University of Exeter

Final Report

ECM3401 Individual Literature Review and Project

Babel Analytics Platform

Author:
Kieran Bacon

Module Leader and Supervisor:
Yulei Wu

I certify that all material in this document which is not my own work has been identified.

.....

May 3, 2017

Abstract

The report documents the process of developing two pieces of complementary enterprise software. The purpose of products is to help companies fully utilise the potential of analytical insight, by hosting analytical models on a simplistic RESTful API with the means to manage and operate them through a graphical interface. I review the literature to help inform and assess the legitimacy of my interpretation and examine some business cases that may benefit from the use of the product. An in depth discussion on the design and implementation with respect to the initial project specification and any design pivots during the process is presented. Outline testing was done on each product to verify robustness. Two critical evaluations appraise each piece of software both individually and combined against the initial requirements of the project, I conclude that both products are fit for purpose.

Contents

Acknowledgements

I would like to express my gratitude to the following people for the time they have given to me in sparking my imagination and discussing technical issues whilst I worked through the project: Dr Jacqueline Christmas, Dr Yulei Wu and Mr Thomas Brady.
Your help has been invaluable, Thank you.

1 Introduction

The Analytics Server is a product that aims to help businesses produce and utilise analytical power both day to day and in real time. More specifically it aims to breach the gap between the developers who generate services and the users who operate their implementation. Traditionally beneficiaries of services do not have a hand in the development of models, instead they essentially commission them when a problem is identified and implement them when they are produced. In the process of developing an analytical solution for which a model is to be deployed, the developer has to acquire a large amount of knowledge regarding the current state of a user, the resources they have access to and any interfaces. This inevitably leads to a reliance on them to subsequently develop solutions with respect to interface compatibility and in some cases, consult with the team to trains users in the use of the service. This can result is higher costs to the firm because of the over qualified nature of staff being employed on interface and training activities. Furthermore revisions to the model may have to undergo additional validation steps akin to a pitch, as the department may decide the improvement is not worth the new training required, resulting in further input by the original developer. By being able to separate these functions say into three parts: development, in-bedding and training, some tasks can be delegated within the firm to fully utilise the available expertise at optimal cost. In conjunction, with the analytics server is a solution that weaves this dynamic into a company, it creates a method of interaction for a service that is abstracted away from its implementation with equal or improved accessibility. It maintains the developer's ability to improve and interact with the service and it handles all the connectivity issues related to the model. Both users and developers benefit as a consequence allowing everyone to utilise their time more efficiently.

The management console is a complementary piece of software that aids developers in the management and use of multiple analytic servers within a graphical user interface. The Console aims to handle the distribution of services across expansive networks of analytics servers through a system that mimics typical development stages embodied by Waterfall development. The console is flexible in design and modular so that developers are able to add and edit its appearance and functionality to the demands of the company. Acting as an example of what is possible, the console offers convenient methods and functionality that allows customers to implement and begin operating analytics servers immediately.

A distinction must be made between entities that interact with these products. The intended customer base of the platform is large scale businesses that utilise vast amounts of statistical and analytical insight to perform day to day. A user is defined with reference to any entity that intends to use the analytical models that are being hosted by an analytics server. These users may well be other applications, employees, departments or members of the public. The server does not specifically prohibit any entity from utilising a model, and if a model is to be private, its permissions are to be set within the service itself. Users are numerous and encompass all vectors one might imagine linking with the Internet. Developers are a set group of known employees of the customer, pre-platform, who prior to development of the platform produced analytical solutions. Post-platform they are additionally responsible for the management of the services on the platform (deploying, updating and deleting) and general maintenance of the analytics servers.

I chose to follow the principles of Agile development as my design process. Initially declaring each product as an Epic goal, I created a backlog of stories illustrating the precise functionality of each product. Along with each story, I identified who was the stakeholder and the expected difficulty of the task. I then rated their importance against those metrics and began the project taking on three stories a week. The regular re-evaluations meant that I could quickly switch sto-

ries in and out depending on the stage of development I had reached, the weekly retrospectives with my supervisor meant that I continuously had an understanding of the stage the project was at. Like most software development, the project requirements and scope grew constantly as I revisited and researched the problems, though I was eventually able to achieve my original aim by prioritising the critical stories. I also made use of a GIT repository to store and backup my project during development, and it provided a clear visual means of comparing completed stories.

I deviated from the standard agile development process by not being able to produce multiple iterations of a minimal viable product. To be viable, the platform would still have to be rather complex, making it unrealistic to try and achieve. I aimed to separate out the products into a collection of modules so that they could be tested and have their intended functionality demonstrated individually. This allowed me to get a general understanding of the viability of a story long before implementing sections that relied on them. A by-product was that I had to undergo extra work piecing together the modules into a coherent product after the fact, but it was a preferred method of development.

2 Relevant Literature

In order to inform the development of this product it was essential to review the literature of current methods and applications to gain an understanding of what is already possible, and what is required for the project to succeed. My main focus was on the programming languages to be used for the products, the message protocol they should enact and the design patterns of popular server implementations.

2.1 Language choice

The analytics server is envisioned to be able to handle many concurrent requests that operate computationally expensive scripts and programs. It is important that the server runs as fast and efficiently as possible, not only for the user, but to reduce the likelihood that the hardware will become exhausted. The Management Console on the other hand is envisioned to have a fraction of the traffic with smaller message payloads. The consoles does not focus on speed and efficiency rather on the ease of implementation and modularity. As of November 2016, the top five most popular programming languages are Java, C, C++, C# and Python respectively. [1]

Empirically, the compiled languages can be used to generate solutions around five to thirty times faster than interpreted languages when tasked with producing the same output. However, Python's in built structures and functionality set it apart from these languages: when tackling problems that can make use of Python's inbuilt structures, such as lists and dictionaries, Python can operate on an equivalent or better level than Java with consistently less memory costs. In terms of complexity, Python requires around half the amount of coding whilst simultaneously taking a fraction of the construction time in comparison with the compiled implementations. A disadvantage of Java is that its virtual machine dramatically interferes with its efficiency such that it is out performed by C and C++. Resulting in Java taking around three to four times as much computational time and memory to complete similar tasks.[5][8] When comparing C and C++, both operate to a similar standard with performance between them being only marginally different. C++ provides a larger scope of control over its environment and with a greater user base, has a more expansive library of packages and frameworks. Furthermore, C++ can fully utilise the subset C so all practical advantages of the C language can be obtained. C++ is extremely fast and efficient and would be the correct language to build a resource intensive server, for this reason it has been selected it to build the analytics server. The Management

Console is likely to perform perfectly without this aforementioned level of efficiency. Therefore Python is the correct language to use for the management console as its performance is adequate for its needs and it would allow for possible adaptations and modifications to be made in the future.

2.2 Message Protocols

The feasibility of the Server is linked to the manner in which it communicates its analytical insight. A single solution is intended to supply usable outputs regardless of the origin of any a requests. In this vein, it is necessary to communicate via a protocol which is well supported across languages, applications and systems. Additionally, the protocol is required to efficiently transfer both small, to exceptionally large payloads seamlessly and robustly. In doing so it is imperative that no message packages can be lost in the transfer, therefore UDP based protocols, which are known to suffer from this problem must be ruled out of this review.

After reviewing the available protocols, it would seem that the Web Application Messaging Protocol method of communication is superior to HTTP in a contest of speed, however due to its infancy, WAMP does not currently maintain a sufficiently wide enough scope of language support or functionality to rival that of HTTP. Functionalities such as user authentication are still in development and therefore infringe its ability to pass payloads that could potentially contain large amounts of sensitive data.[4] The multiple platform support of HTTP and its powerful functionality set it apart from RTMP and WAMP.[11] Another powerful attribute of HTTP is its ability to dynamically define whether a connection should persist, which reduces the overhead cost associated with forming the TCP connection parameters, which in turn avoids the laborious handshake issue and provides speeds like a UDP protocol.[3] Lastly, the extensive compatibility of HTTP with a majority of languages, and the certainty that it shall continue to be developed and supported is the reason I selected it as my messaging protocol.

2.3 Server Implementation

The product's ability to integrate itself into a working environment is somewhat more of a business consideration that its ability to deliver. The Server aims to support and encourage the use of analytics regardless of the companies development dynamic and therefore its flexibility and functionality needs to be large enough to effectively work in a range of configurations. Primarily the design should consider multiple implementations spanning several machines, creating a distributed network of Analytics Server Nodes. Secondly, it is plausible that multiple implementations may be instantiated on a single machine, for environments that are not expected to have large volumes of traffic or are mainly for the use of developers. The ability to plan the use of resources appropriately and effectively is a priceless feature the product should embody. I have looked at three server implementations to take inspiration from their design: Apache, Node.js and NGINX.

NGINXs use of a single worker process per core allows it to dramatically improve a systems performance. Processes are pinned to a core which reduces the overhead generated by content switching that would occur with systems that generate a thread or process for each request.[9] Another benefit is that creating a new connection within the worker processes requires very small amounts of additional memory in comparison to memory usage of a new thread/process with a connection. Unfortunately, despite NGINXs improved performance, due to the nature of the content on the Analytics Server being complex; where objects benefit from being able to utilise large sections of memory when being run, it is not feasible to have a single process managing and controlling multiple connections in this manner. The overhead of the server is likely caused by the processing of the scripts and programs rather than blocking periods of waiting for client

responses. NGINXs and Node.jss implementation would remove the ability for the connections to be switched in and out and this will in turn likely increase the time taken for responses to be generated when dealing with large volumes of traffic.[2][12] Another consideration is that meaningful change could have occurred to the complex object implementing a script, this object may need to be reset after responding to a request which can be expensive. Apaches set up allows not only for a large amount of resources to be allocated to a scripts computation but, allows the thread to continue to resolve and work on an object after a response has been given, without entirely blocking other request threads. Also, Apache's operation is heavily dictated by a set of configuration files which define the intended behaviour of the server, this has been one of the main reasons it has been so popular as it can adapt to multiple environments. I have therefore decided to draw inspiration mainly from the Apache design pattern.

3 Analytics Server

The Analytics Server is a product that acts as an analytical platform for companies to host, develop and operate statistical programs and models as web services. It provides a simplistic RESTful API that allows users to access and connect to models without concern for the underlying architecture of the service. Additionally, it aims to enhance the performance over conventional usage without limiting the functionality or connectivity of the service.

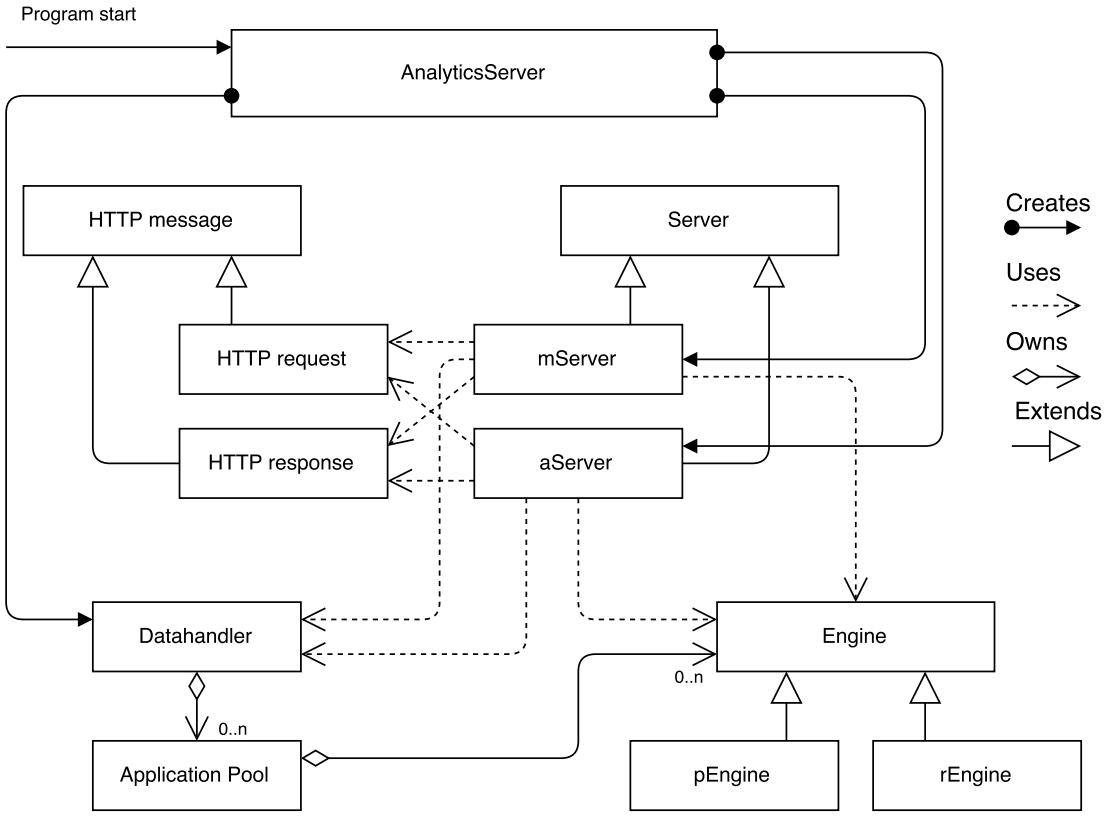


Figure 1: A component diagram to illustrate the underlying structure of the Analytics Server, excluding auxiliary classes and structures.

Despite the accomplishments of the Analytics Server to date, it is a product that will be in continuous development to keep the product competitive and viable in the business world. The main intention is to enhance a firm's capability to utilise analytical power, giving a firm the ability to provide solutions to problems in real-time, with models that can form a part

of a departments tool set, without overburdening them with responsibility. The adoption of an Analytics Server creates a mutually beneficial environment for departments and software developers alike, providing departments access and developers control.

3.1 Original Specification

Along with conducting my literature review at the start of the project, I also defined a set of criteria for the evaluation of the project. Identifying what the analytics server should encompass and the initial scope of its functionality. In doing so, I defined a means of defining how well the product fits its purpose. Meeting the criteria will be testimony to the project's fitness of purpose and general success.

Communicate via the HTTP Protocol: The server needs to accept messages that abide to the HTTP standard to allow the server to be inclusive and flexible with its ability to integrate with other systems, local or external. The Server is intended to supply both applications/systems, and human users, with usable analytical information and therefore needs a common simply transfer protocol.

The analytics server both accepts and responds to HTTP messages via the HTTP Protocol on both the analytics and management port. The server provides a well defined RESTful interface that allow users of all types to be able to simplistically interact with the system.

Host service information pages internally: The server needs to provide functionality to host information pages for services such that a human user can use it as a reference when attempting to understand the requirements of the service.

The server provides developers with the ability to produce service specific HTML web-pages that may illustrate to a user the intended use of a service, for each service. Users are then able to access this page by simply requesting the resource from any typical Internet browser.

Redirect to externally hosted service Dashboards: Products such as Spotfire and Tableau provide and generate great visual 'dashboards' for datasets and services alike. It is important that the Analytics Server has to ability to integrate with these systems in place of a locally hosted solution. Suitable functionality should exist to allow developers to integrate externally hosted dashboards for the services.

The server provides developers with the ability to choose for a service whether their information page is internal or external. The server correctly handles this selection and redirects users to the location given in place of returning a locally hosted HTML page.

Handle large volumes of concurrent requests: During normal computation, the server should be able to handle multiple concurrent requests for multiple services, and management commands, without causing a disruption to the server as a whole or any given service.

The server's structure allows it to handle multiple concurrent requests for analytical services quickly and safely by threading each request, while protecting against overwhelming volumes of traffic. The management port does not handle requests concurrently, which was a design choice mid project. The management requests are still run along side analytical request without disrupting the availability of services. Both ports implement a variable length request queue that ensures that a manageable volume of requests persist while the server decides how to handle them.

The management port in is not threaded, as the port is responsible for causing large fundamental changes to the state of the system which could be potentially hazardous if multiple threads were simultaneously operating. Despite this, as the management port is meant for developers only, who are generally few in number, there is no negative impact on its availability. Furthermore, less resources will be required for the management port, partitioning resources favourably for the analytics requests. I decided therefore that it was beneficial for the server to make management requests consecutive.

Host analytics solutions with enhanced performance: The server must handle the interaction between users and services, passing inputs in while protecting against errors, and passing out outputs with an indication of their validity. Services are expected to perform at a rate that is at least equivalent to their normal execution, although services should have reduced times due to preprocessing of service dependencies.

Speed ups have been seen in both Python and R for models that require pre-processing of script data. Python gets an additional speed up from being compiled at initialisation via the C API, removing the need for the scripts to be interpreted and thus lowering the computational effort of the script.

Host Analytics solutions without restriction on connectivity: Services should not be restricted in their ability to integrate with other applications and processes. i.e., the service should maintain its ability to integrate with any database method that would have been accessible naturally in the language.

Both Python and R operate in a private workspace with private environment variables. As long as the user permissions have been given to the analytics server prior to instantiating those environments the services are not restricted in their operations. They may access any package or library they would have had access to naturally.

Deploy, Update and Delete Analytical solutions in real-time: The server must facilitate the action of hosting a newly created service during the server's natural runtime, with functionality to determine the state of the service. The server must also facilitate the actions of updating and deleting the service information.

The management port provides APIs to construct and edit services and permits developers to host new services in real-time and determine at the point of initialisation the status of the service. State can then be changed in real-time and the service information can be edited, all of which are done through dedicated APIs that are thread safe. The server also allows the deletion of services from the server and handles the removal of the service file on the local machine.

3.2 Design and Implementation

The design for the server has largely focused on developing a future-proof solution, with abstracted sections that open the server up to multiple configurations. Taking inspiration from the successful Apache web-server, the system relies on configuration files to define the internal function of the server. A single configuration file known as the *master*, defines all the initial communication information for the server, and dictates the locations of file directories. The master may also hold references to service configurations files that are to be initialised at server start up. The service configuration file defines how the service is meant to be operated and the location of its source material.[A.1][A.2]

Currently the server is started via the terminal and accepts a small set of tags to edit its initial behaviour. The program reads through the tags and records their inputs, exiting at any point if a tag is invalid. The last tag is used to identify the location of the master configuration file. In the event the location is not found, the program searches nearby locations for a configuration file. If no file can be found, the program exits with the cause. If found, the program continues to determine from the tags given, the type of running procedure the analytics servicer is to undergo. An example of a command for starting the server would be:

./AnalyticsServer -r DEBUG4 -m 5665 -a 7980 ./server.conf

Tag information:

-h -help	Displays the information of the available tags and provides a small description on how the program is intended to be started and used.
-d -start_daemon	Identifies that during the start up process, the program should separate itself from the current terminal and user and persist. This tag also results in the request for terminal output to be suppressed.
-t -test_start	Runs the initialisation and shut-down sequences only, binding to ports without accepting requests, so that the developer can ensure the server has been installed correctly.
-a -analytics_port	Starts the server accepting analytical requests on the following port number.
-m -management_port	Starts the server accepting management requests on the following port number.
-r -report	Forces the server to print log information to terminal rather than file, at the debug level that follows.
-l -log_location	Overwrites the log location given in the master configuration file.

The configuration address is passed to the Data-handler which begins to parse the file and extract the server's parameters into a singleton instance for global access. Tags that were read from the terminal replace any extracted information at this point. Both ports are then initialised and the program begins to accept requests.

Throughout the operation of the analytics server, components are required to produce logging information to flag unexpected or undesired behaviour. This logging feature was required to provide granular levels of detail to the logs and be accessible; I am happy to report that I have been able to achieve this with an in-line class.

I used the libconfig package for C/C++ to manipulate the configuration files for the server.[7] What drew me to this package was the fact that it uses and creates light weight and readable configuration files, and that it parses files with a fully reentrant parser that can operate on multiple threads simultaneously. The package is very flexible in terms of types and function and provides lots of convenience. I am confident that the package will provide additional utility in future iterations of the analytics server and it is a fundamental aspect to the function of the server.

Configuration files not only provide developers more control and flexibility over the operation of the server, but they facilitate future expansion of parameter scope. New message protocols

could be added without affecting the method of interaction for the developer. The greatest asset the server will eventually provide is the range of languages it will be able to host simultaneously. The server currently runs services abstractly as illustrated in figure 1, ensuring that the future inclusion of languages will be a simple task.

The server is essentially split to handle the separate stakeholders. Developers are intended to interact with the server through a management port providing them with functionality to change the state of the server. Users are intended to interact with the analytics port, providing them a simplistic API for the services. The server is essentially running two main threads of computation, the analytics thread and the management thread.

3.2.1 Analytics Thread

The analytics thread handles all of the user request traffic for the server, and spawns request threads to interpret the HTTP messages received to determine what content should be provided. After the master configuration file has been read, a server object is created that binds to the analytics port. The analytics thread is then spawned to wait on the port and to listen for incoming requests. Incoming requests join a queue structure on the port and wait to be moved by the analytics thread to a new socket to be handled. The analytics thread spawns a new request thread to process the HTTP message, and in doing so returns to listening to the port for new requests.

The Request thread converts the incoming HTTP message into a request object separating out the relevant information from the message. A response object is also generated to encapsulate the server output for the request. The request intent is then identified from the method and resource combination. The server currently accepts four methods:

GET messages identifies that the user would like the generic HTML representation for the service. Depending on the service, this could be an internal HTML file or a link to an external location.

POST messages identify that the requests body content is to be used as the payload to the service running under the requests resource name.

OPTIONS messages asks for a list of available interactions with the analytics port.

TRACE messages asks the server to simply reply with the request message content, to verify that it is accepting and reading the message body correctly.

All other methods are currently not implemented but the functionality to accommodate them is present for future potential development. Methods not implemented are rejected appropriately. This simplistic interface provides everything necessary for connecting with an analytical model. Utilising the trace method, users can ensure that the server is correctly receiving their message contents. With the ability to receive the service representation via a 'get request' from a browser, users can therefore be confident that they are using the service correctly. Fundamentally, a user can access an analytical model by generating a post request with a valid service payload, to reap the performance benefits from a enhanced script without needing to understand programming.

Understandably any errors that are produced throughout the processing of a request are communicated to the user via the use of the HTTP status code response message. More specification is given in the event that the error cannot be conveyed entirely though status alone. An example of which would be no content responses: when requests for services come in, the thread initially

locates the position of the application pool for the service, failure to find one, or if no services are being hosted at that time, results in a 204/404 error being returned to the user accompanied by its cause.

For GET messages the thread collects the HTML contents for the service set when the service was initialised. Depending on the status code flagged by the service, the content forms either the message body, or a part of the response header determining the location of the service dashboard.

POST messages first collect a service engine for the service specific application pool and abstractly executes the message payload. The string output from the engine forms the response message body and the status of the engine, which may be determined by the script, this sets the response status.

Immediately after the response has been finalised it is returned to the user. The thread logs the request information and the time taken to produce a response for that particular service. If the thread used an engine during its processing, it is responsible for returning it to the application pool. Depending on the state of the used engine, the thread may have to generate a new engine to replace the original. When the engine is returned, the request thread terminates.

3.2.2 Management Thread

The management thread handles all of the developers requests to fundamentally change the state of the server. Operating in a similar fashion to the analytics thread, the management thread interprets incoming HTTP messages and performs an action, returning a HTTP response message at completion. After the master configuration is read, a server object is created to bind with the defined management port. The management thread is then spawned and is initially required to handle small managerial tasks. The thread is responsible for running and operating the Bin-man thread that is dedicated to freeing used resources during the lifetime of the server. The management thread also sets up and edits the global logging parameters as defined by the user arguments or configuration file. This is done after the analytic thread has been successfully started and has begun waiting on requests. The management thread then begins waiting on the bound port for HTTP requests to arrive.

Upon receiving a connection on the port, the management thread moves to process the request itself rather than spawning additional threads. This ensures that a management request is entirely processed before accepting another request that may potentially conflict. The thread behaves like an analytics request thread by converting the HTTP message into a request object and produces a response object for the server output. However, before the server acts upon the contents of the request, the user's validity is determined. Management requests are required to provide as headers a Username and Access-Token value, both values are checked against authorised usernames and key values that were dictated in the master configuration. The username is separate from the access-token that is used only to determine the degree of the rights of the user. At this stage if the user is found to not be genuine, permission is denied and a response illustrating that is returned. Otherwise the requests action is determined from its method and resource combination. The management thread accepts five methods:

GET messages are to access actions that the server does not need additional information to complete. The resource is split into an action identifier and a target service taking the form */ACTION/TARGET*. Actions are as follows:

Activate Access the service known to the server with the resource *target*, and switch the service into an accepting state refreshing its application pool. In the event the

service is already active, do nothing.

Deactivate Access the service known to the server with the resource *target*, and switch the service into an rejecting state emptying its application pool of engines. In the event the service is already inactive, do nothing.

Status If the special character * is provided as the *target*, return the current status of the analytics server. Otherwise access the services known to the server with the resource *target*, and return a string representation of its current state.

Address If the special character * is provided as the *target*, return the the absolute path to the master configuration file, the service directory and the log directory. Otherwise access the services known to the server with the resource *target*, and return the absolute path of the service configuration file.

LS List the services known to the server along with their status irrespective of the *target*.

Shutdown Gracefully shutdown the server irrespective of the *target*.

POST messages indicate that the developer intends to provide new information to the server. With the resource set as */NEW* the server understands that the message body should be used as the initial variables for a new service. Name, status and directory location should be provided allowing the server to initialise the new service and append the service into the data store. The server itself does not handle the the OS operations for moving or creating service files but instead locates the files that should be present on the machine.

PUT message indicates that the developer intends to update or change information currently being hosted by the server. The *resource* indicates what service is to be edited. Provided in the message body, should be a absolute file path to a service configuration that should replace the current service information. The application pool is emptied and refreshed with the new information.

OPTIONS messages ask for the list of actions that can be performed on the management port.

TRACE messages ask the management port to ping the requests message body back to verify that the port is responding.

DELETE messages ask that the service running under the resource are removed from the server.

In the event that the management thread is required to make changes to the data store, the thread duplicates the stored variables and makes an atomic switch with the new information. This atomic action ensures that the analytics threads are safe and that no service is disrupted when a change is made. Services that are being removed do pose a risk, as they are meant to cause a disruption. The management thread in this instance removes the service from the global data store and sends its information to the Bin-man thread. The service persists at this stage so that requests threads that still hold references to the service with unfinished business with the application pool may continue to operate. New requests that attempt to use the service are unable to get a reference effectively removing the service from the server. File information for the service is destroyed immediately to protect against the potential risk of a new service with similar information being deployed into the formed resources location.

At all points the server is protected against potential errors caused by the developer or the operation of the script. Errors generated are illustrated in the responses status code and the

reason is communicated back the to developer for correction; any additionally information concerning an error is logged for later reference. When the management thread is satisfied that the request has been handled correctly will it return, it is important that the developer knows that an actions have definitely been performed. The thread then goes back to listening on the management port for new requests.

3.2.3 Communication - HTTP

Within the server, message objects are responsible for parsing the content received from a socket and writing a response to the socket. The active thread loops over the socket contents extracting information into a variably sized buffer. Initially the buffer is checked for the method, resource and protocol, failure to validate this information results in the request being rejected. Otherwise the thread continues to parse header information into the request object, logging the inputs and flagging up any headers that are unrecognised. Based on the requests method, the active thread then moves on to parsing the message body until no new content can be extracted from the socket.

The request object implements two separate time-out features during message parsing, one for the header and one for the body. In an effort to protect the server from malicious users, it is import that the server does not waste resources processing malformed requests. The header is only meant to pass context on the message content and is therefore expected to be short. Alternatively, the message body is unpredictable and potentially vast. Having separate time-out functions allows for greater efficiency on the sockets. After the request has been read, the response object is created to collect a buffer of information the thread may produce through the processing of the request. When the thread is satisfied, the response object connects to the socket and returns its content that has been converted into a message that follows the response standards set by the HTTP protocol.

3.2.4 Singleton Data Storage

In the effort to provide an increase in performance to models and scripts it is important for the Server to be as efficient as possible with the parsing and operating of its resources, the most notable resource being Engines. During the operation of the Server, multiple threads will be spawned to make fundamental changes to the internal state of the system, additionally the management thread and the bin-man thread will consistently need access to said resources. To facilitate this, the server has a main hub of information stored within a single class with global static access methods to allow threads and other classes, where the ability to retrieve and operate on the shared information is facilitated. This store is generated during the initialisation of the server from the master configuration file and it persists throughout the running of the server. It is currently imitating an immutable structure as it cannot entirely be refreshed or overridden during runtime, however threads will have the ability to make fundamental changes to the stores contents, a vital function for communicating information globally.

3.2.5 Application Pools

An application pool is a object that contains a collection of service specific engines, it handles the interactions between user requests and the engines themselves in a thread safe manner. They are responsible for extracting information from the service configuration files and the continuous initialisation of service engines throughout the lifetime of the analytics server. Application pools have set limits on the number of engines that they can contain and keep track of the number of engines currently in their pool. When requests arrive for the particular service, an engine is detached from the pool and given to the request thread to execute its payload. When the

engine is returned, the application pool assesses the state of the engine and determines if the engine is reusable. If the application pool is still in an active state and the limit has not already been reached, the service is appended to the pool, otherwise it is destroyed. The application pools does not impose a hard limit on the number of service specific engines however, in the event that the pool is exhausted new engines are generated to service the request from scratch. This behaviour is intended to be avoided as these services do not benefit from the enhanced performance of the partially initialised engines.

The engine that was generated *just in time* behaves no differently to any other engine and can continue to be returned and reused by the application pool. This is possible if the application pool is constantly servicing requests and therefore does not ever have a full pool of engines. This allows a service to potentially have a greater number of engines being operated in the system than originally specified in the service configuration. The application pool has effectively swelled to accommodate larger influxes of requests. Furthermore the pool is able to return to its normal size by removing from the system engines that no longer need to be kept alive. As demand drops the pool begins to fill back up and when full any engine that is returned is destroyed.

The application pool is also responsible for reducing its resource usage in the event that the service is deactivated or marked for removal. When a service is not active the application pool empties its contents to free its engines so that their resources can be used by the other machine processes.

3.2.6 Engines - Embedding Python and R

Services are analytical models/programs that accept a payload of information and perform a meaningful action. These services are restricted to the permissions of the user running the analytics server but have the ability to perform as they would in any other environment. The aim of the analytics server is to provide a means of utilising these services via a simplistic RESTful API with little regard to the underlying process of the service. This is achieved by wrapping the services in a C++ wrapper, I have named an Engine.

The generic engine is an encapsulation of a model that has dependencies and a main function, the engine initially partially executes the model to loads and runs a model's preprocessing sequence. The engine is then in a waiting/sleeping state until a requests payload is received and can be passed to the services main function. This is the largest cause of any decrease in time taken to resolve models as computation on the input can begin as soon as the input information is provided. The engine is then expected to produce some type of output that is stringified before being returned to the user. Failure to produce an output or producing an output that cannot be converted into a string (e.g. *complex object*) will result in the server assuming a computational error. Engines protect the server from potential malformed scripts and report back error codes accordingly. Engines that throw an error or are left in a state of *non successes* are destroyed after use and a new instance is made to replace them.

The signature of a models main function is strictly defined so it was beneficial for the user to provide functionality for the unpacking of a requests payload into set variables. Additionally, creating a system that is flexible enough to convert payloads to a standard form, with capacity to include other unpacking methods in the future, was under great consideration for the future development of the server. A requests content type is used by an engine to determine by what means the message payload is written so that it can invoke the main method accordingly. The server benefits from the potential reduction of wasted engines as incorrect/malformed request payloads can be identified before being passed to the service itself. However, this does not impact on the developers ability to perform their own validation on message payloads so this

feature can be ignored entirely to allow all kinds of payloads to be passed. In this event all payloads are passed as strings straight from the message body of the request.

The server is monitoring the state of the engine to confirm if a valid output can be formed and returns relevant information if an error does present itself. Information about the low level cause of an error is logged within the system. During the construction of the service environment (where the service is being run), the global variable *program_status* is generated with a value corresponding to the HTTP status code of success. This provides a developer with the ability to flag internal errors (i.e. non syntax errors) that have been handled by the service itself. A response output can still be formed but the status is used to define the HTTP response status to illustrate to the user that said error occurred.

All language engines inherit from the base class Engine which allows them to be manipulated throughout the server without checking the type specifically. Engine forces the language engines to implement three main functions: an Initialise function; that runs some form of pre-processing for that language, an Execute function; which takes a request payload to be run against the service's main function after its variables are unpacked, and a Reset function; that performs any language specific operations to ensure the engine is suitable to be used again.

Fundamental properties of a Engine are:

- Takes an input and produces a meaningful output, either a string or a side effect.
- Does not interact with other engines of the same service, or type.

Embedding Python has been a tricky task, but Python being a highly used statistical language, is a vital addition to the server in terms of usability. Initially I had hoped that embedding Python would be a case of initialising a Python interpreter within a engine and executing the contents of a service. This was not possible as the Python API is written for C rather than C++ and has no concept of instance specific interpreters. Instead Python initialises itself for the lifetime of the program causing engines operating Python scripts to interact by sharing variables and overwriting functions.

A promising solution at the time was to instead produce a Python module for the server that would act as a single point of interaction, handling and running each service itself in a controlled environment within Python. This idea meant that the server itself would not be handling or invoking scripts but rather would communicate to the model through a set interface. In pursuit of this solution, issues arose concerning the capability of communicating the results correctly with each engine. Either no service was truly preloaded, as running of the script occurred at the point of request or, outputs from scripts would have to be stored in files with a name determined from the request thread so that they could communicate properly. In either set up, the system failed to removed the computational overheads associated with the initial service, with the later incurring great amounts of IO overhead costs.

In the end, I was able to embed Python without a service specific cost by relying heavily on the threaded mechanics of the of the Python API. Each Python engine contains a threaded component that stores the individual Python state, this thread is then detached from the analytics thread. This thread contains a pointer to an $interface$ structure that is unique to the engine and allows for communication between them. The thread communicates its state during the set up sequence so that the engine can be sure that the thread was set up correctly and that the service is viable. It was important to keep the engine as self contained as possible as engines are to be used interchangeably throughout the server and could not be treated as special cases.

Embedding R was thankfully a lot more straight forward as there were a variety of packages written to help efficiently embed R within other programming languages. I decided to use the Rserve package for R that acts as a socket server (TCP/IP or local sockets) allowing commands to be interpreted within R in a separate workspace with its own environment parameters. This helps ensure that no engine interacts with the other engines within the system and provides a fast and effective way of running R scripts. The R server requires the R TCP server to be running as a separate process on the machine for it to make connections, which in fact benefits developers by providing more flexibility.

Analytics servers that are not going to be running services written in the R language are not required to start the R TCP server as no engine will attempt to connect to it. Furthermore, the machine does not even require the R language and its packages to be installed for the analytics server to operate. Analytics servers that do run R services can additionally be safely upgraded without potentially causing a disruption to the analytics server as the R services run remotely. The R TCP server can be deactivated and the language upgraded and then reconnected. Services during that time would fail to produce outputs as the engines would not be able to make any connections but most importantly it is by far easier than upgrading Python and can be done externally to the analytics server.

3.2.7 Bin Man Thread

The other side of resource management is the efficient and reliable destruction of unused or terminated resources. Services are essentially represented within the system by the application pool object, as they store the service specific information and their engines. These pools are accessed in a heavily threaded environment and it is not possible to determine the number of threads that at any one point own a reference to the application pool. Due to the fact that threads are unpredictable in nature and the services they are interacting with takes an unpredictable amount of time to compute, the process of removing an application pool has led me to program a thread tasked with application pool termination.

Initially when a service is scheduled for deletion, the application pool is switched into a dying state and emptied of service engines. The state change prevents new request threads from collecting engines, but does not stop the requests from interacting with the application pool. The services specific files, configuration file and script files, are then removed from the host machine. This does not affect engines that are being operated at that time as these scripts are already executed in the engines interpreter environment. Reference to the service in the master configuration file is removed and the application pool and relative information is removed from the singleton data structure and placed into the Bin-man's deletion queue. Threads that still have references to the application pool in the queue can continue to interact with it, however, new request threads will no longer be able to gain a reference as it will not exist globally. After a substantial amount of time has passed, the application pool is deleted. The pool would be in a low resource state as soon as it was marked for deletion so it does not waste an unacceptable amount of resources during this time. The time is necessary to ensure that all threads that at some point owned a reference to the Application pool have expired.

The Bin-man thread sleeps before deleting a given application to ensure that enough time has been given to the request threads to expire. In the event that no services have been earmarked for deletion, the bin-man sleeps, checking periodically to see if any work is to be done.

3.3 Testing and Validation

The server was constructed in stages as to minimise the potential sources of bugs and errors. Being confident at each stage that the current system worked as intended, I could incorporate different components and perform integration tests easily and locate any errors quickly. Although, as the scope of the project expanded and existing components were edited, I decided to switch my testing method to system testing which I was able to carry out via a Python module, I created.

HTTP messages - Appendix item ??

Valid Request	Returns successfully from server.
Garbage Requests	Sending requests that do not adhere to the HTTP standard results in the server returning a *400 - bad request*.
Malformed headers	Sending requests that contain unrecognised headers or malformed header content, is generally accepted, as only a set subset of headers is used by the server. Headers that are required are checked during their processing and the server returns a *400 - bad request* if found to be malformed.
Gigantic requests	Sending requests that are valid but contain a repeated set of message headers, result in the server rejecting the connection with a *408 - request time-out*.

Server operation

Get Requests	For services providing a dashboard, the service correctly responses successfully with the HTML page. For servers that redirect the user the server returns *303 - see other* with the location header set.
Post Requests	Requests with empty message bodies returning with success, if the service produces an output. Requests that do not abide by the main signature of the service are returned with *400 - bad request*. Correct messages return with success and service output.
Malformed configuration	Supplying a configuration file that contains syntax error causes the start-up process to abort. Information about where the syntax error was found is printed to the terminal.
Unauthorised Access	Requests without providing a username or access-token that are not valid are rejected with a *403 - forbidden*. Requests that supply valid information but attempt to access actions they are not permitted too equally results in a *403 - forbidden*.

Services actions

Activating/Deactivating Activating/Deactivating a server in that is in an initial state results in a successful response. This is a desired feature as it reassures the developer that the service is accepting/rejecting as they intend. Activating/Deactivating the service from another state results in a successful response.

Malformed Updating Updating a service with information data will result in a deactivation of the service and a *500 - internal server error* being produced. Information concerning the cause of the issue is returned as the message body and logged on the server.

Activating Malformed Attempting to activate a service that has been deactivated as a protective measure results in the service denying action with a *403 - Forbidden* response.

Deleting no content The server attempts to delete the services files when marked for deletion. Failure to do so is logged on the server but a successful response is still returned. Access to the machine should already be restricted to developers, so the movement of files is acceptable and it is assumed the developer would already have knowledge of the file transfer.

Resource spam Spamming a service with message results, do not incur a drop in availability. Time taken does increase as *JIT* responses begin to occur. Tested with small pool-sizes and large server queues.

Malformed config During the initialisation of the server, the server aborts and prints the relative information about which configuration file and what line the error arose, to the terminal. During the run-time of the server, a new service added with a malformed config results in the server rejecting with a *400 - bad request*.

Malformed script During initialisation of the server, the server aborts and prints relevant information about the cause to the terminal and exits. During the run-time of the server, a new service added with a malformed script results in the server rejecting with a *400 - bad request*.

Malicious scripts

Variables For both Python and R, scripts that purposely edits the *program_status* variable to a non integer value during its main execution is handled by the server. The service is valid and the output it creates is still returned to the user however the server interprets the signal as a script error. The response is a *500 - internal service error* is with the resource and cause being logged.

Outputs Python scripts that produce outputs that cannot be converted to a string with the *str*(), *_str_()* are not returned and the response given is *503 - Service unavailable*. R scripts that do not produce a printable output does not currently flag any issue, the service responds with nothing.

3.4 Evaluation

A critical aspect of the server is that it provides an enhancement to the performance of a service, allowing solutions to be realised sooner, so that informed decisions can happen in real-time. The results shown below are from a series of tests run on a single machine. At the time of testing, the server was hosting all scripts simultaneously where the terminal scripts were run individually with only the natural OS processes running. This is done to take into account the fact that the server is meant to host multiple services at any one time, the impact of this is important to factor into the measurement of performance enhancement. I aimed to illustrate the kinds of performance increases for services that were pre-processing both light and heavy.

Python Script	T1	T2	T3	T4	T5	Avg.
Factorising Terminal	11.16443	11.46656	11.19199	11.21521	11.27478	11.26259
Factorising Server	10.78970	9.53602	11.29180	9.54912	11.06880	10.44709
DB lookup Terminal	4.11782	3.94536	3.93578	3.94562	4.00450	3.98982
DB lookup Server	0.03314	0.03230	0.00980	0.03837	0.03214	0.02915

Figure 2: Execution time of python scripts in seconds.

For Python's Factorising script, all execution was done at the point of request and the server implementation was able to produce almost consistently better speeds than executing the service via the terminal. What is noteworthy is that the terminal has less variance in its time values over the server. This may be as a results of the server running multiple iterations of multiple services, and the overhead incurred by switching the thread request in and out of computation is causing fluctuations in the time values for the service. Despite this, it is clear to see the service is subject to a slight performance improvement over the conventional method. [B.1]

For Python's Database lookup script, the largest aspect of computation was prior to executing the main function, which has given the server a massive advantage over the script. As shown, the server is able to execute a request averagely over 130x quicker. [B.2]

R Script	T1	T2	T3	T4	T5	Avg.
Factorising Terminal	59.71423	57.82161	60.21632	60.00577	58.24187	59.59996
Factorising Server	59.88665	59.91267	60.13381	60.41902	59.97111	60.06463
Setup Terminal	2.58650	2.58974	2.56341	2.61391	2.62315	2.59534
Setup Server	0.08197	0.08002	0.08263	0.08254	0.08345	0.08212

Figure 3: Execution time of R scripts in seconds.

For R's factoring script, it would seem that the terminal implementation of the script is slightly quicker at producing a result. This is to be expected, the server not only runs the script but is responsible for navigating the analytics server and communicating with the R TCP server for the script execution. What is surprising is the appearance of greater variance between the values of the terminal over the server. One would expect that the terminal would be give greater consistently.

For R's Setup Script, the server clearly out performs the script as its preprocessing aspect removes the need for the time consuming computation aspect of the script.

Overall all traditional models have aspects of preprocessing, even the importing of modules has its respective overhead cost. The results illustrate that in the worst case scenario the server

operates equivalently to the standard form of model execution, and in the best case permits a dramatic reduction in the time taken for a result to be produced, evidence that confirms that the project has successfully achieved its objective to perform service enhancement.

Having passed all the tests that I have set, the product is proven to be a robust and viable solution to the problem of analytical model hosting. By utilising the HTTP protocol as its method of communication, users benefit from being able to integrate any model into any application with minimal effort. It allows users and stakeholders the ability to use services without having to be informed about the underlying service with the confidence that the service will continue to run correctly in the future. Developers are able to update and improve services without affecting accessibility and can focus their time on more import things. The business benefits from being able to quickly implement an analytical solution with real performance enhancement to solve time critical business problems.

However, there are aspects of the Analytics Server that I feel could be improved upon in future iterations, and these are aspects that are left in my backlog. Request payloads are only minimally validated before being passed to a service, depending on the content-type of the request. Incorrect type signatures for a service's main function are still queried against their source, and it is expected to be handled by the service itself. If the server had access to the main functions prototype or descriptive information about its structure, the server can reject an invalid request earlier without having to expend an engine resource.

Generally, better management of engine structures could yield improved resource efficiency. Application pools currently maintain a constant volume of service engines regardless of dwindling demand. Volume can swell in the event the service receives multiple requests but inversely does not shrink in the event demand falls. By lowering the pool size to a fraction of its original service defined size when the service has not been interacted with reduces the resource intensive state of the application pool. This would have to be governed by some sort of timer, or a genetic algorithm that generates some form of service specific pool size to time structure, trained on recorded information of service use. The system would then be responsible for shifting the application pool back in the event demands rises. The server is likely to benefit from a system that frees unnecessary engines and refreshes unused or stale engines.

Python Engines currently terminate their threaded interpreter component after executing a request payload. The server suffers a slight overhead cost re-initialising the service engine irrespective of the state of the service and is akin to generating an entirely new engine. Changes to ensure the persistence of valid engines would help efficient use of the resources of the server.

The Bin-man thread is very resource light but its performs its actions currently on a arbitrarily set time out interval. Implementing a system that may correctly identify a more appropriate value could reduce the inefficiency of having an application pool linger, although as previously mentioned this inefficiency is very low. Alternatively, providing the developer the ability to determine what value may be more efficient and allows more granular control.

Features I would like to include in future iterations of the server would aim to improve accessibility to server statistics and provide more control. Information about the machine hardware, CPU and memory usage of the server and services, and potentially information regarding competing system processes could be valuable information for the developer. This informations is already available to the developer but providing a method to access this would be appreciated.

Through testing and implementation I discovered that the server could benefit from implement-

ing functionality to define a global soft pool size limit, a limit that overwrites the pool size of services. The idea being that servers that are intended to be used as a development suite do not require large pools of engines. This feature combined with a global hard limit that limits a pools ability to generate *JIT* engines, allow developers to exert more control.

A concern is that the server does not have internal authentication for services natively. I had initially considered forming a authentication system based on local IP addresses or some form of password, however the extension was just not possible in the time frame. Services may still be protected by restricting access with the use of authentication services native to their languages but this requires the service resource to be used to reject a request which is a waste of a resource. Developers may appreciate the ability to set up authentication on the analytics port or a servicer specifically.

4 Management Console

The Management Console is a complimentary product designed to operate and manage multiple Analytics Servers. The intuitive interface provides a concrete development environment that handles the problem of distributing services. Additionally it provides features that are designed to help implement development stages to follow typical development processes, so a firm may construct its personal structure within the console.

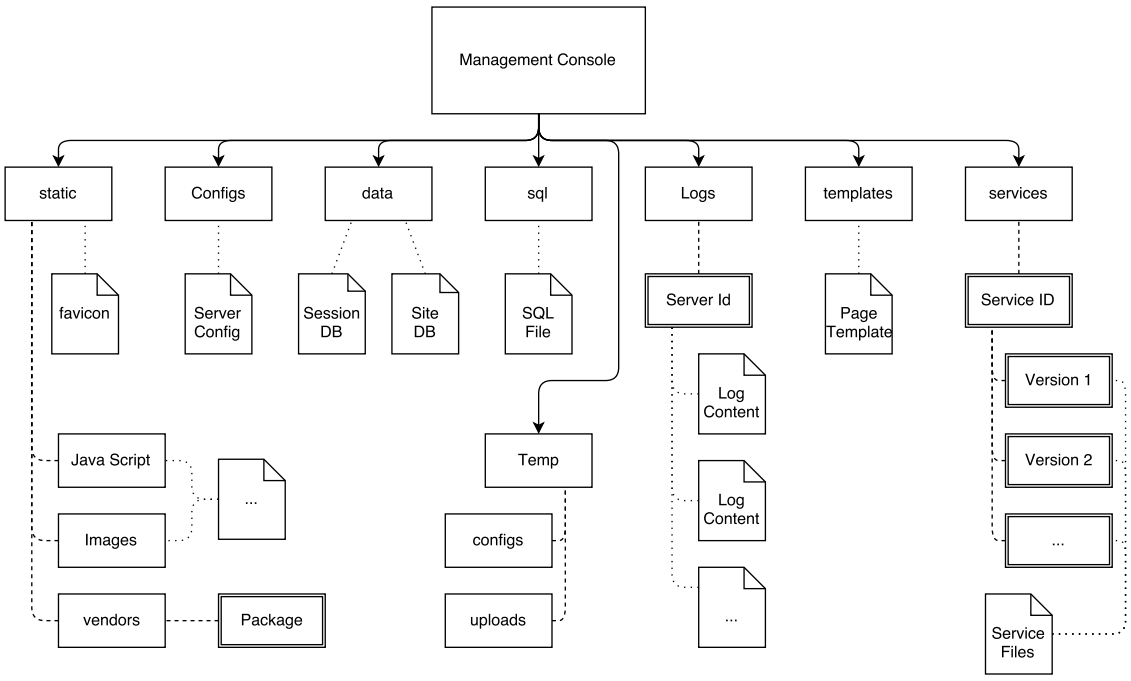


Figure 4: A illustration of the file structure for the Management Console.

The product comes in the form of a modular web-server as it is meant to act primarily as a base management option and be improved upon. As the requirements of the system shall differ in each firm, the console has each section split so that components may be adapted to better suit its purpose, it illustrates what is possible and implements the basic functionality necessary.

4.1 Original Specification

Along with the specification for the analytics server, I identified the criteria for the management console. The consoles focus is much more oriented towards facilitating the use of the analyitcs

server, however it is important that the design and visual aspects of the management console allow developers to quickly understand the current state of a server/service and the possible actions that they can take.

Create and manage Server Environments: The management console needs to implement environment structures to mimic the typical development stages found within a waterfall development process. Environments are collections of servers and facilitates the deployment and management of services.

Environments have been correctly implemented with four main types, each time is a typical stage in a waterfall development process and they provide a rigid structure for developers to build on to. Through the use of development paths, and the functionalities that have been implemented, to add, edit and delete environments, developers could additionally mimic other stages.

Create and manage Development Paths: The management console needs to implement development paths to allow services to undergo different life cycle stages, and allow developers the ability to manage specifically what services are developed on which environments. The paths are formed from a collection of ordered environments.

The console implements development paths that allow developers to drag and drop environments into set orders for services to traverse. Paths also ensure that a service must transition through the main environment types, *development* to *live*.

Create and manage Individual Server Implementations: The management console is required to facilitate the management and operation of a Analytics Server. The console should provide functionality to add, edit and delete a server implementation, and a page to view server information.

The console implements server structures that can interact with analytics server. The console handles the deployment of service files to target analytics machines, and the collection of logs and configuration files.

Deploy, Update and Delete Services and their versions upon Environments: The console is required to provide functionality to deploy a service to an environment, and subsequently all servers within an environment. The console should be able to handle multiple versions of a specific service and handling the interaction of updating an environment version. The removal of a service from an environment and the console should be handled gracefully.

The console ensures that services are enrolled on a set path at their conception, and in doing so ensures that it correctly handles the deployment of services to environments within the path. The console also handles any change that may occur to either environments of the path, or the path of the service. The console removes and deploys service versions whenever it is necessary.

Manage Services remotely with performance visualizations: The server would benefit from having pages dedicated to performance visuals and statistical data of the operation of services that are deployed on environments. This facilitates developers in their management of the services allowing them to quickly get an understanding of the operation of services and environments.

There are visual representations of the operation of services on both the environment page and the service group page. General visualizations of the operation of services can also be found on

the main dashboard. My intention was to have have distinct pages for service versions that were accepting live traffic, however, I was not been able to fully implement this feature although the current setup of the console would allow for its development.

Manage and organise developers into groups with set permissions: The console should provide functionality to create users, and set their personal permissions on the console. A developer should also be able to separate users into groups with group level permissions.

I have been able to implement users accounts and the console does correctly handle the generation of new users, but this has been accompanied with a few restrictions and I have not been able to implement user permissions. The underlying architecture does lend itself nicely to future inclusion of this aspect but as of yet, all users have access to all console commands. The Analytics Servers will however, reject actions that are not permitted by a user.

Manage the packages and libraries used by services on Environments: The console should either, provide the user with a panel to edit the packages and libraries for particular languages within an environment, or handle the installation of missing packages libraries internally.

I have not been able to implement a method of managing language packages. Services that are deployed onto servers without the necessary dependencies do flag this issue, and the relevant information is displayed to the user to resolve externally.

4.2 Design and Implementation

The management console is built on top of two SQLite databases that store state of the system and connected servers, and session and site information. The primary function of the console is to manipulate this data to handling the deployment and management of services on a grand scale, with a abstracted interface.

I build the console using the web.py python framework as it is simplistic and powerful in its ability to instantiate a functioning HTTP websever.[10] The framework is in fact rather popular as it powers a handful of site to data and was initially exclusively running 'reddit.com'. The framework has allowed me to set specific Python classes as the target for requests and in turn, this has meant that I have been able to separate out the functionality of the site into set modules. This has aided in my development of the console and it benefits customers who might intend to change the functionality of a page/component, or even replace the functionality entirely. I believe it was the right decision as webserver platform from which to build.[6]

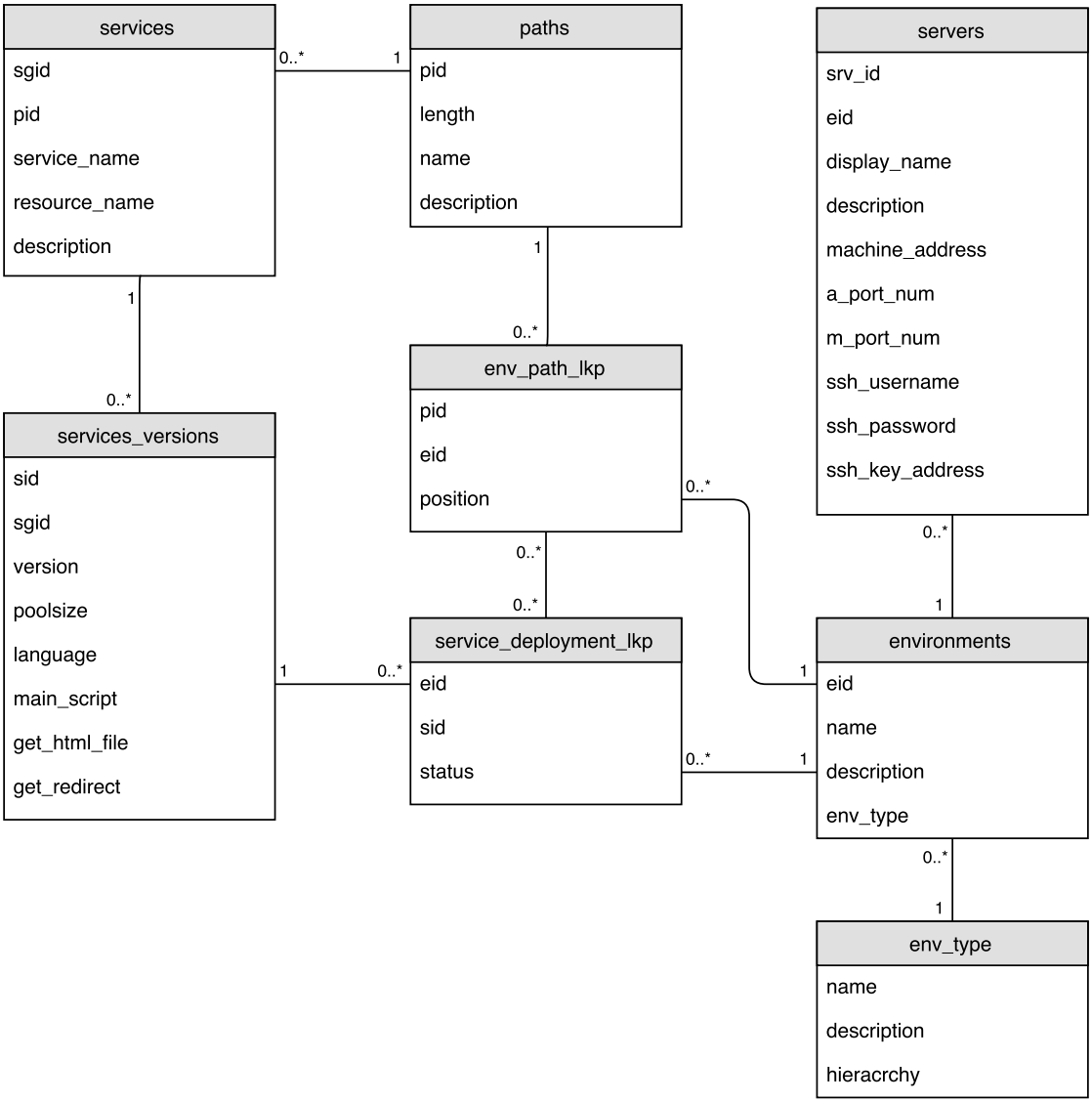


Figure 5: A basic entity relational diagram to illustrate the underlying structure of the database holding the service and server information.

With the use of the Python SQLite database package, I have been able to generate and interact with a number of relational databases in-line within the console. The ability to do so without requiring a database management system has meant that the console has been able to keep its light weight profile, and be as self contained as necessary.

4.2.1 Access

The console itself correctly handles the requirement of permitting only authorised users to access the core console. Before handling of any request, the session information on the request is extracted and used to identify whether or not the user is known to the console. In the event that the user is either logged out, locked or unknown, the console redirects them accordingly. When a user does attempt to log in, the supplied username is used to query the table of users to find relevant information. Passwords are stored on the console in a hashed form, along with a uniquely generated salt. These are extracted if a user by that name exists and compared with the salted and hashed version of the supplied password. At which point the user gains access to the console and their information is stored along with their session key in the database to allow them continual access to the site while their browser remains open. Developers cannot

create or request an account through the console if they do not already have an account, as to do so would be a security issue.

In future iterations of the console, I intend to allow a developer to generate a uniquely named hidden page in the event they would like to add a user. This link could then be sent to the new developer for them to input their information personally. These changes would accompany other system user updates I have in mind.

4.2.2 Environments

An Environment is the name given to a collection of servers that have a similar intended use and parameters. They are intended to form a single entity from the prospective of a user, and work together to produce meaningful answers and speed ups. Environment parameters define the situation servers are operating in and can be used to simulate a whole host of potential scenarios to allow for a better understanding of the capabilities of the system. Environments offer a rigid structure for the implementation of analytical services, and they come in four types.

DEV Hosting services that are still in development producing potentially unintended side affects. Intended this to be used by developers themselves, with no live traffic use.

TEST Hosting services that are fit for purpose but have not been validated, potentially containing undesired bugs and affects. Intended to be used by a test group of users, or small volumes of live traffic.

STAGE Hosting services that have satisfied all development tests, services should be complete. The stage environment is meant to bridge the gap between test environments and Live, a limited amount of live traffic can be directed towards the service until confidence is built and can be officially moved to live.

LIVE Hosting and operate reliable and complete services taking on high amounts of traffic.

Developers may create a new environment at any point and the environment can be any of the types mentioned. An environment does not necessary have to contain any server implementation, however, it would also not provide any function if it was left in that state. Developers are free to add and removed servers from an environment by navigating through the environment details page. When a server is added or removed from an environment, the console deploys/removes the highest versions listed as running on the environment. Failure to do so is flagged to the user so that the cause of the issue can be resolved.

Environments cannot change their type whilst a member of a development path, when free the environment does not host any services as it has been removed during the process. Environments can then be edited and reintroduced into development paths seamlessly.

4.2.3 Development paths

Development Paths are a sequence of environments, in type order, that identify the projected environment a service will become a member of during its production. This helps implement a waterfall-esque development style, as services seemingly move from stage to stage when developers are satisfied the service has passed certain requirements. Additionally, it communicates the expected importance, scope and intended recipient of a project by identifying what environments it is expected to transition through.

Initially Paths were designed to be followed by individual versions of services, allowing for service versions to branch and coexist on different environments and for development of both to

happen simultaneously. However, it became apparent that this behaviour was not beneficial to either the developer nor the users and was already implemented in the system. Additionally, it caused tremendous amounts of complexity when developing the console, as determining which version should be run on a particular environment when changes occurred, was huge. A service should not have its means of interaction nor expected output changed from version to version, which led to the conclusion that having two versions under development for a particular service would be misguided. This is because both versions would be aiming to fulfill the same need. They should not be separated or placed in conflict since if they do fundamentally alter their method of interaction they are not longer versions of the same service, and fundamentally, not interchangeable. Instead I decided to implement the paths as a service group feature which retains all services on the same path. Branching of a service can be handled by creating another service with the same content. A secondary benefit would be the fact that services no longer compete with each other as they would require different resources and therefore could co-exist on the same path.

4.2.4 Servers

The server pages provide a basic overview of the information the console has of the server. At this current point, the analytics server does not generate statistical information about its operation. The console however, can be used to determine if the server is accepting or rejecting requests.

The server may only be a member of one environment and is meant to host all services that are deployed to that environment. Upon set up, a connection is formed between the console and the Analytics Server to ensure that the server is valid. When confirmed the server provides the absolute file paths to the master configuration file and the log files. The console then opens an SFTP connection with the server and collects all the files for reference and statistical data on the requests and and operation of the server. It is my intention to implement a method of viewing such log files easily in the console itself in a future iteration.

4.2.5 Services

The console acts as a repository for services and their versions. All service files are stored locally and the information used to form the service configuration file is stored within the database. Services are added to a server by opening a SSH connection with the target and moving the service files across into the service directory specified by the server. A configuration file is generated at the point of transfer specific for the machine and is placed in the execution folder of the service. The management console then messages the server with its resource name, status and configuration location so that the server can begin to initialise the service. A service group may have zero to many service versions and each is viewable in the group details page, with visualisations on operations on the environments.

Services follow development paths which outline the development stages the service is to transition through. After the service has successfully met the requirements of an environment (as dictated by the developer) the service may be promoted to the next stage. Upon reaching an environment of type *LIVE*, the service is given a link on the site navigation bar. Initially Paths were designed to be followed by individual service versions rather than the service groups, however it became apparent that this behaviour was not beneficial to either the developer or the user. The original idea was that service versions could branch develop and coexist on different environments simultaneously, allowing more flexibility in the development and implementation of services. This was a misguided consideration. A service should not have its means of interaction or its expected output changed from version to version, doing so would break external

applications that relied on the service. So versions are interchangeable and separating them, or developing on two branches would be an error. Fortunately the console already provided functionality to facilitate branching, the developers could simply create a new service that follows its own path.

When creating a new service, a service group is created along with its first version. Version files are initially uploaded into a temp folder under the developers username, and are moved in to permanent position when the service form is finalised. The version is then immediately promoted to the first environment of the development path selected. Failure to push the servers into the environment results in the causes being displayed to the developer, but this does not stop the console from accepting the files. The version exists on the console without any deployment information, and may be promoted onto the first environment whenever the developer is confident that the errors have been resolved. Idealistically the version is successfully deployed and can begin accepting requests as soon as it is uploaded.

Editing service information is largely restricted as its content is vital to the operation of its versions that may be running. The resource name for example cannot be changed as it forms a part of the address of the service on every server, changing it could cause linked applications and other systems to fail. Any change like this should instead be handled by the generation of a new service with the desired resource name. This would allow for a safe transition from the old resource to the new, removing the older service when confident no application still relies upon it. Changing the path a service belongs too, results in the system calculating the services version information for all environments related to the change. The system removes the service from environments that are no longer a member of the services path and uploads the service version to the new environments if the environment is positioned beneath a environment with a version running.

The editing of service versions is currently deprecated as its use case is underwhelming. Initially I had planned that the editing of version information could occur at any stage of development as adjustments and tweaks are common place, except it became ever more evident that this should not be possible. Changes to the language or dashboard require potentially fundamental changes to the files of the services that would suddenly be untested and unchecked. These changes would have to occur in an environment that was not accepting legitimate requests and additionally it would be a waste to open the service to testing users if the service did not even run as a result of the changes. I concluded that the changes should only be allowed on development environments. This mixed with the idea that changes regardless of how small, would have to result in the redistribution of service files, indicated that it there was little benefit in this, over simply creating a new version with the updated file; thus reducing the risk of potential errors. I intend to fix this feature in a future story.

4.2.6 Database handling

Database interactions have been extracted from the console modules into a separate module referred to as the DatabaseHandler. The DBH provides a set of functions to be able to interact with a SQLite database which holds all the information for the console, but its intention is to be replaceable with any SQL accepting any database setup a firm requires. SQL commands are stored within files that are specific for each section and are read in by the server at its initialisation. The SQL files were created to be used by a specific component, so if a component was changed by a developer they would be able to quickly edit the relevant SQL commands .

4.2.7 Server Module

Analytics Server interactions are handled by a separate module named AnalyticsServerInterface. It provides the functionality necessary to communicate with an analytics server and handles the process of opening up *SSH* and *SFTP* connections to interact with files. Once again the module is meant to be operational individually so changes to the aspects of the site do not impact on the ability of the analytics module.

4.2.8 Logging module

The service request information is extracted from the log files collected from the Analytics Servers the console connects too. The logging module provides an abstracted parsing of the log files with multiple convenient functions to manipulate the information that is to be extracted.

4.3 Testing and Validation

Throughout the development of the console, I tested the UI thoroughly to ensure that it acted as expected and that the actions and methods it was meant to implement produced meaningful interaction in the back end.

Environments

Adding/Removing	For connected servers: Services were correctly hosted and deleted, console redirected to environment details as expected with success message. For non connected servers: Services were rejected, server action still applied. Cause for issue was displayed.
Deleting	Path lengths correctly shortened, information about failed removal of services were displayed with cause. Redirected to Manage environments page.
Type change	Server rejects change with specific warning, redirected to form to correct.

Servers

Invalid Editing	Rejected, error message displays reason for rejection.
Swiching Environments	Server correctly deletes services of previous environment and uploads the services of the new environment. Deployment failures are displayed with their cause.
Deleteing	Server information is removed correctly. No reference to server stored.

Paths

Adding	Path information and length edited correctly. Adding servers to the path correctly deploys to environments positioned lower than environments. Correctly deploys versions, although previous versions are not recorded as being members of the environment. If top version removed or demoted server will not roll back.
Removing	Path information and length edited correctly. Removes services correctly from the environments.
Deleting	Console rejects delete if services are still enrolled. Deletes successfully otherwise. Redirected to the Manage paths page.

Services

Promotion	New Environment: Console is correctly deployed to the environment. New Version: Console correctly replaces the version on the environment. End of path: Console correctly rejects action. Not connected: Console rejects, no action is performed.
Demotion	Only Version: The server correctly removes the service from the environment. Top version: The console correctly rolls back the service version on the environment. Beginning of path: The console correctly deletes the service information and files. Not connected: Console rejects, no action is performed.
Edit Path	Correctly moved to the new path environments, deploying to environments that are behind their current position in the path. Service is correctly removed from removed environments. Not connected: Rejection origins and causes displayed, update continues.
Deleting	Service Group: Correctly removes from all environments with deployments. Version: Correctly removes from environments where running, rolls back if applicable. Not connected: Both reject, no action is performed.

It is a desired aspect of the server to assume none database actions are a success (except when all targets reject), as the console itself cannot rectify those issues. Storing the new information and displaying the errors and causes allows the developer to resolve the issues and have the management console in a state to continue.

4.4 Evaluation

Critically the console does fulfill most of the essential criteria that would determine its fitness for purpose. The console does provide the overall sense of a development environment and it does facilitate the distribution of services to a network of any size in a extremely simply interface. It provides all the necessary functionality to manage the operation of services, and servers. The console additionally allows for the management of servers in bulk via the environment structures. Although, the idea of the console is to be built upon for a tailored experience, the console could benefit from having the interface utilising more of its space for information about the servers and services it is hosting. Having a base that displays more of the potential features of the system would likely lead to a better understand of the potential of the console

and server from the prospective of the developer. Additionally, developers would find it easier to construct and build their own environment if these parts were already implemented.

There are a few things I would have like to have improved, such as the SQL interface. The SQL command management works as a concept, but it removes far too much information for the development of the piece. In-line the commands are referenced by an index rather than a description of their action, which makes it confusing as to what the command achieves. Having the SQL system changes to be action orientated, rather than component orientated with names rather than indexes, could be positive for both continual development of the console and developer changes within a company. But this does not detract from the capability the setup provides. Users are able to switch different database models in and out via the adaption of a single module. For the addition of service information, the console is designed in a manner that can handle reading, and communicating multi-layered service files, but specifically uploading files to the console it is not. Files that are uploaded within folders are extracted from the folders and placed on the same level. This is an unnecessary burden on the developer and can render services unusable if the developer does not give them attention.

The user aspect of the console is workable and does protect the console from unauthorised individuals but it is missing fundamental aspects such as groups and permissions that would make it fully meet the criteria. The backbone infrastructure for personal permissions is present but has not been implemented as throughout development, the kinds of actions and permission levels associated to them were constantly in flux, because of this I deliberately did not attempt to set these in stone until confident the console and server were more consistent.

Overall I am very happy with the progress I have been able to make on the console and I believe that any user could find the interface easy to navigate and aesthetically pleasing. If this was not the case, they should at least be content that it can be customised to their liking.

5 Use Cases

I believe with the functionality that the server currently provides, that companies can improve the effectiveness of the knowledge they generate through their analytical solutions. In the following sections I briefly highlight sections of business that can benefit from the functionality the server naturally brings.

5.1 Fraud Detection

Insurance companies are constant targets for fraudulent claims and with the expansion of the Internet of things, so to has the number of frauds and methods of attack. The far-reaching affects of such fraudulent claims effect not only the financial stability of the company but negatively impact upon their loyal customer base who are then subject to increases in premiums. In the competitive market insurance is situated, this can have a significant impact on the reputation and trust investors and clients have in a company.

However, with the power of data analytics, insurers are able to identify claims and clients that are likely to be illegitimate. Machine techniques such as genetic algorithms and neural networks, are employed to generate classifiers from the vast reservoir of historic insurance claims information. These classifiers once trained can be applied in real-time to determine if a claim or client may be fraudulent, and in doing so associate some value of risk. Classifiers in their basic form, are instructions to for the generation of a network structure with values for nodes, weights and input output vectors. When a classifier is to be used, this metadata is interpreted

into a program instance and Inputs can quickly be run through the graph to produce an output. Within insurance, fraud classifiers are formed by evaluating the information of known fraudulent claims against legitimate claims. The learned distinction can then be applied to new claim information to assess the likelihood the claim is legitimate, any hidden underlying indicators of fraud are learned by the classifier.

The analytics server offers the perfect means of implementing such a classifier for a number of applications simultaneously. An insurers website that allows users to fill in claims information on-line can seamlessly integrate the classifier into its operation via a simple HTTP call to an analytics server. The information can be processed in any one of the statistical languages the insurer team would decided create the classifier in, and the probability of fraud can be returned to the page for inform their client if the claim is has been accepted. Additionally the wbsite might make another call to another analytics server services that checks a databases of known frauds, potentially providing a definitive answer. The services used could also be employed without requiring any adaptation, into desktop applications used by brokers handling claims personally in the insures high street outlets.

5.2 Zurich Insurance

During the initial stages of the project, I got in touch with an developer at Zurich Insurance in Bratislava. From our discussions they mentioned a model they had worked on that was meant for their finance department. Vast amounts of sales information was being collected and stored for the European branch of Zurich on a distributed database network that split collections into weekly installments. The finance team required some additional information to be extract to aid the department in deciding their direction that week. The developer's team was able to deliver this service to the department but, the finance team was not confident that they would be able to operate it correctly. Subsequently they insisted that the team handle the services operation, which resulted in the developer taking responsibility for the use of the model.

The dynamic that followed was extremely inefficient. The department head would get in contact with the developer each week via e-mail and outline what they needed for that given week. The developer would then have to run their model and handle the output, emailing the insight back to the department head when it was completed. This module took supposedly 30 minutes in computational time to run and it was only deployed on the developers machine, which rendering it unusable for other more important tasks.

This formed part of my motivation for the project as this process seemed inefficient and problematic whilst being totally solvable. Hosting the service on the analytics server the finance team would be able to interact with the service in a manner as simple as creating an e-mail. The developer would not have to waste their time operating the model and the finance team would be confident the service was being maintained appropriately. They would know that changes by the developers would not affect availability and they would be not need to be concerned with updates the development team would make. Additionally the performance of the model would be dramatically improved as dependencies could be load beforehand (especially with the smart engine managment I intend to implement) from the data bank and it would be run on a dedicated analytics server machine who's hardware is likely significantly more capable.

6 Conclusion

I set out to create two pieces of complimentary software that would enable companies to generate usable analytical insight in a number of scenarios. With an aesthetically pleasing management

console that can operate a number of services over a distributed network, on analytic servers that can provide genuine performance enhancements while abstracting away the complex nature of the models it would be running. As these as my objectives, I have successfully achieved the initial intention of my project and have developed a system that surpasses my original expectation. Both products can continue to adapt to better fit its purpose and their design allows for continuous development to ensure the product will stay viable and successful.

Over the course of the project I have gained a huge insight into the practicality of undertaking a more wholesome project that focuses on creating distributed systems utilising the underlying network protocols of the Internet. In particular I have developed a deep appreciation for the challenges of asset management via the standard forms of communication. I have benefited significantly from experiencing the breath of scope and complexity an enterprise level piece of software entails. Though my scope expanded greatly in the mist of the project, I was able to ensure that I met the project criteria by strictly following my development methodology. Having continuous self evaluations has allowed me to determine the precise stories and functionality that was necessary, as well as functionality that was consequential.

This being said, I did unintentionally lead myself in the wrong direction at times by attempting to work on both products simultaneously. My original train of thought was that development of the one would inform me of the requirements of the other, which is a wishful thinking approach to development. In turn, I spent time rewriting sections of the management console to correct misalignments in communication with the analytics server after necessary changes were made. Furthermore major functionality on the management console spent more time than I would have liked being partially implemented as the server had not yet implemented the necessary aspects suitable for testing. This led to components being built around stub classes providing canned responses, which adhered to a contract that was not well defined or implementable on the server.

A fundamental aspect of Agile development is the interaction a team is meant to have with the client. Through their interactions the team may gain a better understanding of the requirements of the client and ensure that development is always focused on desired functionality. In retrospect, I should have attempted to reach out to more local businesses and enterprises that could use and benefit from my products, to form a better understanding for the kinds of concerns they may experience and what they individually look for in a product.

But these are small misnomers and they do not detract from how pleased I am with the overall platform. I have been able to fully implement the criteria for the analytics server and provide meaningful improvements to the performance of services. The manner of communication is simplistic and efficient, requiring only the bare minimums of the HTTP standard to connect any application with a service. The means of interacting with a service is intuitive for any loosely and strongly typed languages and the server is robust in its handling of services, configuration files and request information. Jointly the management console provides useful functionality to develop and handle the distribution of services. The console installs structures to mimic the development process of typical enterprises in a manner that I feel is not intrusive but extremely helpful in their potential to help organise and manage systems. The flexibility of the console facilitates its initial objective of simplifying a developers interaction with a group of servers.

7 References

[1] Tiobe index for november 2016. http://www.tiobe.com/tiobe-index/. Accessed: 15/11/2016.

[2] R. Dahl. Node. js from jsconf."nginx vs. apache,", 2009.

[3] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol–http/1.1. Technical report, 1999.

[4] F. Huang. Web technologies for the internet of things. *Mather's Thesis, Degree Program of Computer Science and Engineering, School of Science, Aalto University*, 2013.

[5] D. Jelovic. Why java will always be slower than c++. *Website, http://www. jelovic. com/articles/why_java_is_slow. htm*, 2008.

[6] T. P. S. Library. Db api 2.0 interface for sqlite databases. *https://docs.python.org/2/library/sqlite3.html*, Accessed 3/5/2017.

[7] M. A. Lindner. libconfig. *http://www.hyperrealm.com*, Accessed 3/5/2017.

[8] L. Prechelt. An empirical comparison of seven programming languages. *Computer*, 33(10):23–29, 2000.

[9] W. Reese. Nginx: the high-performance web server and reverse proxy. *Linux Journal*, 2008(173):2, 2008.

[10] A. Swartz. Welcome to web.py! *http://webpy.org/*, Accessed 3/5/2017.

[11] M. Thornburgh. Adobe's secure real-time media flow protocol. 2013.

[12] S. Tilkov and S. Vinoski. Node. js: Using javascript to build high-performance network programs. *IEEE Internet Computing*, 14(6):80, 2010.

Appendices

A Configuration Files

A.1 Master Analytics Configuration File Example

```
Logging :
{
    root_directory = "/var/logs/AnalyticsServer/";
    detail_level = "DEBUG3";
};

Analytic_Settings :
{
    bufferSize = 256;
    queueSize = 100;
    portNum = 7980;
    protocol = "HTTP";
    service_info :
    {
        root_directory = "/var/local/AnalyticsServices/";
        services = (
            {
                 resource = "/range";
                 status = "ACTIVE";
                 configPath = "/var/configs/range.cfg";
            },
            {
                 resource = "/bank";
                 status = "ACTIVE";
                 configPath = "/var/configs/z7.cfg";
             });
      };
};

Management_Settings :
{
    bufferSize = 256;
    portNum = 5665;
    queueSize = 5;
    usernames = ( "USERNAME", "USERNAME" );
    permission_keys :
    {
        admin = "KEY";
        level_one = "KEY";
        level_two = "KEY";
    };
};
```

A.2 Service Configuration File Example

```
#Adding Configuration File

version = "1.0.0.1";

serviceInfo =
{
    workingDirectory = "./AddingService/";
    mainScript = "adding.py";
    language = "PYTHON";
    poolSize = 2;
    HTTP_Status = 303;
    GET_HTML = "www.google.com";
};
```

B Performance Tested Scripts

B.1 pFactorise

```
import time

def main( value ):
    values = []
    for i in range( 1, int(value)):
        if value % i == 0:
            values.append([i, value/i])
    return values

if __name__ == "__main__":
    start_time = time.time()
    main( 96724316 )
    print( "time: ", time.time() - start_time )
```

B.2 pDBlookup

```
import sqlite3
import time

def main():
    global empNames
    global cusNames
    return empNames + cusNames

def setup():
    global empNames
    global cusNames
    empNames, cusNames = [],[]

    database = sqlite3.connect('./chinook.db')
    database.row_factory = sqlite3.Row
    cursor = database.cursor()

    cursor.execute( "SELECT count(*) FROM employees", [])
    count = cursor.fetchone()[0]

    for i in range( int(count)*10 ):
        cursor.execute( "SELECT * FROM employees", [] )
        response = cursor.fetchall()
        for a in response:
            empNames.append( [i] + list(a) )

    cursor.execute( "SELECT count(*) FROM customers", [])
    count = cursor.fetchone()[0]

    for i in range( int(count)*10 ):
        cursor.execute( "SELECT * FROM customers", [] )
        response = cursor.fetchall()
        for a in response:
            cusNames.append( [i] + list(a) )

    database.commit()
    database.close()

if __name__ == "__main__":
    start_time = time.time()
    setup()
    main()
    print( 'time: ', time.time() - start_time )
```

B.3 rFactorise

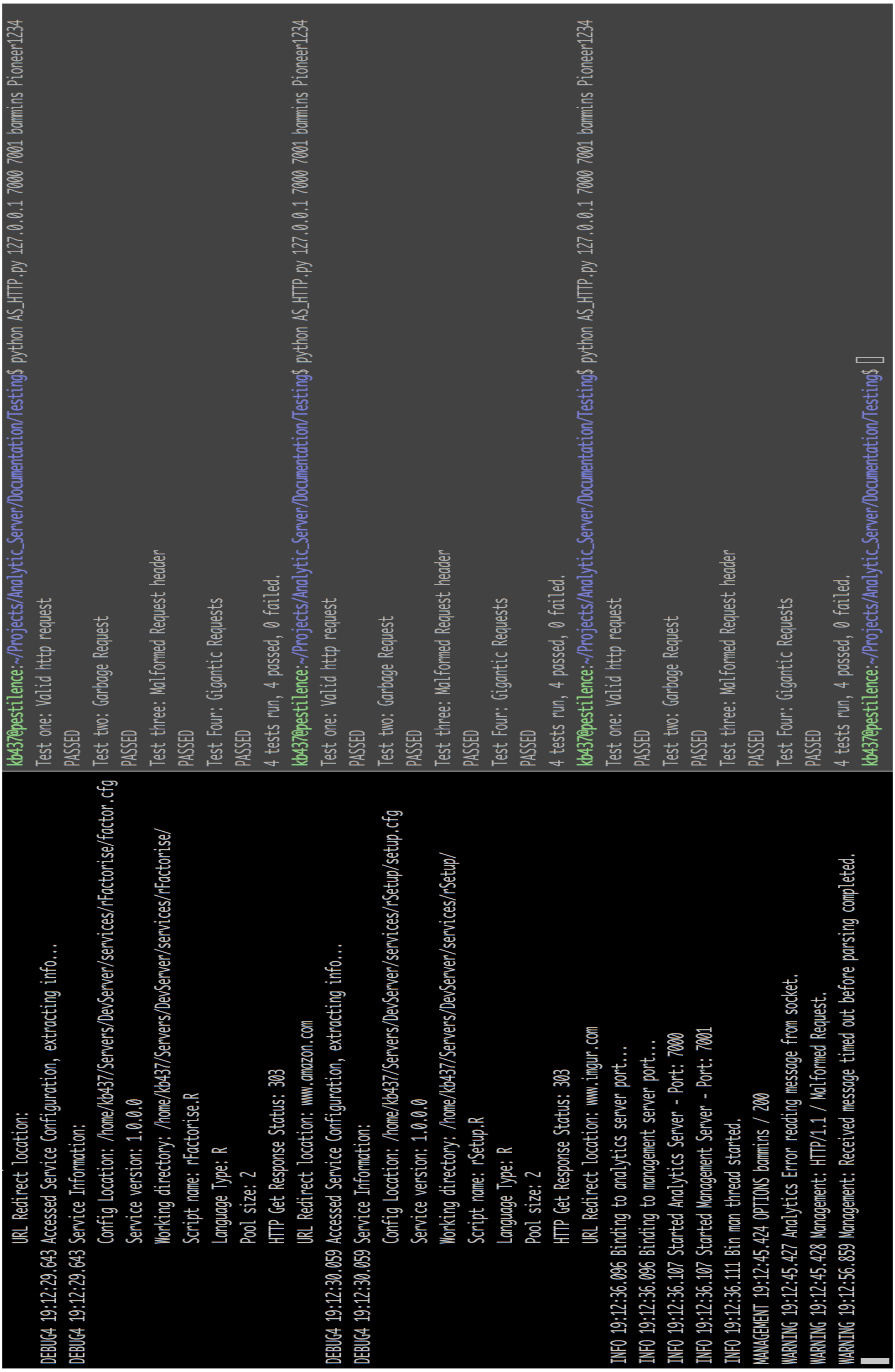
```
main <- function( number ){

    collection <- (1)

    for( i in 2:number ){
        if( number%%i == 0){
            collection <- c( collection, i )
        }
    }

    return( collection )
}

value <- 96724316

paste0( "Begining" )
start.time <- Sys.time()
main(value)
end.time <- Sys.time()
time.taken <- end.time - start.time
time.taken
paste0( "End" )
```

B.4 rSetup

```
main <- function(){
    return( collection )
}

setup <- function(){
    collection <<- (1)

    for( i in 2:4568782 ){
        if( 4568782%%i == 0){
            collection <- c( collection, i )
        }
    }

    return( collection )
}

paste0( "Begining" )
start.time <- Sys.time()
setup()
main()
end.time <- Sys.time()
time.taken <- end.time - start.time
time.taken
paste0( "End" )
```

C Testing printouts

C.1 HTTP Tests



C.2 Service Tests

```
kb437@pestilence:~/Projects/Analytic_Server/Documentation/Testing$ python AS_services.py 127.0.0.1 7000 7001 bammins Pioneer1234
Test set one: Valid requests
LS
PASSED
SERVER STATUS
PASSED
SERVER ADDRESS
PASSED
NEW SERVICE
PASSED
SERVICE GET
PASSED
SERVER POST
PASSED
DELETE NEW SERVICE
PASSED
Test two: Malfored Config
(u'400', u'Service configuration error.')
Test two: Malfored service
PASSED
Test set three: Updating malformed and activating malformed
NEW SERVICE
PASSED
Update
('404', 'Error within script file, service deactivated.')
Activate
('404', 'Initialisation of service error.')
Delete
('200', '')
13 tests run, 13 passed, 0 failed.
```