  - Collective/individual accesses
- Which guarantees?
  - Consistency
  - Performance
  - Persistence

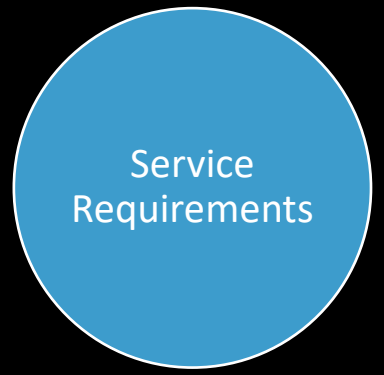
Identifying application needs

- Which data model?
  - Arrays, meshes, objects
  - Namespace, metadata
- Which access pattern?
  - Characteristics (e.g. access sizes)
  - Collective/individual accesses
- Which guarantees?
  - Consistency
  - Performance
  - Persistence

Identifying application needs

- Which data model?
  - Arrays, meshes, objects
  - Namespace, metadata
- Which access pattern?
  - Characteristics (e.g. access sizes)
  - Collective/individual accesses
- Which guarantees?
  - Consistency
  - Performance
  - Persistence

Defining service requirements

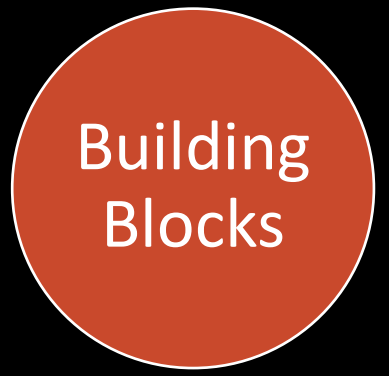


- Which data model?
  - Arrays, meshes, objects
  - Namespace, metadata
- Which access pattern?
  - Characteristics (e.g. access sizes)
  - Collective/individual accesses
- Which guarantees?
  - Consistency
  - Performance
  - Persistence

- How should data be organized?
  - Sharding, distribution, replication
- How should metadata be organized?
  - Distribution, content, indexing
- How do clients interface with the service?
  - Programming language, API

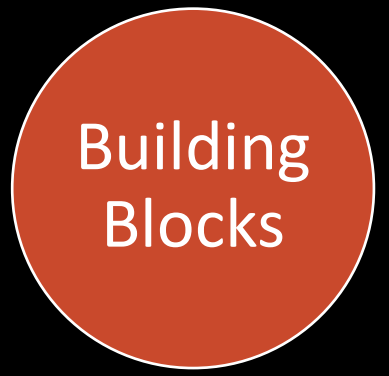
What do components look like?

Components: engineering challenges



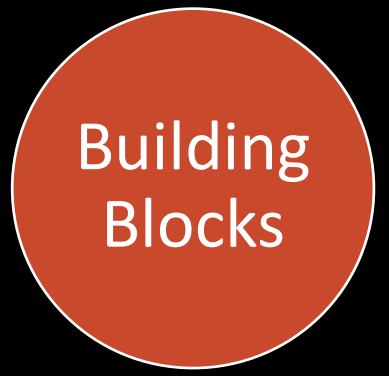
- How do components share resource (CPU, network, memory) without interfering with one another?
  - Bad approach: each component has its own progress loop
- How do we leverage massively multi-core nodes to, for instance, assign components to cores, make components efficiently share a core, prevent components from interfering with network progress?…
  - Bad approach: each component manages its own thread(s)
- How can we support a wide range of networks?
  - Bad approach: reimplement for new transport every time the code is ported to a new platform

Components: engineering challenges



- How do components share resource (CPU, network, memory) without interfering with one another?
  - Bad approach: each component has its own progress loop
- How do we leverage massively multi-core nodes to, for instance, assign components to cores, make components efficiently share a core, prevent components from interfering with network progress?…
  - Bad approach: each component manages its own thread(s)
- How can we support a wide range of networks?
  - Bad approach: reimplement for new transport every time the code is ported to a new platform

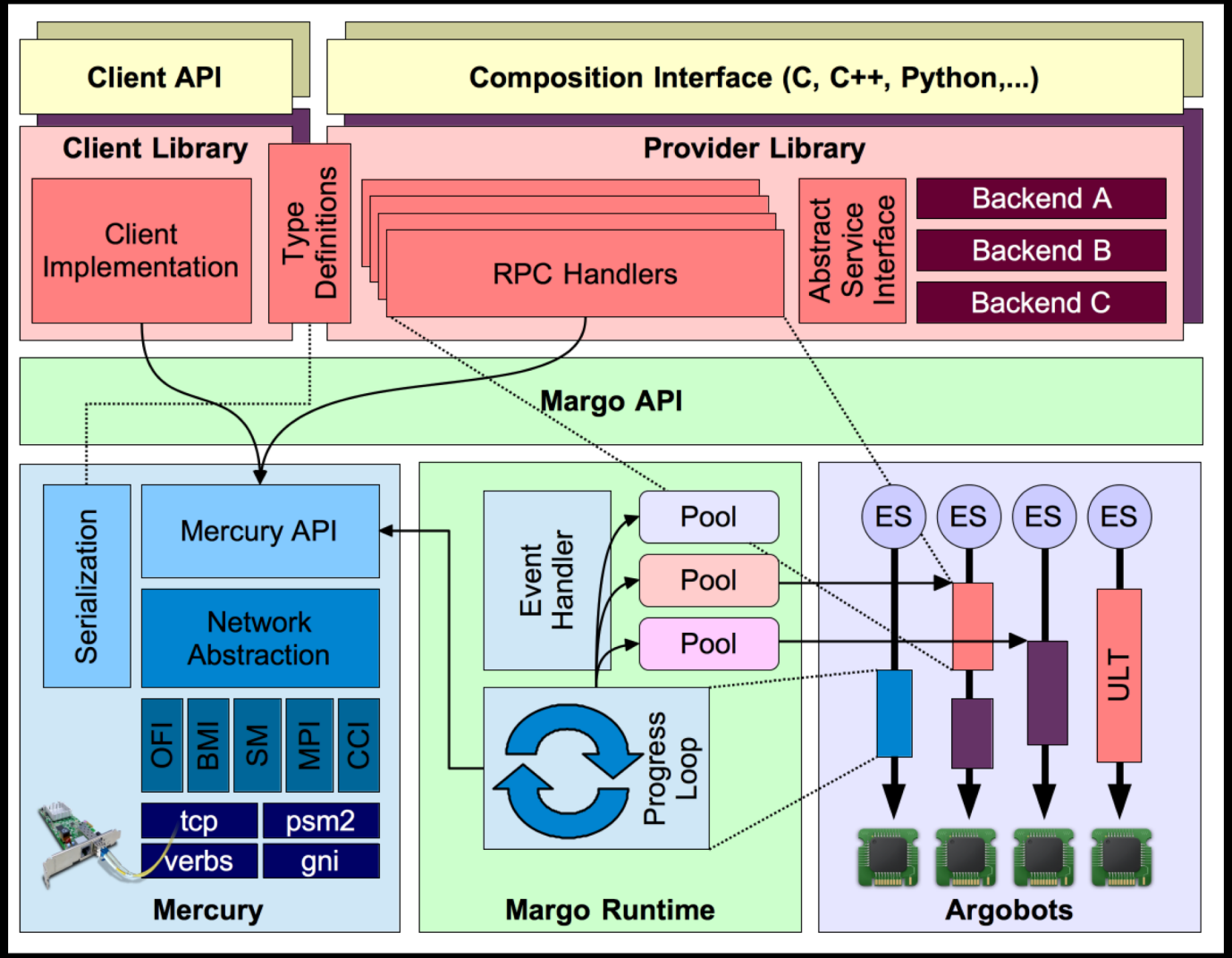
Components: engineering challenges



- How do components share resource (CPU, network, memory) without interfering with one another?
  - Bad approach: each component has its own progress loop
- How do we leverage massively multi-core nodes to, for instance, assign components to cores, make components efficiently share a core, prevent components from interfering with network progress?…
  - Bad approach: each component manages its own thread(s)
- How can we support a wide range of networks?
  - Bad approach: reimplement for new transport every time the code is ported to a new platform

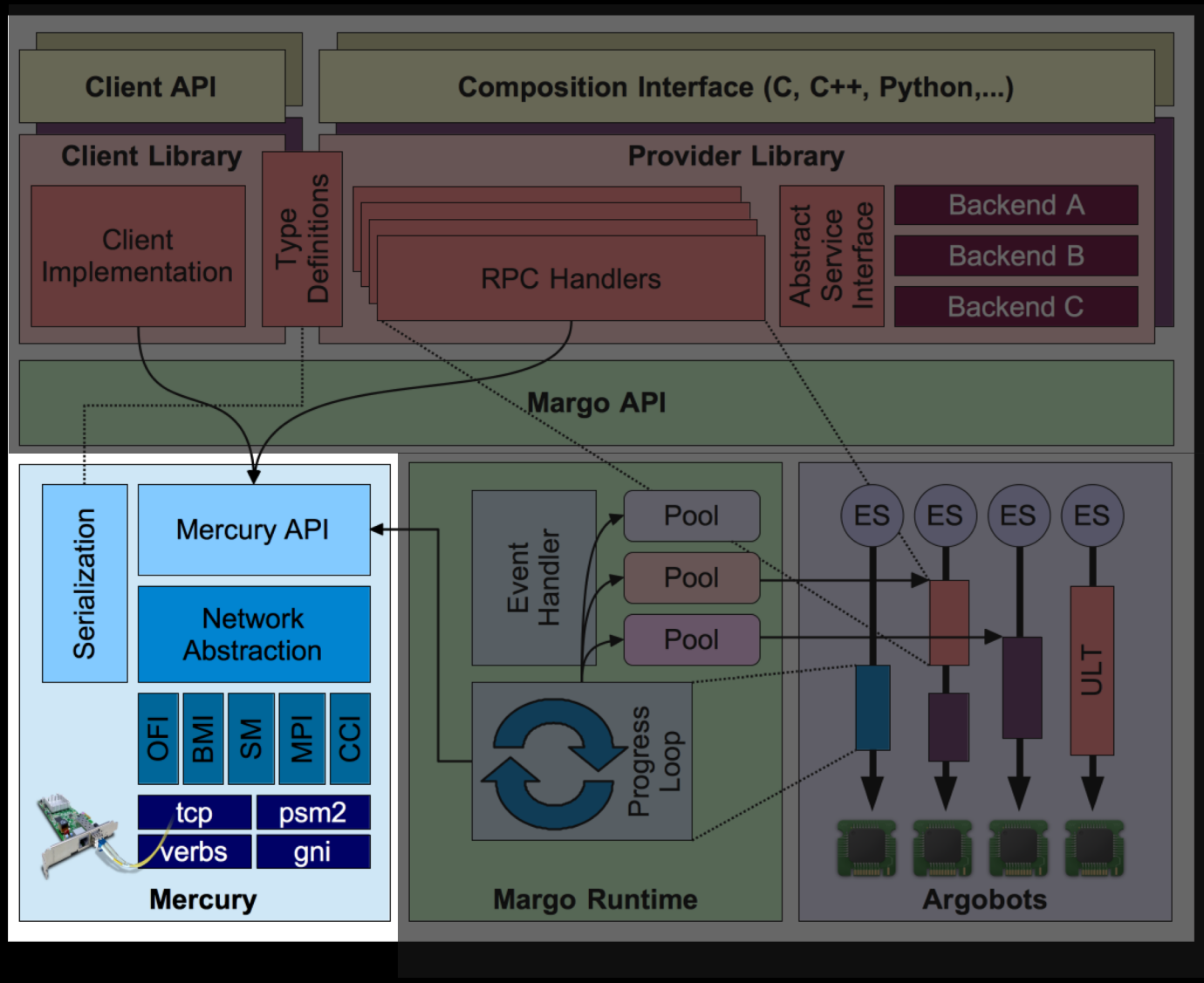
Building Blocks

Anatomy of a Mochi component



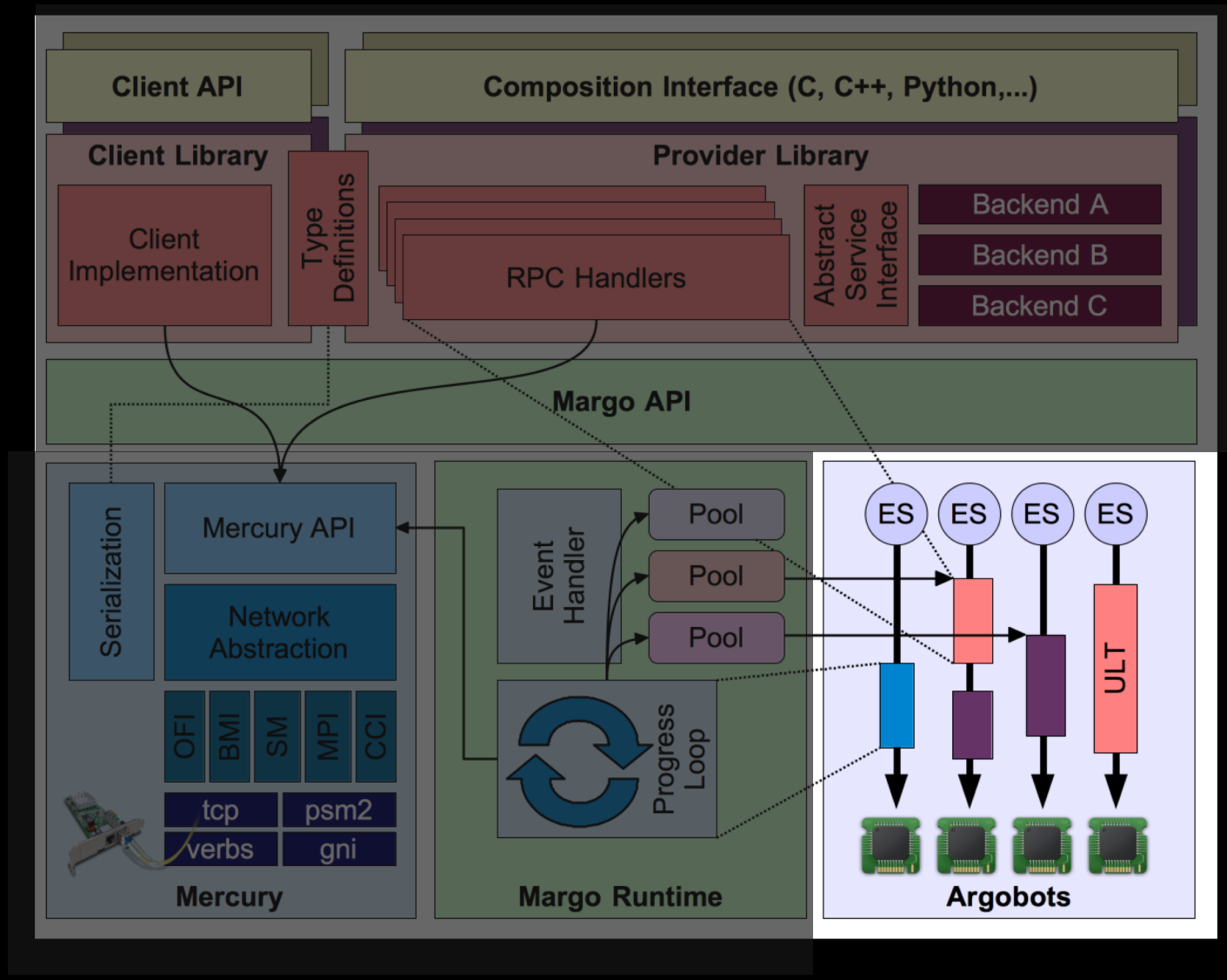
Building Blocks

Anatomy of a Mochi component



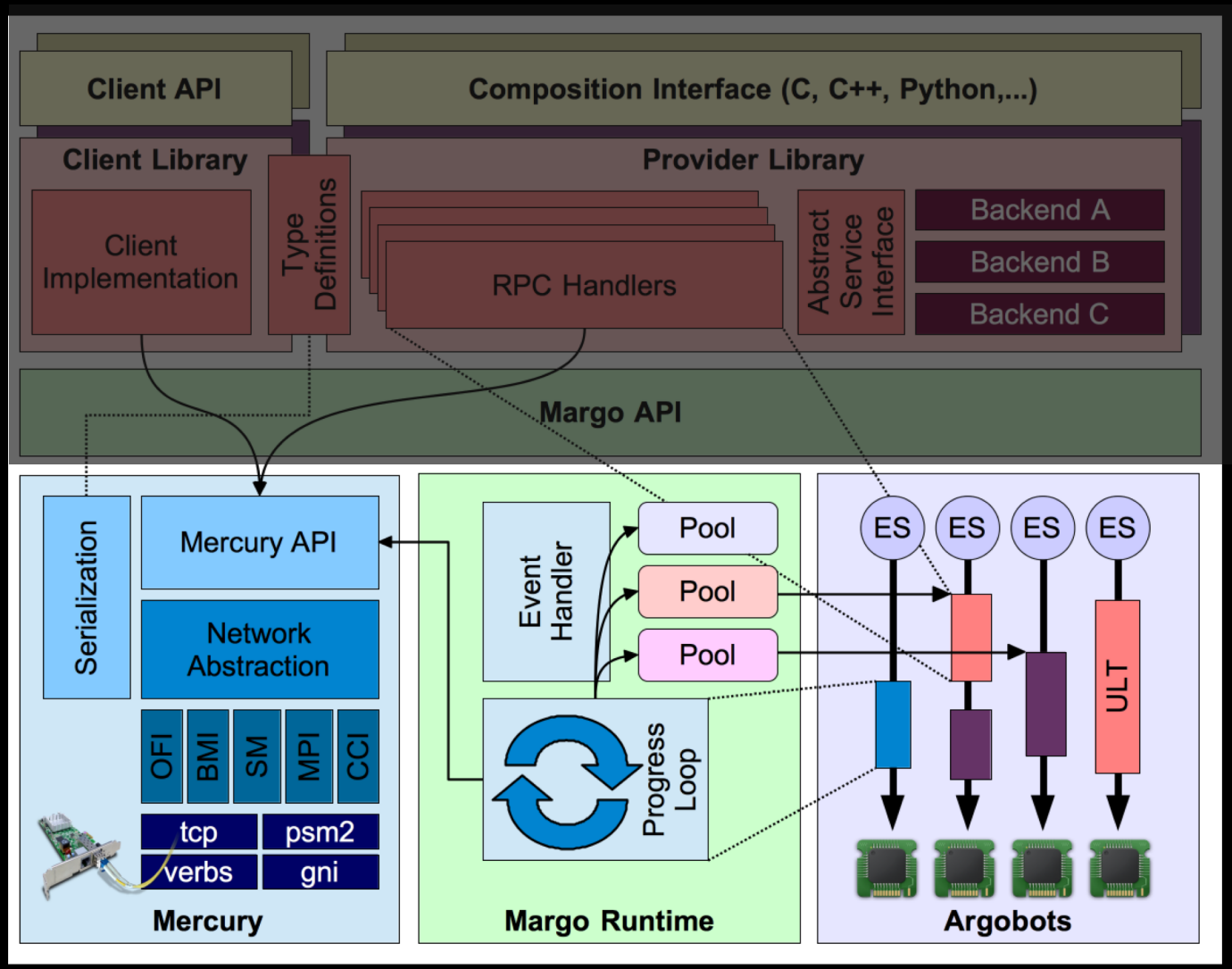
Building Blocks

Anatomy of a Mochi component



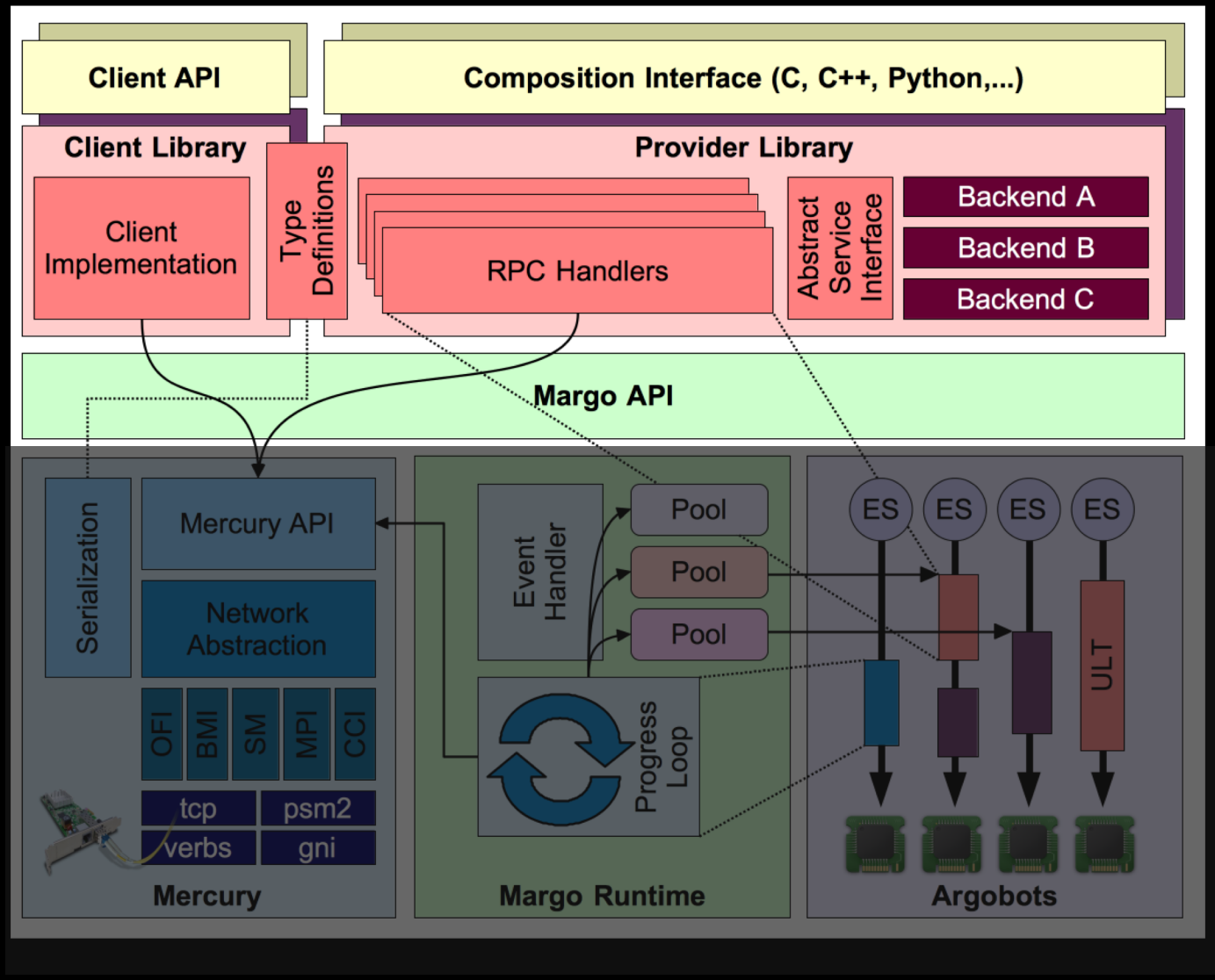
Building Blocks

Anatomy of a Mochi component



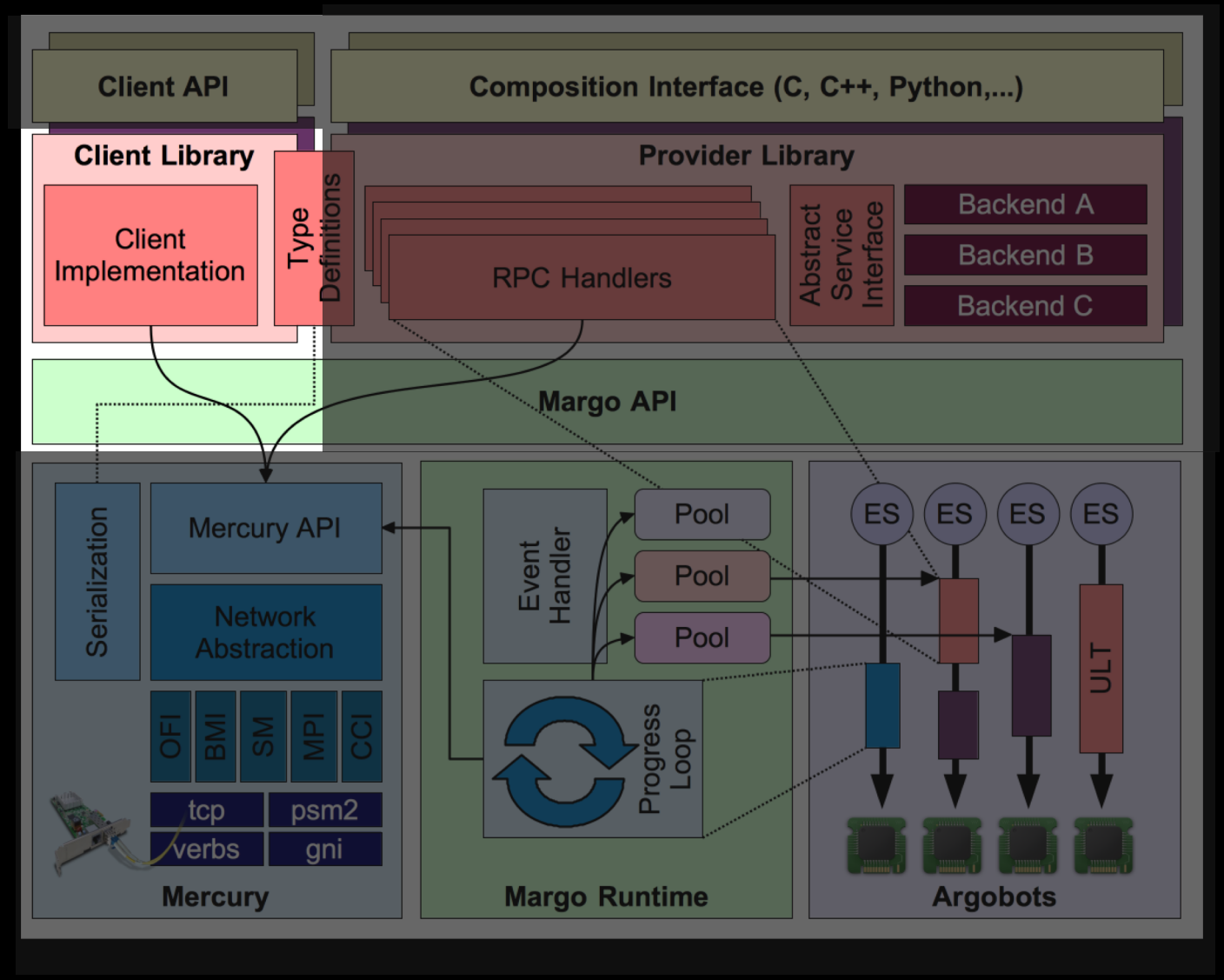
Building Blocks

Anatomy of a Mochi component



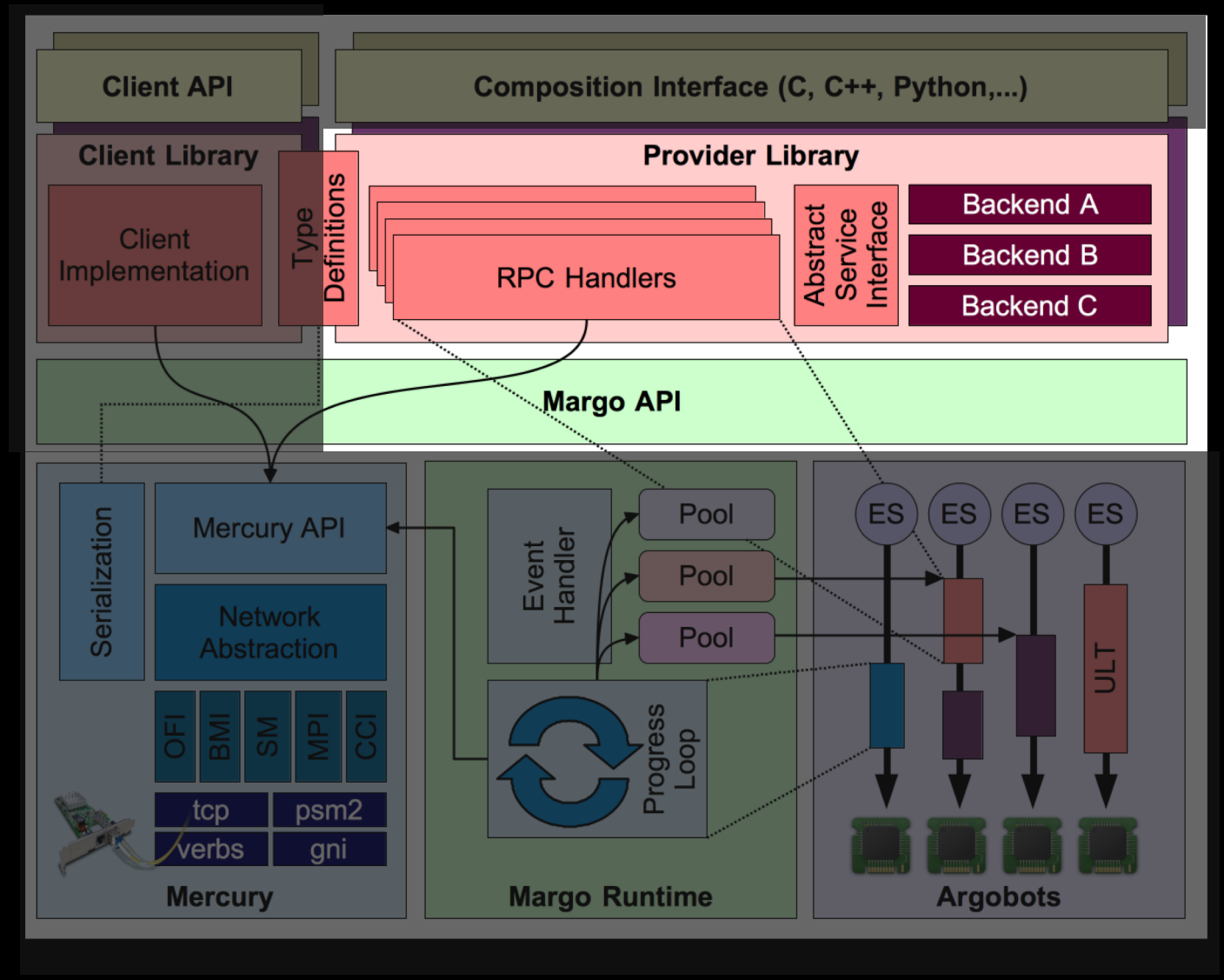
Building Blocks

Anatomy of a Mochi component



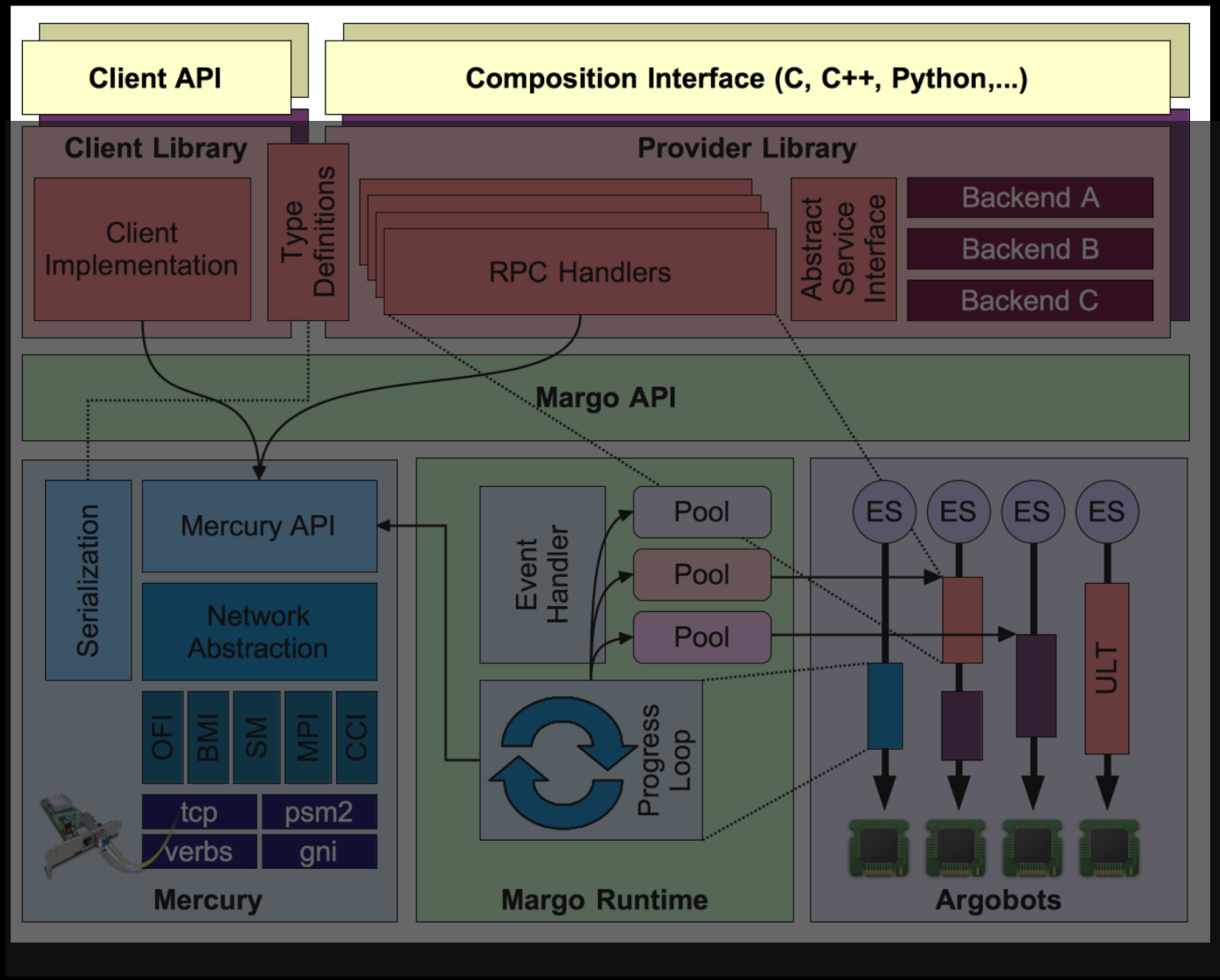
Building Blocks

Anatomy of a Mochi component



Building Blocks

Anatomy of a Mochi component



Composition made very easy

- Example composition code in Python
  - (components themselves are programmed in C or C++)

Initializing Margo runtime (using Cray GNI network for Mercury)

BAKE component managing RDMA to storage target

SDSKV component managing a database

```
# some import statements here

mid = MargoInstance("gni")

bake_provider = BakeProvider(mid, 1)
bake_provider.add_storage_target("/local/ssd/space.dat")

sdskv_provider = SDSKVProvider(mid, 1)
sdskv_provider.add_database("mydatabase", "/tmp/sdskv",
                            pysdskv.server.leveldb))

mid.wait_for_finalize()
```

HEPnOS

Fast event-store for High Energy Physics experiments

Fermilab

Storing “products”

- From experiments
- From simulations or analysis workflows

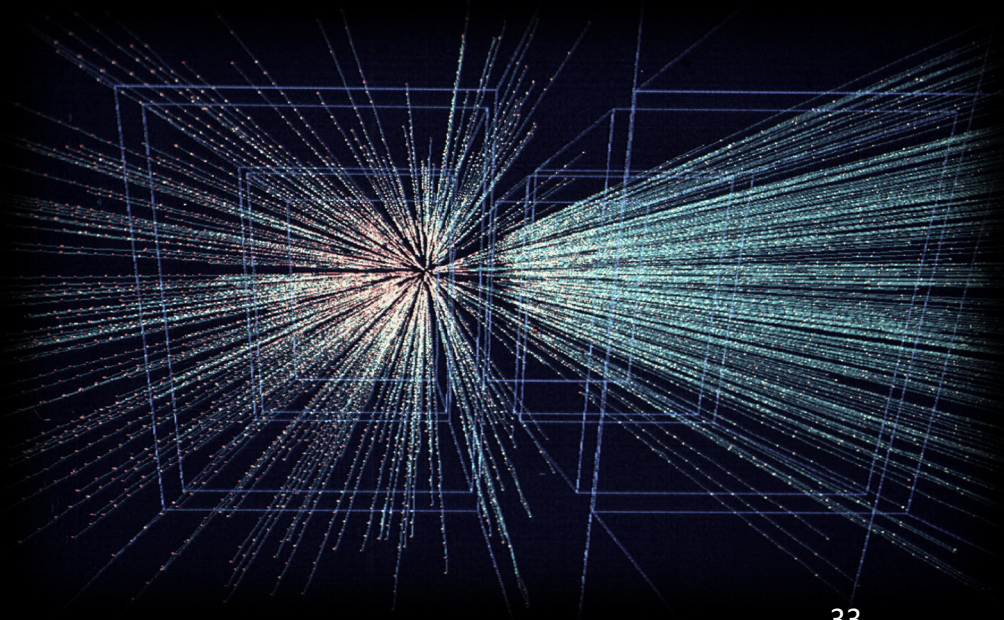
User Requirements

Data model

- Products are instances of C++ objects
- Hierarchy: datasets, runs, subruns, events
- Products are labeled by an “input tag”

Access pattern

- Write-once-read-many
- Products accessed atomically
- Access by input tag and by type
- Iterators to navigate the hierarchy



Storing “products”

- From experiments
- From simulations or analysis workflows

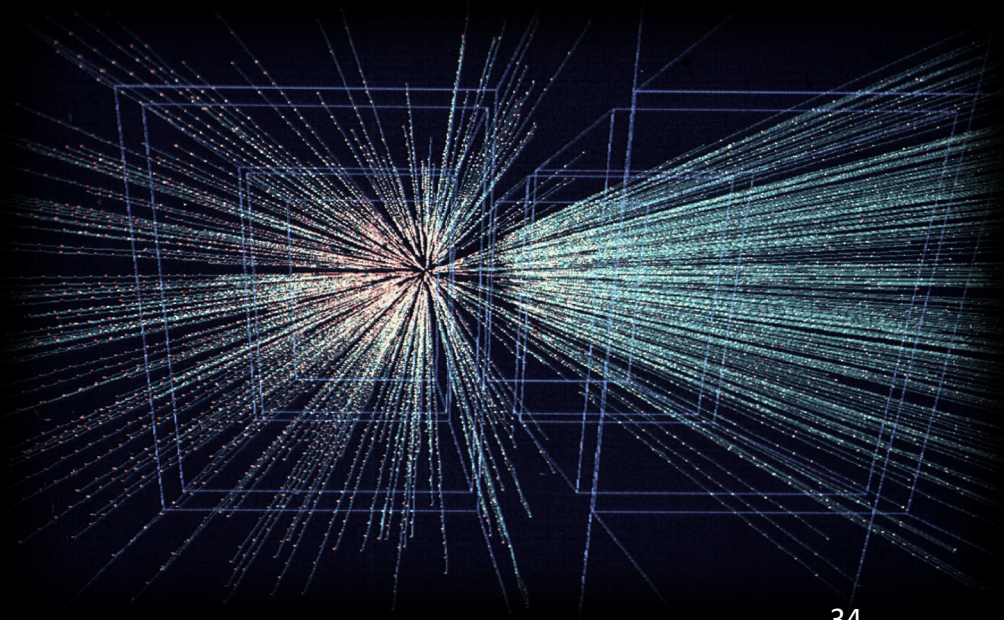
Data model

- Products are instances of C++ objects
- Hierarchy: datasets, runs, subruns, events
- Products are labeled by an “input tag”

Access pattern

- Write-once-read-many
- Products accessed atomically
- Access by input tag and by type
- Iterators to navigate the hierarchy

User Requirements



Storing “products”

- From experiments
- From simulations or analysis workflows

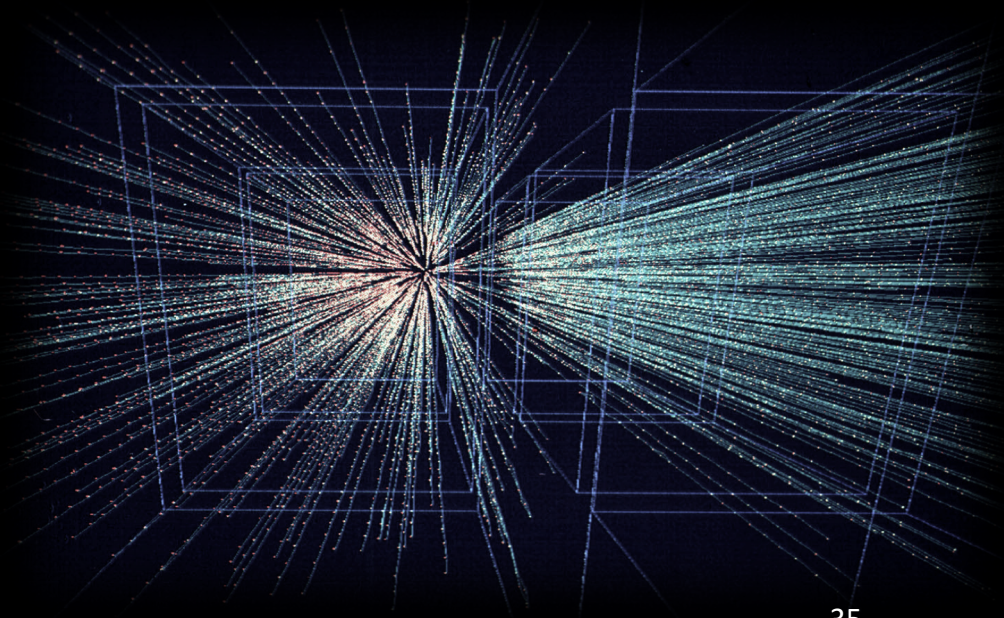
Data model

- Products are instances of C++ objects
- Hierarchy: datasets, runs, subruns, events
- Products are labeled by an “input tag”

Access pattern

- Write-once-read-many
- Products accessed atomically
- Access by input tag and by type
- Iterators to navigate the hierarchy

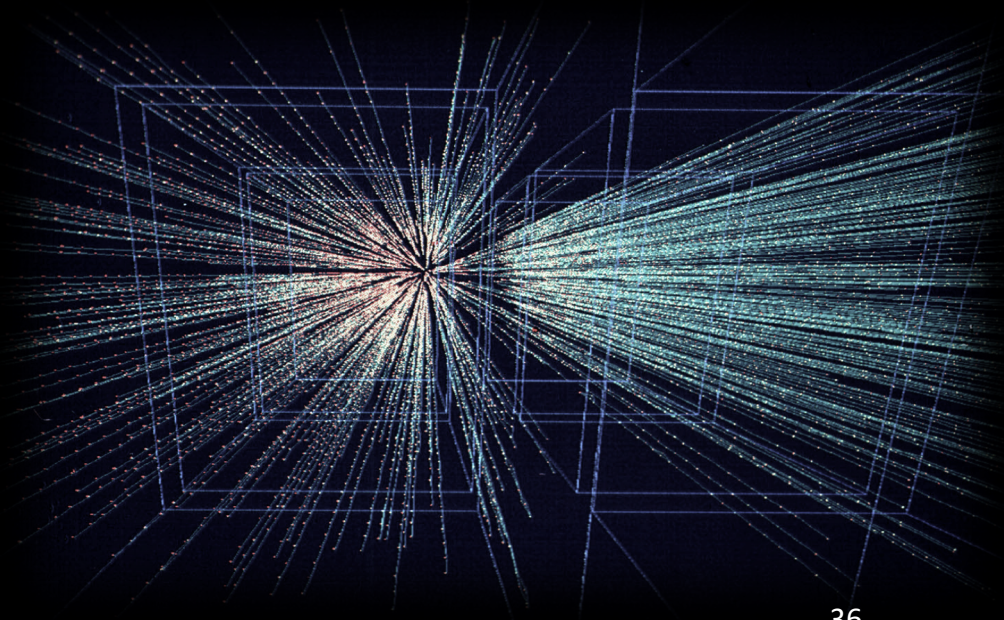
User Requirements



Envisioned usage

User Requirements

- Long-running (weeks), resizable cache based on fast, in-compute-node storage (SSDs, NVRAM, local memory)
- Accessed by multiple applications concurrently
- Backed-up by a more permanent storage system (parallel file system, archive system, object store) when undeployed



How objects should be distributed?

- Based on the hash of a “path-like” string
- <dataset>/<run>/<subrun>/<event>/<input-tag>/<object-type>
- myproject/mydata%45%23%678#exp1_alpha_std::map<int,Particle>

Should objects be sharded?

- No

Should objects be replicated?

- Maybe

How should metadata be managed?

- Same path-like strings as products
- Hash is based on the “parent” path in the hierarchy so that all containers belonging to the same parent end up on the same node

What should the API look like?

- C++ with template metaprogramming to handle storage of any type of object, and iterator constructs to navigate the hierarchy

Service Requirements

How objects should be distributed?

- Based on the hash of a “path-like” string
- <dataset>/<run>/<subrun>/<event>/<input-tag>/<object-type>
- myproject/mydata%45%23%678#exp1_alpha_std::map<int,Particle>

Should objects be sharded?

- No

Should objects be replicated?

- Maybe

How should metadata be managed?

- Same path-like strings as products
- Hash is based on the “parent” path in the hierarchy so that all containers belonging to the same parent end up on the same node

What should the API look like?

- C++ with template metaprogramming to handle storage of any type of object, and iterator constructs to navigate the hierarchy

Service Requirements

How objects should be distributed?

- Based on the hash of a “path-like” string
- <dataset>/<run>/<subrun>/<event>/<input-tag>/<object-type>
- myproject/mydata%45%23%678#exp1_alpha_std::map<int,Particle>

Should objects be sharded?

- No

Should objects be replicated?

- Maybe

How should metadata be managed?

- Same path-like strings as products
- Hash is based on the “parent” path in the hierarchy so that all containers belonging to the same parent end up on the same node

What should the API look like?

- C++ with template metaprogramming to handle storage of any type of object, and iterator constructs to navigate the hierarchy

Service Requirements

How objects should be distributed?

- Based on the hash of a “path-like” string
- <dataset>/<run>/<subrun>/<event>/<input-tag>/<object-type>
- myproject/mydata%45%23%678#exp1_alpha_std::map<int,Particle>

Should objects be sharded?

- No

Should objects be replicated?

- Maybe

How should metadata be managed?

- Same path-like strings as products
- Hash is based on the “parent” path in the hierarchy so that all containers belonging to the same parent end up on the same node

What should the API look like?

- C++ with template metaprogramming to handle storage of any type of object, and iterator constructs to navigate the hierarchy

Service Requirements

How objects should be distributed?

- Based on the hash of a “path-like” string
- <dataset>/<run>/<subrun>/<event>/<input-tag>/<object-type>
- myproject/mydata%45%23%678#exp1_alpha_std::map<int,Particle>

Should objects be sharded?

- No

Should objects be replicated?

- Maybe

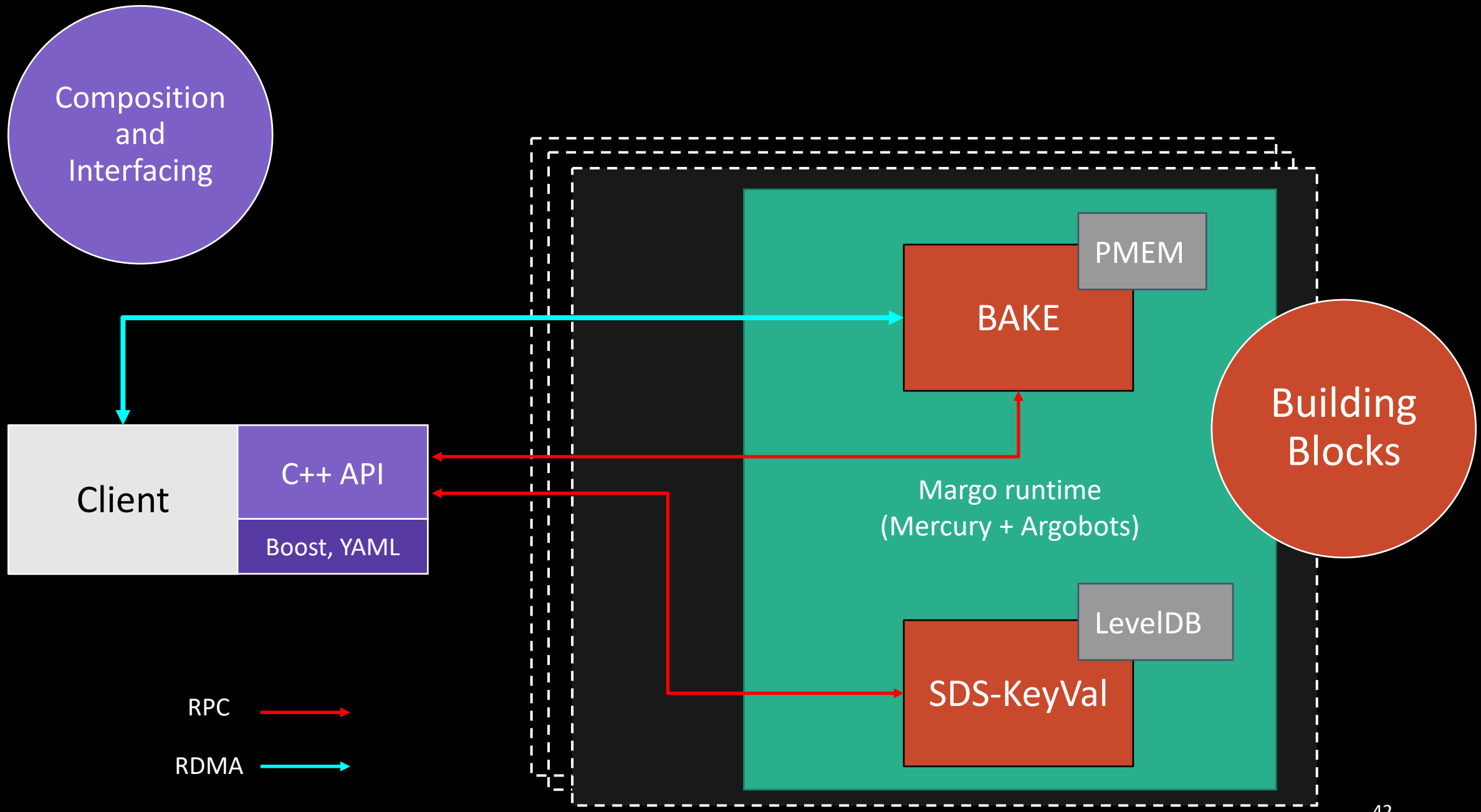
How should metadata be managed?

- Same path-like strings as products
- Hash is based on the “parent” path in the hierarchy so that all containers belonging to the same parent end up on the same node

What should the API look like?

- C++ with template metaprogramming to handle storage of any type of object, and iterator constructs to navigate the hierarchy

Service Requirements


Composition and Interfacing
PMEM
BAKE
Building Blocks
Client
C++ API
Boost, YAML
Margo runtime
(Mercury + Argobots)
LevelDB
SDS-KeyVal
RPC
RDMA

```
#include <hepnos.hpp>

// example structure
struct Particle {
        float x, y, z; // member variables
        // serialization function for boost to use
        template<typename A>
        void serialize(A& a, unsigned long version) {
                ar & x & y & z;
        }
};
// initialize a handle to the HEPnOS datastore
hepnos::DataStore datastore( "config.yaml" );
// access a nested dataset
hepnos::DataSet ds = datastore[ "path/to/dataset" ];
// access run 43 in the dataset
hepnos::Run run = ds[43];
// access subrun 56
hepnos::SubRun subrun = run[56];
// access event 25
hepnos::Event ev = subrun[25];
// store data (an std::vector of Particle)
st::vector<Particle> vp1 = ...;
ev.store(“mylabel”, vp1);
// load data
std::vector<Particle> vp2;
sv.load(“mylabel”, vp2);
// iterate over the subruns in a run
// using a C++ range-based for
for(auto& subrun : run) { ... }
```

Code sample of HEPnOS

- Boost for serialization of C++ classes
- “map”-like interface in DataStore, DataSet, Run, and Subrun classes
- Template “load” and “store” methods
- Iterators to navigate the hierarchy

Taking a step back: other Mochi services

- FlameStore
  - Python interface, Python composition
  - Stores Deep Neural networks
  - Flat namespace
  - BAKE storing NumPy arrays
  - SDSKeyVal storing model metadata in JSON format
  - Embedded python interpreter to modify models within storage

- SDSDKV
  - C interface, C++ composition
  - Distributed key/value store
  - Used for the ParSplice application (molecular dynamics)

Lightweight: Source Lines of Code (SLOC)

	Component	Client	Server	Other	External Users
Core					
	Argobots			15,193	Intel, LLNL, Mainz
	Mercury			27,979	Intel, LBL, LLNL, Mainz
	Margo			1,625	Intel, LLNL, Mainz
	Thallium			3,913	
	SSG			2,203 + 131 (py-ssg)	
	MDCS			906	
Microservices					
	SDSKV	1,392	2,881	234 (py-sdskv)	
	BAKE	949	1,273	514 (py-bake)	
	POESIE	343	689		
Composed Services					
	HEPnOS	2,689	321		FNAL
	FlameStore	334	438		
	Mobject	1,498	5,044		
	SDSDKV	407	601		

Conclusion: use componentization!

- Monolithic file systems are often suboptimal
- Data services are better
- Efficiently building custom data services is a challenge
- Composed data services is key to productivity

Thank you! Questions?

This work is in part supported by the Director, Office of Advanced Scientific Computing Research, Office of Science, of the U.S. Department of Energy under Contract No. DE-AC02-06CH11357; in part supported by the Exascale Computing Project (17-SC-20-SC), a joint project of the U.S. Department of Energy's Office of Science and National Nuclear Security Administration, responsible for delivering a capable exascale ecosystem, including software, applications, and hardware technology, to support the nation's exascale computing imperative; and in part supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Scientific Discovery through Advanced Computing (SciDAC) program.
This work was done in the context of the DOE SSIO project "Mochi" (https://www.mcs.anl.gov/research/projects/mochi/), a Software Defined Storage Approach to Exascale Storage Services.

Personal thanks to the Spack developers who make our lives much easier as we develop these services!