Millenial: Modular Microservice Macrobenchmarks

Generating highly reconfigurable microservice benchmarks for systems research!

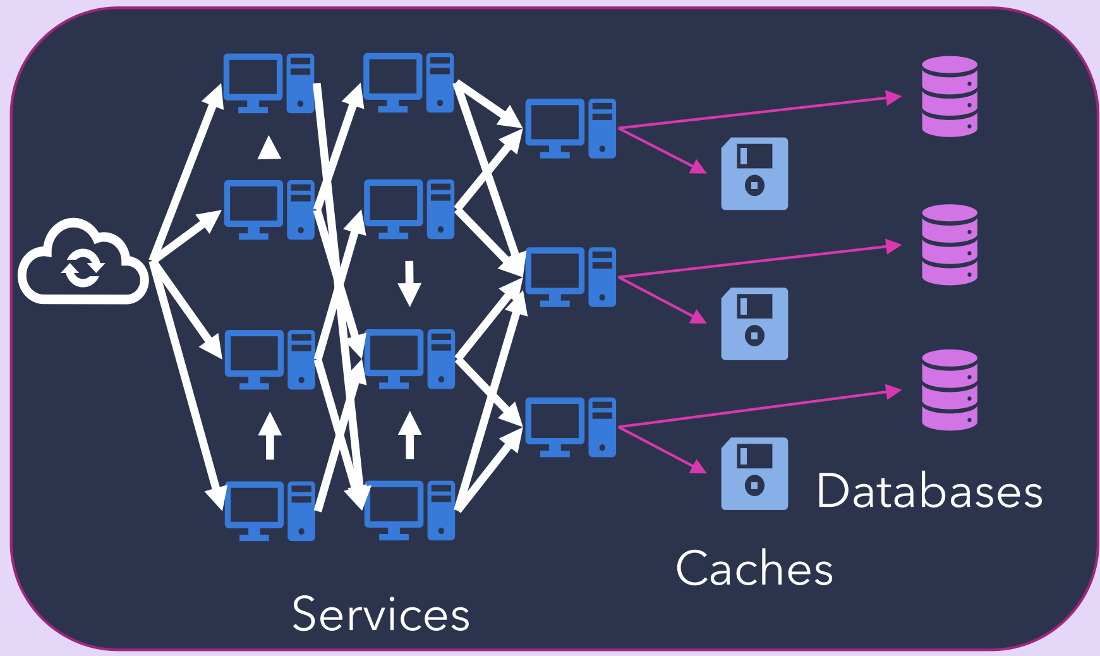
Vaastav Anand (vaastav@mpi-sws.org)
Gerd Alliu (galliu@mpi-sws.org)
Antoine Kaufmann (antoinek@mpi-sws.org)
Deepak Garg (dg@mpi-sws.org)
Jonathan Mace (jcmace@mpi-sws.org)

MAX PLANCK INSTITUTE FOR SOFTWARE SYSTEMS

MILLENIAL
SOSP 2021
STUDENT RESEARCH COMPETITION acm
Association for Computing Machinery
Advancing Computing as a Science & Profession

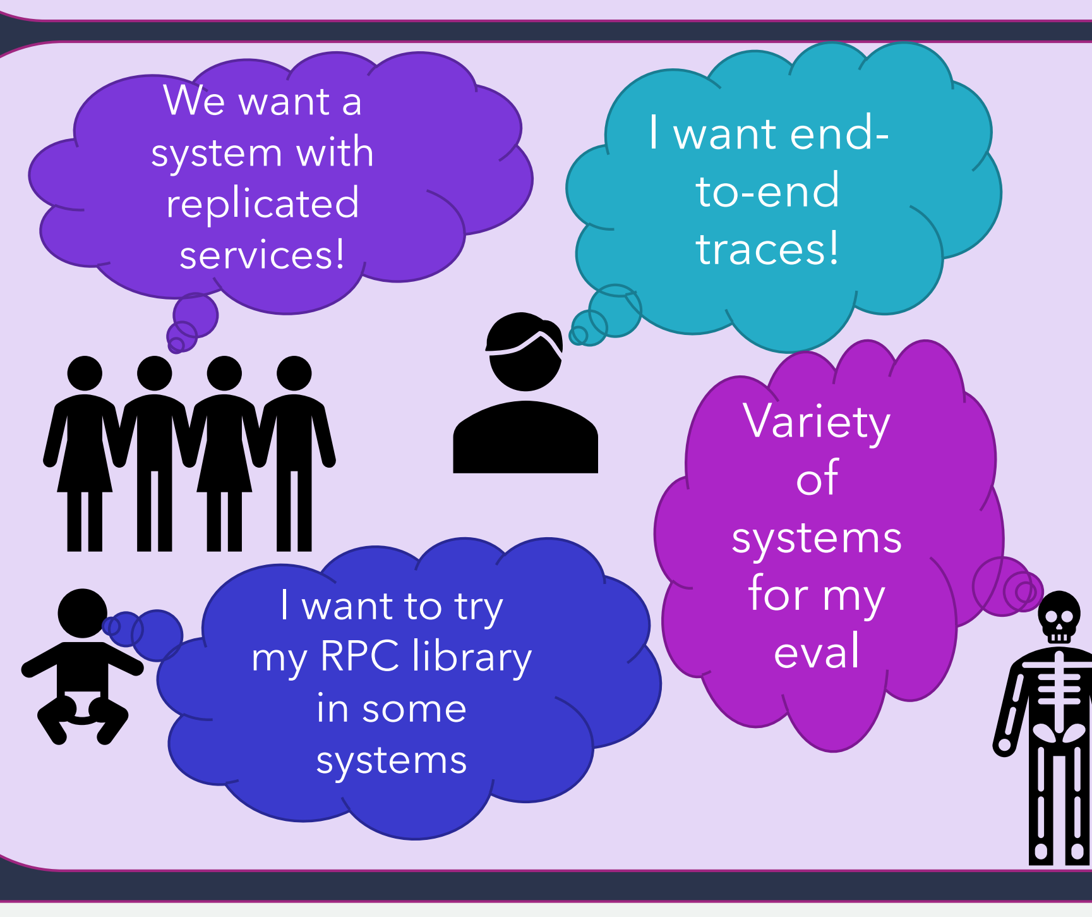
Background

- ❖ Microservices increasingly popular for cloud apps.
- ❖ Present a gold mine of research problems.
- ❖ Good research requires variety of systems.



GOAL

GOAL: Generate implementations of microservice systems on-demand based on user requirements while providing the flexibility to enable/disable features and making it easy to integrate new components.


We want a system with replicated services!
I want end-to-end traces!
Variety of systems for my eval
I want to try my RPC library in some systems

What do researchers want?

- ❖ Multiple diverse systems for robust evaluation!
  - ❖ Existing systems make choice of features (tracing, replication, etc) fixed with no flexibility.
- ❖ Most papers end up using limited number of systems because of high amount of effort required to test ideas on even 1 system

Challenges

- ❖ **Flexibility:** Should be easy to reuse and generate multiple implementations of the same application
- ❖ **Extensibility:** The generation process should be extensible with new features.
- ❖ **Systematic:** Generation can't be ad-hoc.

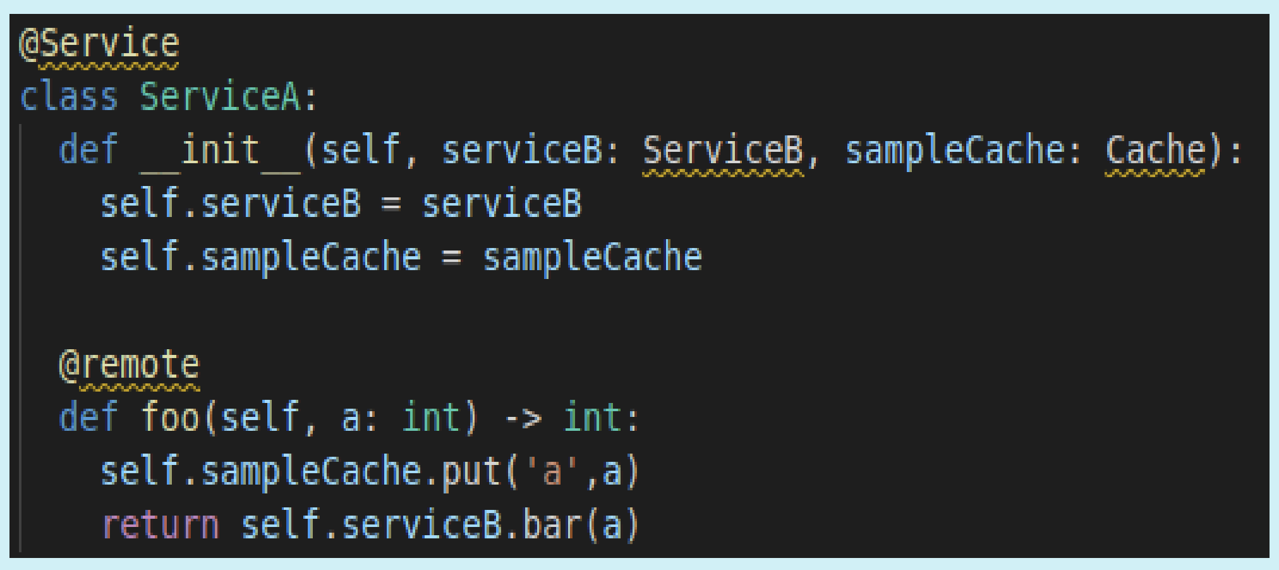
Key Insights

- ❖ **Abstract Application**: The business logic of the app is independent of the features and impl choices.
- ❖ **Reusable Features/Components:** Features are implemented once and used many times.

Millenial Overview

Input

❶ **App. Spec**: Core business logic of various services

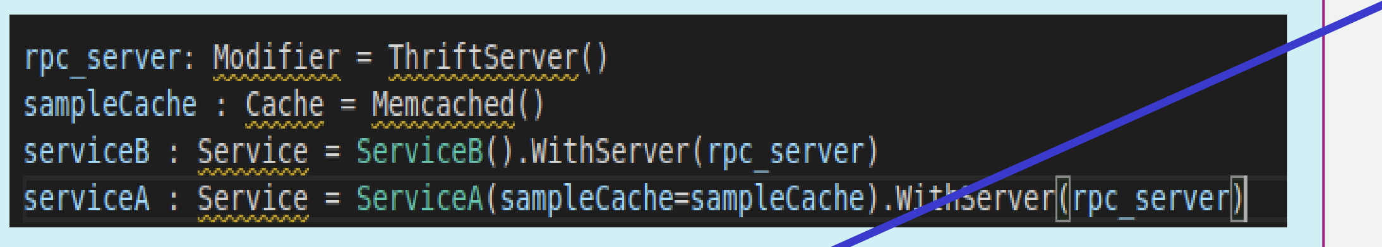


```
@Service
class ServiceA:
  def __init__(self, serviceB: ServiceB, sampleCache: Cache):
    self.serviceB = serviceB
    self.sampleCache = sampleCache

  @remote
  def foo(self, a: int) -> int:
    self.sampleCache.put('a',a)
    return self.serviceB.bar(a)
```

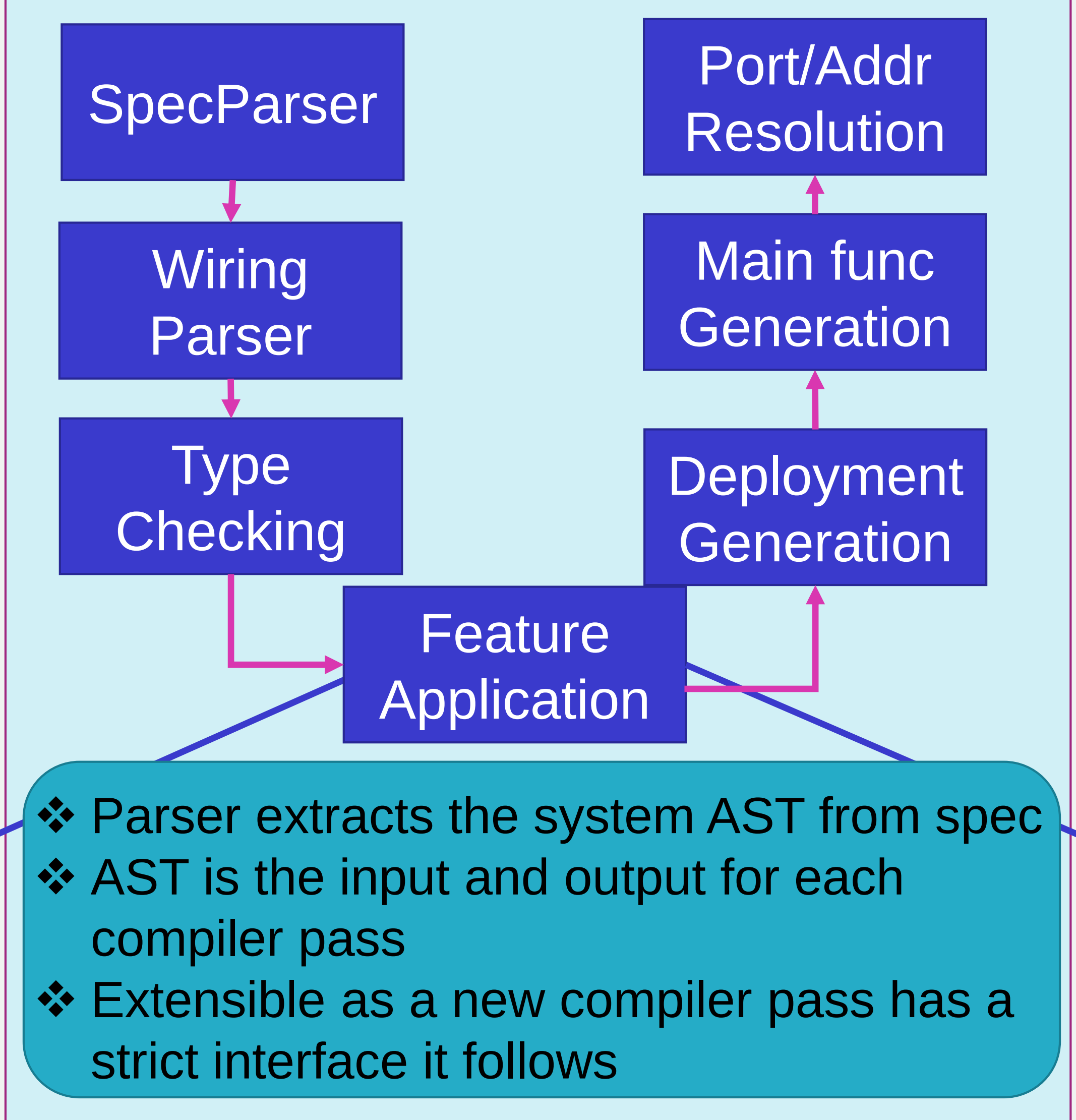
❷ **Wiring Spec**

- ❖ Implementation choices for services
- ❖ Apply add-on features like tracing, replication



```
rpc_server: Modifier = ThriftServer()
sampleCache : Cache = Memcached()
serviceB : Service = ServiceB().WithServer(rpc_server)
serviceA : Service = ServiceA(sampleCache=sampleCache).WithServer(rpc_server)
```

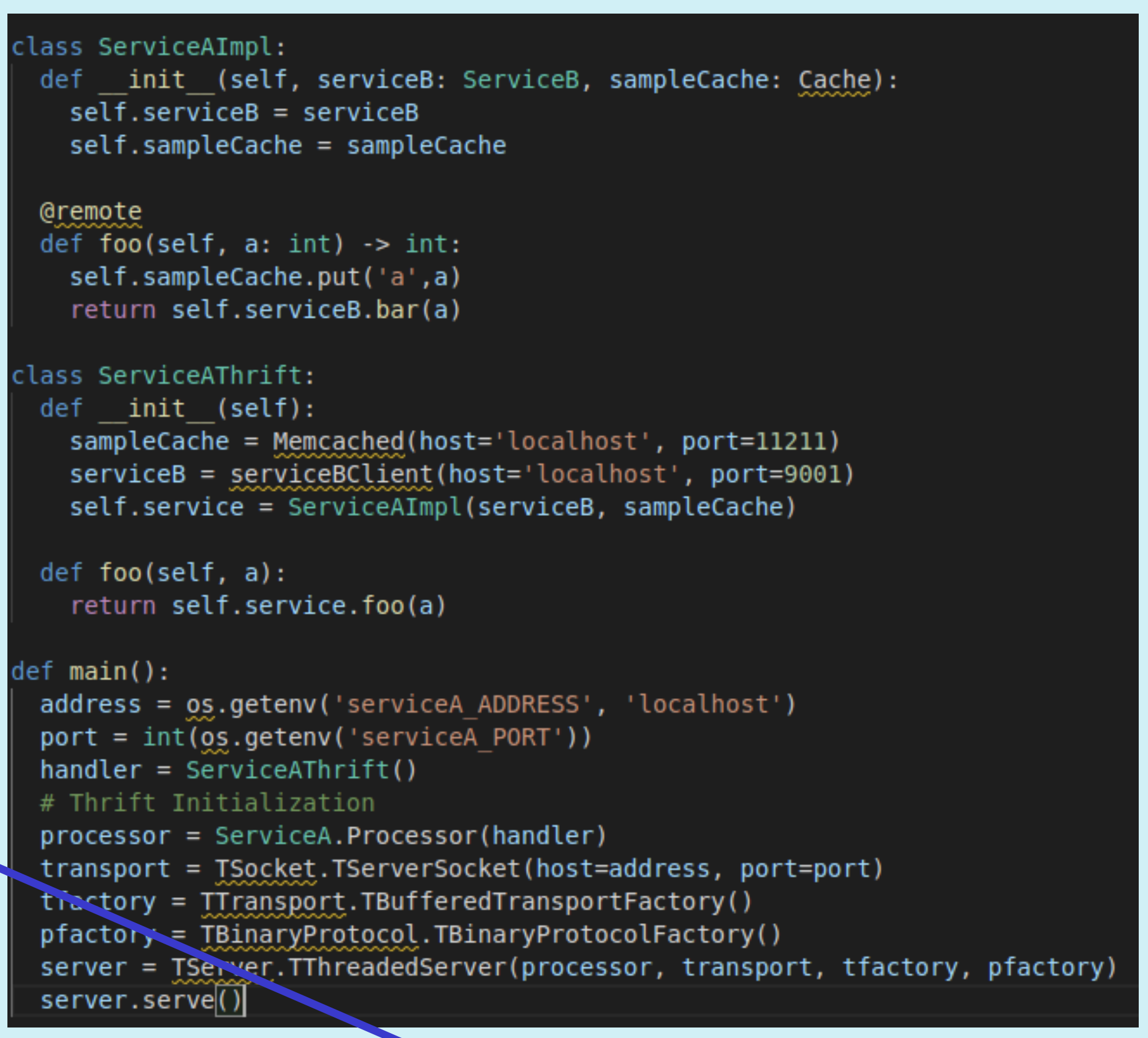
Compiler



- ❖ Parser extracts the system AST from spec
- ❖ AST is the input and output for each compiler pass
- ❖ Extensible as a new compiler pass has a strict interface it follows

Output

❶ Source Code ❷ Deployment Files



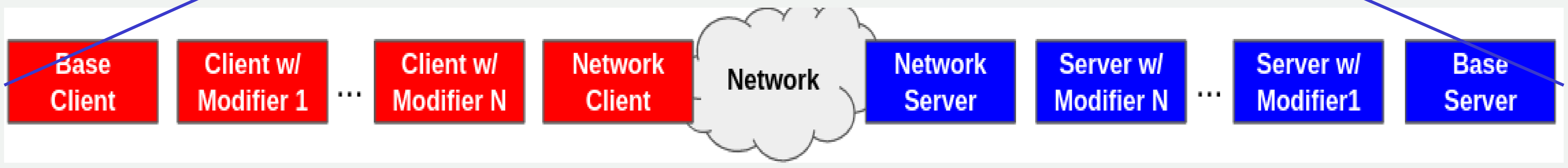
```
class ServiceAImpl:
  def __init__(self, serviceB: ServiceB, sampleCache: Cache):
    self.serviceB = serviceB
    self.sampleCache = sampleCache

  @remote
  def foo(self, a: int) -> int:
    self.sampleCache.put('a',a)
    return self.serviceB.bar(a)

class ServiceAThrift:
  def __init__(self):
    sampleCache = Memcached(host='localhost', port=11211)
    serviceB = serviceBClient(host='localhost', port=9001)
    self.service = ServiceAImpl(serviceB, sampleCache)

  def foo(self, a):
    return self.service.foo(a)

def main():
  address = os.getenv('serviceA_ADDRESS', 'localhost')
  port = int(os.getenv('serviceA_PORT'))
  handler = ServiceAThrift()
  # Thrift Initialization
  processor = ServiceA.Processor(handler)
  transport = TSocket.TServerSocket(host=address, port=port)
  tfactory = TTransport.TBufferedTransportFactory()
  pfactory = TBinaryProtocol.TBinaryProtocolFactory()
  server = TServer.TThreadedServer(processor, transport, tfactory, pfactory)
  server.serve()
```



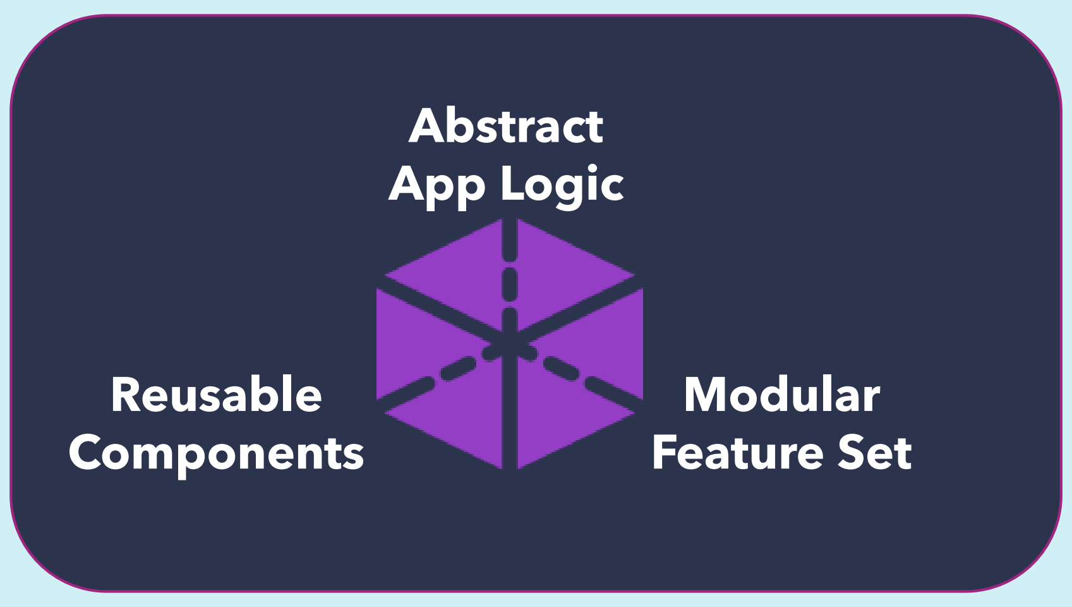
Systems as Millenial Applications (LoC)

System	Original	Millenial Spec	Millenial Wiring	Millenial Generated
DSB-SN	8209	1601	59	6012
DSB-MM	7794	1146	42	6308
DSB-HR	5160	977	63	6081
TrainTicket	54466	10264	166	45151
SockShop	13987	2015	40	7413

- ❖ Lines of Code numbers shown from an early prototype.
- ❖ In addition to being **highly reconfigurable**, Millenial application offers a **significant reduction** in the lines of code that a user needs to write.
- ❖ The large fraction of the code generated by Millenial is the "glue code" to bind the services with features such as tracing, replication, etc and concrete choices of caches, databases, and queues.

Implementation

- ❖ Early prototype implemented in 6K lines of Python
- ❖ Custom DSL for wiring.
- ❖ Input Spec and Output will be in Go for 2 reasons
  - ❖ Good performance!
  - ❖ Easy to write specs in Go!



On the Road to Evaluation

- ❖ Can Millenial generate equivalent replicas of existing microservice systems?
- ❖ Do the systems generated by Millenial have realistic performance?
- ❖ How easy is it to reconfigure applications with Millenial?