Parallel Unconcentrated Management for Applications (PUMA) Applied to Dialog Systems

Kevin Jesse
UC Davis
krjesse@ucdavis.edu

Austin Chau
UC Davis
amchau@ucdavis.edu

Abstract

Dialog systems are versatile, providing services from weather, alarms, movie recommendations, traffic, advice and more. Their functionality depends on the underlying micro-services, which are interchangeable applications running in real time. Since most modern dialog systems are hosted on network clusters, these micro-services must be able to scale under a distributed environment with minimal impact on performance and data integrity. Furthermore, micro-services must not compete for resources or impact performance of other micro-services. This paper proposes a new framework which adopts existing distributed system managers to scale applications, like dialog systems, while maintaining programmability. To test the effectiveness of this new framework, PUMA, we developed a movie recommendation dialog service that runs exclusively on the proposed framework. This implementation is then put through a series of benchmarks that simulates high service usage in a variety of capacities. We expect PUMA to outperform existing designs and be ideal for applications with many independent micro-services such as dialog systems.

1 Introduction

dialog systems are extremely useful by the skills they present to a user. An interaction consists of several utterances that may illicit the help of several micro-services. These micro-services must scale and perform with minimal latency in order for the core service to return a timely response. These services often operate in the cloud where the largest obstacles are availability, performance unpredictability, and auto scaling [Armbrust *et al.*, 2010]. While these obstacles are central to the quality of the service, they are often neglected in favor implementation simplicity. Novel dialog systems, like Tick-Tock, [Yu *et al.*, 2015], are naively implemented to run on a single Linux instance, despite intentions to integrate with high throughput systems. We continue to see this trend as applications like dialog systems increase complexity both computationally and as a service [Gustafson *et al.*, 2000], [Levin *et al.*, 1998], [Henderson *et al.*, 2013]. Moreover, these systems are not built with scalability, availability, concurrency, and parallelized computation in mind at the application level. This needs to be addressed.

PUMA is inspired from Amazon Lambda and Elastic Load Balancing services. Complex dialog systems of today like Alexa are built on serverless platforms with instant scaling response on triggers. Code runs parallel for requests and triggers individually, scaling only for the size of the workload; this is extremely cost efficient [Villamizar *et al.*, 2016]. While practical for simple and isolation driven architectures, this platform suffers from several drawbacks. Namely, inherent portability complexity of existing code for stateless server design, a lack of visibility in scaling and load balancing services, and a commitment to using AWS. Organizations are in need of an open source application scalar and load balancer that is IaaS/PaaS independent. Moreover, scaling monolithic servers running multiple services on demand is not feasible on traditional providers like RackSpace and AWS [Villamizar *et al.*, 2015]. Scaling traditional monolithic applications continues to be a challenge because they typically offer a variety of services, some more popular than others. If popular or high utilization services need to be scaled because of high demand, the entire set of services must also be scaled [Villamizar *et al.*, 2015]. PUMA attempts to alleviate monolithic service scaling and load balancing difficulties by enabling service definition. PUMA proxies requests for application defined micro-services that are scaled and balanced automatically. PUMA can run on both single instance servers and in a distributed manner using commodity nodes.

PUMA automatically enables distributed computing resources to the application, proxies micro-services, ensures availability, and provides computational resource responsibility. PUMA routes core functionality of standalone applications across distributed computing nodes. Nodes can execute requests and application defined services in parallel and coordinate high intensity workloads by automatically spawning more nodes. Application defined services are not limited to services accessed only by the end user, but all areas of computation where distributed computing can be leveraged. Influenced by Google's MapReduce [Dean and Ghemawat, 2008], PUMA aims to enable data-segmented parallelized computation on dynamically spawnable CPU nodes. This framework prevents blocking of new services by ensuring enough computational resources exist for new requests. Thus, the CPU is available for requests, while leaving heavy lifting such as distributed matrix factorization [Gemulla *et al.*, 2011], neural network training [Dean *et al.*, 2012], and classification tasks to scalable computing workers.

These workers are delegated by the load supervisor. PUMA leverages Envoy, an open source L7 service proxy and communication bus [Klein, 2016]. By using Envoy as the load supervisor, PUMA can efficiently manage a mesh of nodes and service endpoints. Envoy runs natively with any application language, providing a rich service architecture in a latency-optimized manner [Klein, 2016]; ideal for real time applications such as dialog systems. Envoy also provides gRPC support, ideal for encapsulating code for remote execution [Klein, 2016]. PUMA is designed for a centralized application designed service architecture, that is scalable and accessible with API-like connectors to key features and services like database operations and background service operations. With Envoy's abstracted upstream mesh and simple service architecture, PUMA does the heavy lifting by configuring Envoy to the applications specifications. Integration simplicity for existing architectures is a central focus of this work, and to evaluate this metric, PUMA was tested on an existing monolithic dialog system. This system utilizes several married services that when configured with PUMA, could be decentralized and optimized to the dynamic workload, running and scaling independently. This system was then tested on a series of benchmarks that measured latency, availability, adaptation to unpredictable workloads, and scalability.

Portability is a central focus of this work. Most applications are first developed as a standalone application and when scaled to distributed computing environments, like Amazon Lambda, there is a significant modifications required. Additionally, designing a new standalone service with future distributed computing design in mind, hurts deployment time and increase expense of development. As distributed computing advances or service agreements change, legacy architectures embed in applications must be supported. PUMA abstracts the distributed computing software from the application and fundamental design choices translates directly into easy maintainability throughout the product life cycle. Our framework achieves high portability and maintainability on our sample monolithic dialog system while increasing performance by scaling and balancing various application defined services.

Some key questions in evaluating this framework are: what are implementation costs, how difficult is programmability, and how much of an improvement over a traditional non-distributed micro service implementations. Precisely, how many lines of code is required to port an existing monolithic application and what is the independent service speedup. In deciding what metrics to use for programmability, we relied on operating system techniques such as measuring the number of lines of modified code. Due to ease of migration and implementation, this framework can be adopted in various domains for any application. To measure independent service speedup, we evaluate common data computations on PUMA that can simulate potential speedup for more complex operations like matrix factorization.

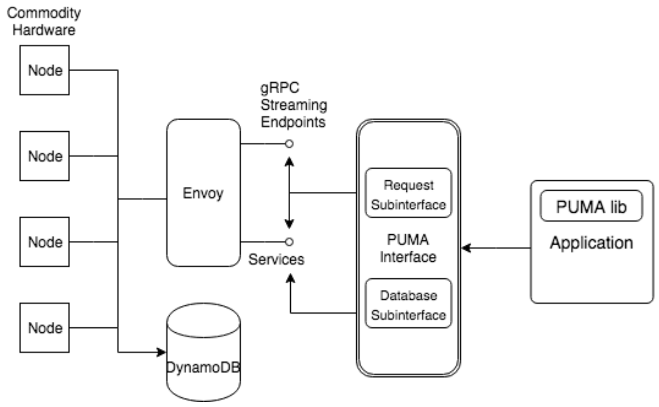


Figure 1: PUMA System Framework PUMA configures standalone applications to interface with application defined services in Envoy. PUMA configured applications serve requests by proxy to distributed computing nodes. Envoy load balances requests with commodity hardware to provide service availability. A lightweight PUMA python library is available to provide API-like communication with application defined services, thus providing parallel computing on a distributed mesh.

The contributions of this paper are,

- A framework for automatically configuring standalone monolithic applications to a set of application defined micro-services capable of load balancing and scaling according to the workload
- A set of benchmarks that demonstrate the effectiveness of such framework on computationally expensive tasks
- A ported monolithic style application with less than 15 lines of altered code.

We published our source code[1] and maintain a running instance[2] of the ported chatbox application. Our team is hoping to publish a python library for application defined PUMA configurations to automatically port standard python applications to PUMA enabled applications.

2 System Overview

PUMA library is at the root of application portability. With the PUMA library imported to standalone applications, major functionality can be migrated to the service mesh. This requires well defined application service boundaries, but these can be specified by the application, further ensuring maximum flexibility during portability.

The PUMA interface extends Envoy by exposing proxy functionality and providing abstractions to Envoy features like load balancing and automatic computational scaling. With application service mappings defined by the PUMA library, independent services can be routed to computational nodes running on Envoy's native code base at the service endpoint. Code intended for gRPC streaming endpoints are

[1]https://github.com/kevinjesse/puma

[2]http://puma251.ddns.net:8080

encapsulated and forwarded while complementary functionality like routing and load balancing for gRPC still occurs on the Envoy substrate. The database sub-interface translates database requests from locally defined database connections to a balanced Envoy service routing directly to DynamoDB. This is critical for frequent or large database queries as the database service can scale according to usage in addition to software logic; thus not becoming a common bottleneck for scaled software logic. The PUMA interface registers defined configurations of services from the PUMA lib and propagates those configurations to Envoy in a seamless fashion. PUMA provides application developers with the tools necessary to scale and balance the application, without having to have specific domain knowledge of L7 proxies, load balancing, and scaling technologies.

Envoy is a self contained process that can run alongside the application server. The Envoy mesh is completely transparent to the application in which the application sends traditional GET and POST requests to and from the localhost, completely unaware of the underlying network topology [Klein, 2016]. This is important for several reasons: applications do not need to be configured with distributed computing in mind for portability. Moreover, with REST API-like communication, applications can be written in any language; service orient architectures tend to use multiple application frameworks and languages. While Envoy is typically a service to service communication system, PUMA uses Envoy as a front edge proxy. This brings the same benefits of observability, management, identical service discovery, load balancing, and scaling at first interaction with the workload. This interaction with the workload at then edge provides Envoy with a distinct scaling advantage as there is little delay of recognizing when more computational resources are necessary and no up time to expand the mesh because the future resources are configured at start up. Lastly, the transparent Envoy mesh means scaling and library upgrades can be deployed and upgraded across the infrastructure efficiently and quickly. This distributed mesh is composed of a variety of commodity hardware.

Envoy relies on commodity hardware for the execution of its services. The envoy master thread controls coordination while worker threads accept connections and is bounded to a set of resources. These threads are ran natively so that the architectural components can get out of the way and are optimized for highest hardware performance.

Services expansion provides a potential bottleneck at database operations. Even services that do not directly access the database, commodity nodes will eventually need information that reside elsewhere. This tail latency is dangerous, especially when sharing a structure such as a database. To demonstrate the need for conscious structural decisions, we recommend using a database like DynamoDB which is optimized for Envoy and has all of the advantages previously discussed.

The detail of PUMA's configuration methods will demonstrate the easy of migrating an existing application to this system architecture.

3 Design

The design for PUMA was inspired by the Lambda service architecture used in the Alexa Prize. Novel dialog and multimodal systems [Yu *et al.*, 2015][Thomason *et al.*, 2016] cannot easily be built on serverless platforms and future concerns such as load balancing, is often neglected in favor of feature development and integration. PUMA inherits many advantages of serverless applications with Envoy, while maintaining flexibility and exclusiveness of monolithic cloud environments (IaaS/PaaS).

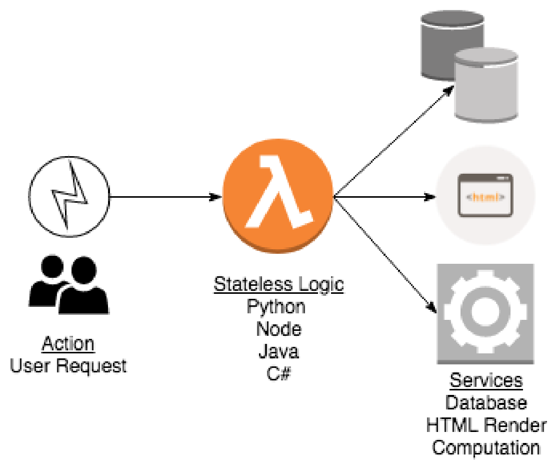


Figure 2: Serverless Model Applications must be written for existing serverless platforms like AWS Lambda. Applications in serverless platforms can easily be scaled and load balanced while cost of operation is less than traditional server hosts like EC2.

3.1 Serverless Design

Serverless computing denotes a special software architecture where application logic is execution in an environment without visible processes, servers, virtual machines, or operating systems [Stigler, 2018]. While these operating system and architectural principles still exist in underlying hardware and virtualization, they are abstracted away so the developer can solely focus on writing deploy-able code. Service providers are responsible for efficiently serving all requests from its clients and do not have to run a permanent workload for a specific client. The application developer does not need to worry about scalablity and load balancing and the service provider can utilize their infrastructure more efficiently; a true win win.

Serverless architecture has several advantages over traditional PaaS providers. Many PaaS providers do not guarantee automatic scaling, and the application developer has to manually scale the application, potentially, still without necessarily meeting dynamic needs. PUMA solves this scaling issue within PaaS systems by leveraging Envoy. Moreover, PUMA maintains the performance and abstractions serverless architectures provide without any of the drawbacks.

Serverless architectures have some fundamental drawbacks. Long running applications or services with tail latency are often more expensive than a dedicated server or virtual machine [Baldini *et al.*, 2017]. Serverless code is written for the platform it is running on like AWS Lambda or Azure Functions which introduces a learning curve and vendor lock-in. Additionally, when the application is completely dependent on a third-party provider, there is less control of the application and changing the provider leads to significant in application changes. As serverless costs change, it might be more affordable to revert the code base for other providers, a difficult and expensive procedure. In contrast, PUMA runs on any platform PaaS and is completely portable because it runs in a Docker container. When running applications in a serverless environment, service providers might run software from several different customers on the same physical server to utilize their resources more efficiently [Baldini *et al.*, 2017]. Not only will your application not be guaranteed to maximally scale according to it's workload, other customer application and provider infrastructure bugs can lead to security vulnerabilities. PUMA can run in any PaaS or monolithic server and scale without restrictions as long as the underlying hardware is available. In practice, serverless platforms suffer from a cold start problem in which there is a delay to initialize internal resources when the application is offline or there hasn't been requests to the function for a while [Baldini *et al.*, 2017]. For dialog systems and real time systems, delays within application functions is unacceptable [Patil *et al.*, 2017]. In contrast, PUMA always runs a single master thread and will spawn worker threads as needed so response time is always instant. Lastly, FaaS services like AWS Lambda do not provide out-of-the-box tools to test functions locally; this incurs a cost for testing the application on AWS Lambda [Baldini *et al.*, 2017]. Due to PUMAs design to run in a Docker container, testing and extensive logging can be done anywhere and does not incur additional costs.

PUMA avoids the aforementioned disadvantageous of serverless platforms while providing most of the benefits: scaling and load balancing capabilities, micro-service design, abstractions to allow developers to focus on code and additionally requires little modification to existing monolithic application. This is accomplished by exposing key envoy service endpoints.

3.2 Envoy

Envoy is an open-source L7 proxy and communication bus designed for modern service architectures with the principle of accessible but transparent network resources. PUMA deploys Envoy as a service to service with front proxy to allow communication with the internet from specialized ports. We use these ports to specifically communicate with PUMA from within the web application.

Envoy runs with a single master thread that provides coordination with a number of worker threads. These worker threads listen, filter, and forward connections. Connections through the Envoy proxy are bound to a single worker thread, allowing connections to be embarrassingly parallel with little effort necessary for coordination between worker threads. Envoy is 100% non-blocking and the number of worker threads is typically configured to the number of hardware threads available on the machine. When a connection is re-

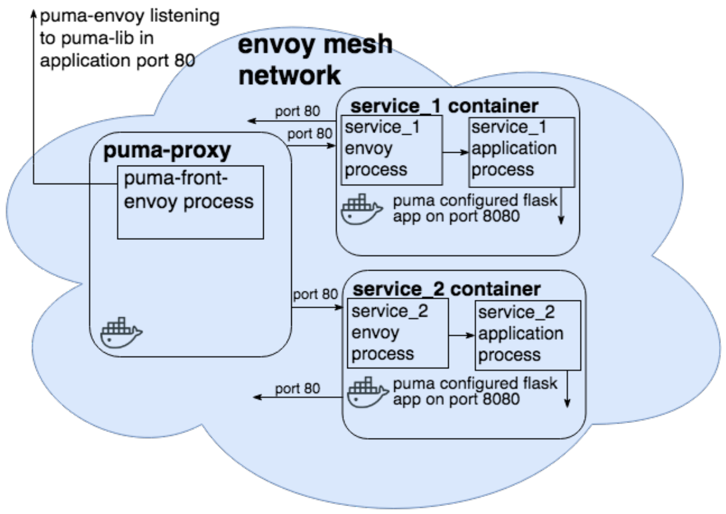

puma-envoy listening to puma-lib in application port 80
envoy mesh network
port 80
port 80
service_1 container
service_1 envoy process
service_1 application process
puma configured flask app on port 8080
puma-proxy
puma-front-envoy process
port 80
service_2 container
service_2 envoy process
service_2 application process
port 80
puma configured flask app on port 8080

Figure 3: PUMA Configured Envoy Mesh PUMA defined services at the PUMA library are configured in Envoy front proxy mesh network. These services use envoy listeners defined in the application using the PUMA library. Workers operate through a PUMA flask application that maintains application defined service logic.

ceived on a listener, or worker thread, it can perform a variety of different proxy tasks: rate limiting, TLS client authentication, HTTP connection management, filtering, and forwarding.

Each Envoy deployment has a HTTP connection manager responsible for HTTP multiplexing. PUMA interfaces with Envoy thought the HTTP connection manager on port 8080 for statically established routes in the configuration. As PUMA becomes more robust, we hope to dynamically establish routes on running deployments using Envoys RDS API (Route Discovery Service). The RDS API is an optional API that Envoy calls dynamically to fetch route configurations. PUMA also leverages Envoys HTTP routing to efficiently route incoming HTTP requests to upstream clusters, acquires a connection pool to host the upstream cluster, and forwards the request. Benchmarking PUMA and envoy saw that the forward proxy acquired connection pools up to 250 cluster nodes. Furthermore, as PUMA becomes more feature rich, we hope to support priority based routing; while sufficient for our real time system, other applications with more significant demands might benefit from priority routing. Expose simple load balancing options is in PUMA's future.

Currently PUMA relies on Envoy's simple load balancing policy of round robin. This means each service defined in PUMA lib gets an equal distribution of computational resources. With multi-modal systems of various importance and impact to real time, round robin might be less than ideal. With PUMA we hope to integrate load balancer options of: weighted least request, ring hash, Maglev, random, and original destination. Circuit breaking is another critical component of distributed systems that PUMA leverages with Envoy; although we do not plan to expose any configurable options. In contrast, global rate limiting configurations are available

```
{
  "load_balancer": "round_robin",
  "logging": {
    "filter": "{...}",
    "config": "{...}",
    "name": "..."
  },
  "rate_limit": {
    "cluster_name": "...",
  },
  "dynamodb": {
    "config": "{...}",
  },
  "puma_services": {
    "service3": "/face",
    "service2": "/haptic",
    "service1": "/policy",
    "service5": "/db",
    "service4": "/sentiment"
  },
  "http_connection_manager": {
    "http_filters": [],
    "drain_timeout": "{...}",
    "rds": "{...}",
    "tracing": "{...}",
    "idle_timeout": "{...}",
    "route_config": "{...}",
  }
}
```

Figure 4: Example PUMA Core Configuration JSON This configuration is passed to *puma_config* to define fundamental Envoy principles. The configuration is parsed and PUMA configures Envoy according to the definitions. Note puma_services is the list of micro-services to be established with their respective url access points on the localhost.

in PUMA because various applications will have vastly different use cases where, for example, a large number of hosts are forwarding to a small number of hosts with a low average request latency. This is a common issue with increasing low latency connections to a single database server. This use-case exemplifies why horizontally scalable distributed databases like DynamoDB should be configured with the PUMA library, thus enabling seamless scaling and access of a single table over hundreds on Envoy mesh instances. Lastly, PUMA will expose basic asynchronous logging functionality and formats.

3.3 PUMA and Portability

The PUMA package is distributed in two parts: PUMA library for application configuration and service definition, and PUMA core which contains the modifications to Envoy to accept such configurations.

PUMA Library

The PUMA library is a python library on PyPi[3] available as early as April 2018. It provides a set of functions used to configure PUMA Core in an application. These functions include: puma_config, puma_service, and puma_start.

`puma_config`

`puma_config` is a configuration method used to set PUMA Core configurations for logging, load balancing, databases, service definitions, rate limiting, and more. The only input parameter is a JSON or python dictionary with PUMA configuration keys and values. For example, configuring PUMA core can look like the JSON in Figure 4.

`puma_service`

`puma_service` is an code encapsulation method that takes python source code encoded as a string and passed to the corresponding service. Similar to the `puma_config`, the encoded source code is assigned to a service key and loaded into the runtime of PUMA's envoy flask management app on port 8080. For PUMA data bench-marking, `puma_service` is was used to define map reduce in the service space.

`puma_start`

`puma_start` is a wrapper to start the PUMA flask app and launch the envoy mesh. `puma_start` should be declared in the final steps of starting the application.

PUMA Core

The PUMA Core is an PUMA configured Envoy[4] front proxy. The PUMA Core contains modified configuration files and a master Flask[5] application. The Flask application serves to enable configurations, routes, database connections and more. All Envoy deployments rely on a master Flask application and PUMA is no different. However, the PUMA Core master application maintains configurations and PUMA service definitions. With the defined services and the encapsulated source code from PUMA Lib, PUMA Core can deploy such source code to the corresponding service and service URLs. Finally PUMA deploys the PUMA configured Envoy front proxy in a docker container, starts the upstream mesh, and enables worker listeners on all commodity hardware. Ideally service definitions would be established in the Envoy configurations as declared dynamically, however because of the short developmental time frame, we have manually configured the maximum number of service definitions to be five.

Porting With PUMA

Porting with PUMA requires minor modifications to existing applications. PUMA Core configuration JSONs are variable in length, but at a minimum require at least one service definition; this amounts to a single line of code. Another modification to a traditional application is including the `puma_start` method. Encoding application python code into a list requires approximately five lines of modified code. Most applications no longer need the server infrastructure code because it exists within PUMA and so applications typically become more lightweight; this is consistent with serverless migrations. Lastly, if URLs are defined differently as microservices than in the traditional socket implementation, then they need to be replaced with the correct PUMA service endpoints. Section 4.1 Porting An Existing Application, will demonstrate the simplicity of porting an existing application and how our application lost weight.

3.4 Database Optimization

Traditional database connections to legacy databases are still supported in PUMA, however conducive for optimal performance. Legacy database architectures are supported but may require Docker modifications to ensure that the proper

[3]https://pypi.python.org/pypi

[4]envoyproxy.io/

[5]flask.pocoo.org/

libraries are available at deployment. The PUMA migration of the dialog movie recommendation application, maintained the existing PostgreSQL database, but required psycopg2 package at deployment in order to connect to the PostgreSQL database.

Low request latency to database structures provides a challenge in distributed systems. Systems where micro-services are embarrassingly parallel and contend for the same structures provides potential hazards and impedes the ability for timely responses. While aggregating database structures by service helps, it does not scale with the PUMA deployment. Ideal databases for PUMA are DynamoDB[6] [DeCandia *et al.*, 2007] or MongoDB[7] because of the ability to horizontally scale with ease of configuration in Envoy by PUMA. Other database configurations should be considered if running Envoy on alternative elastic computing platforms such as Microsoft Azure[8] and Google Compute Engine[9]. Databases like Google's BigTable [Chang *et al.*, 2008] or Spanner [Corbett *et al.*, 2013], are suitable for PUMA because they are capable to handle an influx of requests as Envoy scales. Configuring any of these database solutions does impact portability, but with DynamoDB, it is to a minimum.

4 Implementation

PUMA was deployed by porting an existing monolithic application onto the Envoy Mesh. Additionally, to determine if the Envoy mesh would provide speedup in computationally expensive tasks, we benchmark PUMA with a simple map-reduce implementation.

4.1 Porting An Existing Application

Porting the existing application like our monolithic socket based python server application was incredibly simple. With only nine lines of additional code and 13 lines of retained logic, our application shed 49 lines of code. Appendix A demonstrates how using PUMA and moving to a pseudo-serverless architecture significantly reduces the responsibility of the developer by removing system code. With PUMA the developer can focus on the application code knowing their application can scale and load balance accordingly.

4.2 MapReduce

A branch of our PUMA system is created for the MapReduce benchmark. The mechanism and system framework are no different from the main PUMA implementation, only that some code is modified to aid the collection of experimental data.

The PUMA MapReduce system consists of the client API library and the server Envoy service. The client API library is responsible for providing the transparent function in which the application will call. The function takes in the sequential data and the mapping function as arguments. The library will then chunk the data into the size of the square root of the number of data given, so that the size of each chunk is similar to the number of chunks. We recognize this to be a point of optimization. The library then package the chunked data and mapping function into discrete JSON string, and is sent to the server Envoy endpoint via a RESTful API. The library then waits for all the response to be returned, re-combines the chunks and returns the result to the application.

[6]aws.amazon.com/dynamodb

[7]mongodb.com/

[8]azure.microsoft.com

[9]cloud.google.com/compute/

The server endpoint of PUMA utilizes Envoy's port services and load balancing. RESTful requests from PUMA client is received using the python Flask library, which maps the requested URL endpoint to their respective service. Envoy routes each incoming requests in a round robin fashion to a fixed number of service instances, which processes the request and return the results as a JSON RESTful API response to the original sender.

For our MapReduce implementation, the mapping function is covered into JSON by obtaining the string representation of the source using the built-in inspect module's `getsouceline()` [10]. The data is sent as a JSON array. On the server, the mapping function is converted to local object using the `exec()` function, and is then called with the data parameters using `eval()` [11]. In future work, we intended to streamline this process and migrate away from using `eval()` due to the inherent security vulnerability, but for this paper and the simplicity of our implementation, we included the client and the server connection into our TCB.

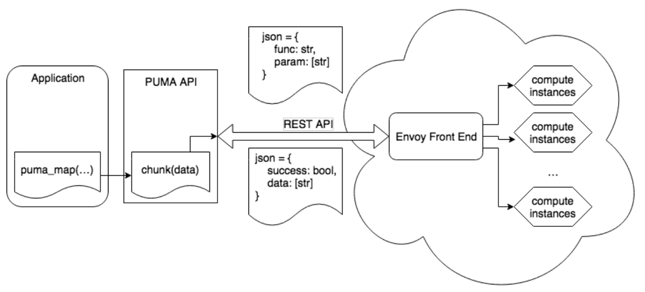


Figure 5: PUMA API service This figure describes the system diagram of PUMA MapReduce API. Applications call a function provided by PUMA's API library, for instance, `puma_map()`, where the API library will chunk the data array into multiple smaller arrays. The data and mapping function is sent as JSON via RESTful API to the target PUMA-Envoy service, where Envoy load balances the incoming requests into respective compute instances. Once the instance processed the data chunk, it is sent back to the client's API library via JSON. The API library then rejoins the chunked data arrays and return to the application.

5 Evaluation

A major goal of the PUMA system is to provide standalone applications transparent access to a distributed system framework. In order to measure the effectiveness of the PUMA sys-

[10]https://docs.python.org/3.5/library/inspect.html

[11]https://docs.python.org/3.5/library/functions.html

tem, we gauged the system based on three categories: Ease of adopting the PUMA library, throughput and latency.

5.1 Ease of Adoption

We qualitatively recorded the required effort of adopting a task or applications to using distributed systems using PUMA. Our point of consideration includes the transparency of using the PUMA system, the lines of code needed to be changed, and the number of factors needed to be considered by the application developer when porting their apps onto a distributed system.

5.2 Throughput and latency

We benchmarked the PUMA system on a MapReduce problem. MapReduce is a common problem both for standalone applications, such as the use of for loop on a list of data, and for distributed system applications, for processing a large dataset on server clusters [Dean and Ghemawat, 2008]. A MapReduce problem will also gauge the throughput and latency of our system via the size of the data being mapped and the complexity of the mapping function, respectively.

Benchmarking System

We created our distributed system on an AWS EC2 m5.24xlarge instances with 96 virtual CPUs and 384GB of memory [Amazon Web Services, 2018]. The distributed system is created by Envoy via docker-compose. We scaled our distributed system to having 250 instances for our benchmarking service using Envoy, where each instance is simulated using child processes behind the scene.

We ran two versions of the trial with different clients. The first trial was run with a 2016 15" MacBook Pro with an Intel Core i7 Skylake at 2.6GHz and 16GB of memory [Apple, 2017]. This is to simulate real-world usage of PUMA, where the developer of the standalone application is most likely to be running their application on their computer. The second trial was run directly on the EC2 instance. By running the client directly on the machine, it eliminates latency of going through the Wide Area Network from our results. The performance impact to Envoy and the distributed system is negligible due to the amount of resource on the EC2 instance.

Experimental Setup

We varied the data size of the mapping problem for benchmarking the throughput of the system. We created a MapReduce problem of 100 to 1,000,000 item with a step size of a factor of 10. Each item consists of the integer 100, each to be mapped by the factorial function. Each step size is run five times to reduce statistical variation.

We varied the integer to be mapped to benchmark the latency of the system. We vary the input integer for the factorial function, using 100, 500, 100 and 1500. Each trail is run with the data size of 100,000. Each input integer trials is run five times to reduce statistical variation.

The total latency of the system is measured by recording the time taken starting from the creation of the data array by the client until the client has collected all results from PUMA and returned the sum of all the mapped factorials.

```python
import asyncio
import puma

async def app_function(
        data: list
    ) -> list:

    def fac_1(n):
        total = n
        for i in range(n - 1, 0, -1):
            total *= i
        return total

    result = await puma.puma_map(
        fac_1, data
    )
    return result
```

Figure 6: Paralleling Data Operations with PUMA PUMA is more than a service definition and configuration tool. PUMA has can support computational tasks that are naturally advantageous to execute on a distributed system. With PUMA, a simple map reduce function can leverage the entire Envoy mesh. Data operations are balanced and scaled concurrent with other services.

6 Results

We collected data according to our evaluation metrics to gauge the performance and effectiveness of PUMA.

6.1 Ease of Adoption

We measured PUMA's ease of adoption according to its transparency, lines of code changes needed and abstractions of factors needed for consideration when porting application to distributed systems.

Transparency

PUMA enables transparent usage of the distributed system by abstracting communications with the servers, chunking of the data and functions and gathering of server responses away from the API user, as seen in Figure 5 and Appendix A . Application services only have to wait for the PUMA library to return the results, as the data must go through the network stack. Figure 6 shows that the PUMA system uses asyncio, a modern built-in Python library that allows for asynchronous tasks to be written like synchronous code[12]. The *await* keyword ensures code will only continue executing when all tasks within `puma_map()` is completed. The API is also designed to be as similar to the non-PUMA counterpart as possible. Application services may simply substitute `map()` from the Python built-in library with `puma.puma_map()`, and provide the correct configuration to PUMA, such as the address of the distributed system, and PUMA will transparently run the map function on the distributed system.

Lines of Code

Appendix A demonstrates less than 9 additional lines of code needed to be modified to migrate application logic to a microservice architecture. Figure 6 shows that custom computational operations such as map-reduce require little modified lines of code to changed to a distributed system task. A standard `map(function, iterable, ...)` function

[12]https://docs.python.org/3.5/library/asyncio.html

in python requires a function type and an iterable type. Our `puma_map(function, data)` implementation of map has a very similar function signature. This means the API users only have to change the function signature in order to use PUMA and the distributed Envoy mesh.

Factors for Consideration

Migrating applications to a distributed environment, or converting existing applications to a software as a service (SaaS) requires consideration of multiple factors in the application architecture. For instance, while most applications to be monolithic, SaaS such as our dialog systems needs to be created as a microservice. Traditional applications assume data to come from a structure that has a complete view of the data, while SaaS runs on distributed systems where databases are stored fragmented across multiple clusters. In order to take advantage of a distributed system, SaaS developers must rethink their application architecture and distribute their application as components piece by piece [Golding, 2018]. SaaS developers must also understand the restraints of the respective distributed system and choose the application components carefully to avoid bottleneck.

Our PUMA system abstracts away most of these considerations behind the API, allowing the application developer to convert components of their applications with ease. For instance, application developer converting a map function can simply call the `puma_map()` instead. The developer only needs to recognize components of their application that can be distributed, and the library manages the distribution and integration with the distributed system service. Load balancing and data consistency are done by PUMA as well, where PUMA manages the load balancing via Envoy and data consistency by combining the returned data before passing the result to the user.

6.2 Throughput and Latency

We measured the performance of PUMA quantitatively using MapReduce with varying data size and input integer for the factorial function.

Varying Data Size

From our result, PUMA has demonstrated to speedup computation when a certain data size is reached. As seen from Figure 7a, the graph for total latency when using PUMA intersects the graph for not using PUMA at between 10,000 and 100,000 when WAN overhead is present and at between 100 and 1,000 when WAN overhead is not present. This means that at their respective data size threshold, it is more efficient to run the MapReduce problem using PUMA on the distributed system than on the local machine. This is reasonable since using distributed systems allow the MapReduce task to be run in parallel across a large number of computing instances.

The differences in threshold where PUMA with local client performs better than PUMA with client over WAN illustrates the bottleneck of the system being the HTTP requests, which will be demonstrated in Figure 8. Since data is chucked to $sqrt(datasize)$ number of arrays each with similar sizes, the number of HTTP requests sent to the envoy services grows exponentially.

Both blue lines for MapReduce without PUMA is strictly exponential, meaning the MapReduce algorithm have not introduced noises to the result. The different x-intercept for both lines illustrates the latency due to differences in computing power between MapReduce running on the EC2 instance and running on a laptop.

Varying Factorial Integer

With data size kept consistent, PUMA's the speedup threshold is more prominent as seen in Figure 7b. PUMA systems, i.e. the orange lines, with and without WAN overhead are seen to perform better than without PUMA at between 100 and 500 factorial. This is reasonable since applications without PUMA has to run each computation after another, whereas PUMA can run the mapping in parallel. The overhead of HTTP requests is also constant since the number of HTTP requests made is constant.

The orange lines are seen to converge at 1000 factorial. This means that at high enough mapping complexity, The overhead of going through the wide area network becomes negligible due to the latency of the actual data processing.

Both blue lines for MapReduce without PUMA are vertically transpositional of another, illustrating the MapReduce algorithm yields predictable increase in total latency when the mapping complexity is increased. It also illustrates that differences in client computing power have a consistent effect at any mapping complexity.

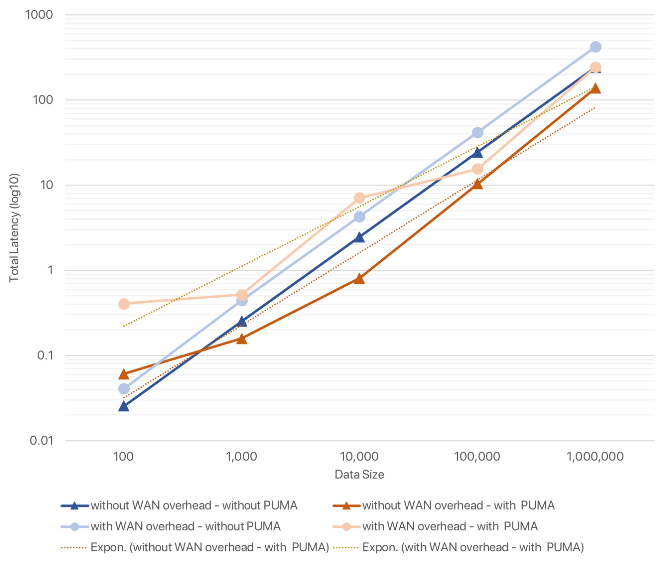
WAN Effects

The results from varying MapReduce's data size and factorial integer has shown an overhead with HTTP request. We plot the differences in total latency between trials with and without PUMA for both experimental parameters in Figure 8. These graphs demonstrate the change in network overhead with the change in data size and factorial integer.

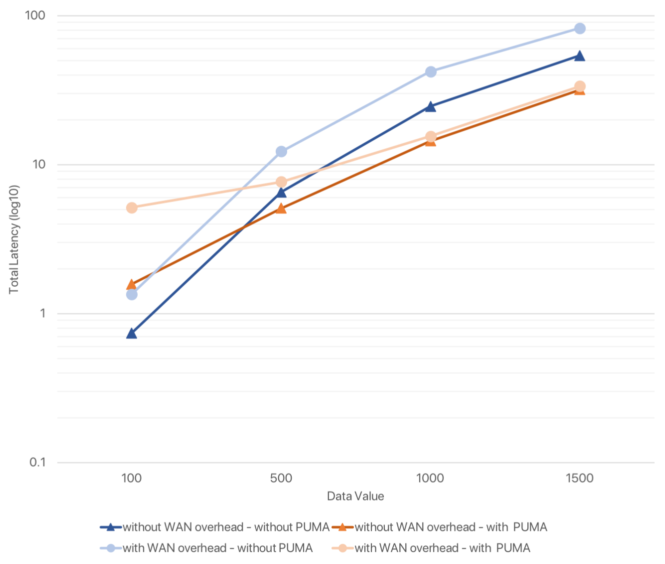
As seen in Figure 8a, the difference in total latency increases exponentially as data size increases. Since the only difference between the two PUMA systems is the usage of wide area network, the difference in latency observed is attributed to the network latency and variability during the trials. This will also explain the high fluctuation of the data. The difference in latency is increasing exponentially as the data size increases exponentially. This poses a potential problem with scalability as the data size grows even larger. However, since the issues lie with the number of network requests, optimizations on reducing network requests or intelligently balances the sending of resources should mitigate the issues. Having dedicated connections between client and PUMA, such as using the client within a company network, would also reduce the overhead.

Figure 8b further illustrates the effect of network overhead, and that complexity of the mapped function does not increase the latency of the MapReduce task. The difference in total latency decreases slightly as the factorial integer increases. This means the difference in total latency is mostly independent of the complexity of the mapped function. The slight decrease can be attributed to the fact that at large mapped function complexity, the time difference becomes less significant against the total time needed to complete the algorithm.

In addition to calculating the differences between using



(a) with Varying Data Size



(b) with Varying Factorial Integer

Figure 7: Total Latency of Performing MapReduce with Iterative Factorial Function Both graphs show the average total latency in logarithmic scale for the trials with varying data sizes in Figure 7a and with varying factorial input integer in Figure 7b. The two blue lines represent the trials where the MapReduce is run solely on the client machine. The two orange lines represent the trials the algorithm is run using PUMA – where the mapping of individual data is done using the distributed system. The lines with triangle represent the trials where the client machine is part of the EC2 instance. Thus there is now WAN overhead between the client-server communication. Exponential trend-lines are given for the orange lines. The lines with circles represent the trials where the client machine is the MacBook Pro. Thus each request to PUMA must be sent via the WAN to reach the distributed system.

PUMA with different clients, we calculated the difference between running MapReduce on the EC2 instance and a laptop. The green lines represent the difference in total latency due to the difference in computing power. Varying data size yields a standard exponential graph, since the number of data needs to be processed increases exponentially. Varying the factorial integer yields a linear graph, since the number of data needed to be processed is constant, but the complexity of each mapping increases.

7 Related Work

Our project involves multiple technologies within the area of distributed systems architecture. PUMA requires a distributed system framework to manage system resources. It also requires a data distribution mechanism to break apart computations on large datasets. Since PUMA's goal is to distribute standalone applications with minimal re-engineering, it needs to adapt the applications using interfaces or encapsulations. Below are some of the researches and tools in the community that can provide PUMA ways to fulfill those requirements.

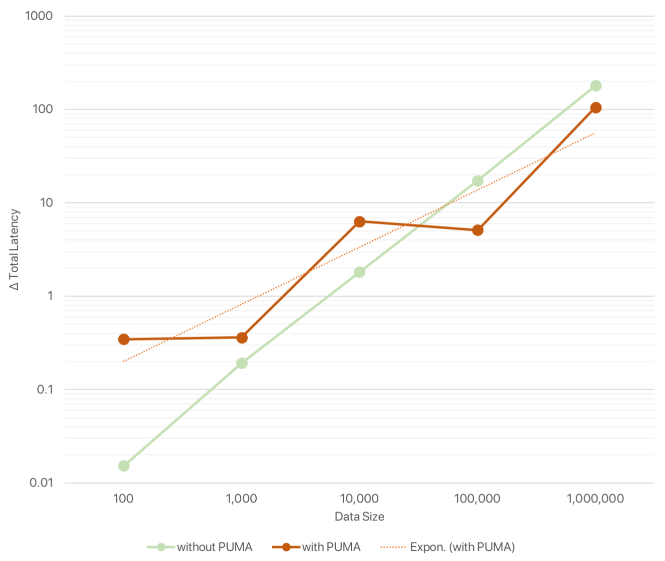
7.1 Distributed System Frameworks

A core part of a distributed system is the distributed system manager. Software, such as Envoy and Amazon Elastic Load Balancer, manage system resources over the network and distribute workload accordingly. It also serves as the interface between the application and the network, where common I/O such as database access are provided by the manager.

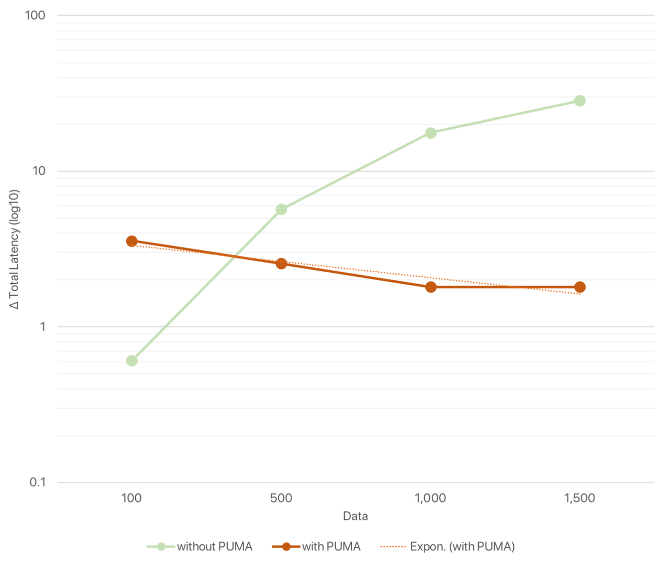
AWS Elastic Load Balancing is a scalable distributed system framework for the AWS cloud service. It manages incoming traffic either at request level or network level, then routes traffic to targeted network clients that are running identical or different micro-services [Amazon Web Services, 2018]. Amazon Web Services ELB balances incoming traffic within the network so that no one system is overloaded. This load balancer can provide a framework for managing our distributed systems. However, our project is not focusing on distributing incoming task within a network, but instead distributing internal processes across clients for optimal computing.

Envoy is an enterprise-sized distributed system framework that runs and manages services across a distributed system [Klein, 2016]. It enables network distribution transparency by providing a communication mesh for applications over the network. While it is feature-rich and comprehensive, our project is not focusing on managing or load balancing between services, but how we can distribute workload on an application level using a load supervisor. However, its design for abstracting the network can be used to provide network distribution at the application level.

Chubby is a course-grained lock service designed to scale applications across networked machine clusters [Burrows, 2006]. For example, the Google File System uses Chubby to appoint master servers and as the root of their distributed data



(a) with Varying Data Size



(b) with Varying Factorial Integer

Figure 8: Total Latency Penalty from WAN overhead Both graphs show the difference in total latency between trials with the client running on the EC2 instances and running on the local MacBook Pro in logarithmic scale. Figure 7a shows the results for trials with varying data sizes and Figure 8b shows the results for trials with varying factorial input integers. Green lines show the difference in latency without PUMA, i.e., the difference in latency due to the computing power of the two clients. The orange lines show the difference in latency for the PUMA system, i.e., the latency due to the overhead of sending requests over the wide area network.

structures [Burrows, 2006]. By using a lock-based paradigm, Chubby abstracts management of distributed consensus and allow applications to be implemented and scaled with relatively low effort. However, Chubby focuses on reliability and deprioritizes performance, which dialog system microservices require to achieve low interaction latency.

Zookeeper is a distributed database management software designed by Yahoo for Hadoop and eventually integrated with Apache [Hunt *et al.*, 2010]. Similar to Chubby in principle, Zookeeper provides intuitive interface using locks, registers and group messaging to the underlying distributed file system. It is a useful and powerful tool at the data-level, which our framework can utilize for managing data concurrency over the network. Apache and Hadoop, which zookeeper is currently part of, can be used as a foundation for managing data distribution in our framework.

7.2 Distributed Data Computation

Distributing partial data for machine learning across a distributed domain has been proposed for image processing tasks [Alonso-Calvo *et al.*, 2010]. Images are divided into different sub-images that can be stored and processed separately, which the system manger can distribute the processing of each sub-image over the network. One of PUMA's responsibility is to allow applications that work on large continuous datasets to be distributed across the network. It must be able to manage the division of the data and distribute the task over the network. PUMA is also responsible for aggregating the results of the completed distributed task once all sub-images are processed. Unlike Google's TensorFlow [Abadi *et al.*, 2016], PUMA optimizes both request and computational resources for current usage needs and is not exclusive to rigid and specific training optimization that are leveraged exclusively on Google clusters.

7.3 Existing Software Adaptation

A major focus in distributed systems is the adaptation of legacy software into the distributed network. Most legacy software is written with outdated technology and is not portable to modern network-centric architectures. One method of deploying legacy software components into modern distributed systems is through encapsulation [Sneed, 2000]. A wrapper is used to encapsulate the legacy software and provide a bridge between modern architecture and the software components. Wrapping technology provides modern applications access to the legacy components, such as wrapping legacy databases to provide object-oriented semantics to modern software. This concept of wrapping can be adapted to our framework, either by providing bidirectional interfaces using the wrapper, or using the wrapper on the distributed network to that the application can see the network as a standalone, non-distributed hardware.

8 Conclusion

We have described PUMA and its ability to provide distributed computing environments for traditional monolithic applications. PUMA provides applications with serverless advantages such as scaling, load balancing, deniability of infrastructure provisioning and management all without the disadvantages of vendor lock-in, lack of security, and cold start. We have shown that porting applications with PUMA is flexible enough for most use cases and potentially suitable for wide adoption. Moreover, we demonstrated that PUMA outperforms traditional computational heavy tasks by leveraging the Envoy mesh. With applications becoming more complex and dependent on simultaneous high performing services, it becomes increasingly important to delegate resources efficiently and optimally. PUMA accomplishes this task and provides significant ease of adoption.

References

[Abadi *et al.*, 2016] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Gregory S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian J. Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Józefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Gordon Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul A. Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda B. Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *CoRR*, abs/1603.04467, 2016.

[Alonso-Calvo *et al.*, 2010] Raúl Alonso-Calvo, Jose Crespo, Miguel García-Remesal, Alberto Anguita, and Victor Maojo. On distributing load in cloud computing: A real application for very-large image datasets. *Procedia Computer Science*, 1(1):2669 – 2677, 2010. ICCS 2010.

[Amazon Web Services, 2018] Inc. Amazon Web Services. What is elastic load balancing?, 2018.

[Apple, 2017] Inc. Apple. Apple, Sep 2017.

[Armbrust *et al.*, 2010] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, April 2010.

[Baldini *et al.*, 2017] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, et al. Serverless computing: Current trends and open problems. In *Research Advances in Cloud Computing*, pages 1–20. Springer, 2017.

[Burrows, 2006] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 335–350, Berkeley, CA, USA, 2006. USENIX Association.

[Chang *et al.*, 2008] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[Corbett *et al.*, 2013] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.

[Dean and Ghemawat, 2008] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[Dean *et al.*, 2012] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Quoc V. Le, and Andrew Y. Ng. Large scale distributed deep networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1223–1231. Curran Associates, Inc., 2012.

[DeCandia *et al.*, 2007] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *ACM SIGOPS operating systems review*, volume 41, pages 205–220. ACM, 2007.

[Gemulla *et al.*, 2011] Rainer Gemulla, Erik Nijkamp, Peter J. Haas, and Yannis Sismanis. Large-scale matrix factorization with distributed stochastic gradient descent. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '11, pages 69–77, New York, NY, USA, 2011. ACM.

[Golding, 2018] Tod Golding. Migrating applications to saas: A minimally invasive approach, Mar 2018.

[Gustafson *et al.*, 2000] Joakim Gustafson, Linda Bell, Jonas Beskow, Johan Boye, Rolf Carlson, Jens Edlund, Björn Granström, David House, and Mats Wirén. Adapt-a multimodal conversational dialogue system in an apartment domain. In *Sixth International Conference on Spoken Language Processing*, 2000.

[Henderson *et al.*, 2013] Matthew Henderson, Blaise Thomson, and Steve Young. Deep neural network approach for the dialog state tracking challenge. In *Proceedings of the SIGDIAL 2013 Conference*, pages 467–471, 2013.

[Hunt *et al.*, 2010] Patrick Hunt, Mahadev Konar, Yahoo ! Grid, Flavio P Junqueira, Benjamin Reed, and Yahoo ! Research. Zookeeper: Wait-free coordination for internet-scale systems. 8, 06 2010.

[Klein, 2016] Matt Klein. Announcing envoy: C l7 proxy and communication bus, Sep 2016.

[Levin *et al.*, 1998] Esther Levin, Roberto Pieraccini, and Wieland Eckert. Using markov decision process for learning dialogue strategies. In *Acoustics, Speech and Signal*

Processing, 1998. Proceedings of the 1998 IEEE International Conference on, volume 1, pages 201–204. IEEE, 1998.

[Patil *et al.*, 2017] Amit Patil, K Marimuthu, R Niranchana, et al. Comparative study of cloud platforms to develop a chatbot. *International Journal of Engineering & Technology*, 6(3):57–61, 2017.

[Sneed, 2000] Harry Sneed. Encapsulation of legacy software: A technique for reusing legacy software components. 9:293–313, 05 2000.

[Stigler, 2018] Maddie Stigler. Understanding serverless computing. In *Beginning Serverless Computing*, pages 1–14. Springer, 2018.

[Thomason *et al.*, 2016] Jesse Thomason, Jivko Sinapov, Maxwell Svetlik, Peter Stone, and Raymond J Mooney. Learning multi-modal grounded linguistic semantics by playing" i spy". In *IJCAI*, pages 3477–3483, 2016.

[Villamizar *et al.*, 2015] Mario Villamizar, Oscar Garcés, Harold Castro, Mauricio Verano, Lorena Salamanca, Rubby Casallas, and Santiago Gil. Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In *Computing Colombian Conference (10CCC), 2015 10th*, pages 583–590. IEEE, 2015.

[Villamizar *et al.*, 2016] Mario Villamizar, Oscar Garces, Lina Ochoa, Harold Castro, Lorena Salamanca, Mauricio Verano, Rubby Casallas, Santiago Gil, Carlos Valencia, Angee Zambrano, et al. Infrastructure cost comparison of running web applications in the cloud using aws lambda and monolithic and microservice architectures. In *Cluster, Cloud and Grid Computing (CCGrid), 2016 16th IEEE/ACM International Symposium on*, pages 179–182. IEEE, 2016.

[Yu *et al.*, 2015] Zhou Yu, Alexandros Papangelis, and Alexander Rudnicky. Ticktock: A non-goal-oriented multimodal dialog system with engagement awareness. In *Proceedings of the AAAI Spring Symposium*, volume 100, 2015.

Appendix A: Migrating Application To Distributed Mesh With PUMA

Listing 1: Before PUMA Migration Monolithic threading application used traditional sockets. Most migration logic occurred here within `listenToClient`.

```
import socket
import threading
import dialogueCtrl as dCtrl
from dialogueCtrl import dialogueCtrl, initResources, dialogueIdle
import json
import sys
import time
import traceback

debug = False
passive = {}


def chatbox_socket():
  if debug:
    return 13120
  else:
    return 13113


class ThreadingServer(object):
  def __init__(self, host, port):
    self.host = host
    self.port = port
    self.sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    self.sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    self.sock.bind((self.host, self.port))

  def listen(self):
    self.sock.listen(5)
    self.sock.settimeout(None)
    while True:
      client, address = self.sock.accept()
      client.settimeout(60)
      threading.Thread(target=self.listenToClient, args=(client, address)).start()

  def listenToClient(self, client, address):
    size = 2048
    while True:
      try:
        data = client.recv(size)
        if data:
          try:
            print "data: {}".format(data)
            response, userid, passiveLen, signal = dialogueCtrl(data)
            if response == dCtrl.end_dialogue:
              signal = 'end'
            responseJson = json.dumps(
              {'response': response, 'userid': userid, 'signal': signal, 'passiveLen': passiveLen})
            print responseJson
            client.send(responseJson)
            if signal != "listen":
              dialogueIdle(userid, debug)
          except Exception as e:
            exc_type, exc_value, exc_traceback = sys.exc_info()
            traceback.print_tb(exc_traceback, limit=1, )
            traceback.print_exc()
        else:
          raise error('Client disconnected')
      except:
        client.close()
        return False
      client.close()

if __name__ == "__main__":
  initResources()
  if 'debug' in sys.argv:
    debug = True
    print "Running in debug mode. (socket: {})".format(chatbox_socket())
  while True:
    ThreadingServer('localhost', chatbox_socket()).listen()
```

Listing 2: After PUMA Migration Only code retain was application logic which was encoded and sent to PUMA Core for deployment to service endpoints

```
from dialogueCtrl import dialogueCtrl, initResources, dialogueIdle
import json
import puma_lib
import inspect

def service1():
  _text = request.form['chatText']
  _mode = request.form['mode']
  _id = str(request.form['UUID'])
  if _id == '-1':
    _id = str(uuid.uuid4())
  # app specific python container
  response, userid, passiveLen, signal = dctrl.dialogueCtrl(json.dumps({'text': _text, 'mode': _mode, 'id': _id}))
  if signal != "listen":
    dctrl.dialogueIdle(userid, debug=True)
  return json.dumps({'response': response, 'userid': userid, 'signal': signal, 'passiveLen': passiveLen})

conf = {}
conf['puma_services'] = {
  "service1": "/agent",
  "service2": "/listener",
}
puma_config(json.dumps(conf))

service = {
  "service1": inspect.getsourcelines(service1())
}

puma_service(json.dumps(service))

if __name__ == "__main__":
  initResources()
  puma_start()
```