Service-Oriented Architecture based Cloud Computing Framework For Renewable Energy Forecasting

Rakesh Sehgal

Thesis submitted to the Faculty of the Virginia Polytechnic Institute and State University in partial fulfillment of the requirements for the degree of

Master of Science

in

Electrical Engineering

February 17, 2014,

Blacksburg, Virginia

Keywords: Microsoft Azure, Cloud Computing, Forecasting, Service-Oriented Architecture

Service-Oriented Architecture based Cloud Computing Framework For Renewable Energy Forecasting

Rakesh Sehgal

ABSTRACT

Forecasting has its application in various domains as the decision-makers are provided with a more predictable and reliable estimate of events that are yet to occur. Typically, a user would invest in licensed software or subscribe to a monthly or yearly plan in order to make such forecasts. The framework presented here differs from conventional software in forecasting, as it allows any interested party to use the proposed *services* on a *pay-per-use* basis so that they can avoid investing heavily in the required infrastructure.

The *Framework-as-a-Service* (FaaS) presented here uses Windows Communication Foundation (WCF) to implement Service-Oriented Architecture (SOA). For forecasting, collection of data, its analysis and forecasting responsibilities lies with users, who have to put together other tools or software in order to produce a forecast. FaaS offers each of these responsibilities as a service, namely, *External Data Collection Framework (EDCF), Internal Data Retrieval Framework (IDRF) and Forecast Generation Framework (FGF*). *FaaS Controller,* being a composite service based on the above three, is responsible for coordinating activities between them.

These services are accessible through *Economic Endpoint (EE) or Technical Endpoint (TE)* that can be used by a remote client in order to obtain cost or perform a forecast, respectively. The use of Cloud Computing makes these services available over the network to be used as software to forecast energy for solar or wind resources. These services can also be used as a platform to create new services by merging existing functionality with new *service features* for forecasting. Eventually, this can lead to faster development of newer services where a user can choose which services to use and pay for, presenting the use of FaaS as Platform-as-a-Service (PaaS) in forecasting.

[The National Science Foundation under Grant 1048079, supports this work. The resources provided by the Microsoft Corporation under the Microsoft Cloud Research Engagement Program are gratefully acknowledged]

Dedication

I dedicate this work to my mom, Parsanni Sehgal without her love I couldn’t have come this far.

ACKNOWLEDGEMENTS

First and foremost, I would like to express my deepest gratitude to my advisor, Dr. Kwa-Sur Tam for his guidance and support at every stage during my research. Without his feedback on my work, this thesis would not have materialized.

I would like to thank Dr. Virgilio A Centeno and Dr. Jaime De La Reelopez for graciously agreeing to serve on my committee. I also thank the staff at the Bradley Department of Electrical Engineering and the Graduate School for their help and co-operation in paperwork and administrative matters.

My heartfelt thanks to my roommates and friends at Virginia Tech for keeping me motivated during my stay here.

Finally, I would like to thank my family and friends for their unconditional love, and encouragement at every stage of my life.

Rakesh Sehgal

Table of Contents

List of Abbreviations

EDCF	External Data Capture Framework
FaaS	Framework as a service
FGF	Forecast Generation Framework
IAAS	Infrastructure as a service
IDRF	Internal Data Retrieval Framework
IRC	Inter-Role Communications
MAPE	Mean Absolute Percentage Error
PAAS	Platform as a service
REST	REpresentational State Transfer
SAAS	Software as a service
SDLC	Software Development Life Cycle
SOA	Service Oriented Architecture
SOAP	Simple Object Access Protocol
WCF	Windows Communication Foundation
WSDL	Web Services Description Language
XML	eXtensible Markup Language

List Of Figures

List Of Tables

(This page is intentionally left blank)

Chapter 1

Introduction & Motivation

“Forecasting is commonly defined as 'the activity of predicting a future event or condition'.” [McCalman, S., 2012] where the actual outcomes have not yet been observed. Every forecasting process has a certain risk and uncertainty that is given by the degree of uncertainty. The accuracy of forecasted data will also depend on the quantity and quality of the data as well as the length of time over which the data has been collected. Moreover, the collection of data from various sources is also a major responsibility of users, which then must be analyzed with other software. Currently, there are various software and tools that help researchers to forecast, but they have their own limitation. In the next few sub-sections these limitations and the motivation behind this work will be discussed.

1.1 Problem Definition and its Significance

The problem with current approaches in software for forecasting is that an end user has to buy a license in order to use the software. In cases when such services are offered online, users are typically bound to pay monthly or annual subscription charges even though they might be using it for a limited number of times. The cost of such software is often too high for groups with limited funding, thus preventing them from using such services.

Once a user has invested in such infrastructure, they have to permanently bear the cost incurred from its operation and maintenance, irrespective of whether they are using it or not. The use of cloud computing can free users from such investments as they can *pay-as-they-go* for traditional resource (hardware or software), including storage, database, application development, application services, etc., instead of a making a long-term commitment. Even though there are some risks and limitations involved in working with cloud services, such as dependence on network, privacy, etc., still the advantages from lower cost, ease of use, scalable on-demand infrastructure and reusability, etc., outweighs them.

In addition to cost, the usability of typical software is limited to a particular platform, and these software have to be used with disparate tools and technologies which a user has to put together. This further reduces the reusability of such software, as their usage is limited to the particular group that they are created for. To overcome these limitations, Service-Oriented Architecture (SOA) provides a different approach to software design.

Software designed using SOA use a structured collection of various software modules called *services,* which can be reused by combining with other functionalities needed to build a new software solution. SOA based services are loosely coupled, that is the services are broken into the smallest unit of functionality, and are not dependent on other services in order to function. Such a design ensures that there is a maximum reuse of code; this will eventually bring down the cost and effort of software development.

The lack of software or services that can be used as a platform on a *pay-per-use* basis in forecasting was the primary motivation behind this work. Bringing together SOA and cloud computing increases reusability, interoperability between various vendors, collaboration between different parties, and promotes efficient use of resources. Eventually, this will also allow an independent group of developers to benefit from the individual or collective efforts of others.

1.2. Goal and Scope of Thesis

The primary goal of this research work is to develop and implement a framework that can be used by an end user as software in forecasting. The developed framework can be used as a platform to create new services for forecasting, while paying only for the services they use and not the entire framework. The *pay-per-*

use cost model and the savings from the use of cloud computing results in significant cost reduction for an end user.

Through this work it will also be demonstrated how the use of SOA in the development of such services can increase their reusability, leading to faster development and reduced effort. The services developed in the process will be used by other services that add more features and functionality, to show the use of Framework-as-a-Service (FaaS). These services will essentially be offered as Platform-as-a-Service (PaaS) exposed at given endpoints accessible via the Internet, to be used by anyone, anywhere with access to the network, without worrying about the maintenance of the infrastructure.

The scope of this thesis will be limited to discussions about forecasting for solar and wind energy, and the reusability of the developed framework. The stress will be more on the development of the framework using cloud computing, with lesser focus on the forecasting process. The principles and technologies that were used to create the framework and services will also be discussed. The idea of using SOA to develop distributed applications, and leveraging cloud computing to offer these as network based software services will be emphasized. As there are a variety of cloud services available from various vendors, the choice of using Microsoft Azure as a platform for developing such service will also be justified.

1.3 Thesis Organization

This thesis is organized in seven chapters followed by a bibliography and appendices. In the next chapter, a brief literature review will be provided to explain the motivation that drove the development of the FaaS to be focused on SOA and the use of Microsoft Azure as a cloud-computing platform. The design methodologies that were central to the design of the framework will be discussed in Chapter 3. After explaining these ideas, implementation details of the forecasting framework will be presented in Chapter 4 to show how a solution to the problems discussed in Section 1.1 was achieved using SOA. Chapter 5 will present the cost estimation framework and the pricing strategy that was used to decide the cost associated with each service. Finally, the results obtained using the framework will be discussed, and a conclusion will be drawn in Chapter 7.

Chapter 2

Literature Review

This chapter gives an overview of cloud computing and the technologies used to develop FaaS (Framework-as-a-Service). The use of cloud computing for applications such as forecasting will be justified here, with a brief review of relevant literature. A comparison of major cloud computing platforms will also be provided, followed by the original contribution of this work and the use of Microsoft Azure as development environment to offer FaaS.

2.1. Cloud Computing Framework for Forecasting

Forecasting may be used to estimate the outcome of an event at a future date, by taking into consideration past events. Weather foresting has been performed since time immemorial, with early civilizations using recurring astrological and meteorological data to help monitor weather changes. Modern computers were first used for weather forecasting by a team of scientists at Institute for Advanced Study (IAS) in 1940's. Since then, forecasting has been used in various domains ranging from rainfall forecasting, the harvest of crops and fields of supply chain planning, to revenue forecasting, electrical load forecasting and sales forecasting.

Every forecasting process has a certain risk and uncertainty factor attached to it, indicated by the degree of uncertainty. The uncertainty is due to the lack of data or computing resources that are at disposal of a forecaster. The more the uncertainty in a process, the lesser is the precision of forecast. To increase the precision of a forecasted value, a bulk of data needs to be analyzed, a model to be prepared and based on the input values, results are obtained. [Giebel, G., et. al., 2011] discussed some state-of-the-art methods for short-term prediction of wind power like Numerical Weather Prediction (NWP), Artificial Neural Networks (ANN), which can be applied to multiple domains. The historical data can be analyzed in either time or frequency domain for training and to increase the accuracy of forecast.

Nowcasting [Mass, C. and Mass, C. F., 2011], a fine-grained weather forecasting algorithm, combines a description of the current state of the atmosphere and prepares a short-term forecast (some forms of Nowcasting use NWP model) of how the atmosphere will evolve during the next several hours. In this technique, the data generated by radar, satellite and observational data is used to analyze small-scale features present in an area. Forecasting algorithms like Nowcasting created a requirement for an underlying network to transfer several megabytes of data per sec and so "generating Nowcasts on dedicated infrastructure is economically infeasible due to its enormous computational demands, which scale quadratically with spatial resolution” [Krishnappa, D. K. et. al. 2012].

For energy planners, short-term forecast (hours to days) and long-term forecasts (months to years) are measurable parameters. So, to successfully perform such weather forecasting research, easy access to computing resources and to the measured and meteorological data is required.

2.1.1. Benefits of using Cloud Computing for Forecasting

Scientific experiments essentially involve analysis of data and advances in the field of information technology have changed the way this analysis can be done. As the amount of data increases, more time is required for analysis and thus the requirement of superior computing power. Moreover, as these experiments are discrete in nature, sharing such resources would save the cost of investing and maintaining them. In recent years, a lot of work has been done to offer computing power as a utility. Cluster computing uses a group of tightly coupled computers on the same local area network (LAN) to share resources by using a number of machines with similar hardware. Grid computing overcomes the limitations of cluster by providing computing resources, each with different capabilities, over geographically spread out areas. These resources can be used as and when needed in order to solve CPU-intensive research problems.

[Vecchiola, C. et. al., 2009] and [Hoffa, C. et. al., 2008] discuss limitations and inefficiencies of grid computing. In grid computing, owners of the resource decide who will receive what and for how long, which leads to under-utilization of resources. In addition to this, smaller projects or groups might not get a fair treatment if their interests do not line up. Also, there might be times when all the resources put together still are not able to achieve the desired results, which leads to the problem of scalability.

Cloud computing addresses all such issues that enables an on-demand network access "..to a shared pool of configurable computing resources" that can be "rapidly provisioned and released with minimal management effort or service provider interaction" [Mell, P. and T. Grance, 2011]. [Vecchiola, C. et. al., 2009] also presented two case studies: a classification of gene expression and the execution of an fMRI brain imaging workflow that show how such computationally expensive scientific experiments can benefit from cloud computing.

Unlike Grid or Cluster computing, Cloud computing caters to consumers' needs by scaling up and down as necessary and at the same time it "ensures the desired Quality of Service" [Vecchiola, C. et. al., 2009]. Rapid elasticity of the system is an integral part of Cloud computing. For enterprises, the economy of scale means reduction in cost, whereas for individual consumers and small-scale research labs, it enables them to *pay-per-use* so that they do not have to make a permanent commitment for access to such resources. This is in comparison to traditional software, where vendors make a huge investment in the infrastructure before they begin to rollout copies of the software, which are then bought by customers who pay a license fee.

CloudCast [Krishnappa, D. K. et. al. 2012] is an application that leverages cloud computing to generate accurate forecasts tens of minutes in the future. It uses Amazon's Elastic Computing Cloud (EC2) to produce short-term forecast using Nowcasting and sends automatic alerts to a user in case a severe weather condition is approaching the user's specified area. As Krishnappa et. al. (2012) discuss in the paper, CloudCast has strict time and network constraints in order for the forecasts to be useful for a user. For these purposes the authors compare different cloud services for this application in terms of computing, data, storage, networking and cost. The results show that using cloud services can "mitigate the impact of staging data and computation latency" [Krishnappa, D. K. et. al. 2012].

Large high performance applications can benefit from on-demand access and scalability of compute and storage resources at the same time, significantly reducing the application execution time. Cloud computing enables application developers to create proof-of-concepts in an easy, timely and cheap manner, before going into full production. Cloud-based outsourcing of computing may be attractive to science applications because it can potentially lower the costs of purchasing, operating, maintaining, and periodically upgrading a local computing infrastructure [Deelman, E. et. al., 2008]. Thus to reduce application development time and produce more reliable software systems, the use of open source cloud computing tools is highly recommended and is demonstrated by a case study of weather forecast presented in [Rodriguez-Martinez, et. al., 2010].

Service providers such as Amazon, Google, Microsoft, Rackspace and Salesforce, etc. offer cloud services at different levels such as Software as a Service (SaaS), Platform as a service (PaaS) and Infrastructure as a Service (IaaS), shown in fig. 2.1. Salesforce.com became a pioneer in SaaS and rolled out many customized applications that were developed as services for other companies. Google's Gmail, Google Docs, Google Calendar, etc. are yet other examples of Software-as-a-Service that one comes across everyday. On the other hand, Google AppEngine and Microsoft offer PaaS solutions where users can create their own applications to run on cloud and leverage their infrastructure to scale up and down on demand. Amazon Web Services (AWS) offers hardware level virtualization and provide computing and storage resources on demand following an IaaS solution (Amazon Elastic Compute Cloud or EC2).

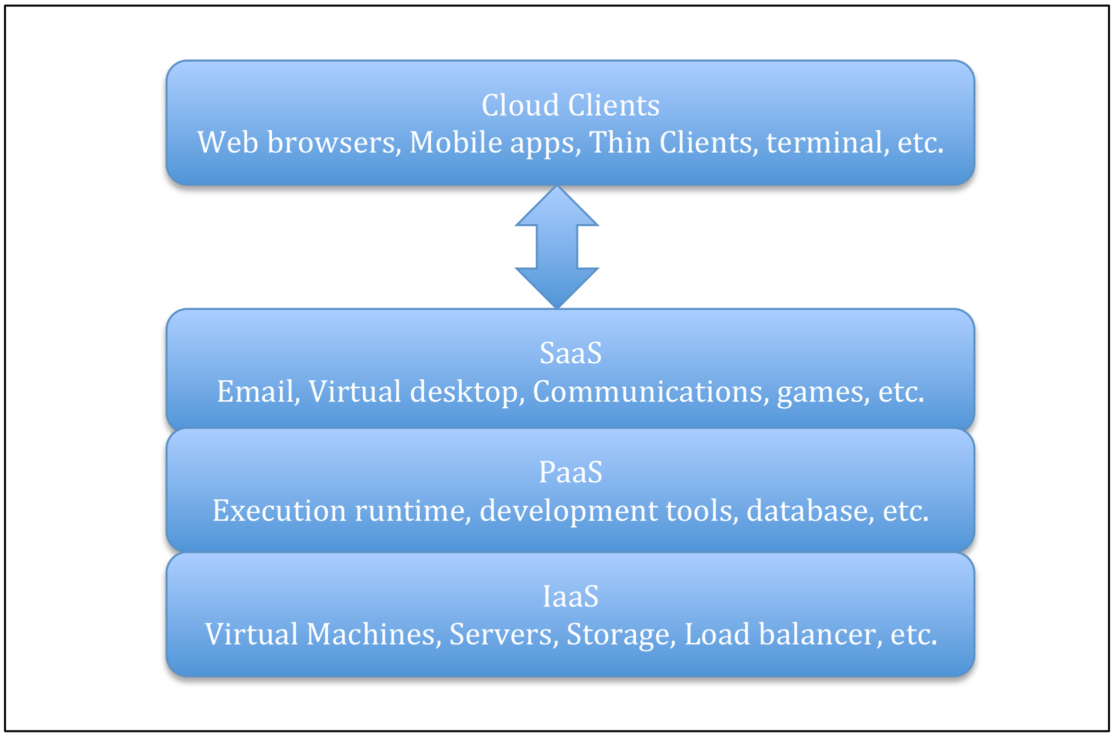

Cloud Clients
Web browsers, Mobile apps, Thin Clients, terminal, etc.
SaaS
Email, Virtual desktop, Communications, games, etc.
PaaS
Execution runtime, development tools, database, etc.
IaaS
Virtual Machines, Servers, Storage, Load balancer, etc.

Figure 2.1 Types of Services offered by different Cloud Service Providers

Aneka [Vecchiola, C. et. al., 2009], commercialized by Manjrasoft, like FaaS is a pure PaaS cloud computing implementation for developing distributed applications on the cloud by using .NET, which can be deployed on Windows and Linux. “The core value of Aneka is a service oriented runtime environment – the Aneka container – that is deployed on both physical and virtual infrastructures and allows the execution of applications developed by means of different programming models" [Vecchiola, C. et. al., 2009]. Aneka provides dynamic scaling of applications to meet increasing demands. There are different programming models, which provide scientists with options to express the logic of their applications. Being a PaaS, applications administrators can deploy the Aneka containers on a variety of platforms or on Amazon Machine Images (AMIs). Users, applications or other services can directly consume the services developed on Aneka. A collection of jobs are automatically moved and executed in the run time environment by guaranteeing a desired QoS along with an infrastructure to facilitate appropriate pricing mechanisms.

2.2. Ways to Implement SOA

2.2.1. Definition of SOA and its Implementation

The term *service-oriented architecture* (SOA) refers to systems structured as a network of loosely coupled, communicating services [D. Booth et al., 2003]. SOA introduces the concept of *services* that are individual units of logic bound together by a set of principles (*service contracts*) and can be accessed via standard interfaces and protocols. Each service becomes a fundamental block that exists autonomously in SOA and can be used to create newer services, known as *composite services*, by re-combining existing functionality to meet changing demands, rapidly.

SOA adds agility to the architecture by giving a user the ability to change the system as a whole by manipulating it at a basic level. SOA and web services bring in the idea that software can be offered as services that can be discovered, extended and executed by service consumers anywhere. By applying SOA across multiple solution environments a good return on investment can be obtained by building reusable and interoperable services as these services will then be vendor-neutral and can work across various platforms [Erl, T., 2005].

The XML data representation architecture is the backbone of SOA and establishes the format and structure of messages traveling within and between services. SOAP [SOAP, 2013] extends this idea to web services by transmitting data between components and serializing them into XML, transport, and then de-serializing them back into their native format to extract useful information. SOA relies on SOAP messaging for communication between services. Distributed Internet applications allow application logic to be placed on the server side and a thin client or browser can be used to access these services through the network. To implement SOA, one must know how the application logic can be partitioned, when it should be processed and how various services can interact with each other.

After the services have been created, they can be orchestrated together by dynamically creating new *workflow* (a set of activities) to perform a specific task. The uniformity of the interfaces through which these services can be accessed results in a distributed network of services such that each has some original content. Each of these services can be offered as a value added product without the concerns of interoperability between different vendors.

2.2.2. Benefits of Cloud Computing in Implementing SOA

Cloud services that are built using SOA can be exposed as Web services using industry standards like WSDL [WSDL, 2013], SOAP, etc. that are accessible across networks. The underlying principles of SOA help make cloud computing more flexible, scalable and reusable by creating abstraction, loose coupling, and encapsulation between various services. Cloud computing provides a scalable on-demand infrastructure so services can be scaled-up to meet increasing demand in traffic or scaled-down to reduce cost when service traffic is low. At the same time, SOA reduces both, cost and effort, by reusing services.

By using SOA, processes are not limited in their functional capabilities, and services can be assembled or reconfigured as per changing requirements. More complex models such as workflows can take advantage of this simple implementation for distributing the execution of tasks. The workflows can begin in a simple manner and their behavior can be changed dynamically by other involved elements.

From an architectural point of view of cloud computing in [Huang, Y., et. al., 2010], the authors described a service-oriented cloud-computing platform that enables web-delivery of application-based services with a set of common business and operational services. [Tsai, W.-T., et al., 2010] also discuss the issues with current Cloud computing architecture such as how migration from one Cloud service provider to another is difficult because of the storage or computing capabilities, and the inherent tight coupling between components. To overcome such limitations, [Tsai, W.-T., et al., 2010] and [Zhang, H., et. al., 2012] suggest Cloud Computing Architectures based on SOA (SOCCA, UC2A), where each layer of cloud services (IaaS, PaaS, and SaaS) can be offered independently.

To highlight the importance of SOA, the authors [Tsai, W.-T., et al., 2010] and [Zhang, H., et. al., 2012] also proposed to introduce an SOA layer to take advantages of the existing research and infrastructure, by integrating frameworks like user-centric service-oriented architecture (UCSOA) [Chang, M., et. al., 2006]. The fundamental difference being that service providers will not host these services. Instead, they publish the services in deployable packages, so that they can be replicated and redeployed to different cloud hosting environments. This would enable, not only services, but also workflow templates, test cases, etc. to be shared thus increasing their reusability.

2.3. Review and Comparison of Major Cloud Computing Platforms

As previously mentioned, different providers offer cloud services at different levels, namely, SaaS, PaaS and IaaS. The usage and needs of both, a developer and end users, are a deciding factor as to which will prove to be more efficient. This section provides an overview of capabilities of these providers and compares them in view of the framework similar to the one presented in this thesis.

Google's AppEngine platform provides a set of APIs and an application model that allows developers to use already available apps from Google and combine with services from other vendors. Their PaaS platform offers developers a choice of multiple programming languages (Java, Python, and JRuby) to create applications and as of this writing, is free of charge for developers, but provides paid solutions for enterprises. Microsoft offers their PaaS services under Microsoft Azure, providing a highly scalable platform based on Windows, with multiple pricing strategies [Microsoft Azure, 2013]. Applications in Azure can be created based on the well-established .NET framework and exposed through various interfaces to external parties. Table 2.1 below shows the comparison between different cloud service providers to highlight levels of services provided by them and the tools that are offered to carry out development.

Table 2.1. Comparison of Different Cloud Service Providers [Vecchiola, C. et. al., 2009], used under fair use, 2014

Property	Amazon	Google	Microsoft	Rackspace
Type of Service	IaaS	IaaS, PaaS	IaaS, PaaS	IaaS, PaaS
Interface	Web & CLI	Web & CLI	Azure Portal	Custom Web UI
Programming Languages	Java, Python, C#/.NET	Python, Java, JRuby	C#/.NET	C#/.NET, Python, PHP, Java, and Ruby
Operating System	Linux, Windows	Linux	Windows	Linux, Windows

Amazon's PaaS services, like Microsoft’s, provides several programming tools for development on the cloud to be run on virtual “machine images” that can run Linux, Windows, and other operating systems by creating Amazon Machine Images. Amazon also provides interfaces to migrate from/to other clouds on-premise servers and charges the user per hour per instance. A user also has the option to run multiple instances thus enabling dynamic scaling. In comparison to this, Rackspace provides PaaS in addition to IaaS and similar development tools and run-time environment.

2.3.1. Benefits of Azure compared to the other Cloud Computing Platforms

Microsoft's Azure is based on Windows and serves as a platform to develop, run and control cloud services. Applications running on Azure run virtual machines and can have multiple instances of a single application, each running in its own VM (64-bit Windows Server 2008). A developer is provided with Microsoft Azure SDK where services can be created by leveraging the existing capabilities of the .NET Framework and then published on the cloud using the Azure portal.

Applications in Azure can be implemented by using a web role or worker role instances, each running their own separate VM, whereas the cloud fabric handles physical provisioning. A web role provides support for HTTP and HTTPS and is hosted in IIS. This is where the frontend to the application is normally deployed. Whereas a worker role can also be exposed on endpoints other than HTTP and HTTPS and is not hosted in the IIS. A worker role is suitable to do heavy processing work in the background similar to processing data for forecasting.

Figure 2.2 shows how this type of infrastructure is offered as a service in Microsoft Azure. The instances of web or worker role can be increased dynamically or configured statically to do higher computational tasks or to meet increasing service requests, thus making applications scalable. As the web and worker role communicate through a queue, this also helps in balancing of data requests (discussed in Chapter 3 and 4).

Windows Azure AppFabric helps in connecting services within or across cloud and enterprises by providing a Service Bus for connectivity using standard protocols. The Service Bus and Access Control capabilities of Azure make it easier to extend already existing .NET applications to the cloud. This makes it easier for the enterprise to migrate and decide what applications should go on cloud while allowing other to run on current infrastructure. Reliability and security are an integral part of Azure as all the data is backed up and multiple copies of the data are stored on various servers and across data centers.

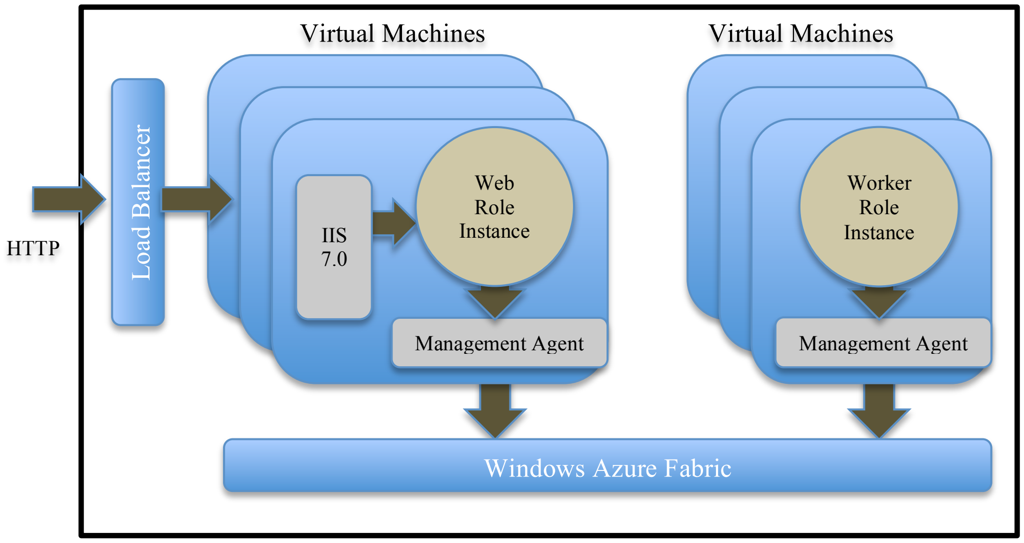


Figure 2.2. Infrastructure services in Microsoft Azure [Li, Henry 2009], used under fair use, 2014

In addition to this, the simplified access to scalable storage services in Azure through Blob, Tables and Queues (shown in fig. 2.3) takes away the time consuming process of maintaining complex databases and customized storage. The Azure storage, unlike other services is not relational, so this allows storage of data in blob, queue, and simple tables that are available through REST API based on HTTP requests. The data can be manipulated through traditional POST, PUT, and DELETE requests, and can be retrieved from the storage using simple GET requests.

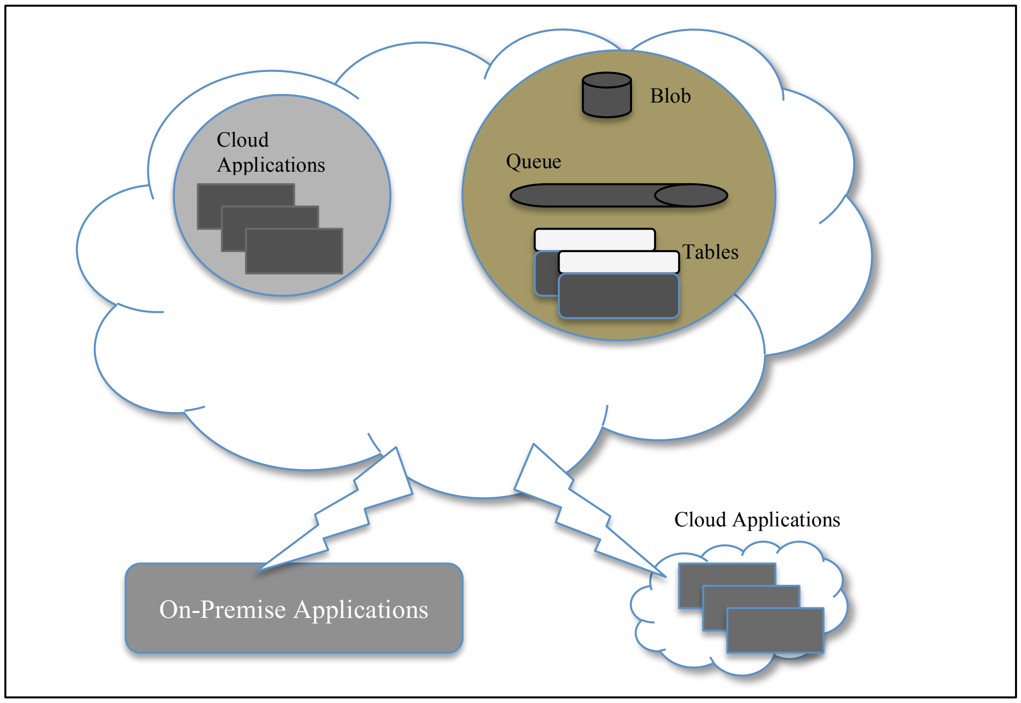


Figure 2.3. Infrastructure of Microsoft Azure Cloud Platform [Li, Henry 2009], used under fair use, 2014

Microsoft's WCF is a tool that is seamlessly integrated with other technologies and is commonly used to implement and deploy services that are based on SOA. As WCF itself is designed using principles of SOA it is a powerful tool in supporting distributed computing where services are offered to remote clients, making them platform independent. Interfaces like WSDL and protocols like SOAP, XML, etc. are an integral part of WCF services making it easier for developers to adopt these instead of using other tools to explicitly put them together.

2.4. Original Contribution

The *pay-per-use* pricing strategy developed for this work, lets a user pay only for the services they use instead of paying a license or subscription fee as in traditional software. The prototype developed as a part of this research, was used to forecast energy that can be generated at a particular location through solar and wind resources. In a broader view, it also presents the idea of how the services presented here can be used for forecasting in other domains.

The work presented here applies the concepts of SOA combined together with cloud computing to present the use of Framework-as-a-Service (FaaS) that can be used as a platform for forecasting. As discussed in the *section 2.1*, scientific experiments can be data and computationally intensive and thus cloud computing becomes an essential part of forecasting processes too. Cloud computing helps speed-up the development of a system as the infrastructure is available on demand, thereby researchers have at disposal various means to analyze if a proof of concept works or not. Cloud computing also meets the strict time and network constraints, discussed in Section 2.1.1, that applications such as forecasting place on them. The *pay-as-you-go* cloud services reduce the cost so users do not have to maintain the resources if in future they decide to do away with it. For small research groups or enterprises this means they can have access to the same capabilities, with a given QoS, as large research groups do.

The cost model for the FaaS takes in consideration various factors some of which are discussed in [Truong, H.-L., et. al., 2010], and takes into account the computation time that would be required, the storage cost for data, networking, labor, infrastructure cost, etc., (discussed in detail in Chapter 5). As opposed to traditional software development, cost estimation for services that are based on SOA is more complex and thus a Divide and Conquer [Li, Z., & Keung, J., 2010] approach was used. In Divide and Conquer, each service developed has a cost associated with it and as a client can consume multiple services, the costs are accumulated accordingly. In addition to this, Autonomic pricing [Yeo, C. S., et. al., 2010] has been used within the framework with the aim of maximizing revenue by charging different customers based on the usage of their computing services.

For the implementation of FaaS, Microsoft's Azure was chosen because the .NET capabilities are themselves SOA oriented, so extending other services from them becomes easier. In contrast to other service providers, Microsoft's PaaS API's to access the cloud storage are simplified and do not need as much preparation in order to use. This is particularly beneficial for users who have little to no experience with cloud services. Assuming the same computational power, storage, data transfer and database usage, it was found that for FaaS, with lower data storage requirement, the use of Microsoft Azure could have lower costs than other cloud service providers.

The control unit of the framework is the Framework-as-a-Service (FaaS) Controller and is responsible for creating workflows to carry out forecasting. In comparison to Aneka, discussed in Section 2.1.1, the FaaS follows a PaaS approach, and contains services that are available at specific endpoints for use as a platform. These services are developed using WCF (Section 2.2) so an end user does not have to be aware of where the applications are deployed, stored or executed, thus making the application extendable for other uses. A developer can extend and reuse these services as a platform to create more *composite* services similar to FaaS.

The FaaS being a composite service is based on three services, namely, External Data Collection (EDCF), Internal Data Retrieval (IDRF) and Forecast Generation

Framework (FGF). Each of these services runs multiple instances to service simultaneous request and each has two endpoints namely, Technical Endpoint (TE) and Economic Endpoint (EE). A consumer of these services can use the economic endpoint to know beforehand what the cost of using these services will be and the technical endpoint can be invoked after the user submits their requests to carry out the actual workflow. These endpoints also make it possible for an end user to choose which services they want to use and pay for.

When service requests enter service *queue,* FaaS Controller partitions the workflow and uses the technical endpoints of EDCF, IDRF, and FGF. Then each service creates a workflow that is based on various *service features*; to download data from external sources, store them internally in blobs, tables, etc. and to produce a forecast. The service features used by these services are implemented as *classes* that contain data member and overloaded member functions to achieve a desired functionality. As these service features are available as library files, by using the technical endpoints along with these library files, a developer can extend the use of these services. So, FaaS essentially acts as a PaaS in creation of new services for forecasting in other domains.

Chapter 3

Methodologies

Since the time when SOA was introduced as a design methodology, many industry sources have participated in publishing principles of SOA. In [Erl, T., 2004], the author discusses the importance of designing “quality endpoint services” with “streamlined interface” and how services in SOA inter-relate with each other to achieve a better application architecture. SOA enables such a design as services in SOA are based on bottom up approach, which helps in breaking them down to more granular and independent pieces. After successful creation of services using SOA, cloud computing helps add agility to the architecture and these services can be accessed at various endpoints.

The benefits of using SOA as mentioned in the previous chapter, will be discussed here at length and it will be shown how they were extensively used to create the framework for forecasting.

3.1. Service Features, Services and SOA

As mentioned in Section 2.2, a service is an independent block which when combined with other services powers an application and must be completely independent of other services [Erl, T., 2004]. As each service has distinct design characteristics, like encapsulation of functionality, abstraction of logic and modularity, which enables them to achieve platform independence. Applications designed in this way are easier to upgrade, as changing one service does not affect the functioning of other services. The lower levels of services are common to a wide range of applications and are loosely coupled from other services at the same level. Thus, they can be used to create a *composite* service, which "is an aggregate of services collectively composed to automate a particular task or business process" [Erl, T., et. al., 2010] and are based on lower level service to add more functionality.

SOA is based on application frontend, service, service repository and service bus. The frontend is responsible for providing access to the processes whereas services provide functionality to it through various service features. The service repository stores the service contracts of the individual services, and the service bus runs along to connect the application frontends and services. The benefit of using such an approach is that "although the services can often remain unaltered, the application frontends are subject to change" [Krafzig, D., et. al., 2005]. Thus, designing the services and data is of more importance than the frontend.

Access to services in SOA is guided by a contract, which comprises "of one or more published documents that express meta information about a service" [Erl, T., et. al., 2010]. The business logic and data contained in such services is accessible via a standard interface, much like in traditional API's. The difference being that traditional applications were never "built with interoperability in mind, and therefore do not provide generic interfaces." [Erl, T., et. al., 2010]. Also, as these services are agnostic and stateless, each service may be invoked any number of instances.

3.1.1 Using WCF to Implement SOA

As discussed in Section 2.2.1, SOA can be implemented using a variety of technologies, including SOAP, REST, WCF and Web Services. The design presented here uses WCF as it greatly simplifies implementation of SOA by putting together various technologies that otherwise had to be put together to achieve the same results. As WCF supports distributed computing where services have remote consumers, multiple clients can simultaneously consume services using endpoints that are comprised of Address, Binding and Contract or ABC as shown in fig. 3.1 below. Address (A) indicates *where* the service is, Binding (B) dictates *how* to access the service and the Contract (C) essentially tells *what* the services can do.

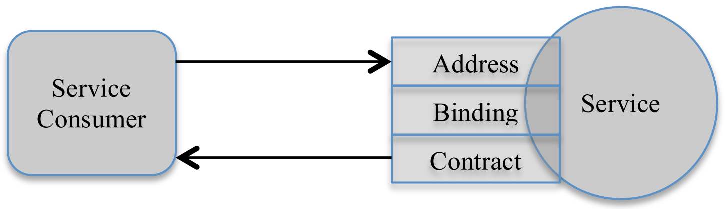


Figure 3.1 A service consumer communicating with service with information provided by the endpoint

WCF can be thought of as a messaging infrastructure, where service operations receive messages, process them, and send response messages. WCF supports operations modeled on either the Remote Procedure Call (RPC) style or the messaging style. One of the biggest capabilities of the .NET framework is that it enables developers who are not familiar with SOAP and SOAP messages to easily create such messages by choosing the type of serialization that they require, which controls the structure of the data in the underlying messages.

An interface contract is defined to provide an abstract description of a WSDL document where the operation contracts are defined. Its major function is to associate various data and message contracts with their corresponding operations or methods. By implementing the operation contracts of a WCF service interface, a required functionality can be achieved.

A WCF data contract exists as an XML schema that defines the data structure and data types that are exchanged between different services. Although a data contract is sufficient to define the schema of messages exchanged, sometimes more control is required over the structure of the SOAP messages transmitted over the wire. To achieve this, WCF services allow the use of message contract, where custom SOAP headers can be inserted in messages and an XML is automatically generated corresponding to these messages.

Services can choose to exchange messages in request or response, or they can choose to exchange empty messages between themselves, as shown in fig 3.2. A service can also choose not to return any message (one-way) as shown in fig 3.3. This can be useful in a scenario where a service takes longer time to perform operations and might not want to inform the calling service about the outcome of the operation. Other way to achieve this is by declaring contracts as *asynchronous*, so that the services can communicate whenever they have data ready to exchange.

```
[OperationContract]
public string GetFilename();
```

```
[OperationContract]
public void SetFilename(string testname);
```

Figure 3.2. An empty request message vs. empty reply message

```
[OperationContract (IsOneWay=true)]
public void CreateFileIfNotExists(string filename);
```

Figure 3.3. A service operation that returns no message

Each service in Framework-as-a-Service (FaaS), uses interface contracts to declare operations that a service consumer is permitted to perform. These operations can be defined as per the consumer's requirement by using different *service features* (Section 3.1.2) and thus creating different *workflows* (Section 3.2) for forecasting. A combination of one-way and two-way operations allows messages to be exchanged between services through a data contract (discussed in detail in Chapter 4 and 5).

3.1.2 Service Features used by Services

The implementation of service and data contracts for the services, EDCF, IDRF and FGF, involves the use of various fields like mathematics, statistics, networking, etc. These WCF services are built using various service features that are classes with data members and member functions that gives them their respective functionality. These classes were partitioned in to various *namespaces*, where each namespace broadly belongs to a particular field. The namespaces acts as a container for these services features and were developed as Dynamic Library Files (DLL) using C#/.NET as follows:

1. *Communication Tools* is used to communicate with the outside world via email. It contains service features that enable the framework to send results to the user. Details for the service features in this namespace are in Appendix A.1.

2. *Costing Tools*: contains functionality that helps a developer, add or update costs of service features and infrastructure to keep track of changing costs. Details for the service features in this namespace are in Appendix A.2.

3. *Forecast Tools*: contains service features to forecast solar and wind data using data from internal sources for both operational and project prospecting forecasts. Details for the service features in this namespace are in Appendix A.3.

4. *Foundation Tools*: This tool contains functionality that help process the data obtained from external sources and store them internally in a fixed, pre-decided format. It also helps read and write data to/from Blobs and Tables. Details for the service features in this namespace are in Appendix A.4.

5. *General Support Tools* is used to prepare reports that are sent to the user and help developer customize dates to a particular format. Details for the service features in this namespace are in Appendix A.5.

6. *IO Tools* contains various service features that offer the Azure cloud infrastructure as platform-as-a-service to access the storage services. Other than this, some of the features that are not natively available on Azure are implemented here like reading or writing excel files, unzip and tar functionality, etc. Details for this service features in the namespace are in Appendix A.6.

7. *Solar Analysis Tools* provides functionality to read solar historical data and store it in tables in a given format for later use during forecasting. Details for

this service features in the namespace are in Appendix A.7.

8. *Statistical Tools* provides various basic statistical tools and features that a user needs in order to analyze data. Details for this service features in the namespace are in Appendix A.8.

9. *Transfer Tools* helps a user fetch data from external sources and store them in blob for later retrieval by other services. Details for the service features in this namespace are in Appendix A.9.

The tools and API's used for development of these service features are listed in Appendix A.10. These namespaces have been partitioned so that no two classes have anything in common, but can be used together to create various workflows for forecasting. A list of service features used by each service is given in Appendix B.

The services are responsible for calling an appropriate service feature and also providing data to it. As the framework was first developed for wind forecasting, some of the service features were also reused by solar forecasting, which eventually led to reduced time and cost of development of services for solar forecasting. In all, 32 service features were developed for both wind and solar forecasting.

3.2. SOA based Services for Forecasting

The forecasting process for this work can be broadly categorized as project prospecting and operational forecast for wind and solar energy. Project prospecting gives a user an estimate of how much energy can be produced over a 12 month period at any given location, so that energy planners can decide if it is worth investing at the location or not. Project prospecting involves use of historical data from various sources to calculate the relevant forecast. Whereas, operational forecasting gives users a weekly forecast of energy produced at a location and is dependent on future predictions taking into calculations the forecasted wind speeds or cloud cover data.

The framework was divided into three WCF services that are used by the FaaS Controller, such that each service has a function to perform. EDCF is responsible for collecting data from external sources and storing them in Azure blob, IDRF standardizes the data and stores it in blob (if files need to be unzipped) and tables and FGF uses the standardized data to produce a forecast. The way each service performs its duty is by creating a workflow that involves using a combination of *service features* as explained in section 3.1.2.

3.2.1. Workflows for Forecasting

When service consumers submit a request, FaaS Controller decides which of the services out of EDCF, IDRF and FGF, will be participating in order to create the forecast. In section 3.1, it was discussed how the services can use service features to build a given functionality; in comparison to that, organizing a set of activities that need to be performed in order to execute a given task creates a workflow. The workflows are abstracted and exposed through the WCF service contracts; to a consumer of services they behave as a black box so they do not have to be aware of the internal specifics.

A WCF service is defined by an interface, which a developer can implement to perform tasks that are specific for them. This work implements workflows for solar and wind forecast, which can be accessed through well-established endpoints. Fig 3.4 below shows how different workflows can be created for two separate processes where each can use any number of service features. Some services use data that is available in internal storage, whereas others might interact with data from outside sources.

Each service including the FaaS Controller has two implementations of the interface contract, Solar and Wind, to create workflows for solar and wind forecasting, respectively. Further, each service also has different ABC's for their respective economic and technical endpoints. Depending on the endpoint invoked, corresponding instance of solar or wind is called to process a request as shown in figure 3.5.

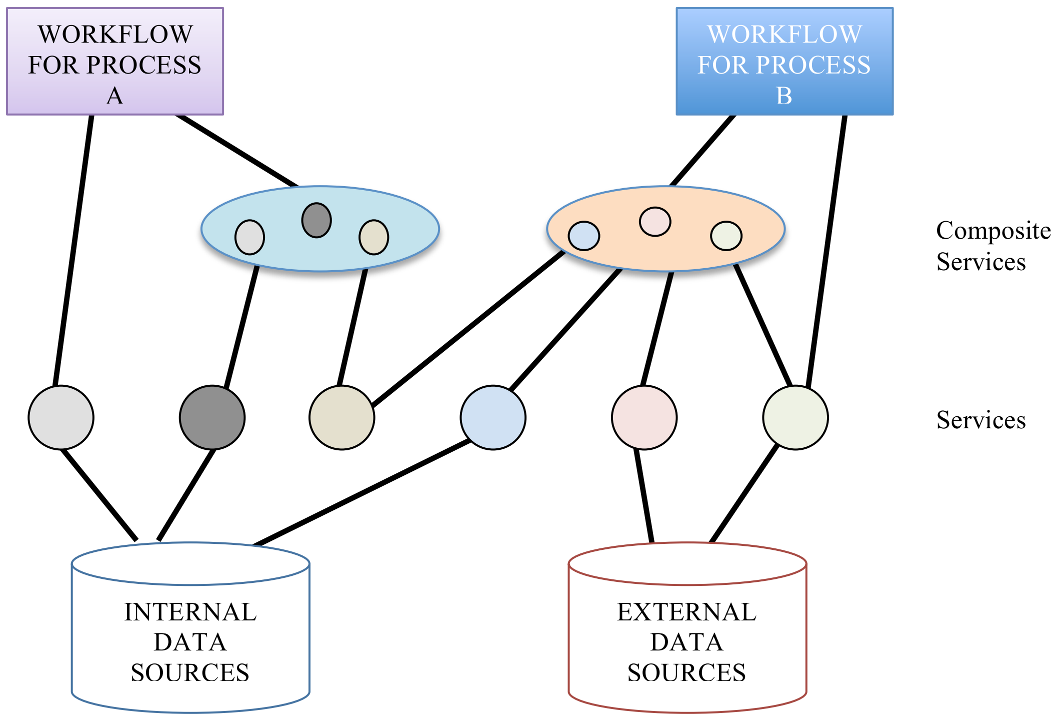


Figure 3.4. Creation of workflows for processes

A Metadata table contains a list of locations, for which historical data can be obtained from external sources. It essentially acts like a checklist of services and service features that the FaaS Controller must invoke in order to prepare a forecast. If the data for a particular location can be obtained from external sources, but was never downloaded to the internal storage of Azure, FaaS Controller requests EDCF to download the data and store it in the blob. Upon receiving a request, EDCF downloads the data and updates metadata table to point to the location in the blob where file is stored. This helps subsequent users save computation cost and time, as the data that has been downloaded once can be reused for other users.

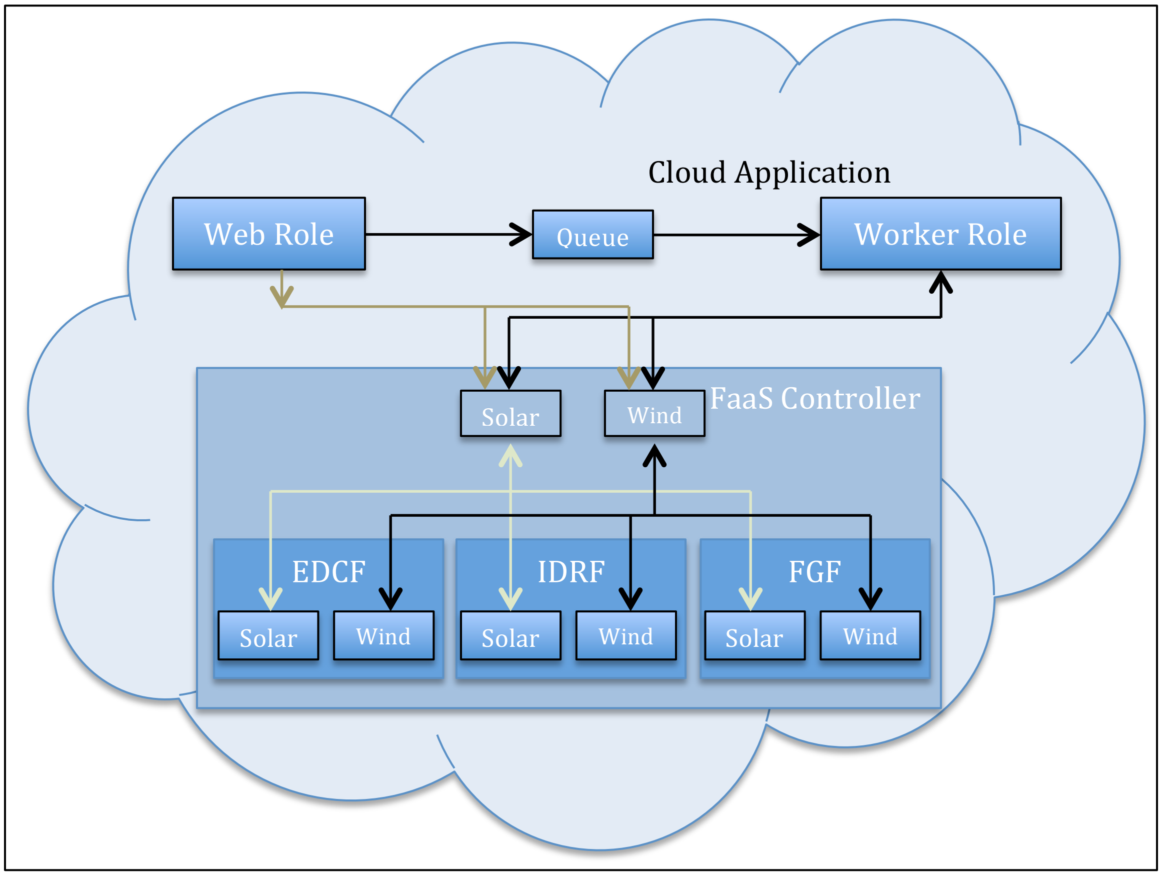


Figure 3.5. A high level view of the FaaS Controller in Azure

As data from different sources can be in different formats, it must be standardized from the raw format using various algorithms (discussed in Chapter 4), which is the responsibility of IDRF. IDRF does so by reading the data downloaded by EDCF from blob and converting it to a pre-decided format and storing the data as entities in Azure tables. When IDRF processes historical data, it updates the entry in metadata table to point to the location of the standardized historical data in data table.

FGF mainly deals with processing of the data standardized by IDRF and it uses various mathematical and statistical tools to generate a forecast. After successfully creating forecasts, it uses other service features to prepare a report and email results to the user. As services are decoupled from each other, they can be individually used in case a developer does not need other services. This essentially helps in offering these services a platform (PaaS) to develop composite services in forecasting.

3.3. SOA and Cloud Computing

Cloud computing is an abstraction of services and resources, where the underlying complexities of the technical implementations are encapsulated and transparent from user programs interacting with the cloud [Erl, T., et. al., 2010]. Using cloud, services created in an SOA fashion are highly reliable and can be scaled to support any number of users. Microsoft achieves this through *Windows Azure Fabric Controller* built on top of its underlying infrastructure that automatically manages resources and performs load balancing.

Windows Azure serves as an application container, where all the cost and logic is deployed. The PaaS delivery model enables the end users to focus more on their application and fabric controller takes control of the infrastructure. It brings applications to a world market, where developers have ready-to-use services and can select and pay only for the services they want.

Cloud services in Azure can run multiple instances of the code and the number of instances running can be controlled dynamically. Each WCF service discussed in Section 3.2 has multiple instances running to service simultaneous requests. Azure has different roles, namely, Web role and Worker role (discussed in Section 2.2.2) that represent the type of applications that are running. Each role runs on a *Virtual Machine*, which are created at the time of project deployment. Each service is deployed on a separate virtual machine and thus can be independently scaled up or down.

Figure 3.6 below presents a high level view of the FaaS. The Web role is used to host webpages so that the services of FaaS can be offered as SaaS. In Windows Azure Queue, messages are inserted in the queue at the back end by a Web role and read from the front end by a Worker role. When the request has been processed, these messages are deleted from the queue.

Azure provides easy access for storage and retrieval of data through blob and tables. FaaS uses separate metadata tables for solar and wind to access data stored in blob and data tables. Thus, each service can independently locate the data by looking up in the metadata table and operate on it. Due to such an approach, services can be used as a platform as they are self sufficient to perform their task.

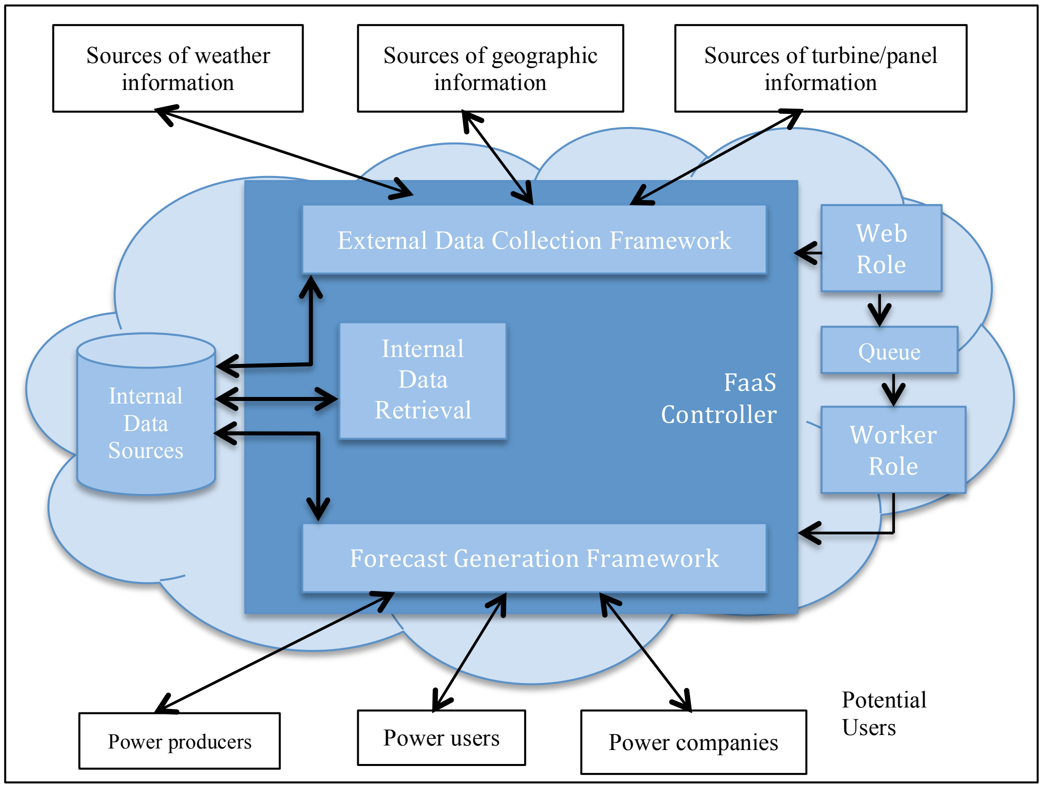

Sources of weather information
Sources of geographic information
Sources of turbine/panel information
External Data Collection Framework
Web Role
Queue
Internal Data Retrieval
Internal Data Sources
FaaS Controller
Worker Role
Forecast Generation Framework
Power producers
Power users
Power companies
Potential Users

Figure 3.6. An implementation of the FaaS with Microsoft Azure

In the next chapter, details of EDCF, IDRF, FGF, and the architecture of the FaaS Controller will be discussed in detail.

Chapter 4

Implementation of FaaS Framework

Discussion in Section 3.1 and Section 3.2 showed how the services provided by the FaaS could be consumed by WCF clients. A developer can also use these services as a platform to build composite services along with their own code or workflows. The FaaS can also be used as software for forecasting solar and wind power through the Web Role that consists of a series of web pages. So, in this chapter a detailed architecture of the FaaS and the services used by it will be discussed, followed by the reuse and extendibility of the framework.

4.1. Designing the Services

Discussion in Section 3.2 showed how the framework is divided in to three WCF services, namely, EDCF, IDRF and FGF and how the FaaS Controller uses them to perform forecasting. These services in-turn use various service features. The service features were designed such that they have minimum coupling between them and can be used along with other classes by a service to create workflows. This design philosophy was used to ensure that when services are reused or service features are updated, they are independent of the messages being exchanged between the services.

Each service feature is basically a class as explained in Section 3.1. Services can use multiple service features to achieve a desired functionality by instantiating an object of a service feature. Services or other service features can access data members by using the *getter* methods to retrieve data from them. As each class can operate only on its data members, data coming from services is serialized by the sending service and the receiving service deserializes it to a form that it understood by the service feature. The .NET framework greatly simplifies this process for WCF services, through the use of inbuilt services by converting data contracts to SOAP messages. The receiving service then assigns the received data to the data members of the class for them to operate on.

The access to Azure infrastructure was also converted to platform as a service, so the user does not have to maintain data tables or blob storage. A call to the service feature’s member function enables a developer to store and retrieve data to and from the blobs or tables. The storage service features developed for this purpose accept a wide range of data formats and have virtually no limitation on size.

Developers, who intend to extend these services, can use other cloud infrastructure for storage while still being able to use the services that perform forecasting.

4.2. Data Organization in Blobs and Tables

Forecasting processes deal with a huge amount of data, this is the reason an optimized metadata design is a very basic need for the framework to function properly. Azure blobs are capable of storing large binary data objects in them, so containers were created to store data together that belong to the same category. The raw data that is downloaded by EDCF service from external sources is downloaded to different containers depending on whether it is for solar or wind resources. The flow of data from Internet to the blob and from blob to tables is shown in fig 4.1.

A metadata table was used to store locations of every possible site for which data can be obtained from external sources. As each data source website provides a separate metadata file that includes location details, it can be used to create a metadata table. There are service features available in *TransferTools* that can learn where resources can be found by reading the HTML tags or an XML file, in case a separate metadata file is not available. The metadata tables, *SolarMetaDataTable* and *MetaDataTable,* have entities for solar and wind resources, respectively. Each entity in these tables is a location from the metadata file and has property fields that help EDCF determine where data for that location can be obtained.

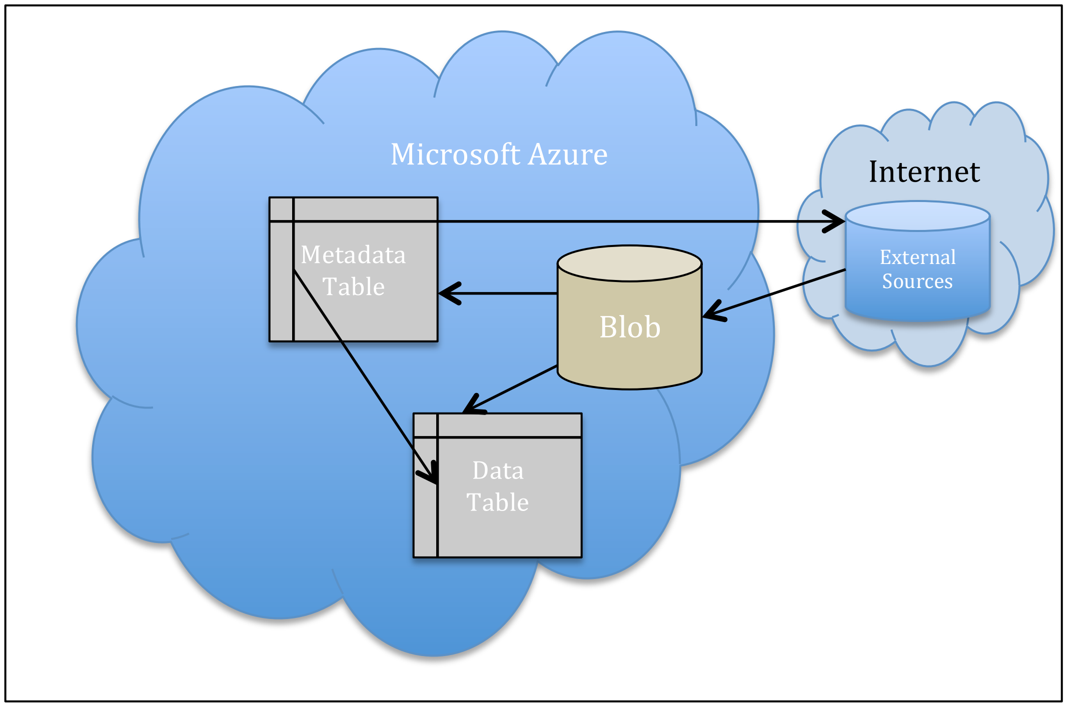


Figure 4.1. Data flow from or to blobs and tables

Fig. 4.2 shows a data contract for an entry corresponding to a location in the metadata table. The *SiteCategory* property helps determine which protocol (HTTP, FTP or XML) must be used to download the data. The *SiteID*, *StateCode*, *LocationName*, *Latitude* and *Longitude* information in the data contract can be used to create a URL to locate the resource on the Internet. The property fields *Data_In_Blob* and *Data_In_Table* are used by services to determine the location of the resource in Azure storage. As Azure tables require a *Source* and *Context* to be created that corresponds to the entities being stored in a table, these were created accordingly.

```
[Data Contract]
public class SiteMetaDataEntry
{
            public string SiteCategory { get; set; }
            public string StateCode { get; set; }
            public string SiteID { get; set; }
            public double Latitude;
            public double Longitude;
            public string Data_In_Blob { get; set; }
            public string Data_In_Table { get; set; }
            public string LocationName { get; set; }
}
```

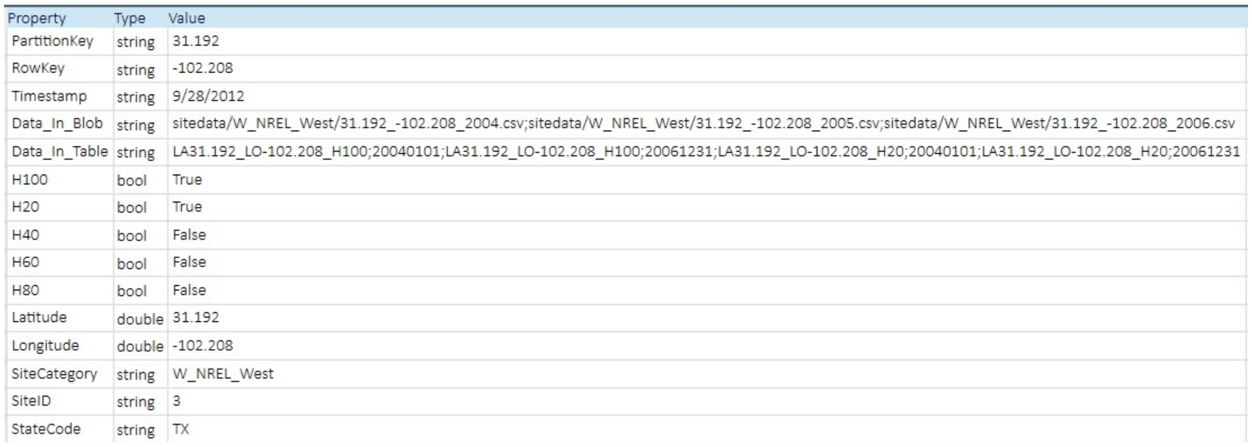
Figure 4.2 Code snippet showing data contract for metadata entry

In fig 4.3 is a code snippet from an operation contract in EDCF service that generates the names of HTTP URL for EDCF to download external resources dynamically. Given a base URL to the web resource, this service can learn how (protocol) and where (URL) the files reside on the Internet, and then it creates names of the files using the metadata information shown in fig. 4.2. When a user requests data for a location, these URL's are used by EDCF to download the actual historical data for those sites.

```
//Generate an HTTP URL ("sourceURL") for the a wind resource file for NREL website for
//location with "SiteID"
//Example: For a location with site id 1000 and data for year 2007, the file can be found at:
// "http://wind.nrel.gov/Web_nrel/data/2007/1000.csv"
string filename = windMetaDataEntry.SiteID + ".csv";
string baseHttpURL = "http://wind.nrel.gov/Web_nrel/data/";

string sourceURL = baseHttpURL + year + "/" + filename;

//Generate a blob URL where the file will be stored in the blob in container ""sitedata",
//based on wind metadata
string destinationURL = string.Format("sitedata/{0}/{1}_{2}_{3}.csv",
                                      windMetaDataEntry.SiteCategory,
                                      windMetaDataEntry.Latitude,
```

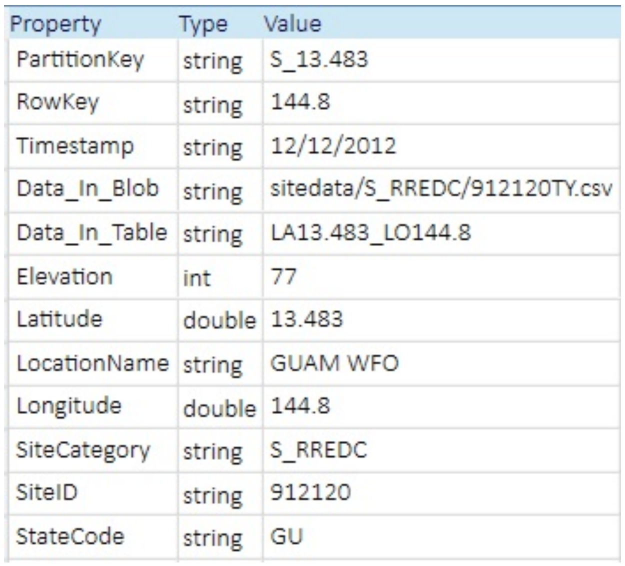
Figure 4.3 Code snippets shows generation of an HTTP URL for external data source

The downloaded data is then processed and stored in data tables, *SolarDataTable* and *DataTable*, for solar and wind respectively, by IDRF. As Wind and Solar metadata entries have different *source* and *context*, they also have different properties as shown in Fig 4.4 and Fig 4.5, respectively. These properties containing height or elevation data can be used by other services if required.



Property	Type	Value
PartitionKey	string	31.192
RowKey	string	-102.208
Timestamp	string	9/28/2012
Data_In_Blob	string	sitedata/W_NREL_West/31.192_-102.208_2004.csv;sitedata/W_NREL_West/31.192_-102.208_2005.csv;sitedata/W_NREL_West/31.192_-102.208_2006.csv
Data_In_Table	string	LA31.192_LO-102.208_H100;20040101;LA31.192_LO-102.208_H100;20061231;LA31.192_LO-102.208_H20;20040101;LA31.192_LO-102.208_H20;20061231
H100	bool	True
H20	bool	True
H40	bool	False
H60	bool	False
H80	bool	False
Latitude	double	31.192
Longitude	double	-102.208
SiteCategory	string	W_NREL_West
SiteID	string	3
StateCode	string	TX

Figure 4.4. Metadata entry for wind resource in *MetaDataTable*



Property	Type	Value
PartitionKey	string	S_13.483
RowKey	string	144.8
Timestamp	string	12/12/2012
Data_In_Blob	string	sitedata/S_RREDC/912120TY.csv
Data_In_Table	string	LA13.483_LO144.8
Elevation	int	77
Latitude	double	13.483
LocationName	string	GUAM WFO
Longitude	double	144.8
SiteCategory	string	S_RREDC
SiteID	string	912120
StateCode	string	GU

Figure 4.5. A typical metadata entry for a solar resource in *SolarMetaDataTable*

The information about wind turbine and solar panels were also stored in their respective metadata tables. For a detailed data schema of the metadata and data table for solar and wind resources, please refer to Appendix C.1.

4.3. Creating the Workflows

As each service creates workflows to perform their respective tasks, so in the next few sub-sections the creation of workflows by each service will be discussed.

4.3.1. EDCF Workflow for Forecasting

As previously stated, the primary purpose of EDCF is to collect data from external sources of information and store them in the Azure storage. If the data doesn't exist internally in blobs and tables, the FaaS Controller invokes EDCF's technical endpoint to capture data from different sources and store the raw data (.csv or .zip or other format) in Azure blob.

Shown in fig 4.6 is the service contract for interface *IEDCF* that allows download of data from external sources given latitude and longitude of a location through operation contract *DownloadDataToBlob*. If metadata doesn't exist for a location, consumers can also specify URL of the link explicitly by using the operation contract *DownloadHistoricalDataToBlob.* This makes the services provided by

EDCF to be reusable in case a developer in unable to obtain the required metadata file, but does know where resources can be found on the Internet.

```
namespace EDCF
{
  [ServiceContract]
  public interface IEDCF
  {
    [OperationContract(IsOneWay = true)]
    void DownloadDataToBlob(double latitude, double longitude);

    [OperationContract]
    string DownloadHistoricalDataToBlob(double latitude, double longitude, string URL);
  }
}
```

Figure 4.6 EDCF service contract interface that enables download for a location

The above service contract shows an interface that declares the operation contracts that are available in EDCF. This service contract is implemented by both, solar (*EDCFSolar*) and wind (*EDCFWind*) instances of EDCF. As the implementation of *EDCFWind* was done prior to the development of *EDCFSolar*, many service features developed for the wind instance were directly used to implement *EDCFSolar*. This shows that by sharing a single service contract, service development can have a positive impact of development of other services saving time and cost.

As discussed in section 3.1, the operation contract *DownloadDataToBlob* is a service operation that does not return any message, whereas the operation contract *DownloadHistoricalDataToBlob* returns a message to the caller that contains location of the file in the blob. The reason for this difference arises from the requirement that *DownloadHistoricalDataToBlob* is used to download files that are smaller in size (several hundred megabytes) and thus the underlying channel between the caller and called service can be closed within the default 1-minute channel timeout. Whereas *DownloadDataToBlob* can be used to download files that are larger in size (several gigabytes of order), so the caller might want to close the underlying channel after sending a one-way, no response message to prevent a time-out from occurring.

Downloading data from an external website, involves HTTP or FTP or XML to be implemented within the functionality of the service as a service feature. As there are multiple sources from where the files can be downloaded, each site has a different URL where the resource can be found and downloaded from. Due to the large number of locations and various available sources, these URL's cannot be known in advance and must be generated at run-time, thus creating a need for such service features. To generate URL's of external sources a detailed study of the metadata is essential, as discussed in section 4.2, and the design of metadata table is thus crucial for this service to generate correct URL's. *EDCFWind* and *EDCFSolar* both use the same service feature *HTTPDownload*, to download data.

The difference between the two is the generation of base URL by the service using *HTTPDownload*.

Fig 4.7 shows a general algorithm to obtain data from external sources. EDCF uses the metadata entry to generate a URL (as discussed in Section 4.2) and then uses a specific protocol to download files as a *Stream (.NET)* and stores them in the blob. After files are successfully saved in the blob, EDCF creates a blob URL corresponding to each file's location in the blob and updates the metadata entry with this link. This allows other services to directly operate on data from the blob by reading value of property *Data_In_Blob*. For a list of service feature used by each service and detailed algorithms used, please refer to Appendix B and Appendix C.8-C.9, respectively for operational forecast download.

1. Check metadata table for source of data using *SiteCategory* field.
2. Generate a URL where from the data file can be obtained.
3. Depending on the source of data, use FTP, HTTP or XML service feature to download the file.
4. Store the downloaded data to the blob using BlobWrite.
5. Generate a name for the stored file.
6. Update the Metadata table to point to the location of the resource in blob.

Figure 4.7 Algorithm to obtain data from external sources

It was found that download of historical data on the cloud can take anywhere from 2 minutes to 45 minutes depending on the size of file. So, in order to allow multiple downloads to be processed, a multi-threaded approach was taken. In addition, multiple instances of the EDCF service run in parallel to allow servicing of multiple requests from different users. Figure 4.8 shows a code segment to show how data is downloaded by EDCF using various service features developed in C# to create a workflow:

```
//Using different namespaces involved
using IOTools;
using TransferTools;

//Invoking a service feature
HTTPDataDownload httpDownloadClient = new HTTPDataDownload(httpURL);
Stream streamOfFile = httpDownloadClient.DownloadData();          //Download file as a stream

writeToBlob.WriteFileToBlob(destinationURL, streamOfFile);                    //Store file in Blob

UpdateWindMetaDataField(entry, "Data_In_Blob", destinationURL);   //Updating metadata entry
```

Figure 4.8 C# code to download data using HTTP

4.3.2. IDRF Workflow for Forecasting

IDRF is responsible for storing only relevant data from the raw data in tables. The raw data in time domain can be converted to frequency-domain for analysis. When a service request is received by IDRF to standardize data, property fields *Data_In_Blob and Data_In_Table* for an entry in the metadata table, *SolarMetaDataTable* or *MetaDataTable*, are read. IDRF then downloads required files from the blob, and uses other service features to store this data in data tables, *DataTable and SolarDataTable,* for wind and solar data, respectively. As this process needs to be done only once, the computation and data processing time gets reduced for subsequent users.

The data contract used by IDRF to communicate with the FaaS Controller or other services is same as shown in fig. 4.2 previously and is used by the FaaS Controller to check if IDRF has done standardization of data. For wind, IDRF uses the property field H100, H80, H60, H40 or H20 to check if the data has been standardized for a given height. If it is, *Data_In_Table* field from *MetaDataTable* can used to find location of the data in *DataTable*. For solar, location of data can directly be read from the *Data_In_Table* field from *SolarMetaDataTable*.

Fig 4.9 shows the interface with service contract for IDRF. The *IIDRF* service contract is same for solar and wind instances, similar to that of EDCF and thus they also share some common service features. The operation contract *GetMetaDataEntry* uses great circle distance (Appendix C.10) to return details of a location that is in close proximity to the user provided latitude and longitude. The location details are retuned in data contract *SiteMetaDataEntry*. The operation contract *StoreDataInTable* in *IIDRF* uses the metadata tables to locate resources in the blob and then uses other service features to standardize the data and store it in data table, whereas *StoreHistoricalData* can perform similar operation but only for a given period of time instead of an entire year.

```
namespace IDRF
{
   [ServiceContract]
   public interface IIDRF
   {
      [OperationContract]
      SiteMetaDataEntry GetMetaDataEntry(double latitude, double longitude, int height);

      [OperationContract(IsOneWay = true)]
      void StoreDataInTable(SiteMetaDataEntry entry, int height);

      [OperationContract(IsOneWay = true)]
      void StoreHistoricalData(string startDate, string endData, double latitude, double longitude);
   }
}
```

Figure 4.9. Service contract for IDRF showing various operation contracts

The above operation contracts create workflows as described in fig 4.10 below by using various service features. The data can be stored in time or frequency domain. Steps 7, 8, and 9 use algorithm in [Zhong and Tam, 2012] to convert raw data to frequency domain and store it in *DataTable* for wind.

IDRF as a framework has various capabilities exposed through the WCF contracts. A developer, to suit the needs of an individual, can further customize these service contracts by providing a different implementation and using custom code. For example, the capabilities of IDRF were extended to solar energy forecast, where the data was stored in time domain in data table, *SolarDataTable* and then used in forecasting. Appendix C.2-C.3 give detailed algorithms that are used for data standardization by IDRF for project prospecting, whereas Appendix C.8 and C.9 show algorithms used by IDRF for operation forecast.

1. Lookup Metadata table to check if data was already standardized.
2. If not, download file from the blob by using *Data_In_Blob* from metadata.
3. Check if the file is a comma separated, excel or text file. If yes, go to step 7.
4. If not, use service feature *unzip* to extract a csv or text file.
5. Upload the extracted file to the blob.
6. Update the metadata to point to the extracted file.
7. Read data from the file, serialize the data, and convert it to frequency domain.
8. Reduce the number of harmonics to minimum required (within 2% MAPE).
9. Store the reduced frequency domain data in *DataTable* for wind.
10. Update the metadata field *Data_In_Table* to the entities in the DataTable.
11. Repeat steps 3 to 10 for each year of data.

Figure 4.10 Algorithm to convert wind speed data from time to frequency domain

IDRF consumes maximum computation time, as most of the service features are used here in addition to the slower table and blob operations. Data standardization for yearly data is also performed here, which requires functionality that is not natively supported on the Azure platform (unzip, Microsoft Office, etc.), so service features were developed to create such functionality. A good use case for developing such service features is when an already existing desktop application needs to be used on the cloud. This is also corroborated by the fact that all service features were initially developed as C# desktop applications, where the input and output data were being read from and written to local hard disk. It was during the later stage of development that these features were seamlessly migrated to the cloud.

4.3.3. FGF Workflow for Forecasting

FGF works with data processed by IDRF, so service features in FGF are aware of the type of data they have to work with. It performs the forecasting operation on data stored in data table by using statistical and mathematical models. FGF is more flexible with the type of data it has to work, this requirement arises from the fact that others who intend to use these services as a platform, must be able to input data in a variety of format and still be able to get correct results.

In fig 4.11 is the service contract *IFGF* for FGF and the current implementation extends the interface shown here to forecast energy for solar (*FGFSolar*) and wind (*FGFWind*). The operation *ForecastPower* is one-way, so no response message is sent back to the caller that is, the FaaS Controller or other WCF service consumer. It accepts data that the caller of service wishes to pass on to it for generation of forecast including details such as solar panel or wind turbine data.

```
namespace FGF
{
   [ServiceContract]
   public interface IFGF
   {
            [OperationContract(IsOneWay = true)]
            void ForecastPower(string requestTime, double cost, double latitude, double longitude,
                        double distance, string locationName, string manufacturerName, string ID, int
                        height, bool UCQ, bool operationForecasting, string emailID);
   }
}
```

Figure 4.11 A service contract for FGF showing the operation contract to forecast

FGF uses the data passed in *ForecastPower* (latitude and longitude) to check the metadata table field *Data_In_Table* or *Data_In_Blob* to locate where the wind or solar data is stored in their respective data tables or blob. The data is then read from the table or blob, converted back to time domain if needed and statistical service features are applied to this data. To calculate the energy that can be generated, it takes in consideration the efficiency of solar panel or power curve of the wind turbine. If the data is in minutes, hourly data is calculated, which is again converted to daily energy values. Daily data can be summed up for monthly and yearly energy values.

Fig 4.12 shows a general algorithm that is used in case of solar and wind forecasting (with detailed algorithm in Appendix C.4 and C.6). Informing the user about the uncertainty in the forecasted value is of prime importance as it can have direct consequence on the reliability of the data, so service features were developed using various algorithms (given in Appendix C.5 and C.7) that were used to calculate the uncertainty values for wind and solar data respectively for project prospecting. Detailed algorithms for operation forecasting for wind and solar are given in Appendix C.8 and C.9.

1. Read the turbine or solar panel data chosen by the user from metadata table.
2. Read the wind speed or GHI data from the data table.
3. Calculate the energy produced per hour.
4. Add hourly energy values to calculate daily energy produced.
5. Add the daily energy values to obtain monthly energy produced.
6. Repeat step 3 to 5 for each month.
7. Repeat Step 3 to 6 for each year of data available.
8. Calculate three-year average of the monthly energy produced for each year.
9. Calculate uncertainty (P95, P90, P75 and P50) for monthly and yearly energy.
10. Create a report in the blob with information required by user.
11. Use email service feature to email the file in blob to the user.

Figure 4.12 Algorithm to generate energy forecast for wind or solar

4.4. Architecture of the Framework

The aim of the architecture presented in this thesis is to create a framework of SOA based services that can be used on a *pay-per-use* basis. The framework promotes the use of these services as a platform in forecasting. This is advantageous in a situation where a developer wishes to reuse the platform and still be able to implement their specific workflows. To achieve the use of platform as a service, these services must be accessible to the outside world through various endpoints.

Each service in FaaS exposes two endpoints, economic (EE) and technical (TE), which are used for different purposes, as shown in fig 4.13. The economic endpoint, as the name suggests, calculates the cost associated with using each service. The FaaS Controller uses the economic endpoint of each service to gather the cost of using them, each service in-turn adds up the cost of all service features that will be used by them to create a workflow. Each service returns their cost in a data contract and the FaaS Controller adds these costs and presents this cost to the user in another data contract.

After a user decides to use the services, the FaaS Controller uses the technical endpoints in order to create a workflow to perform actual forecast operation through EDCF, IDRF and FGF. Web and Worker roles or a WCF client can consume the services at their respective endpoints to perform a forecast.

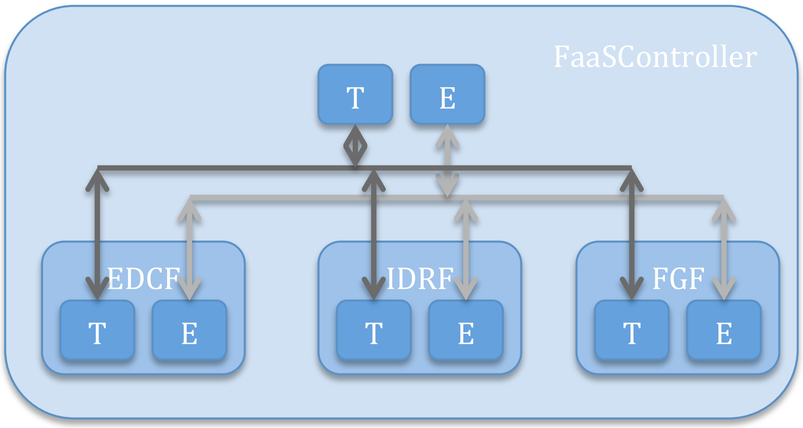


Figure 4.13 A model of the FaaS showing TE and EE

A detailed architecture of the framework is shown in figure 4.14 and it shows how various services and components of Azure interact with each other. Each service runs multiple instances to support servicing of simultaneous requests from users. This is where the scalable infrastructure of cloud computing comes into play as the number of instances these services are running can be scaled up to meet increasing demand or scaled down if the number of service consumer decreases, dynamically. This scalability also helps in reducing the turn-around time for a forecast request.

The Web role presents to a user webpages (shown in Appendix D) to interact with the framework. A user who wishes to know the forecast can input the coordinates of a location (latitude, longitude) and select whether they want to perform project prospecting, with or without uncertainty data or an operational forecast. To keep interaction minimal, a user can choose not to select turbine data or solar panel data so a default value can be applied.

The Web role uses economic endpoint of the FaaS Controller to find a location close to the user specified location and to estimate the cost of using services offered by FaaS. The FaaS Controller in-turn passes the location coordinates to IDRF through the operation contract *GetMetaDataEntry* of *IFaaSController* as shown in Fig. 4.15. IDRF refers to the metadata table and returns a location that has minimum great circle distance from what the user specified. These details are shown to the user on the next page along with the cost.

After user enters their email address and submits the request it is placed in a queue, *cloudservicequeue*. The Worker role reads these messages from *cloudservicequeue* and uses the technical endpoint of the FaaS Controller to process these requests. Using the property fields, *Data_In_Blob* and *Data_In_Table,* FaaS Controller can determine whether the data exists for that location and if data standardization has already been done.

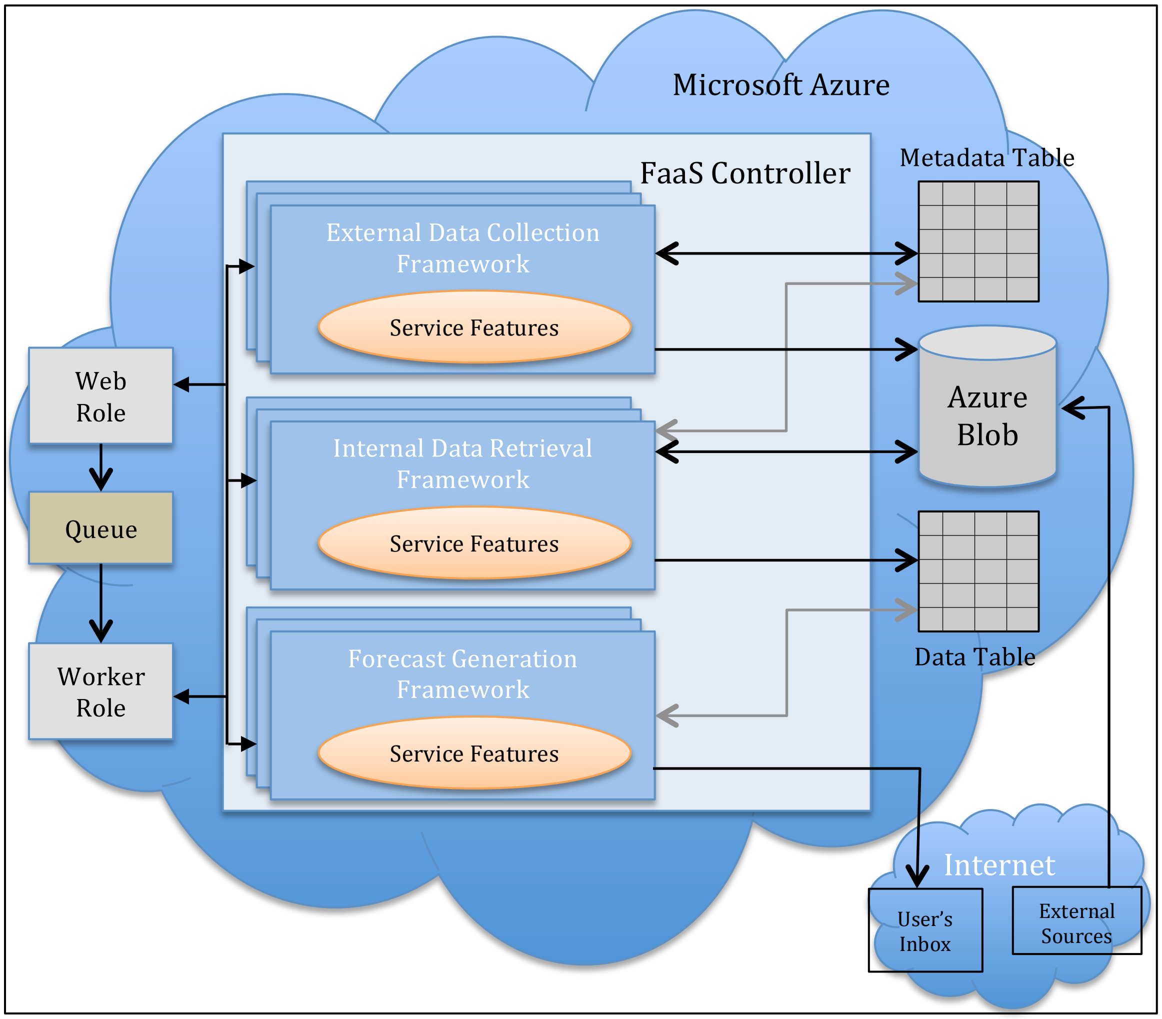


Figure 4.14 A detailed view of the architecture of the framework showing interaction of services inside and outside the cloud

Depending on the data received from IDRF by FaaS Controller, EDCF is invoked by the FaaS Controller if data needs to be downloaded or/and IDRF to store the data in tables. Finally, FGF is invoked to generate forecast for that location with required information in the operation contract *ForecastPower* in IFGF. The details of how EDCF, IDRF and FGF work have already been discussed in section 4.4.1, 4.4.2 and 4.4.3 respectively. After requests are processed, they are deleted from the queue. As each service runs multiple instances, in case if any of the service is not required for a request, the FaaS Controller can directly forward the request to the next service, thus shortening servicing time. For the purpose of FaaS, the number of instances was set statically to three.

```
namespace FaaSController
{
   [ServiceContract]
   public interface IFaaSController
   {
      [OperationContract(IsOneWay = true)]
      void DownloadData(double latitude, double longitude);

      [OperationContract]
      string DownloadHistoricalData(double latitude, double longitude, string URL);

      [OperationContract]
      SiteMetaDataEntry GetMetaDataEntry(double latitude, double longitude, int height);

      [OperationContract(IsOneWay = true)]
      void StoreDataInTable(SiteMetaDataEntry entry, int height);

      [OperationContract(IsOneWay = true)]
      void StoreHistoricalData(string startDate, string endData, double latitude, double longitude);

      [OperationContract(IsOneWay = true)]
      void ForecastPower(string requestTime, double cost, SiteMetaDataEntry entry, string locationName,
                         string manufacturerName, string turbineModelName, int height, bool UCQ,
                         bool operationalForecasting, string emailID);
   }
}
```

Figure 4.15 Service contract for Technical Endpoint (TE) of *IFaaSController* showing various available operation contracts

Shown in fig 4.16 is a detailed view of call flow for solar project prospecting. Also shown in figure are various services and service features that are involved in the workflow. The flow of data and control in the figure is from top to bottom, starting with FaaS Controller. FaaS Controller acts essentially as a client for other services; depending on which instance (solar or wind) is called by the user it invokes an appropriate service contract. The implementation of the service contract uses service features shown in the figure in a given sequence to carry out the forecast. While EDCF involves interaction with the metadata table and the blob; IDRF uses blob, metadata and data table. FGF's interaction is limited to the data table and blob only.

Fig 4.17 shows creation of workflow by the FaaS Controller in case of operational forecasting for wind and solar. Similar to project prospecting, economic endpoints are used to estimate the cost of using the services. After users submit their request, technical endpoints are used to forecast operational data for the next day. As operational forecast uses some service features developed for project prospecting, the cost and effort in development was significantly less.
Here, *XMLDataDownload* service feature is used to obtain data, but the data is not stored in the blob it is liable to change frequently. Also, as data is not stored, IDRF is only limited to interpreting the XML data and passing it to FGF in another service contract for using it in forecasting. A detailed algorithm for solar and wind operational forecasting is provided in Appendix C.8 and C.9, respectively.

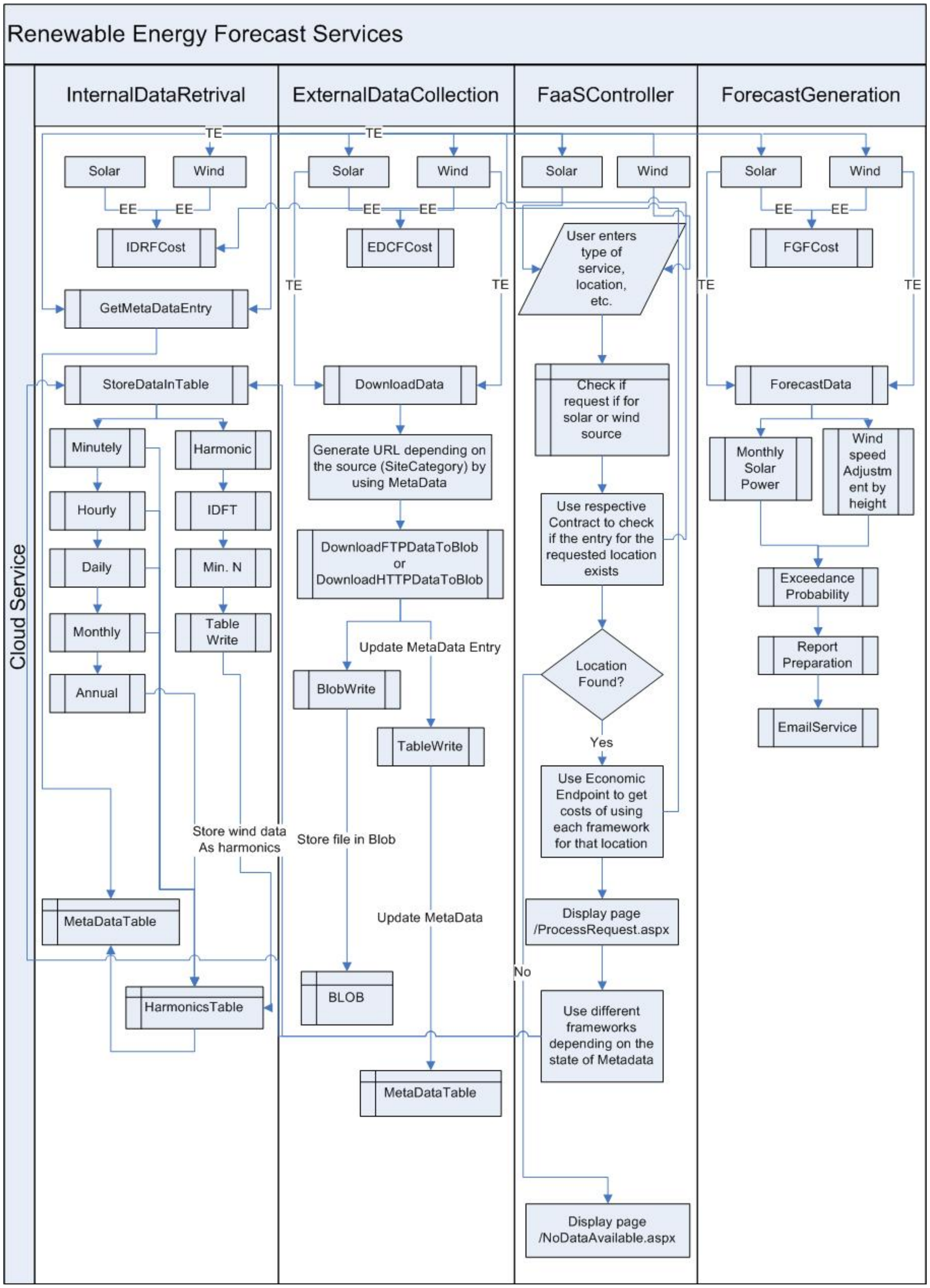

Renewable Energy Forecast Services
Cloud Service
InternalDataRetrival
ExternalDataCollection
FaaSController
ForecastGeneration
TE
EE
Solar
Wind
IDRFCost
GetMetaDataEntry
StoreDataInTable
Minutely
Hourly
Daily
Monthly
Annual
Harmonic
IDFT
Min. N
Table Write
Store wind data As harmonics
MetaDataTable
HarmonicsTable
EDCFCost
DownloadData
Generate URL depending on the source (SiteCategory) by using MetaData
DownloadFTPDataToBlob or DownloadHTTPDataToBlob
Update MetaData Entry
BlobWrite
TableWrite
Store file in Blob
Update MetaData
BLOB
MetaDataTable
User enters type of service, location, etc.
Check if request if for solar or wind source
Use respective Contract to check if the entry for the requested location exists
Location Found?
Yes
No
Use Economic Endpoint to get costs of using each framework for that location
Display page /ProcessRequest.aspx
Use different frameworks depending on the state of Metadata
Display page /NoDataAvailable.aspx
FGFCost
ForecastData
Monthly Solar Power
Wind speed Adjustment by height
Exceedance Probability
Report Preparation
EmailService

Figure 4.16. A high level view of the FaaSController for solar project prospecting showing the use of Technical Endpoint (TE) to create workflows

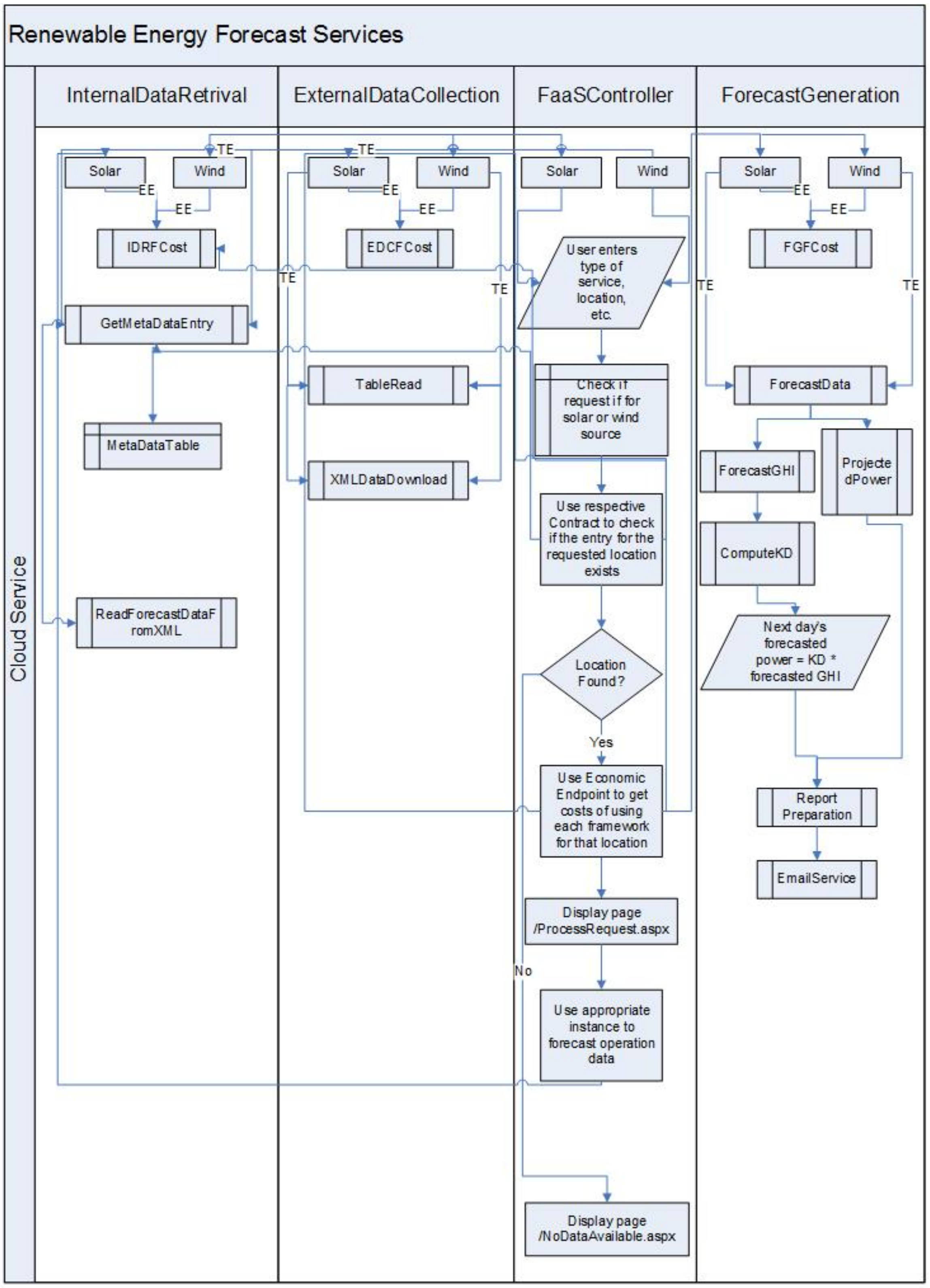

Renewable Energy Forecast Services
Cloud Service
InternalDataRetrival
ExternalDataCollection
FaaSController
ForecastGeneration
Solar
Wind
TE
EE
IDRFCost
EDCFCost
FGFCost
GetMetaDataEntry
MetaDataTable
ReadForecastDataFromXML
TableRead
XMLDataDownload
User enters type of service, location, etc.
Check if request if for solar or wind source
Use respective Contract to check if the entry for the requested location exists
Location Found?
Yes
No
Use Economic Endpoint to get costs of using each framework for that location
Display page /ProcessRequest.aspx
Use appropriate instance to forecast operation data
Display page /NoDataAvailable.aspx
ForecastData
ForecastGHI
ProjectedPower
ComputeKD
Next day's forecasted power = KD * forecasted GHI
Report Preparation
EmailService

Figure 4.17. A high level view of the FaaSController for solar operational forecasting showing the creation of workflow using Technical Endpoint (TE)

4.5. FaaS Service Reusability

The reuse of FaaS can be viewed in terms of service and service features. By redefining the service and operation contracts, developers can provide their own implementation of the services. The service features that are available as library files can also be used as a platform to achieve a given functionality.

The reuse of EDCF and IDRF as a platform was successfully demonstrated to download historical solar data containing cloud cover (details can be found in Appendix E.1). Fig 4.18 shows the redefined interface contract for EDCF that uses the operation contract, *DownloadHistoricalDataToBlob*, while still using the same service features *HTTPDataDownload* and *WriteDataToBlob*. The operation contract in *IEDCF* and it's implementation in *EDCFSolar* is capable of downloading TAR files from the Internet [NWS, 2013] provided an HTTP URL giving the location of the resource as shown in fig 4.19.

```
namespace EDCF
{
  [ServiceContract]
  public interface IEDCF
  {
    [OperationContract(IsOneWay = true)]
    void DownloadDataToBlob(double latitude, double longitude);

    [OperationContract]
    string DownloadHistoricalDataToBlob(double latitude, double longitude, string URL);
  }
}
```

Figure 4.18. Redefined service contract shows download of data using a user provided URL to the resource

```
public string DownloadHistoricalDataToBlob(double latitude, double longitude, string httpURL)
{
        //Generate a filename for blob
        var parts = httpURL.Split('/');
        string destinationURL = "sitedata/HistoricalData/" + parts[parts.Length - 1];

        try
        {
          //Use HTTP to download the file
          httpDownloadClient = new HTTPDataDownload(httpURL);
          Stream str = httpDownloadClient.DownloadFileToStream();
          //Write file to the Blob
          writeToBlob.WriteFileToBlob(destinationURL, str);
}
```

Figure 4.19. Code showing implementation of *DownloadHistoricalDataToBlob* using already existing service features

The redefined operation contract allows a user to download files without using the metadata table, as opposed to the case of *DownloadDataToBlob*. After files are downloaded from the Internet, the operation returns a URL to the file in the blob. This blob URL can be used by other services, such as IDRF or FGF or other custom services to process data.

By introducing a new operation contract, *StoreHistoricalData*, as shown in fig 4.20, IDRF's capabilities were reused. New service features were created in order to help IDRF to process the data downloaded from NWS. As NWS uses a custom compression format to store historical data files and provides an executable file to decompress these files. Separate service features like, *UntarFile and Degribber*, were developed that un-tar a file and use the exe file to decompress data files present in the blob, respectively. A detailed code for the implementation of *StoreHistoricalData* in *IDRFSolar* is given in Appendix E.1.

```
namespace IDRF
{
  [ServiceContract]
  public interface IIDRF
  {
    [OperationContract]
    SiteMetaDataEntry GetMetaDataEntry(double latitude, double longitude, int height);

    [OperationContract(IsOneWay = true)]
    void StoreDataInTable(SiteMetaDataEntry entry, int height);

    [OperationContract(IsOneWay = true)]
    void StoreHistoricalData(string startDate, string endData, double latitude, double longitude);
  }
}
```

Figure 4.20. Redefined service contract shows *StoreHistoricalData* operation contract that decodes data from NWS

The Historical Data Download allows a user to specify the dates, as shown in Appendix E.1, for which they want raw (encoded) or uncompressed data. As in this case the user was interested in downloading and standardization of data, only EDCF and IDRF were used and charged for. The implementation of historical data was only done for solar, but can also be extended to wind historical data through *EDCFWind* and *IDRFWind*.

4.6. Extending the Forecasting Framework

Initially, the framework was developed for wind forecasting and further extended to solar forecasting. As the services were build using SOA the time of development for other services was greatly reduced because of service and service feature reuse, thus also reducing the cost of development. The service features that provide access to cloud infrastructure were particularly of much use from this aspect, as the usual process to access Azure storage requires additional development effort.

The services in FaaS can be used as a Platform-as-a-Service to develop further services in forecasting. A developer, who needs to reuse the services, can add relevant endpoints of EDCF, IDRF or FGF as a service reference to their project, in addition to custom service features and code. The extension of FaaS is possible primarily due to the interface contract defined for each service. By redefining service contracts using the service features already developed for wind and solar to create a workflow, a developer can extend the use of FaaS to other domains in forecasting.

Appendix E.2 discusses technical details of extending the framework for electrical load forecasting by using the already existing service features. For extending FaaS in other domains, each service needs to be extended with custom workflows that are created by using service features.

A metadata table must be created in order for the services to be able to download and process data. The interface contracts, namely, *IEDCF*, *IIDRF* and *IFGF*, must be provided with an implementation *EDCFElectric*, *IDRFElectric* and *FGFElectric*, similar to the ones for solar and wind, respectively. As electrical load forecasting might need some of the functionality that does not already exist as a service feature, this must accordingly be created. Each operation contract in their respective interface contracts must be defined using service features.

The new services now must be hosted on the cloud and their technical and economic endpoints be added to FaaS Controller. A Web role and a Worker role can also be created to access these services. The Web role can use the economic endpoint to display cost and other data, through a webpage similar to one presented in Appendix D. The Worker role should use the technical endpoint to process data.

Chapter 5

Cost Estimation for Services in FaaS

The cost estimation framework is an integral part of the framework. In this chapter the cost model and the pricing strategy that was used during the development and implementation of the services in FaaS will be discussed.

5.1. Cost Estimation

Billing a customer is a major part of the infrastructure on cloud. Enterprise offering cloud services should be able to charge end customers, while simultaneously being able to make profit out of their offerings. Also, to be fair to end customers, a customer should be made to pay only for the services used by them, similar to the *pay-per-use* model followed by many service providers.

Fixed pricing in utility computing is not fair to both service provider and customer, so users must be charged based on their actual resource consumption, level of services used and the Quality of Service (QoS) provided to them [Yeo, C.-S., et. al., 2010]. As utility computing services reduce the risk faced by customers because the costs to the customer are proportional to the volume of transactions performed during a certain time interval, this leads ultimately in an increase in traffic for the service. Cost estimation in SOA can be done based on many parameters. Number of man-hours spent in development, number of original services created, numbers of services re-used, composite services, etc. all affect the cost of a service in SOA.

As the offerings presented in this thesis are not for commercial purpose, the model presented here only takes into consideration the cost of cloud infrastructure and the development cost of services to discuss a possible solution to charging. Even though Windows Azure does not provide the necessary API's to bill a customer as per the resources they consume on the cloud, such functionality can be developed with the help of different usage tracking tools and cost models.

5.2. Pricing Models

As discussed in section 2.4, the cost of services for FaaS and the cloud infrastructure are designed in such a way that a user of service is charged only for the services they use instead of paying a license fee in traditional software. The following cost models were taken in consideration to decide the pricing strategy for FaaS.

Estimation of cost and effort for software developed using SOA is more challenging than traditional software due to architectural differences. To overcome such challenges, [Li, Z., and Keung, J., 2010] proposed a framework based on Divide-and-Conquer as even COCOMO II [Boehm, B.W., et. al., 2000] can generally prove to be inadequate for reusable SOA services even though it has a large number of coefficients. As "the advantages of SOA are mainly reusability and composability with an emphasis on extensibility and flexibility, at a high level of granularity and abstraction" [Boehm, B.W., et. al., 2000], the authors propose that services in SOA can naturally be divided into a set of loosely coupled services and thus their costs too as shown in fig 5.1. So, by recursively decomposing a problem into simpler sub problems and then providing solutions to these sub problems leads to a solution that comprises of all these sub solutions.

To determine the cost of these sub problems, Composable Cost Model [Truong, H.-L., et. al., 2010] presents techniques to estimate costs for services and costs associated with scientific applications. It builds an estimate of the cost by taking into consideration infrastructure pricing available from cloud providers like, storage, data transfer size, and data transfer between source and destination. In addition, it also monitors the cost of execution of certain workflows. For this purpose, the authors [Truong, H.-L., et. al., 2010] assume that execution time, and other such parameters related to computing can be obtained. Autonomic Metered Pricing [Yeo, C.-S., et. al., 2010] discusses how self-adjusting pricing parameters can be beneficial for both, the service provider and the end users. Whereas, Price-at-Risk [Paleologo, G. A., 2004] explicitly takes into account multiplexing gain to decrease the average costs, at the same time taking advantage of the market information.

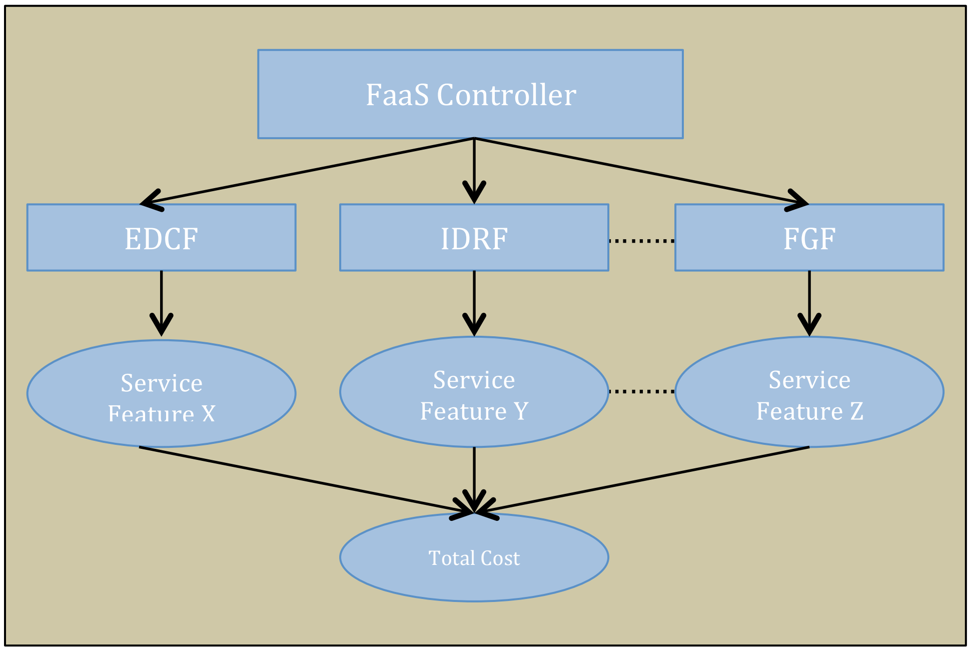


Figure 5.1. Principle of Divide and Conquer for FaaS

Out of all the above models, composable cost model influenced the development of the FaaS costing model. As FaaS is offered as a platform, so the parameters that other authors assume to build their model could not be directly applied in this case. This was because Microsoft Azure does not provide access to parameters such as computation power being used by a single process or service.

For FaaS, the cost estimation for SOA services and cloud infrastructure were done in a manner such that each service feature presents its cost to the service that is using it. The service then sums up the costs of all service features it is using to give a total service cost. The cost returned from each WCF service changes with the number of service features that are used to prepare the forecast. Finally, the FaaS Controller sums up the cost of these services to calculate a composite cost. The FaaS costing model uses parameters such as cost of developing SOA based services and the infrastructure costs as described in the next sub-sections.

5.2.1. Infrastructure Cost

Infrastructure cost is the cost that is incurred from using the cloud platform, which in case of FaaS is Microsoft Azure. This cost was calculated by taking into consideration the following factors that cloud service providers generally provide:

- Cost of bandwidth per GB per month (C_{bw})
- Cost of storing data in blob per GB per month (C_b)
- Cost of table transaction (C_t)
- Compute cost per month (C_C)

Using C_b total cost of bandwidth per GB per year can be obtained as follows:

$$\text{Cost of bandwidth per GB per year, } C_{ybw} = C_{bw} * 12$$

Assuming the maximum amount of data that will be stored in blob to be D_{max}, the total cost from bandwidth usage is given by eq. 5.1.

Total bandwidth cost per year, $T_{BW} = C_{ybw} * D_{max} = 12 * C_{bw} * D_{max}$ …(5.1)

Let, the number of sites the data can be obtained for be N_N, and the number of files that are required to be stored for each location be N_L where size of each file is S_f MB.

Total cost of storing files in blob, $T_B = N_N * N_L * S_f * C_b$ ...(5.2)

Assuming the number of entities per location to be N_{TL}, cost of table transaction becomes:

Total cost of table transaction, $T_{TL} = N_{TL} * C_t$ ...(5.3)

To calculate the total computation cost, let the number of Web or Worker roles required to be N_r:

Total cost computational cost, $T_C = N_r * C_C$ ...(5.4)

The total cost of using any cloud computing platform for FaaS then is given by eq. 5.5, assuming average number of users to be N_u.

Total cost of using cloud infrastructure per year, $T_I = (T_{BW} + T_B + T_{TL} + T_C)$ …(5.5)

Total cost of infrastructure to user, $T_U = T_I / Nu$ …(5.6)

Appendix C.2 shows the cost for deployment of FaaS on Microsoft Azure with assumptions in C.3.

5.2.2. Cost of Service Features

The labor cost associated with each service feature was calculated as given by equation 5.7 and is a product of man-hours spent on development and testing of service features and the cost (USD) per man-hour:

Labor cost = (total man-hours spent * cost per man-hour) ...(5.7)

The indirect cost is the effort spent on designing the service features and is given by equation 5.8 (overhead assumed to be 70%):

Total indirect cost = Labor cost * overhead rate ...(5.8)

So, the total labor cost is a sum of labor cost and indirect cost:

Total labor cost = Labor cost + Total indirect cost ...(5.9)

The final cost of a service feature, assuming the number of users that will use each feature to be N_U, then is given by eq. 5.10:

Final cost to user for using a service feature = Total labor cost / N_U ...(5.10)

Appendix C.1 shows the cost of service features developed for FaaS and their respective break-up as per eq. 5.7, 5.8, 5.9 and 5.10.

5.2.3. Cost of Services

The cost of services can be obtained from their respective Economic Endpoints

(EE). Based on the instance of EDCF called (Solar or Wind), it sums up the cost of all service features EDCF will use in order to download data. The cost always includes the cost of using *TableRead*, *TableWrite* and *BlobWrite*, as these are the service features that must be used. This is because EDCF must read or write from or to table, respectively and store downloaded files in the blob. An additional cost for using *HTTPDataDownload* or *FTPDataDownload* or *XMLDataDownload* will also be incurred. Table 5.1 shows various types of costs that are involved during wind or solar project prospecting. Let C_1 be the cost that will be presented at the EE.

Table 5.1. Costs incurred from service features used during project prospecting

Feature Name	**Costs**
TableRead	2.56
BlobWrite	1.47
TableWrite	2.43
HTTPDownload	4.09
FTPDataDownload	2.55
XMLDataDownload	1.27
Total EDCF Cost (C_1) in USD	**10.55 or 9.01 or 7.73**

Similarly, IDRF has solar and wind instances and they use different service features whose costs are added by IDRF and returned at the EE (a detailed breakup of the costs associated with IDRF will be discussed in chapter 6). In addition to the cost of service features, infrastructure costs from section 5.2.1 (actual costs given in Appendix C.2) are added by IDRF as it deals with data storage in tables and in blobs (unzipped files). Let the cost returned by IDRF be C_2.

The cost incurred for using FGF can be broken down into service features used by it. Common costs are incurred by both instances (Solar and Wind) of FGF and include the cost for *BlobRead*, *BlobWrite* and *ReportPreparationCost*. If a user requested uncertainty data that cost is also added. Solar and wind project prospecting use different service features. Solar instance only reads data from tables and writes data to tables, whereas wind forecasting needs turbine data and uses *PowerForecast* and *RawWindSpeedData* features to produce a forecast as shown in table 5.2. Let C_3 or C_4 be the final cost of using FGF as a service.

Table 5.2. Breakup of costs involved during solar and wind project prospecting

	Service Feature	**Costs (in USD)**
Common Costs incurred by all instances	ReportPreparationCost	2.55
	BlobRead	0.96
	BlobWrite	1.47
Uncertainty Data	ExceedanceProbabilityCost	1.91
	CoefficientOfVariationCost	0.67
Solar	TableReadCost	2.56
	TableWriteCost	5.11
Wind	TurbineDataCost	2.56
	PowerForecastCost	5.11
	RawWindSpeedDataCost	7.67
Total FGF Cost for solar project		**12.56 or 15.2**

prospecting without (C_3) or with uncertainty data in USD	
Total FGF Cost for wind project prospecting without (C_4) or with uncertainty data in USD	**20.32 or 22.87**

5.2.4. Total Cost of Composite Service

As FaaS Controller is a composite service based on EDCF, IDRF and FGF, it invokes the EE of these services to know their respective costs. Data for each location is downloaded by EDCF only once, and standardized and stored in data tables by IDRF. So, if other users request forecasts for the same location, the FaaS Controller need not invoke the technical endpoints of these two services but only for FGF. To be fair to every user, all three costs are added to obtain a final cost of using FaaS Controller irrespective of whether they will be used or not. So,

$$\text{Total Cost} = C_1 + C_2 + C_3 \text{ or } C_4$$

5.3. Cost Estimation Framework for FaaS

As discussed in section 5.2.3, the cost of using services can be obtained using their respective economic endpoint ABC's. Each service has service contract for economic endpoint that is defined for every instance of service, which for FaaS are Solar and Wind. So, each operation contract has two definitions for Solar and Wind.

The service contract for EDCF defined by the interface *IEDCFServiceCost* has operation contracts *DownloadDataToBlobCost* and *DownloadHistoricalDataCost*, as shown in fig 5.2 below. The implementation of this interface, *EDCFWind* and *EDCFSolar*, adds up of the costs of service features that will be used in order to download data. Depending on whether FTP, HTTP or XML service feature will be used, an appropriate cost presented at the EE as a *doubl*e data type.

```
//Economic endpoint contract
namespace EDCF
{
  [ServiceContract]
  public interface IEDCFServiceCost
  {
    [OperationContract]
    double DownloadDataToBlobCost(double latitude, double longitude, bool operational);

    [OperationContract]
    double DownloadHistoricalDataToBlobCost(string startDate, string endDate);
  }
}
//Technical endpoint contract
namespace EDCF
{
  [ServiceContract]
  public interface IEDCF
  {
    [OperationContract(IsOneWay = true)]
    void DownloadDataToBlob(double latitude, double longitude);

    [OperationContract]
    string DownloadHistoricalDataToBlob(double latitude, double longitude, string URL);
  }
}
```

Figure 5.2 Service contracts for EDCF showing a comparison between TE and EE

Depending on the instance of service called, the cost for workflow of solar or wind is returned by the economic endpoint. In contrast to the economic endpoint, lower half of fig 5.2 shows the technical endpoint (TE) service contract, *IEDCF* and uses service features to download data. The operation contract *DownloadDataToBlob* is a one-way contract, as downloading data through this operation takes longer than *DownloadHistoricalData* as discussed in Section 4.3.1.

Similarly, service contract for economic endpoints for IDRF, *IIDRFServiceCost,* is shown in upper half of fig 5.3. The operation contracts, *InternalDataStorageCost* and *HistoricalDataStorageCost* return the cost as a *double* data type to the calling service. *IDRFWind* and *IDRFSolar* contain the implementation of this contract and accordingly add-up the cost of service features that are involved. The technical endpoint service contract, *IIDRF*, as discussed in section 4.3.2, was used to perform data standardization. The difference between the EE and TE operation contracts being that the TE does not return any message to the calling service and is implemented as a one-way contract.

Figure 5.4 shows the service contracts, *IFGFServiceCost* and *IFGF* for economic and technical endpoint of FGF respectively. *FGFSolar* and *FGFWind* implement both these contracts. The operation contract *PowerForecastCost* in *IFGFServiceCost* returns cost as discussed in Section 5.2.3 for either solar or wind instance. Whereas the operation contract *ForecastPower* in IFGF performs the actual forecasting using various service features.

```
namespace IDRF
{
  [ServiceContract]
  public interface IIDRFServiceCost
  {
    [OperationContract]
    double InternalDataStorageCost(SiteMetaDataEntry entry, bool operational);

    [OperationContract]
    double HistoricalDataStorageCost(string startData, string endData);
  }
}
namespace IDRF
{
  [ServiceContract]
  public interface IIDRF
  {
    [OperationContract]
    SiteMetaDataEntry GetMetaDataEntry(double latitude, double longitude, int height);

    [OperationContract(IsOneWay = true)]
    void StoreDataInTable(SiteMetaDataEntry entry, int height);

    [OperationContract(IsOneWay = true)]
    void StoreHistoricalData(string startDate, string endData, double latitude, double longitude);
  }
}
```

Figure 5.3 Economic and technical endpoint service contracts for IDRF

```
namespace IFGF
{
  [ServiceContract]
  public interface IFGFServiceCost
  {
    [OperationContract]
    double PowerForecastCost(bool operationForecast, bool UCQ);
  }
}

namespace IFGF
{
  [ServiceContract]
  public interface IFGF
  {
         [OperationContract(IsOneWay = true)]
         void ForecastPower(string requestTime, double cost, double latitude, double longitude,
                  double distance, string locationName, string manufacturerName, string ID, int
                  height, bool UCQ, bool operationForecasting, string emailID);
  }
}
```

Figure 5.4 Service contracts for EE and TE for FGF

FaaS Controller being a composite service also has EE and TE bound by the service contracts *IFaaSControllerCost* and *IFaaSController* shown in fig 5.5. The operation contracts *ForecastCost* and *HistoricalDataCost* in *IFaaSControllerCost*, return the cost to the calling service as a *double* data type. In the implementation of these operations, it invokes the economic endpoints of EDCF, IDRF and FGF to obtain their respective costs, which are then added and returned to the caller.

In comparison to the economic endpoint, fig 5.6 shows the technical endpoint service contract for FaaS Controller. Technical endpoints of services, EDCF, IDRF and FGF are invoked in the implementation of this service contract in *FaaSControllerWind* or *FaaSControllerSolar*. Similar to their respective operation contracts in EDCF, IDRF and FGF, the contracts in *IFaaSController* are also one-way.

```
namespace FaaSController
{
  [ServiceContract]
  public interface IFaaSControllerCost
  {
    [OperationContract]
    double ForecastCost(SiteMetaDataEntry entry, bool operationForecast, bool UCQ);

    [OperationContract]
    double HistoricalDataCost(bool decodedData, string startData, string endData);
  }
}
```

Figure 5.5 Service contract for economic endpoint for FaaS Controller

```
namespace FaaSController
{
  [ServiceContract]
  public interface IFaaSController
  {
    [OperationContract(IsOneWay = true)]
    void DownloadData(double latitude, double longitude);
    [OperationContract]
    string DownloadHistoricalData(double latitude, double longitude, string URL);

    [OperationContract]
    SiteMetaDataEntry GetMetaDataEntry(double latitude, double longitude, int height);

    [OperationContract(IsOneWay = true)]
    void StoreDataInTable(SiteMetaDataEntry entry, int height);
    [OperationContract(IsOneWay = true)]
    void StoreHistoricalData(string startDate, string endData, double latitude, double longitude);

    [OperationContract(IsOneWay = true)]
    void ForecastPower(string requestTime, double cost, SiteMetaDataEntry entry, string locationName,
                       string manufacturerName, string turbineModelName, int height, bool UCQ,
                       bool operationalForecasting, string emailID);
  }
}
```

Figure 5.6 Service contract for technical endpoint of FaaS Controller

Shown in fig 5.7 below is the cost flow for solar operational forecasting using economic endpoints. The factors that affect the cost of services are manifold and include the labor cost; number of years the data is available for, computation time, storage used, etc. as discussed in Section 5.2. When users select the operations they want to perform, the economic endpoint service contract, *IFaaSControllerCost* invokes appropriate instance of *FaaSControllerWind* or *FaaSControllerSolar* to obtain a composite cost.

Using the divide and conquer approach, FaaS Controller then individually invokes the economic endpoints of each of these services namely, *IEDCFServiceCost*, *IIDRFServiceCost* and *IFGFServiceCost* to obtain cost of using these services indicated by *EDCFCost*, *IDRFCost* and *FGFCost* in figure 5.7. The cost of using service features, shown in green, are added by each of the service, which are then added by FaaS Controller to compute the total cost.

As each service is decoupled from others, in case the service of FaaS are to be reused by users, they can choose which services they want to use and thus only pay for those. Also, as users are not bound to any permanent charges, the services in FaaS are essentially available as PaaS on a *pay-per-use* basis.

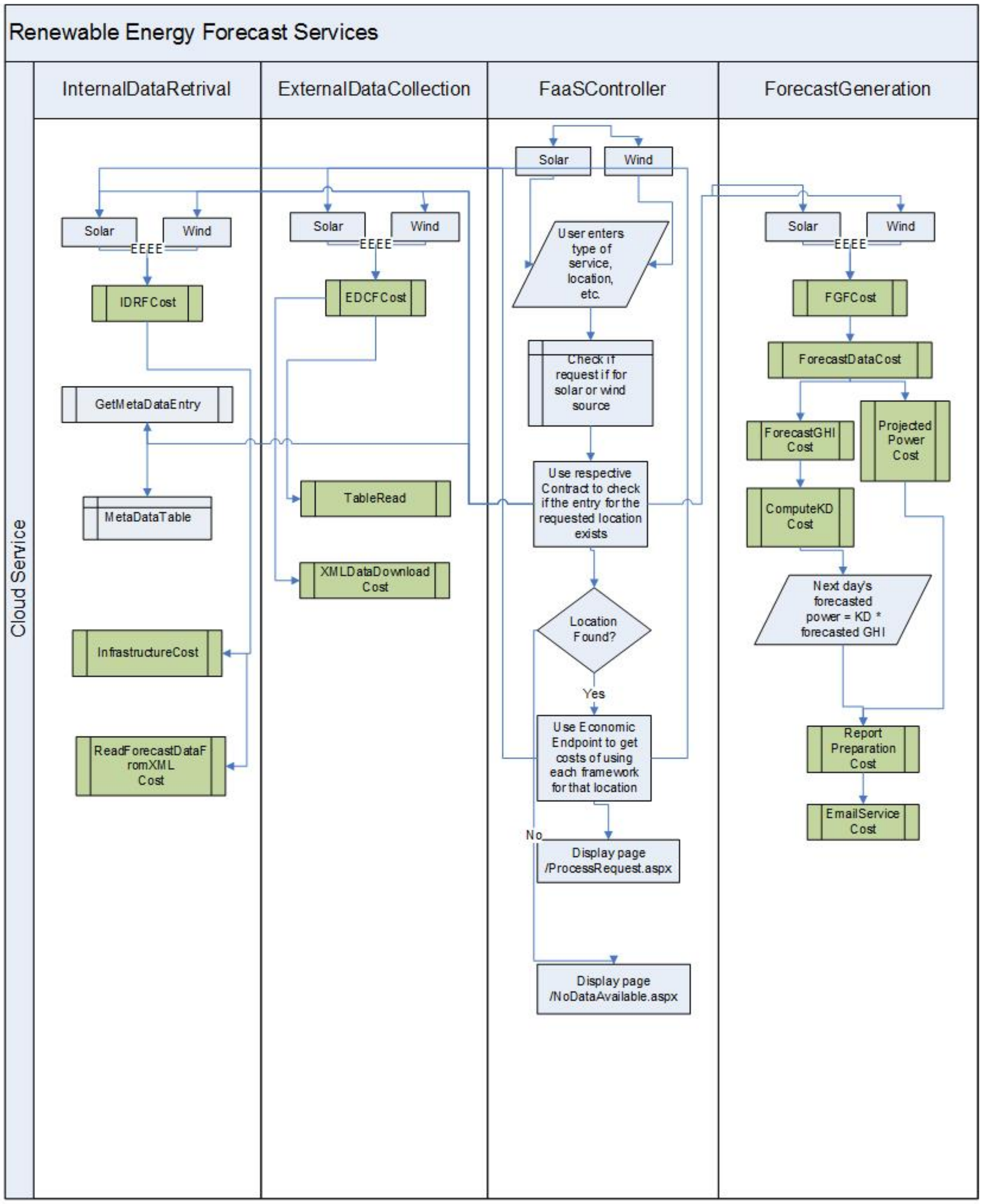

Renewable Energy Forecast Services
Cloud Service
InternalDataRetrival
ExternalDataCollection
FaaSController
ForecastGeneration
Solar
Wind
EEEE
IDRFCost
GetMetaDataEntry
MetaDataTable
InfrastructureCost
ReadForecastDataFromXML Cost
EDCFCost
TableRead
XMLDataDownload Cost
User enters type of service, location, etc.
Check if request if for solar or wind source
Use respective Contract to check if the entry for the requested location exists
Location Found?
Yes
No
Use Economic Endpoint to get costs of using each framework for that location
Display page /ProcessRequest.aspx
Display page /NoDataAvailable.aspx
FGFCost
ForecastDataCost
ForecastGHI Cost
Projected Power Cost
ComputeKD Cost
Next day's forecasted power = KD * forecasted GHI
Report Preparation Cost
EmailService Cost

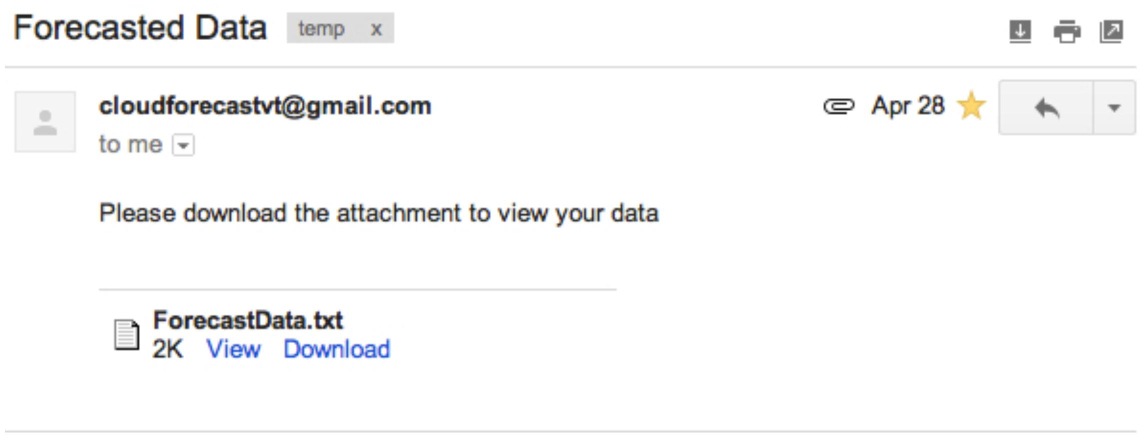
Figure 5.7. Cost accumulation for solar operational forecast using Economic Endpoint (EE)

Chapter 6

Results

6.1. Wind Forecast Results

When a user submits a forecast request using the webpages (shown in fig D.1, D.2 in Appendix D), an email is sent out to them at their provided email address containing the requested data. Fig 6.1 below shows the content of the mail with an attached file in the mail. The email contains an attachment indicating what operations the user requested, time at which the user made the request and time the report was created, and total cost of using the services. Then it has details about the location of data, such as name of the location, co-ordinates entered by the user, co-ordinates where the data is coming from and distance between the two, and state code of the location. Depending on whether the user requested wind or solar forecast, it would have details of the turbine or solar panel respectively. Finally, it contains the annual energy data along with P95, P90, P75 and P50 uncertainty data and monthly data with their individual uncertainty data.


Forecasted Data temp x

cloudforecastvt@gmail.com
to me

Apr 28

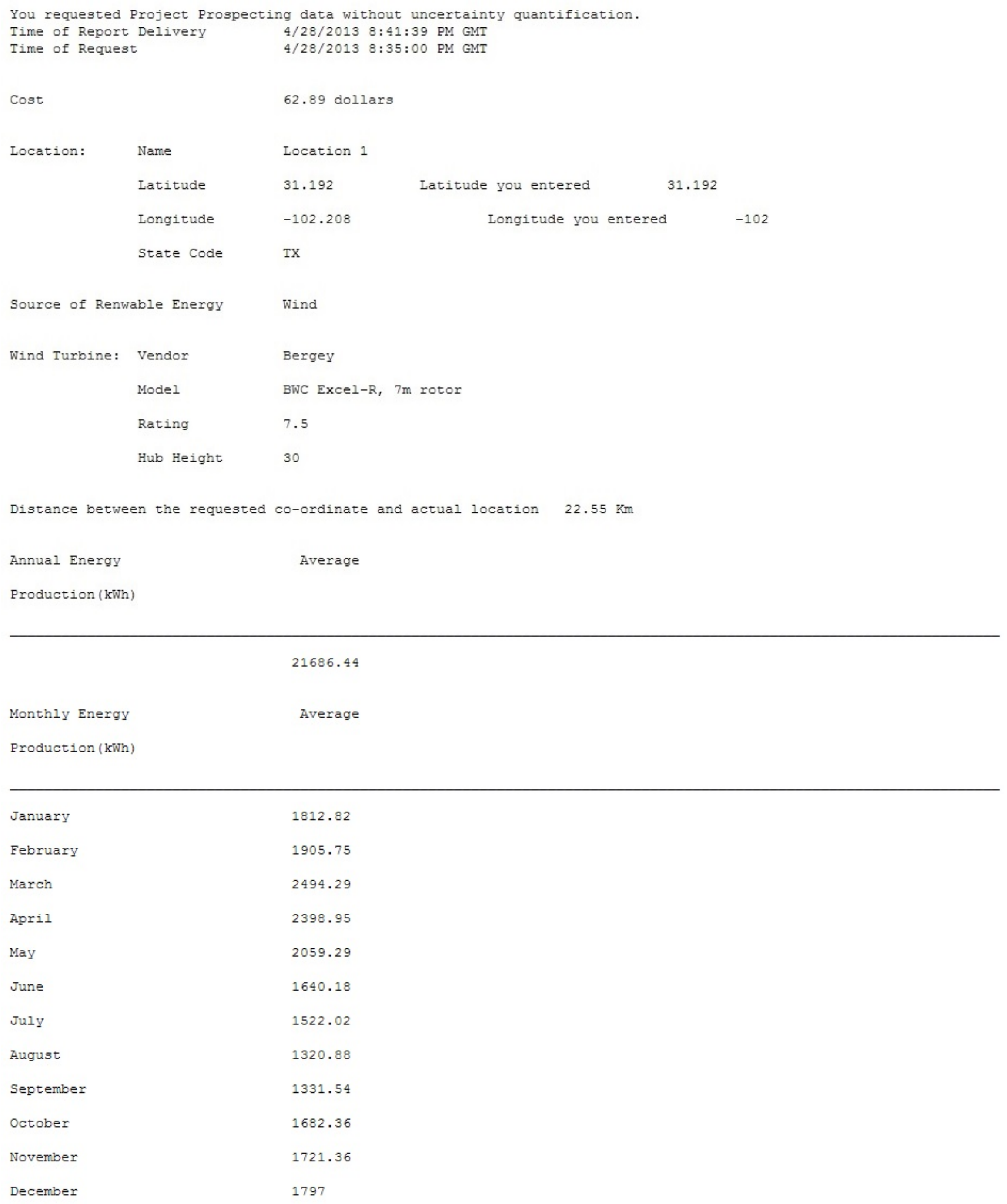
Please download the attachment to view your data

ForecastData.txt
2K View Download

Figure 6.1 Shows content of the mail in the inbox of the user

6.1.1. Project Prospecting

***Without Turbine Data And Without Uncertainty Data*:** The following figure shows content of the file received by the user with default turbine data.



```
You requested Project Prospecting data without uncertainty quantification.
Time of Report Delivery         4/28/2013 8:41:39 PM GMT
Time of Request                 4/28/2013 8:35:00 PM GMT

Cost                            62.89 dollars

Location:      Name             Location 1

               Latitude         31.192          Latitude you entered        31.192

               Longitude        -102.208                Longitude you entered        -102

               State Code       TX

Source of Renwable Energy       Wind

Wind Turbine:  Vendor           Bergey

               Model            BWC Excel-R, 7m rotor

               Rating           7.5

               Hub Height       30

Distance between the requested co-ordinate and actual location   22.55 Km

Annual Energy                     Average

Production(kWh)
_____________________________________________________________________________________

                                 21686.44

Monthly Energy                    Average

Production(kWh)
_____________________________________________________________________________________

January                          1812.82

February                         1905.75

March                            2494.29

April                            2398.95

May                              2059.29

June                             1640.18

July                             1522.02

August                           1320.88

September                        1331.54

October                          1682.36

November                         1721.36

December                         1797
```

Figure 6.2 Forecast data with default turbine and without uncertainty data

A cost break-up (all costs are in USD) for charges incurred by a user is given below for wind project prospecting with default wind turbine *Bergey*. The cost incurred because of services EDCF, IDRF and FGF are given in table 6.1, 6.2, and 6.5, respectively.

Table 6.1 Cost for workflow created using respective service features in EDCF

TableRead	2.56
BlobWrite	1.47
TableWrite	2.43
HTTPDownload	4.09
Total EDCF Cost (C_1) in USD	**10.55**

Break-up of costs in IDRF:

Table 6.2 Cost associated with standardization of data in IDRF

Wind Speed Data Standardization Cost		
RawWindSpeedData		7.67
NumberOfHarmonicsNeeded		0.96
MAPE + IDFT	0.64	0.96
TableWrite		2.43
TableRead		2.56
Total (C_2) in USD	**15.22**	

Table 6.3 Cost accumulated for obtaining turbine data

Turbine Data Cost	
BasicOfficeOperations	3.84
BlobRead	0.96
TableWrite	2.43
Total (C_3) in USD	**7.23**

Table 6.4 Infrastructure cost for using Azure

Infrastructure Cost			
ComputeCost	Years of		Compute Cost * 3 (years of

	Data		data)
1.21	3		3.63
BlobStorageCost	Years of Data	Storage Time	BlobStorageCost * 3 (years of data) * Storage Time
0.54	3	3	4.86
Cost per month for Table Transactions	Number of months	Number of years	TableTransactionCost
0.03	12	3	1.08
Total (C_4) in USD			**9.57**

Total Cost returned by IDRF, C_5

$$= C_2 + C_3 + C_4 = 15.2 + 7.23 + 9.57 = \$\ 32.02$$

Cost break-up for FGF:

Table 6.5 Workflow cost for forecasting by FGF

TurbineDataCost	2.56
PowerForecastCost	5.11
RawWindSpeedDataCost	7.67
ReportPreparationCost	2.55
BlobRead	0.96
BlobWrite	1.47
Total FGF Cost (C_6) in USD	**20.32**

Cost returned by FaaS Controller

= Total EDCF Cost + Total IDRF Cost + Total FGF Cost

$= C_1 + C_5 + C_6$

$= 10.55 + 32.02 + 20.32$

= \$ 62.89

The distance between user provided location (31.192, -102) and the actual location (31.192, -102.208) was calculated using the great-circle distance and is roughly equal to 22.55 Km (please refer to Appendix C.10 for more detailed algorithm and calculations).

Table 6.6 shows the turbine power curve for the default wind turbine used in table 6.7 to calculate the energy produced during a 10-minute period. By summing up the energy produced per 10-minute for the entire day, daily energy produced can be calculated, which can be further used to calculate the monthly energy values as shown in table 6.8.

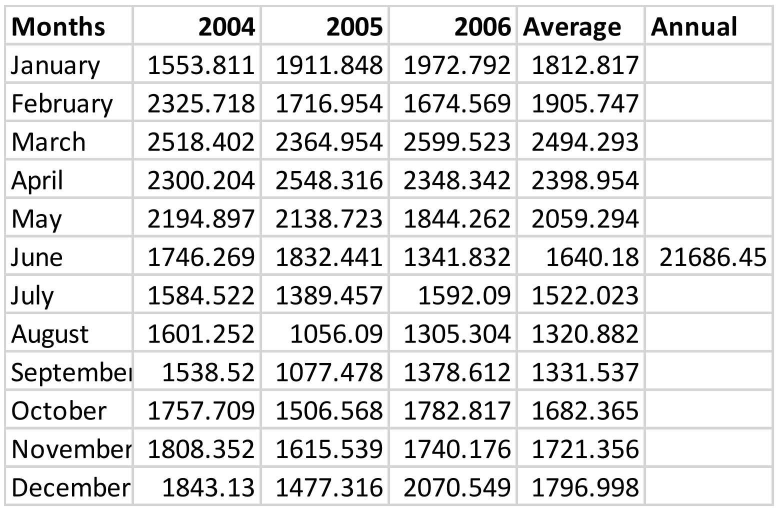
Table 6.6 Power curve table for *Bergey* Wind Turbine

Wind Speed (m/s)	Power (kW)
3.00	0.00
4.00	0.25
5.00	0.80
6.00	1.65
7.00	2.55
8.00	3.65
9.00	4.85
10.00	6.15
11.00	7.50
12.00	8.00
13.00	8.00
14.00	8.00
15.00	7.00
16.00	5.00
17.00	2.70
18.00	3.00
19.00	3.00
20.00	3.00
21.00	0.00

Table 6.7 Sample data table for the location for year 2004 for height of 100 m

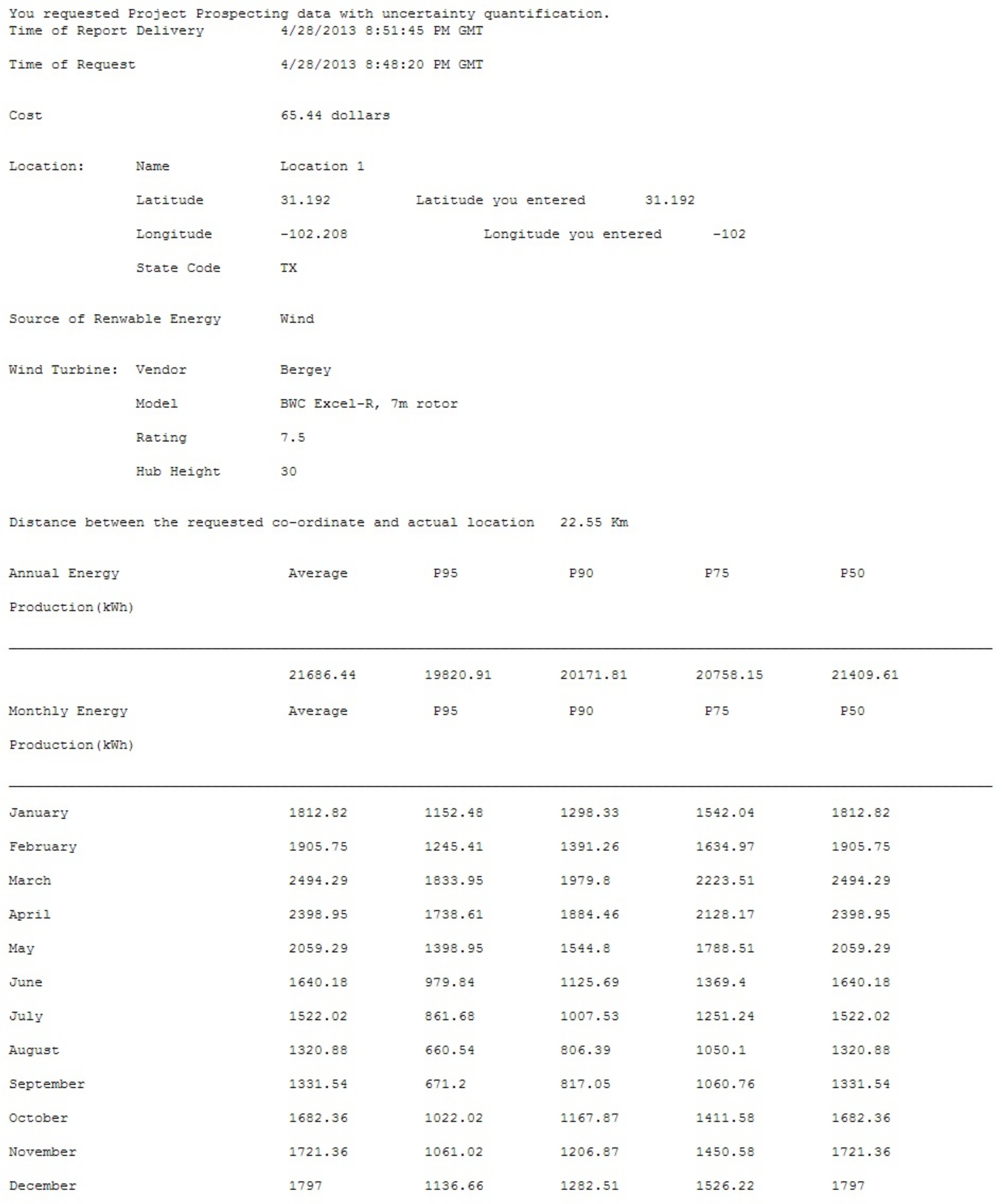
Date & Time	Wind Speed (m/s) at 100 m	Wind Speed at user requested height of 30 m	Interpolated Power from power curve	Energy (kWh)
2004-01-01 00:00:00	10.770	9.068	4.93	4.93*10/60
2004-01-01 00:10:00	11.350	9.556	5.43	5.43*10/60
2004-01-01 00:20:00	11.980	10.086	6.26	6.26*10/60
2004-01-01 00:30:00	11.850	9.977	4.87	4.87*10/60
2004-01-01 00:40:00	12.360	10.406	6.69	6.69*10/60
2004-01-01 00:50:00	12.370	10.415	6.71	6.71*10/60

Table 6.8 Shows the monthly energy values for year 2004, 2005 and 2006 with yearly averages and annual energy produced



Months	2004	2005	2006	Average	Annual
January	1553.811	1911.848	1972.792	1812.817	
February	2325.718	1716.954	1674.569	1905.747	
March	2518.402	2364.954	2599.523	2494.293	
April	2300.204	2548.316	2348.342	2398.954	
May	2194.897	2138.723	1844.262	2059.294	
June	1746.269	1832.441	1341.832	1640.18	21686.45
July	1584.522	1389.457	1592.09	1522.023	
August	1601.252	1056.09	1305.304	1320.882	
Septembe	1538.52	1077.478	1378.612	1331.537	
October	1757.709	1506.568	1782.817	1682.365	
November	1808.352	1615.539	1740.176	1721.356	
December	1843.13	1477.316	2070.549	1796.998	

***Without Turbine Data And With Uncertainty Data*:** The following figure shows the content of file received by the user with default turbine data.



```
You requested Project Prospecting data with uncertainty quantification.
Time of Report Delivery        4/28/2013 8:51:45 PM GMT

Time of Request                4/28/2013 8:48:20 PM GMT


Cost                           65.44 dollars


Location:      Name            Location 1

               Latitude        31.192          Latitude you entered       31.192

               Longitude       -102.208                Longitude you entered      -102

               State Code      TX


Source of Renwable Energy      Wind


Wind Turbine:  Vendor          Bergey

               Model           BWC Excel-R, 7m rotor

               Rating          7.5

               Hub Height      30


Distance between the requested co-ordinate and actual location   22.55 Km


Annual Energy                   Average          P95             P90             P75             P50

Production(kWh)
__________________________________________________________________________________________________________________

                                21686.44        19820.91        20171.81        20758.15        21409.61

Monthly Energy                  Average          P95             P90             P75             P50

Production(kWh)
__________________________________________________________________________________________________________________

January                         1812.82         1152.48         1298.33         1542.04         1812.82

February                        1905.75         1245.41         1391.26         1634.97         1905.75

March                           2494.29         1833.95         1979.8          2223.51         2494.29

April                           2398.95         1738.61         1884.46         2128.17         2398.95

May                             2059.29         1398.95         1544.8          1788.51         2059.29

June                            1640.18         979.84          1125.69         1369.4          1640.18

July                            1522.02         861.68          1007.53         1251.24         1522.02

August                          1320.88         660.54          806.39          1050.1          1320.88

September                       1331.54         671.2           817.05          1060.76         1331.54

October                         1682.36         1022.02         1167.87         1411.58         1682.36

November                        1721.36         1061.02         1206.87         1450.58         1721.36

December                        1797            1136.66         1282.51         1526.22         1797
```

Figure 6.3 Forecast data with default turbine data and with uncertainty data

The difference in cost in this request is because the user selected the uncertainty data as shown in table 6.9. Turbine data cost is automatically added even if the user does not select it because a default turbine is selected in case the user does not specify.

Table 6.9 FGF workflow cost for forecasting with uncertainty data

TurbineDataCost	2.56
PowerForecastCost	5.11
RawWindSpeedDataCost	7.67
ReportPreparationCost	2.55
BlobRead	0.96
ExceedanceProbabilityCost	1.91
CoefficientOfVariationCost	0.67
BlobWrite	1.47
Total FGF Cost (C_7) in USD	**22.87**

Cost returned by FaaS Controller, C_8

= Total EDCF Cost + Total IDRF Cost + Total FGF Cost

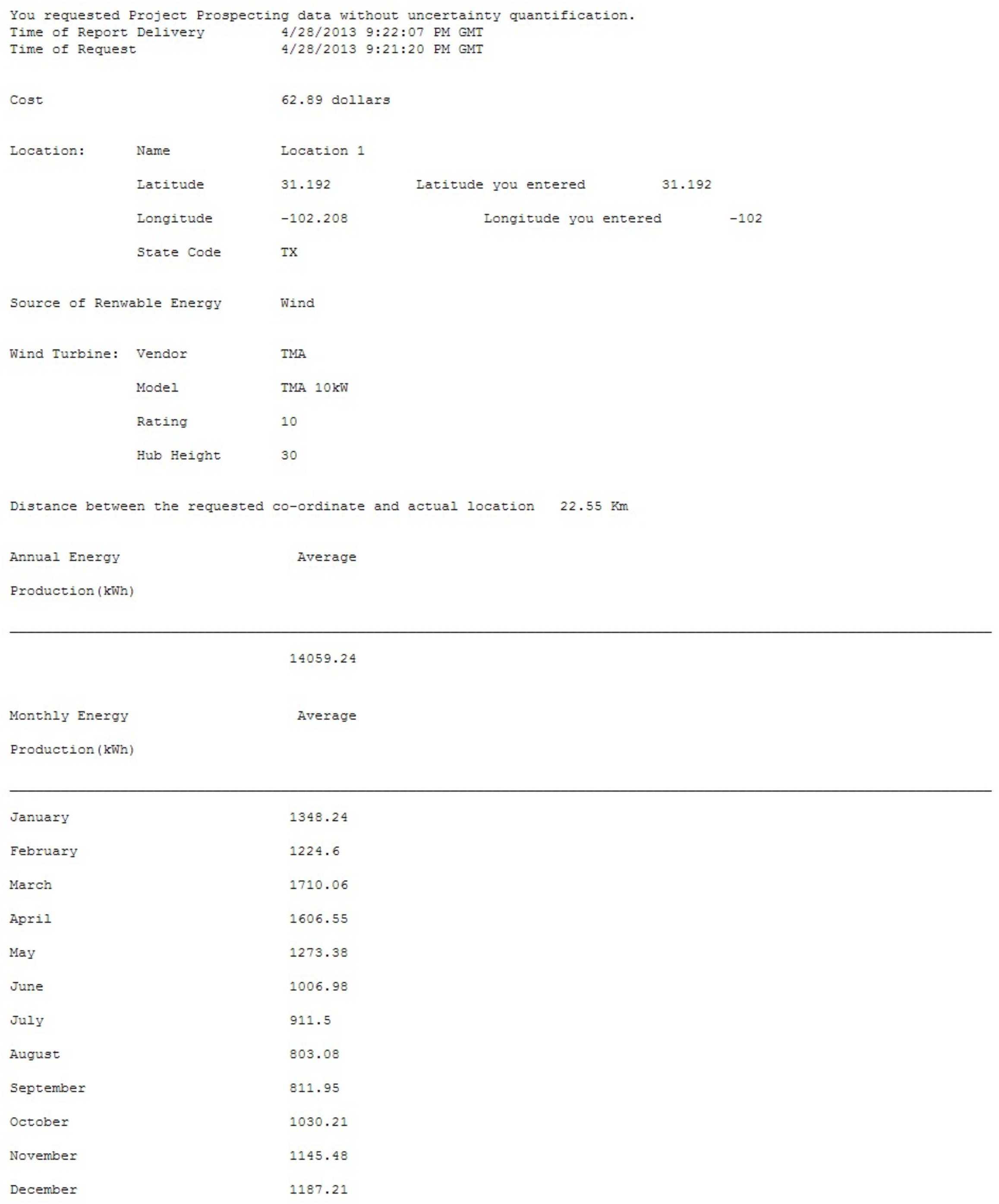
= $C_1 + C_5 + C_7 = 10.55 + 32.02 + 22.87$

= $ 65.44

Table 6.10 Shows energy values for year 2004, 2005 & 2006 with the uncertainty data

Months	**2004**	**2005**	**2006**	**Average**	**P95 for average**	**P90 for average**	**P75 for average**	**P50 for average**
January	1553.811	1911.848	1972.792	1812.82	1152.48	1298.33	1542.04	1812.82
February	2325.718	1716.954	1674.569	1905.75	1245.41	1391.26	1634.97	1905.75
March	2518.402	2364.954	2599.523	2494.29	1833.95	1979.8	2223.51	2494.29
April	2300.204	2548.316	2348.342	2398.95	1738.61	1884.46	2128.17	2398.95
May	2194.897	2138.723	1844.262	2059.29	1398.95	1544.8	1788.51	2059.29
June	1746.269	1832.441	1341.832	1640.18	979.84	1125.69	1369.4	1640.18
July	1584.522	1389.457	1592.09	1522.02	861.68	1007.53	1251.24	1522.02
August	1601.252	1056.09	1305.304	1320.88	660.54	806.39	1050.1	1320.88
Septembe	1538.52	1077.478	1378.612	1331.54	671.2	817.05	1060.76	1331.54
October	1757.709	1506.568	1782.817	1682.36	1022.02	1167.87	1411.58	1682.36
November	1808.352	1615.539	1740.176	1721.36	1061.02	1206.87	1450.58	1721.36
December	1843.13	1477.316	2070.549	1797	1136.66	1282.51	1526.22	1797
Annual		**21686.44**			**19820.91**	**20171.81**	**20758.15**	**21409.61**

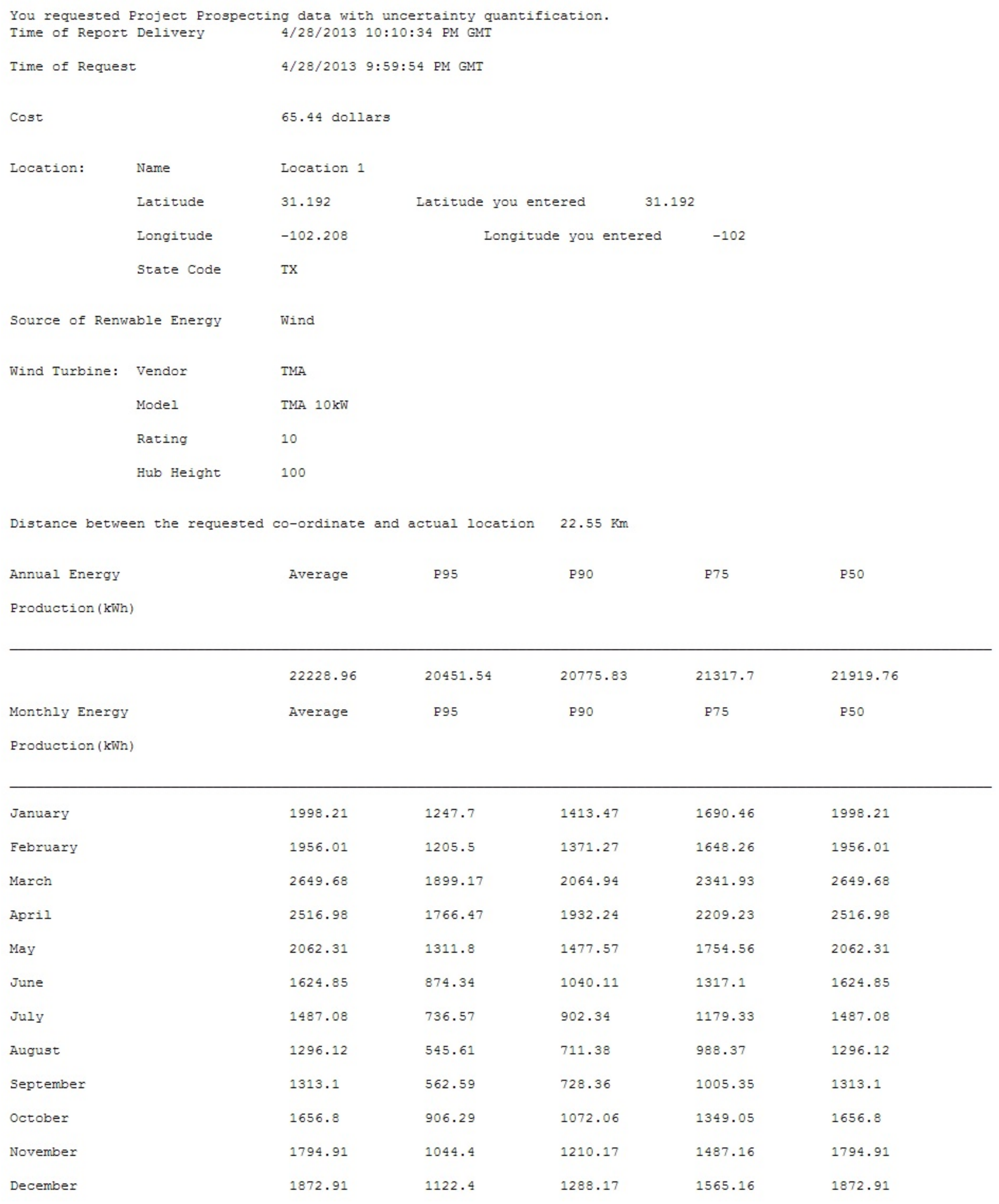
With Turbine Data And Without Uncertainty Data: The result below shows same cost as previous service, the only difference being that the user has selected a turbine in this case and a different location.



```
You requested Project Prospecting data without uncertainty quantification.
Time of Report Delivery        4/28/2013 9:22:07 PM GMT
Time of Request                4/28/2013 9:21:20 PM GMT


Cost                           62.89 dollars


Location:      Name            Location 1

               Latitude        31.192         Latitude you entered        31.192

               Longitude       -102.208               Longitude you entered        -102

               State Code      TX


Source of Renwable Energy      Wind


Wind Turbine:  Vendor          TMA

               Model           TMA 10kW

               Rating          10

               Hub Height      30


Distance between the requested co-ordinate and actual location   22.55 Km


Annual Energy                    Average

Production(kWh)
_____________________________________________________________________________________

                                14059.24


Monthly Energy                   Average

Production(kWh)
_____________________________________________________________________________________

January                         1348.24

February                        1224.6

March                           1710.06

April                           1606.55

May                             1273.38

June                            1006.98

July                            911.5

August                          803.08

September                       811.95

October                         1030.21

November                        1145.48

December                        1187.21
```

Figure 6.4 Forecast data with user specified turbine and without uncertainty data

With Turbine Data And With Uncertainty Data (100m): The results below have different height and turbine values selected by the user.

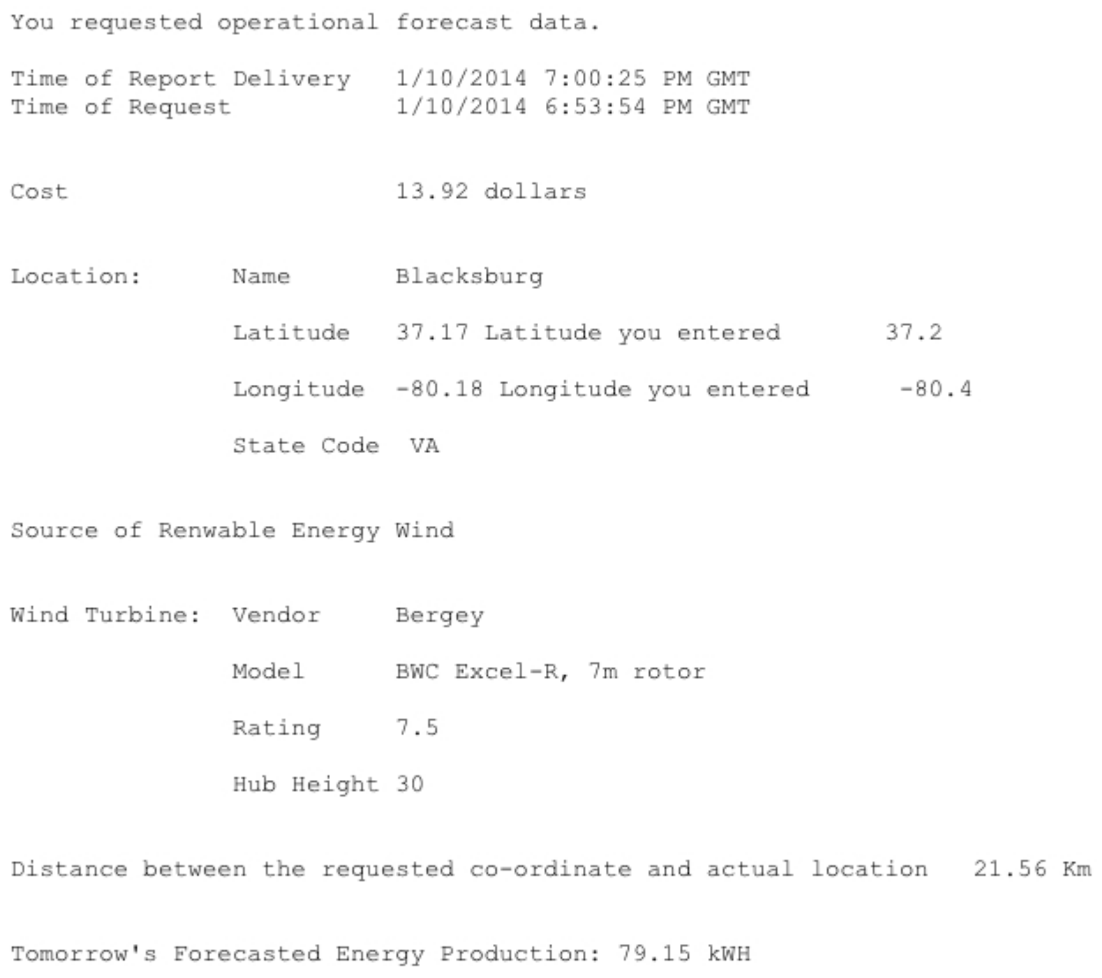


```
You requested Project Prospecting data with uncertainty quantification.
Time of Report Delivery         4/28/2013 10:10:34 PM GMT

Time of Request                 4/28/2013 9:59:54 PM GMT

Cost                            65.44 dollars

Location:       Name            Location 1

                Latitude        31.192          Latitude you entered       31.192

                Longitude       -102.208                Longitude you entered      -102

                State Code      TX

Source of Renwable Energy       Wind

Wind Turbine:   Vendor          TMA

                Model           TMA 10kW

                Rating          10

                Hub Height      100

Distance between the requested co-ordinate and actual location   22.55 Km

Annual Energy                    Average          P95             P90             P75             P50

Production(kWh)
_____________________________________________________________________________________________________

                                 22228.96        20451.54        20775.83        21317.7         21919.76

Monthly Energy                   Average          P95             P90             P75             P50

Production(kWh)
_____________________________________________________________________________________________________

January                          1998.21         1247.7          1413.47         1690.46         1998.21

February                         1956.01         1205.5          1371.27         1648.26         1956.01

March                            2649.68         1899.17         2064.94         2341.93         2649.68

April                            2516.98         1766.47         1932.24         2209.23         2516.98

May                              2062.31         1311.8          1477.57         1754.56         2062.31

June                             1624.85         874.34          1040.11         1317.1          1624.85

July                             1487.08         736.57          902.34          1179.33         1487.08

August                           1296.12         545.61          711.38          988.37          1296.12

September                        1313.1          562.59          728.36          1005.35         1313.1

October                          1656.8          906.29          1072.06         1349.05         1656.8

November                         1794.91         1044.4          1210.17         1487.16         1794.91

December                         1872.91         1122.4          1288.17         1565.16         1872.91
```

Figure 6.5 Forecast data with user specified turbine and with uncertainty data

6.1.2. Operational

***Without Turbine Data*:** The following figure shows the email received by the user for wind operational forecast with default wind turbine selected.



```
You requested operational forecast data.

Time of Report Delivery   1/10/2014 7:00:25 PM GMT
Time of Request           1/10/2014 6:53:54 PM GMT


Cost                      13.92 dollars


Location:      Name       Blacksburg

               Latitude   37.17 Latitude you entered       37.2

               Longitude  -80.18 Longitude you entered      -80.4

               State Code  VA


Source of Renewable Energy Wind


Wind Turbine:  Vendor     Bergey

               Model      BWC Excel-R, 7m rotor

               Rating     7.5

               Hub Height 30


Distance between the requested co-ordinate and actual location   21.56 Km


Tomorrow's Forecasted Energy Production: 79.15 kWH
```

Figure 6.6 Shows the mail content for wind operation data with default turbine

The break-up of cost presented above is given below for EDCF, IDRF and FGF in table 6.11, 6.12 and 6.13, respectively.

Table 6.11 Cost of workflow returned by EDCF for operational forecast

TableRead	2.56
XMLDataDownload	1.28
Total EDCF Cost (C_9)	**3.84**

Table 6.12 Cost of workflow returned by IDRF for operational forecast

ReadDataFromXMLCost	1.28
ComputeCost	1.21
BlobStorageCost	0.54
TableTransactionCost	0.03
Total IDRF Cost (C_{10})	**3.06**

Table 6.13 Cost of workflow returned by FGF for operational forecast

PowerForecastCost	3
ReportPreparationCost	2.55
BlobWrite	1.47
Total FGF Cost (C_{11})	**7.02**

Total cost returned by FaaS Controller, $C_{12} = C_9 + C_{10} + C_{11}$ = $ 13.92

As previously stated, operational forecasting gives an estimate of next day's energy production. For this purpose, it uses next day's forecasted average wind speeds for every three hours available from NOAA (details given in Appendix A.9, *XMLDataDownload*). These average wind speeds were used to calculate average energy that the *Bergey* wind turbine can produce by interpolating its power curve (table 6.6). Table 6.14 shows the forecasted energy values using a set of wind speed data.

Table 6.14 Forecasted energy values calculated using forecasted average wind speed data from NOAA

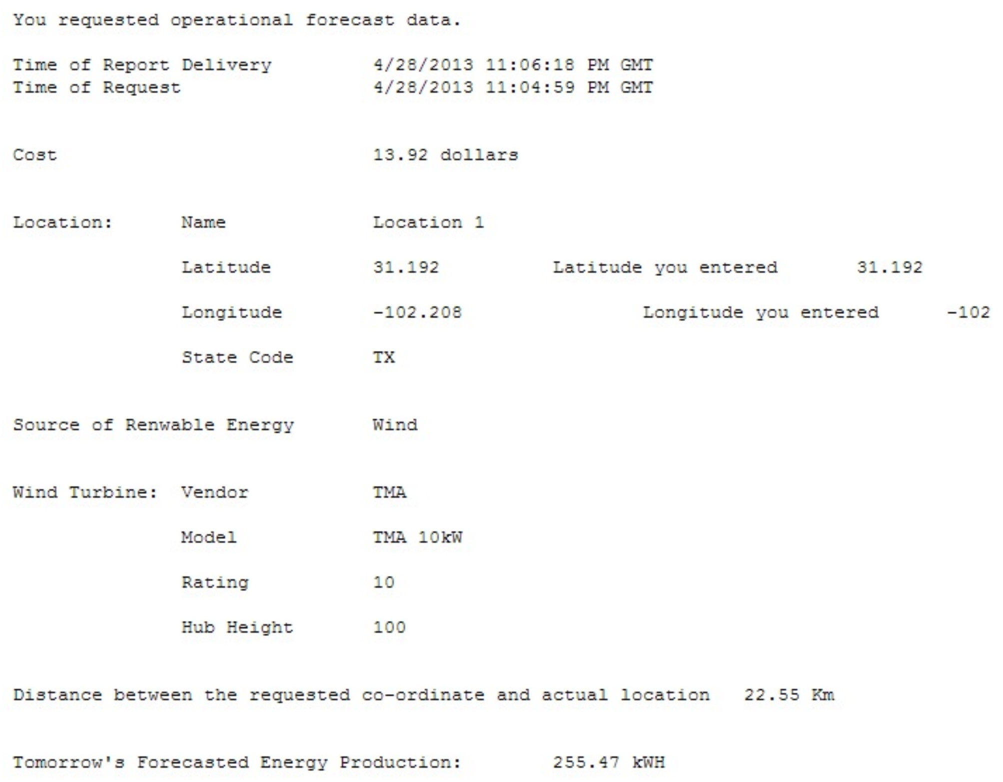
Time	Forecasted Average Wind Speed Data Set 1 (m/s)	Forecasted Average Wind Speed Data Set 2 (m/s)	Forecasted Average Wind Speed Data Set 3 (m/s)
00:00 – 02:29	4.12	10.80	6.69
03:00 – 05:59	5.14	9.77	7.20
06:00 – 08:59	7.20	8.23	6.17
09:00 – 11:59	7.72	7.20	7.72
12:00 – 14:59	9.26	6.17	7.72
15:00 – 17:59	8.75	4.12	7.72
18:00 – 20:59	8.75	2.57	9.26
21:00 – 23:59	10.80	3.09	8.23
Forecasted Energy (kWh)	**79.15**	**61.19**	**73.64**

Table 6.15 shows actual wind speed data for a particular day obtained from [Wunderground, 2013] to calculate the energy that could be produced for the same location. A mean percentage error of 13.7% was observed between the actual and forecasted energy values, mainly due to the difference between forecasted and actual wind speed data.

Table 6.15 Energy values using actual wind speed data for a particular day

Time	Wind Speed Data Set 1 (m/s)	Wind Speed Data Set 2 (m/s)	Wind Speed Data Set 3 (m/s)
00:00 – 02:29	1.79	10.59	6.88
03:00 – 05:59	3.08	9.48	7.51
06:00 – 08:59	5.14	7.87	6.35
09:00 – 11:59	6.17	7.38	7.15
12:00 – 14:59	8.27	6.48	7.69
15:00 – 17:59	7.51	4.69	8.09
18:00 – 20:59	7.78	2.41	9.39
21:00 – 23:59	11.31	2.82	8.72
Energy Produced (kWh)	**62.83**	**64.82**	**81.54**

***With Turbine Data*:** The following figure shows the content of file received by the user



```
You requested operational forecast data.

Time of Report Delivery        4/28/2013 11:06:18 PM GMT
Time of Request                4/28/2013 11:04:59 PM GMT


Cost                           13.92 dollars


Location:      Name            Location 1

               Latitude        31.192         Latitude you entered      31.192

               Longitude       -102.208               Longitude you entered      -102

               State Code      TX


Source of Renwable Energy      Wind


Wind Turbine:  Vendor          TMA

               Model           TMA 10kW

               Rating          10

               Hub Height      100


Distance between the requested co-ordinate and actual location   22.55 Km


Tomorrow's Forecasted Energy Production:        255.47 kWH
```

Figure 6.7 Results for wind operation data with a user specified turbine

Table 6.16 shows power curve for wind turbine from TMA that was selected by the user. These power curve values were used to obtain power values corresponding to a wind speed by interpolation to forecast next day's energy values as shown in table 6.17.

Table 6.16 Power curve for a turbine from TMA

Wind Speed (m/s)	Power (kW)
4.00	0.00
5.00	0.04
6.00	0.07
7.00	0.12

8.00	0.16
9.00	0.26
10.00	0.35
11.00	0.47
12.00	0.58
13.00	0.74
14.00	0.90
15.00	1.10
16.00	1.30
17.00	1.58
18.00	1.85
19.00	2.16
20.00	2.46
21.00	2.83
22.00	3.20
23.00	3.68
24.00	4.15
25.00	4.75
26.00	5.35
27.00	6.02
28.00	6.68
29.00	7.39
30.00	8.10
31.00	9.13
32.00	10.16
33.00	10.58
34.00	11.00
35.00	11.00

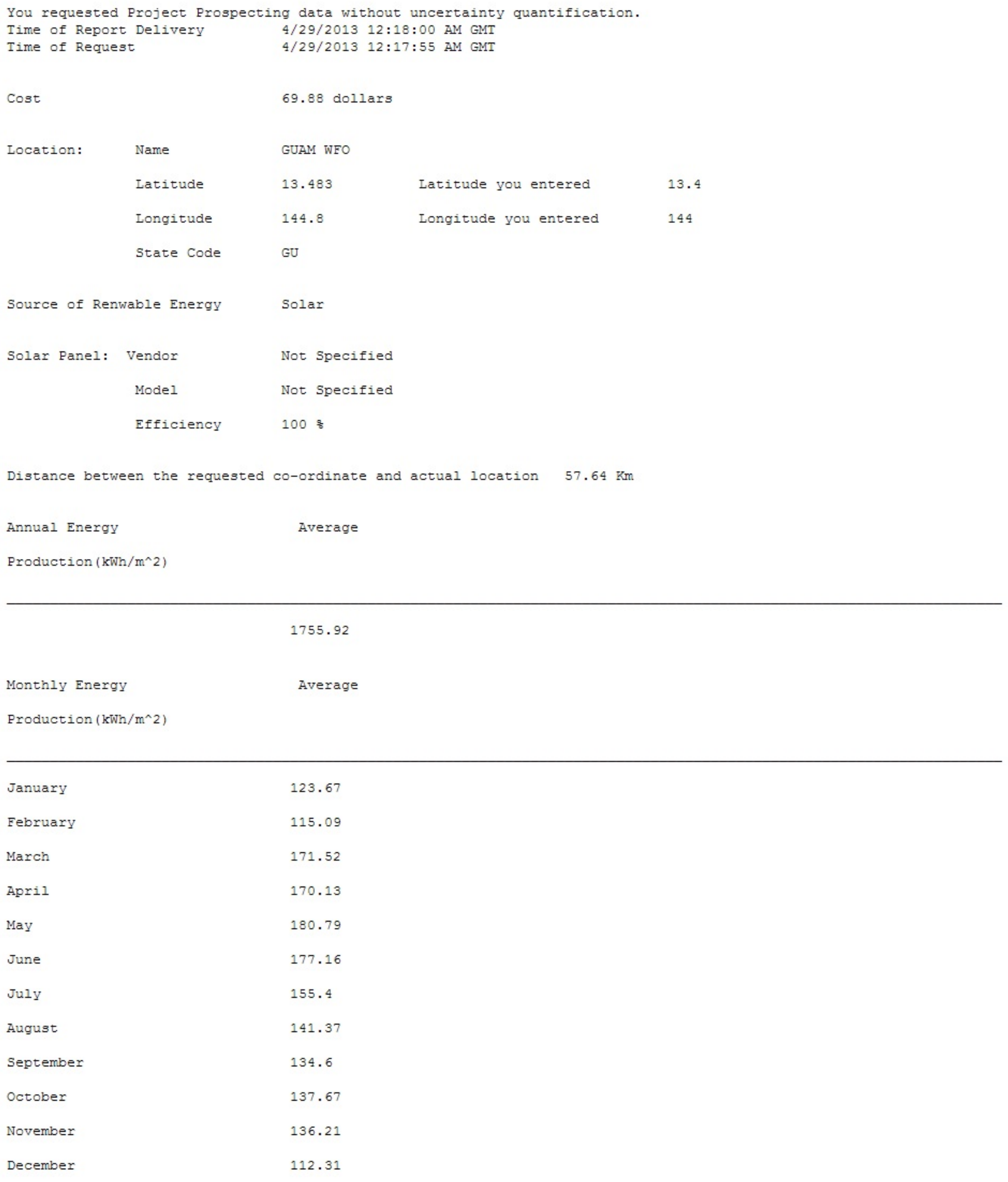
Table 6.17 Shows the next day's forecasted energy using TMA turbine

Average Wind Speeds (m/s) for 3 hours	Projected Power for 1 hour	Projected Power for 3 hours	Total Forecasted Energy (kWh)
14.96	10.62	31.86	255.47
16.11	11.0	33	
13.80	8.54	25.62	
17.26	11.0	33	
17.26	11.0	33	
17.26	11.0	33	
17.26	11.0	33	
17.26	11.0	33	

6.2. Solar Forecast Results

6.2.1. Project Prospecting

Without Solar Panel Data And Without Uncertainty Data: The following figure shows the content of file received by the user.



```
You requested Project Prospecting data without uncertainty quantification.
Time of Report Delivery        4/29/2013 12:18:00 AM GMT
Time of Request                4/29/2013 12:17:55 AM GMT


Cost                           69.88 dollars


Location:      Name            GUAM WFO

               Latitude        13.483          Latitude you entered        13.4

               Longitude       144.8           Longitude you entered       144

               State Code      GU


Source of Renwable Energy      Solar


Solar Panel:  Vendor           Not Specified

               Model           Not Specified

               Efficiency      100 %


Distance between the requested co-ordinate and actual location   57.64 Km


Annual Energy                    Average

Production(kWh/m^2)
_____________________________________________________________________________________

                                1755.92


Monthly Energy                   Average

Production(kWh/m^2)
_____________________________________________________________________________________

January                         123.67

February                        115.09

March                           171.52

April                           170.13

May                             180.79

June                            177.16

July                            155.4

August                          141.37

September                       134.6

October                         137.67

November                        136.21

December                        112.31
```

Figure 6.8 Solar project prospecting without solar panel data and uncertainty data

The cost incurred from services EDCF, IDRF and FGF are given in table 6.18, 6.19, and 6.20, respectively.

EDCF

Table 6.18 Cost returned by EDCF for solar project prospecting workflow

Table Read	2.56
BlobWrite	1.47
TableWrite	2.43
HTTPDownload	4.09
Total EDCF Cost (C_{13})	**10.55**

Cost breakup for IDRF:

Table 6.19 Cost for data standardization for solar project prospecting

Solar Data Standardization Cost				
RawSolarData	7.67			
MinutelyDataCost	5.67	1.28	2.26	2.13
MinutelyDataToHourlyDataCost	6.31	1.92	2.26	2.13
HourlyDataToDailyDataCost	5.59	1.2	2.26	2.13
DailyDataToMonthlyDataCost	5.67	1.28	2.26	2.13
MonthlyDataToAnnualDataCost	5.16	0.77	2.26	2.13
TableWrite	2.43			
BlobRead	0.96			
BlobWrite	1.47			
Total (C_{14})	**40.93**			

Table 6.20 Cost for reading solar panel data from table

Solar Panel Data Cost	
TableRead	2.56
Total (C_{15})	**2.56**

Table 6.21 Infrastructure cost added by IDRF

Infrastructure Cost			
ComputeCost	Years of Data		Compute Cost * 3 (years of data)
1.21	1		1.21
BlobStorageCost	Years of Data	Storage Time	BlobStorageCost * (years of data) * Storage Time
0.54	1	3	1.62
Cost per month for Table Transactions	Number of months	Number of years	TableTransactionCost
0.03	12	1	0.36
Total	**(C_{16})**		**3.19**

Total cost returned by IDRF, C_{17} = $C_{14} + C_{15} + C_{16}$ = \$ 46.68

Cost break-up for FGF:

Table 6.22 Cost returned by FGF for solar project prospecting forecast

TableReadCost	2.56
TableWriteCost	5.11
ReportPreparationCost	2.55
BlobRead	0.96
BlobWrite	1.47
Total FGF Cost (C_{18})	**12.65**

Total cost returned by FaaS Controller, $C_{19} = C_{13} + C_{17} + C_{18}$ = \$ 69.88

Daily GHI and GHI uncertainty data (sample values shown in table 6.23) was used to calculate monthly and annual GHI and GHI uncertainty data (using algorithm in Appendix C.7) as shown in table 6.24.

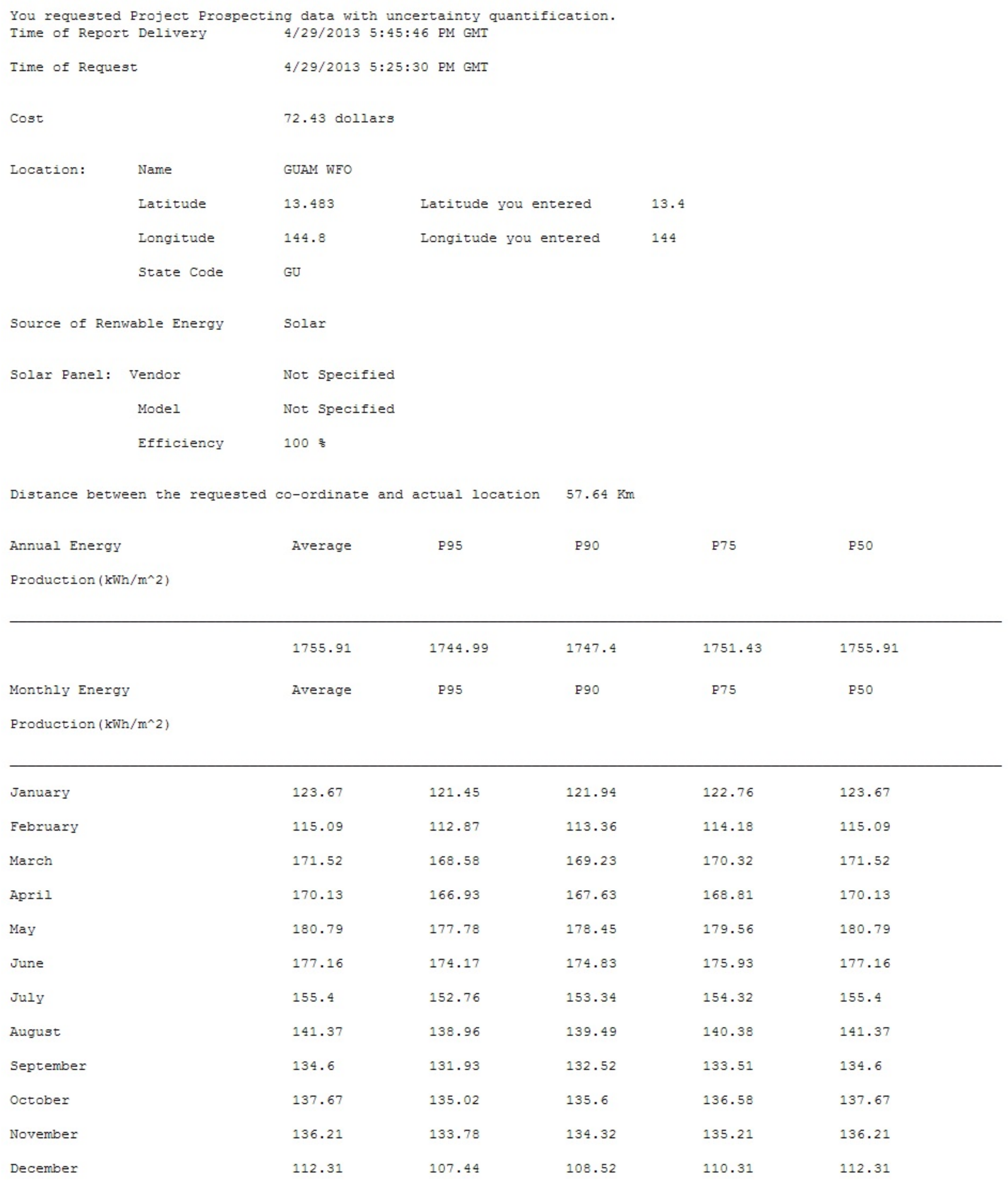
Table 6.23 Sample GHI data table for the location for one day

Date	Time	GHI	GHI Uncertainty	GHI Daily
1/1/1996	7:00	1	11	3919
1/1/1996	8:00	83	11	
1/1/1996	9:00	215	11	
1/1/1996	10:00	383	10	
1/1/1996	11:00	580	10	
1/1/1996	12:00	754	10	
1/1/1996	13:00	411	10	
1/1/1996	14:00	573	10	
1/1/1996	15:00	259	10	
1/1/1996	16:00	407	10	
1/1/1996	17:00	209	10	
1/1/1996	18:00	44	10	
1/1/1996	19:00	0	10	

Table 6.24 Shows monthly GHI and GHI uncertainty values used to calculate annual GHI and annual GHI uncertainty values as explained in Appendix C.7

Month	**Monthly GHI**	**Monthly GHI Uncertainty**	**Annual GHI**	**Annual GHI Uncertainty**
January	123668	1346.3826	1755910	6639.943
February	115090	1350.5052		
March	171525	1791.8682		
April	170126	1945.6345		
May	180792	1830.2707		
June	177156	1817.855		
July	155395	1600.6776		
August	141367	1464.8382		
September	134604	1625.8287		
October	137666	1609.0239		
November	136208	1475.1932		
December	112313	2962.8173		

Without Solar Panel Data And With Uncertainty Data: The following figure shows the content of file received by the user.



```
You requested Project Prospecting data with uncertainty quantification.
Time of Report Delivery        4/29/2013 5:45:46 PM GMT

Time of Request                4/29/2013 5:25:30 PM GMT


Cost                           72.43 dollars


Location:      Name            GUAM WFO

               Latitude        13.483          Latitude you entered       13.4

               Longitude       144.8           Longitude you entered      144

               State Code      GU


Source of Renwable Energy      Solar


Solar Panel:  Vendor           Not Specified

               Model           Not Specified

               Efficiency      100 %


Distance between the requested co-ordinate and actual location   57.64 Km


Annual Energy                   Average         P95             P90             P75             P50

Production(kWh/m^2)

_____________________________________________________________________________________________________

                                1755.91         1744.99         1747.4          1751.43         1755.91

Monthly Energy                  Average         P95             P90             P75             P50

Production(kWh/m^2)

_____________________________________________________________________________________________________

January                         123.67          121.45          121.94          122.76          123.67

February                        115.09          112.87          113.36          114.18          115.09

March                           171.52          168.58          169.23          170.32          171.52

April                           170.13          166.93          167.63          168.81          170.13

May                             180.79          177.78          178.45          179.56          180.79

June                            177.16          174.17          174.83          175.93          177.16

July                            155.4           152.76          153.34          154.32          155.4

August                          141.37          138.96          139.49          140.38          141.37

September                       134.6           131.93          132.52          133.51          134.6

October                         137.67          135.02          135.6           136.58          137.67

November                        136.21          133.78          134.32          135.21          136.21

December                        112.31          107.44          108.52          110.31          112.31
```

Figure 6.9 Results for solar project prospecting without solar panel data and with uncertainty data

As the cost of EDCF and IDRF as same as shown in last section, the cost break-up for FGF is given in table 6.25 below:

Table 6.25 Shows difference in cost of FGF for including uncertainty service

TableReadCost	2.56
TableWriteCost	5.11
ReportPreparationCost	2.55
BlobRead	0.96
BlobWrite	1.47
ExceedanceProbabilityCost	1.91
CoefficientOfVariationCost	0.64
Total FGF Cost (C_{20})	**15.2**

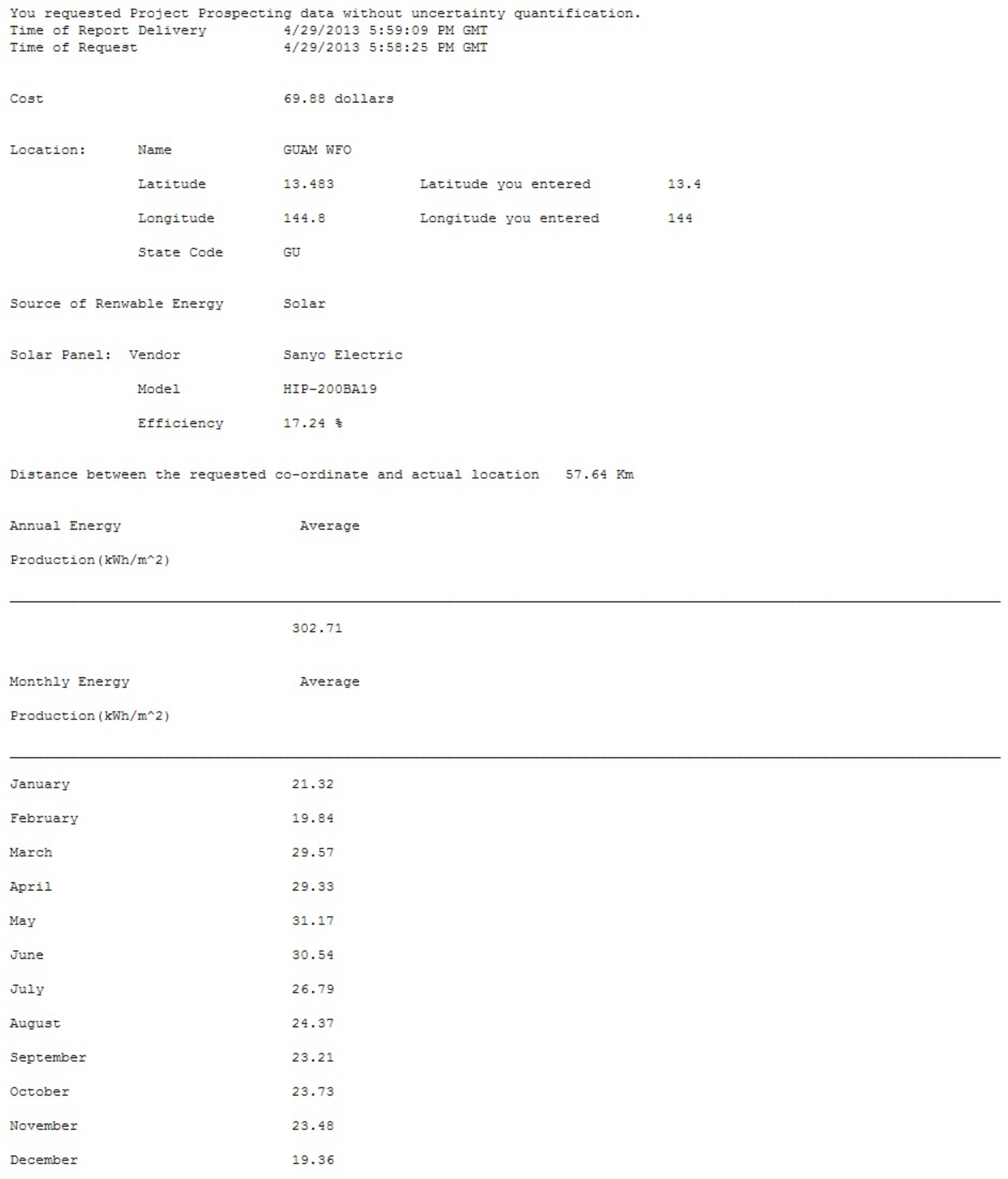
Total cost returned by FaaS Controller, $C_{21} = C_{13} + C_{17} + C_{20} = \$ 72.43$

Table 6.26 below shows the values of monthly and annual GHI and GHI uncertainty values that are calculated using algorithm in Appendix C.7 and sent to the user in mail.

Table 6.26 Shows the monthly values of GHI and uncertainty to calculate annual and monthly P*XX* values

Months	Monthly GHI	Uncertainty	P95	P90	P75	P50
January	123668	1346.3826	121453.4	121942.5	122759.9	123668
February	115090	1350.5052	112868.6	113359.3	114179.1	115090
March	171525	1791.8682	168577.6	169228.6	170316.4	171525
April	170126	1945.6345	166925.7	167632.6	168813.7	170126
May	180792	1830.2707	177781.5	178446.4	179557.5	180792
June	177156	1817.855	174165.9	174826.3	175929.9	177156
July	155395	1600.6776	152762.1	153343.7	154315.4	155395
August	141367	1464.8382	138957.6	139489.7	140379	141367
Septembei	134604	1625.8287	131929.8	132520.4	133507.4	134604
October	137666	1609.0239	135019.4	135604	136580.7	137666
November	136208	1475.1932	133781.5	134317.5	135213	136208
December	112313	2962.8173	107439.6	108516	110314.6	112313
Annual	**1755910**	**6639.9433**	**1744988**	**1747401**	**1751431**	**1755910**

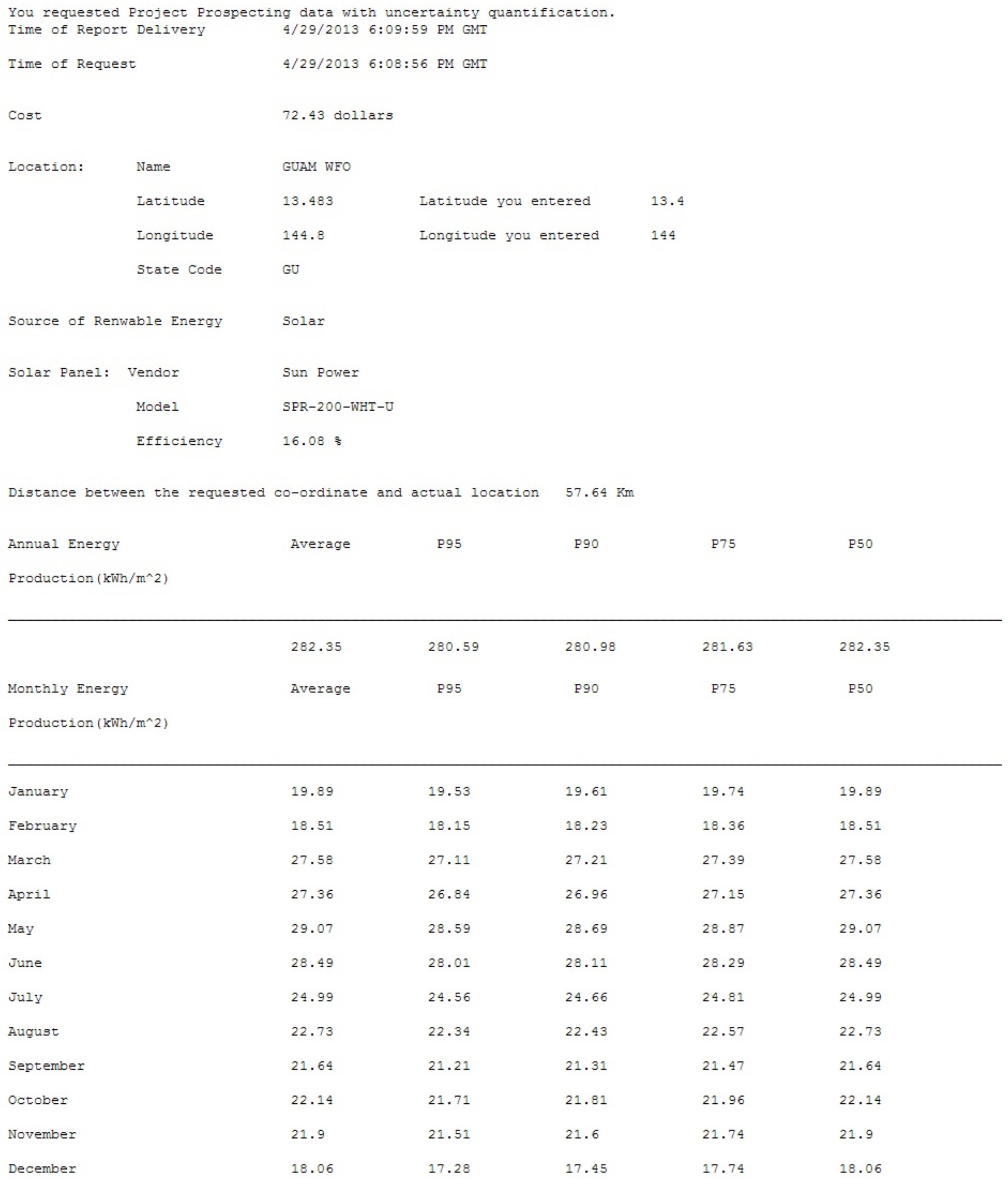
***With Solar Panel Data And Without Uncertainty Data*:** The following figure shows the content of file with a user-selected solar panel.



```
You requested Project Prospecting data without uncertainty quantification.
Time of Report Delivery        4/29/2013 5:59:09 PM GMT
Time of Request                4/29/2013 5:58:25 PM GMT


Cost                           69.88 dollars


Location:      Name            GUAM WFO

               Latitude        13.483          Latitude you entered        13.4

               Longitude       144.8           Longitude you entered       144

               State Code      GU


Source of Renwable Energy      Solar


Solar Panel:  Vendor           Sanyo Electric

               Model           HIP-200BA19

               Efficiency      17.24 %


Distance between the requested co-ordinate and actual location   57.64 Km


Annual Energy                    Average

Production(kWh/m^2)

_____________________________________________________________________________________

                                302.71


Monthly Energy                   Average

Production(kWh/m^2)

_____________________________________________________________________________________

January                         21.32

February                        19.84

March                           29.57

April                           29.33

May                             31.17

June                            30.54

July                            26.79

August                          24.37

September                       23.21

October                         23.73

November                        23.48

December                        19.36
```

Figure 6.10 Mail content for solar project prospecting with panel data but without uncertainty data

With Solar Panel Data and With Uncertainty Data: The following figure shows the content of file with a user-selected solar panel.

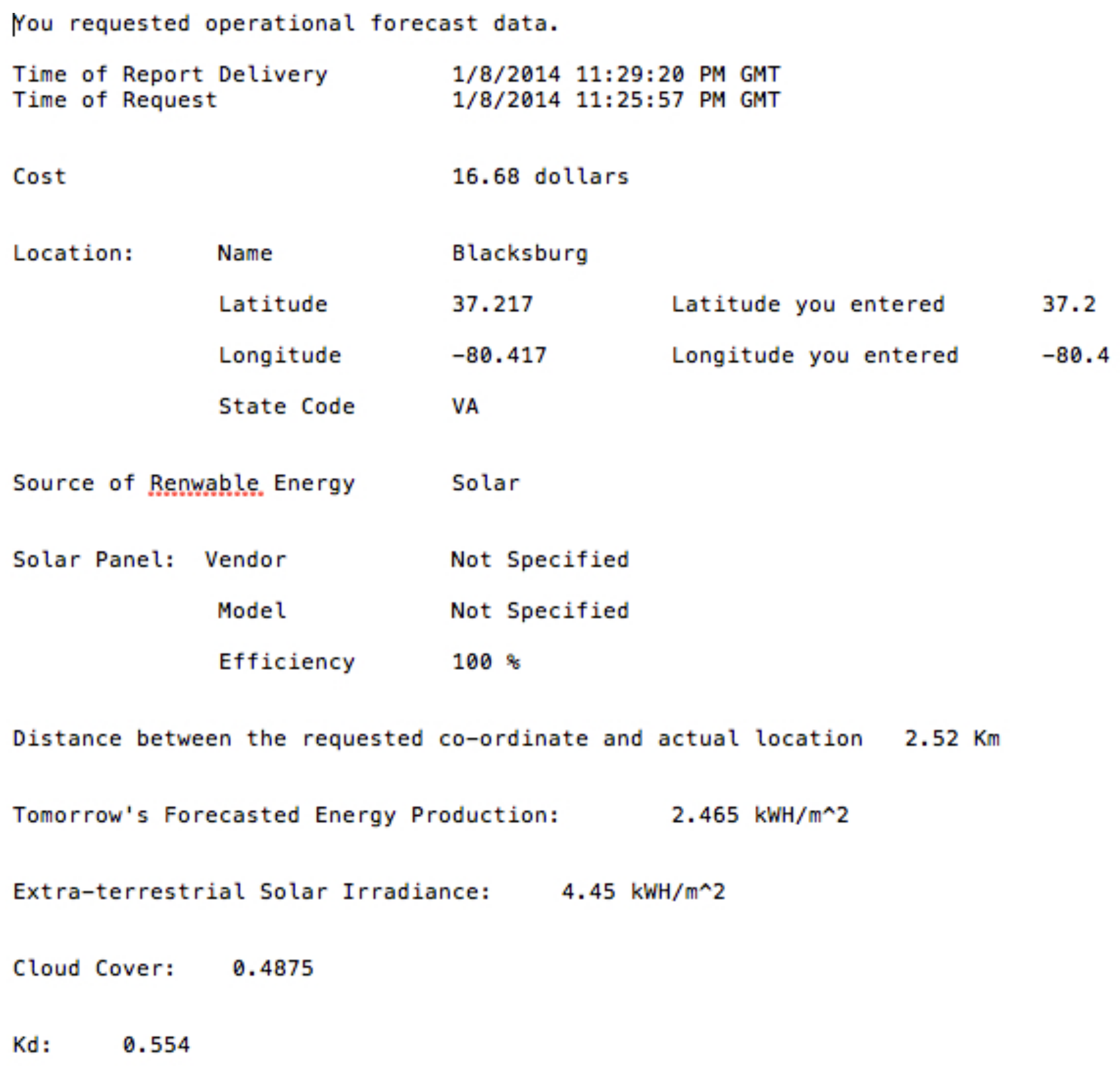


```
You requested Project Prospecting data with uncertainty quantification.
Time of Report Delivery        4/29/2013 6:09:59 PM GMT

Time of Request                4/29/2013 6:08:56 PM GMT


Cost                           72.43 dollars


Location:     Name             GUAM WFO

              Latitude         13.483         Latitude you entered      13.4

              Longitude        144.8          Longitude you entered     144

              State Code       GU


Source of Renwable Energy      Solar


Solar Panel:  Vendor           Sun Power

              Model            SPR-200-WHT-U

              Efficiency       16.08 %


Distance between the requested co-ordinate and actual location   57.64 Km


Annual Energy                   Average        P95           P90           P75           P50

Production(kWh/m^2)

______________________________________________________________________________________________

                                282.35         280.59        280.98        281.63        282.35

Monthly Energy                  Average        P95           P90           P75           P50

Production(kWh/m^2)

______________________________________________________________________________________________

January                         19.89          19.53         19.61         19.74         19.89

February                        18.51          18.15         18.23         18.36         18.51

March                           27.58          27.11         27.21         27.39         27.58

April                           27.36          26.84         26.96         27.15         27.36

May                             29.07          28.59         28.69         28.87         29.07

June                            28.49          28.01         28.11         28.29         28.49

July                            24.99          24.56         24.66         24.81         24.99

August                          22.73          22.34         22.43         22.57         22.73

September                       21.64          21.21         21.31         21.47         21.64

October                         22.14          21.71         21.81         21.96         22.14

November                        21.9           21.51         21.6          21.74         21.9

December                        18.06          17.28         17.45         17.74         18.06
```

Figure 6.11 Mail content for solar project prospecting with panel data and uncertainty

6.2.2 Operational

Without Solar Panel Data: The following figure shows the content of file received by a user for solar operational forecast for Blacksburg, VA location.



```
You requested operational forecast data.

Time of Report Delivery        1/8/2014 11:29:20 PM GMT
Time of Request                1/8/2014 11:25:57 PM GMT


Cost                           16.68 dollars


Location:       Name           Blacksburg

                Latitude       37.217        Latitude you entered      37.2

                Longitude      -80.417       Longitude you entered     -80.4

                State Code     VA


Source of Renwable Energy      Solar


Solar Panel:   Vendor          Not Specified

                Model          Not Specified

                Efficiency     100 %


Distance between the requested co-ordinate and actual location   2.52 Km


Tomorrow's Forecasted Energy Production:       2.465 kWH/m^2


Extra-terrestrial Solar Irradiance:     4.45 kWH/m^2


Cloud Cover:    0.4875


Kd:     0.554
```

Figure 6.12 Mail content for solar operation request without solar panel data

The cost returned by FaaS Controller is the cost for EDCF and IDRF as shown in table 6.11 and 6.12 with a different cost for FGF as shown in table 6.27 below.

FGF

Table 6.27 Shows cost difference in FGF from wind operational forecast

ForecastGHICost	2.56
ComputeKDCost	0.64
CustomDateTimeCost	2.56
ReportPreparationCost	2.55
BlobWrite	1.47
Total FGF Cost (C_{22})	**9.78**

Total cost returned by FaaS Controller, $C_{23} = C_9 + C_{10} + C_{22} = \$ 16.68$

The forecasted energy in figure 6.12 was obtained by using the forecasted average cloud cover data for every three hours from NOAA (details given in Appendix A.9, *XMLDataDownload*). The average cloud cover, from sunrise to sunset, were used to calculate Total Cloud Cover (TCC) and KD, to compute the energy produced at a given location as shown in table 6.28.

Table 6.28 TCC and KD using forecasted cloud cover data calculated using formula from Appendix C.9

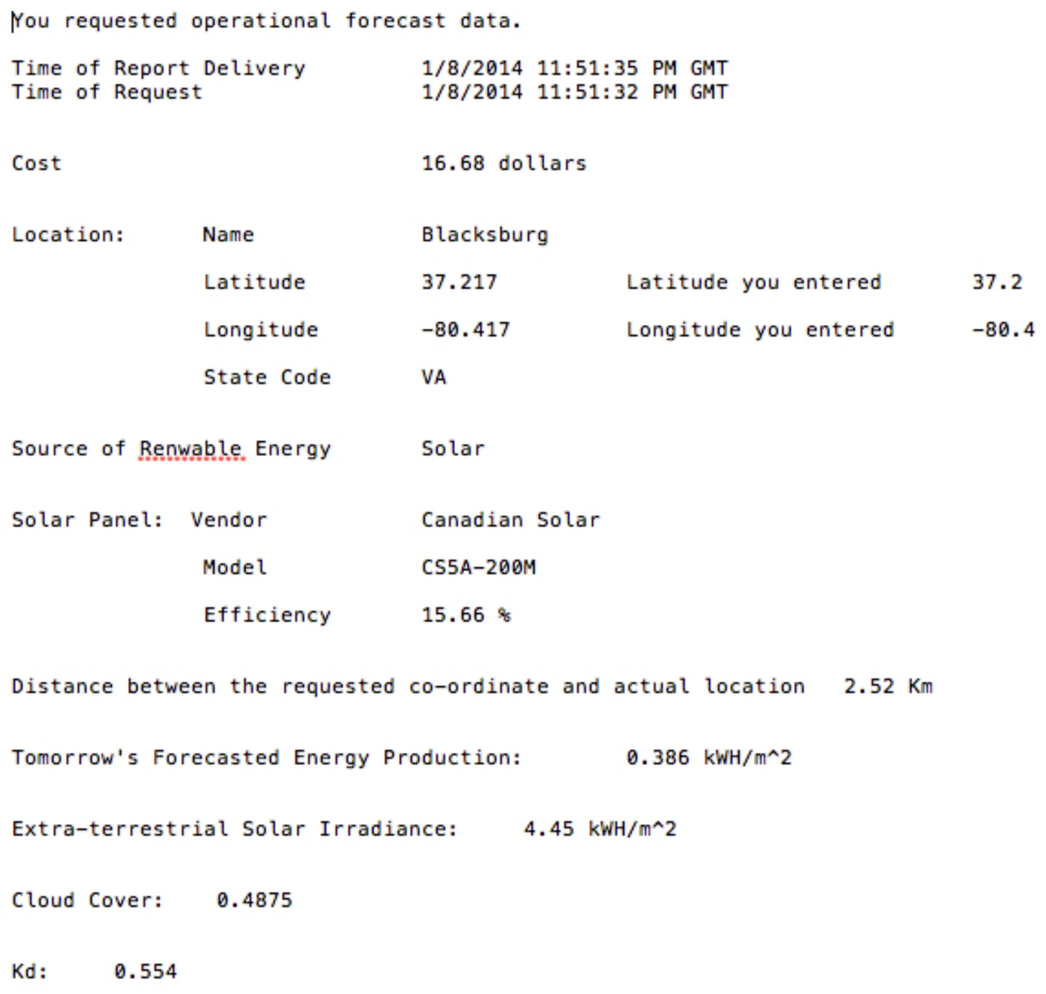
Time	**Forecasted Average Cloud Cover Data Set 1**	**Forecasted Average Cloud Cover Data Set 2**	**Forecasted Average Cloud Cover Data Set 3**
00:00 – 02:29	63	90	24
03:00 – 05:59	56	97	24
06:00 – 08:59	49	98	40
09:00 – 11:59	36	96	33
12:00 – 14:59	50	95	13
15:00 – 17:59	60	96	13
18:00 – 20:59	77	96	16
21:00 – 23:59	83	96	13
TCC (%)	0.4875	0.9625	0.2475
KD	0.553	0.125	0.678
Forecasted Energy (kWh/m^2)	2.465	0.557	3.22

Table 6.29 shows actual cloud cover data for the same days to calculate the energy that could be produced at the same location. A mean percentage error of 5.32% was observed between the actual and forecasted energy values, mainly due to the inaccuracy between forecasted and actual cloud cover data.

Table 6.29 KD and TCC calculated using actual cloud cover data

Time	**Actual Cloud Cover Data Set 1 (%)**	**Actual Cloud Cover Data Set 2 (%)**	**Actual Cloud Cover Data Set 3 (%)**
00:00 – 02:29	59	80	15
03:00 – 05:59	49	98	15
06:00 – 08:59	35	98	20
09:00 – 11:59	12	96	35
12:00 – 14:59	18	95	25
15:00 – 17:59	59	98	10
18:00 – 20:59	61	98	10
21:00 – 23:59	80	85	8
TCC (%)	0.31	0.9625	0.225
KD	0.647	0.125	0.689
Forecasted Energy (kWh/m^2)	2.881	0.557	3.27

With Solar Panel Data: The following figure shows the content of file with a user-selected wind turbine to obtain solar operation forecast.



```
You requested operational forecast data.

Time of Report Delivery        1/8/2014 11:51:35 PM GMT
Time of Request                1/8/2014 11:51:32 PM GMT

Cost                           16.68 dollars

Location:       Name           Blacksburg

                Latitude       37.217         Latitude you entered       37.2

                Longitude      -80.417        Longitude you entered      -80.4

                State Code     VA

Source of Renwable Energy      Solar

Solar Panel:  Vendor           Canadian Solar

                Model          CS5A-200M

                Efficiency     15.66 %

Distance between the requested co-ordinate and actual location   2.52 Km

Tomorrow's Forecasted Energy Production:        0.386 kWH/m^2

Extra-terrestrial Solar Irradiance:     4.45 kWH/m^2

Cloud Cover:    0.4875

Kd:     0.554
```

Figure 6.13 Mail content for solar operation request with panel data

Forecasted energy produce by solar panel *Canadian Solar*

$$= \text{Forecasted energy} * \text{efficiency} = 2.46 * 0.1566 = 0.385 \text{ kWH/m}^2$$

Actual energy produced by solar panel *Canadian Solar*

$$= \text{Actual energy} * \text{efficiency} \quad = 2.881 * 0.1566 = 0.451 \text{ kWH/m}^2$$

Percentage error in energy = 14.68 %

6.3. Analysis of Results

The development effort for the framework was started with project prospect forecasting and thus various service features were created during this phase. From the results in Section 6.1.1, it can be observed that project prospecting cost for SOA services for wind is $62.89 without uncertainty data and $65.44 with uncertainty data. The difference in the cost shows how an added service feature of uncertainty affects the cost of a service. Whereas the costs presented in Section 6.2.1 for solar project prospecting ($69.88/$72.43 without/with uncertainty data), it can be deduced that only an additional cost of $7 was incurred during the development of the services for solar forecasting. These savings in cost result from the reuse of various service features, service contracts & interface contracts.

As the services for operational forecasting were created during the later part of the development and use previously created service features, they have a highly reduced costs of $16.68 and $13.92 for solar and wind respectively. The accuracy of the operation forecast for wind and solar resources, as presented in Section 6.1.2 and 6.2.2, was found to be low (error of 13.7% & 5.32% respectively). This was mainly due to the difference between forecasted and actual values of wind and cloud cover data from respective sources.

Chapter 7

Conclusion

Through the development of this work, it was successfully demonstrated how by applying concepts of SOA, services in forecasting can be offered to an end user on a *pay-per-use* as opposed to traditional fee involved in software services. As shown in Section 2.1, using cloud computing is beneficial for forecasting as a scalable infrastructure is available on demand without long-term commitments.

The architecture of the FaaS presented in Section 4.4 demonstrated how services of FaaS could be used as software to forecast solar and wind data on a *pay-per-use* basis. The FaaS Controller uses services, namely, EDCF, IDRF and FGF by invoking their economic or technical endpoints to obtain cost or produce a forecast. As services are decoupled, they can be individually used in case others are not required. From the analysis in Section 6.3, it can be deduced that using such an approach can result in reduction of cost and development time. As the costs are calculated based on divide and rule, results in Section 6.1 and 6.2 show different costs of \$62.89 or \$68.88 are charged to users based only on the services they use.

As shown in Section 4.5, it can be concluded that the capabilities of EDCF and IDRF can be reused to download and standardize historical data. As the services of FaaS can be extended to forecasting in other domains, discussion in Section 4.6 show one such possible extension in electrical load forecasting. This enables FaaS to be offered as a Platform-as-a-Service to develop composite services for forecasting. By using FaaS, service reuse could results in lower cost for developers and end users as shown by results in Section 6.3.

As every forecasting process can be improved to produce more accurate results, the accuracy of the forecasted data in Section 6.1.2 and Section 6.2.2 can also be increased. This can be done by using wind and cloud cover data from multiple sources as opposed to one.

The ideas, concepts and architecture presented in this thesis are to be further used and enhanced by others working in similar fields. By using the principles of SOA in forecasting together with cloud computing it is believed that extension of FaaS to other domains can enable others to benefit from it too. Further exploration on such frontiers and enhancing the ideas presented in this work in terms of possible applications is a major future work enabled by this thesis.

Bibliography

Aguiar, R. J., et al. (1988). "Simple procedure for generating sequences of daily radiation values using a library of Markov transition matrices." Solar Energy 40(3): 269-279.

Boehm, B.W., C. Abts, Brown, A.W., Chulani, S., Clark, E. Horowitz, B.K., Madachy, R., Reifer, D.J., and Steece, B., Software Cost Estimation with COCOMO II. New Jersey: Prentice Hall PTR, Aug. 2000.

Buyya, Rajkumar, et al. "Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility." Future Generation computer systems 25.6 (2009): 599-616.

Chang, M.; He, J.; Tsai, W.-T.; Bingnan Xiao; Yinong Chen, "UCSOA: User-Centric Service-Oriented Architecture," e-Business Engineering, 2006. ICEBE '06. IEEE International Conference on, vol., no., pp.248,255, Oct. 2006

Computing, Integration, and Assurance. SEUS 2006/WCCIA 2006. The Fourth IEEE Workshop on, vol., no., pp.6 pp.,, 27-28 April 2006

Cost estimation tools: http://www.planforcloud.com/

D. Booth et al., W. C., Working Draft (2003). "Web Services Architecture."

Definition of Cloud Computing: http://cloudcomputing.sys-con.com/node/612375/print

Deelman, E.; Singh, G.; Livny, M.; Berriman, B.; Good, J., "The cost of doing science on the cloud: The Montage example," High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for , vol., no., pp.1,12, 15-21 Nov. 2008

D. Booth et al., Web Services Architecture (W3C, Working Draft, 2003; www.w3.org/TR/2003/WD-ws-arch-20030808).

Erl, T. (2004). Service-oriented architecture: a field guide to integrating XML and web services, Prentice Hall PTR.

Erl, T. (2005). Service-oriented architecture, Prentice Hall, New York.

Erl, T., et al. (2010). SOA with. NET and Windows Azure: Realizing Service-orientation with the Microsoft Platform, Pearson Education.

Foster, Ian, Yong Zhao, Ioan Raicu, and Shiyong Lu. "Cloud computing and grid computing 360-degree compared." In Grid Computing Environments Workshop, 2008. GCE'08, pp. 1-10. Ieee, 2008.

Foster, I. (2005). "Service-oriented science." science 308(5723): 814-817.

Fox, Armando, et al. "Above the clouds: A Berkeley view of cloud computing." Dept. Electrical Eng. and Computer Sciences, University of California, Berkeley, Rep. UCB/EECS 28 (2009).

Great-circle distance, Retrieved 10/10/2013, from http://en.wikipedia.org/wiki/Great-circle_distance.

Giebel G., Brownsword R., Kariniotakis G., Denhard M., Draxl C. The State-Of-The-Art in Short-Term Prediction of Wind Power A Literature Overview, 2nd Edition. Project report for the Anemos.plus and SafeWind projects. 110 pp. Risø, Roskilde, Denmark, 2011

Henry Li. 2009. Introducing Windows Azure. Apress, Berkely, CA, USA.

Hoffa, C., Mehta, G., Freeman, T., Deelman, E., Keahey, K., Berriman, B., & Good, J. (2008, December). On the use of cloud computing for scientific workflows. In eScience, 2008. eScience'08. IEEE Fourth International Conference on (pp. 640-645). IEEE.

Hong-Linh Truong, Schahram Dustdar, Composable cost estimation and monitoring for computational applications in cloud computing environments, Procedia Computer Science, Volume 1, Issue 1, May 2010, Pages 2175-2184, ISSN 1877-0509

Huang, Y., et al. (2010). "Framework for building a low-cost, scalable, and secured platform for web-delivered business services." IBM Journal of Research and Development 54(6): 4: 1-4: 14.

HuiMin Zhang; Xiaolong Yang, "Cloud Computing Architecture Based-On SOA," Computational Intelligence and Design (ISCID), 2012 Fifth International Symposium on, vol.1, no., pp.369,373, 28-29 Oct. 2012

Krishnappa, Dilip Kumar, et al. "Network capabilities of cloud services for a real time scientific application." Local Computer Networks (LCN), 2012 IEEE 37th Conference on. IEEE, 2012.

Krishnappa, Dilip Kumar, et al. "CloudCast: Cloud computing for short-term mobile weather forecasts." Performance Computing and Communications Conference (IPCCC), 2012 IEEE 31st International. IEEE, 2012.

Keahey, K., Freeman, T. “Science Clouds: Early Experiences in Cloud computing for Scientific Applications,” Cloud Computing and Its Applications 2008 (CCA-08), Chicago, IL. October 2008.

Krafzig, D., et al. (2005). Enterprise SOA: service-oriented architecture best practices, Prentice Hall Professional.

Krafzig, D., et. al. (2005). Elements of a Service Oriented Architecture - Enterprise SOA, Prentice Hall.

Li, Zheng, and Jacky Keung. "Software Cost Estimation Framework for Service-Oriented Architecture Systems using Divide-and-Conquer Approach." Service Oriented System Engineering (SOSE), 2010 Fifth IEEE International Symposium on. IEEE, 2010.

Linthicum DS. Cloud computing and SOA convergence in your enterprise: a step-by-step guide: Addison-Wesley; 2010-01-01.

McCalman, S. Forecasting. National Economic Review, 41-43 (2012, 11).

Mass, Clifford, and Clifford F. Mass. "Nowcasting: The Next Revolution in Weather Prediction." Bulletin of the American Meteorological Society (2011).

Mell, P. and T. Grance (2011). "The NIST definition of cloud computing (draft)." NIST special publication 800(145): 7.

Microsoft Azure: http://www.windowsazure.com/en-us/

National Weather Service from http://www.nws.noaa.gov/

National Weather Service data: http://graphical.weather.gov

NREL TMY solar data: http://rredc.nrel.gov/solar/old_data/nsrdb/1991-2010/

NREL MIDC solar data: http://midcdmz.nrel.gov

NREL wind integration datasets
http://www.nrel.gov/electricity/transmission/wind_integration_dataset.html

Paleologo, Giuseppe A. "Price-at-Risk: A methodology for pricing utility computing services." IBM Systems Journal 43.1 (2004): 20-31.

Peng, Junjie, et al. "Comparison of several cloud computing platforms." Information Science and Engineering (ISISE), 2009 Second International Symposium on. IEEE, 2009.

R.J. Aguiar, M. Collares-Pereira, J.P. Conde, Simple procedure for generating sequences of daily radiation values using a library of Markov transition matrices, Solar Energy, Volume 40, Issue 3, 1988, Pages 269-279, ISSN 0038-092X

Rodriguez-Martinez, Manuel, Jaime Seguel, and Melvin Greer. "Open source cloud computing tools: A case study with a weather application." Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on. IEEE, 2010.

Science Cloud: http://scienceclouds.org

Shiyin Zhong; Tam, K., "A Frequency Domain Approach to Characterize and Analyze Load Profiles," Power Systems, IEEE Transactions on , vol.27, no.2, pp.857,865, May 2012

Simple Object Access Protocol (SOAP) [URL]. http://www.w3.org/tr/soap/, access on Nov. 2013.

Truong, Hong-Linh, and Schahram Dustdar. "Composable cost estimation and monitoring for computational applications in cloud computing environments." Procedia Computer Science 1.1 (2010): 2175-2184.

Tsai, W.T.; Bingnan Xiao; Paul, R.A.; Yinong Chen, "Consumer-centric service-oriented architecture: a new approach," Software Technologies for Future Embedded and Ubiquitous Systems, 2006 and the 2006 Second International Workshop on Collaborative

Tsai, W.-T., et al. (2010). Service-oriented cloud computing architecture. Information Technology: New Generations (ITNG), 2010 Seventh International Conference on, IEEE.

Vecchiola, Christian, Xingchen Chu, and Rajkumar Buyya. "Aneka: a software platform for .NET-based cloud computing." High Speed and Large Scale Scientific Computing (2009): 267-295.

Vecchiola, C.; Pandey, S.; Buyya, R., "High-Performance Cloud Computing: A View of Scientific Applications," Pervasive Systems, Algorithms, and Networks (ISPAN), 2009 10th International Symposium on , vol., no., pp.4,16, 14-16 Dec. 2009

Virk, Ishpreet Singh. "Cloud Computing: Windows Azure platform." Journal of Global Research in Computer Science 3, no. 1 (2012): 74-76.

Wang, Lizhe, Jie Tao, Marcel Kunze, Alvaro Canales Castellanos, David Kramer, and Wolfgang Karl. "Scientific cloud computing: Early definition and experience." In High Performance Computing and Communications, 2008. HPCC'08. 10th IEEE International Conference on, pp. 825-830. Ieee, 2008.

Web Service Description Language (WSDL) [URL]. http://www.w3.org/tr/wsdl/, access on Nov. 2013.

Yeo, Chee Shin, Srikumar Venugopal, Xingchen Chu, and Rajkumar Buyya. "Autonomic metered pricing for a utility computing service." Future Generation Computer Systems 26, no. 8 (2010): 1368-1380.

Ying Huang; Hui Su; Wei Sun; Jian Ming Zhang; Chang Jie Guo; Jing Min Xu; Jiang, Z.B.; Yang, S.X.; Jun Zhu, "Framework for building a low-cost, scalable, and secured platform for Web-delivered business services," IBM Journal of Research and Development, vol.54, no.6, pp.4:1,4:14, Nov.-Dec. 2010

Zhang, H. and X. Yang (2012). Cloud Computing Architecture Based-On SOA. Computational Intelligence and Design (ISCID), 2012 Fifth International Symposium on, IEEE.

Zhenkai, Wang Long Wan. "Research on Service-oriented Architecture Cloud Computing and its Implementation [J]." Computer & Digital Engineering 7 (2009): 027.

Zhong, S. and K. Tam (2012). "A Frequency Domain Approach to Characterize and Analyze Load Profiles." Power Systems, IEEE Transactions on 27(2): 857-865.

APPENDIX A – Namespaces & Classes

This appendix lists all the namespaces and their respective classes with their data members and member functions that were developed for the implementation of the services in FaaS. The API's developed are listed as:

Namespace.ClassName

And contains a list of all the data members and member function that are available to be used as a platform to develop other services.

A.1. COMMUNICATION TOOLS

1. EmailServices: This service feature uses .NET API MailMessage and SmtpClient in order to send email to the user.

```
public void MailFileToUser(Stream fileStream, string emailID, string filenameToUser)
```

API list for namespace Communication Tools:

CommunicationTools.EmailServices

Data Members:

I.	**Access modifier:**	private
	Type:	static string
	Name:	adminAccountEmailID
	Description:	stores the mail address of the admin
II.	**Access Modifier:**	private
	Type:	string
	Name:	adminAccountPassword
	Description:	stores the password of the admin email account
III.	**Access Modifier:**	public, private

Type: double
Name: EmailServiceCost, emailServiceCost
Description: To hold the cost of using EmailServices class.

Methods:

I. **Access modifier:** public
Name: MailFileToUser
Description: emails the file provided as a stream from the blob to the user
Parameters: **In:** Stream fileStream, stream of the file to be emailed to the user
string emailID, email id of the user
string filename, as the user would see
Out: None
Return Type: void

A.2. COSTING TOOLS

This tool contains service features to retrieve and update the cost of infrastructure and service costs that are stored in the SolarMetaDataTable and MetaDataTable (for wind).

2. InfrastructureCost: This class is used to initialize the infrastructure costs.

```
public bool UpdateCostInTable(string nameOfService, double newCost)
```

By including the following statement the data members and member functions can be accessed by services:

CostingTools. InfrastructureCost

Data Members:

I. **Access modifier:** private
Type: static TableRead
Name: readFromTable
Description: object to read costs from the table

II. **Access Modifier:** private

Type: static TableWrite
Name: writeToTable
Description: object to write updated costs to the table

III. **Access Modifier:** private
Type: static IEnumberable<CostDataEntry>
Name: costs
Description: holds the values of costs read from table in order to initialize the values used by the code.

Methods:

I. **Access modifier:** None
Name: static ServiceCost
Description: static constructor of the class initializes the private data members
Parameters: **In:** None
Out: None
Return Type: None

II. **Access modifier:** private static
Name: InitializeCosts
Description: Initializes the cost of all the service features in the code
Parameters: **In:** None
Out: None
Return Type: void

III. **Access modifier:** public
Name: UpdateCostInTable
Description: Updates the costs of service features in the table
Parameters: **In:** string nameOfService
double newCost
Out: None
Return Type: bool

3. ServiceFeatureCost: To update or add the cost of service features and services-

```
public bool UpdateCostInTable(string nameOfService, double newCost)
```

By including the following statement the data members and member functions can be accessed by services:

CostingTools. ServiceFeatureCost

Data Members:

I. **Access modifier:** private
Type: static TableRead
Name: readFromTable
Description: object to read costs from the table

II. **Access Modifier:** private
Type: static TableWrite
Name: writeToTable
Description: object to write updated costs to the table

III. **Access Modifier:** private
Type: static IEnumberable<CostDataEntry>
Name: costs
Description: holds the values of costs read from table in order to initialize the values in code

Methods:

I. **Access modifier:** None
Name: static ServiceCost
Description: static constructor of the class initializes the private data members
Parameters: **In:** None
Out: None
Return Type: None

II. **Access modifier:** private static
Name: InitializeCosts
Description: this method initializes the cost of all the service features or services in the code
Parameters: **In:** None
Out: None
Return Type: void

III. **Access modifier:** public
Name: UpdateCostInTable
Description: Updates the costs of service features or services in the table
Parameters: **In:** string nameOfService
double newCost
Out: None
Return Type: bool

A.3. FORECAST TOOLS

4. ComputeKD: This class is used to compute the value of KD given the value of total cloud cover (TCC). It normalizes the value of TCC to not be greater than 0.9009.

```
public ComputeKD(double tcc)
```

By including the following statement the data members and member functions can be accessed by services:

ForecastTools. ComputeKD

<u>**Data Members:**</u>

I. Access modifier:	private, public
Type:	double
Name:	tcc, TCC
Description:	Holds the value of TCC
II. Access Modifier:	public, private
Type:	double
Name:	_kd, _KD
Description:	Holds the computed value of KD.
III. Access Modifier:	public, private
Type:	double
Name:	kdComputationCost, KDComputationCost
Description:	Holds the cost of using ComputerKD feature.

<u>**Methods:**</u>

I. Access modifier:	public
Name:	ComputeKD
Description:	Computes the value of KD for given TCC
Parameters:	**In:** double tcc
	Out: None
Return Type:	The computed value of KD is accessible through data member _KD

5. ForecastGHIET: This class uses the formulas listed below to calculate daily, month and yearly GHI ET values.

By including the following statement the data members and member functions can be accessed by services:

ForecastTools.ForecastGHIET

To compute GHI_ET for next day and to get the sunset and sunrise time -

```
public double ComputeGHIET()
```

This functionality is based on the following formula for calculation of extra-terrestrial radiation:

I0 one hour:

$$I_0 = 1367 * E_0 * cos\theta_z$$

I0 in one day (Wh/m2): sum of the hourly total from sunrise hour to sunset hour for that day (each day is different).

I0 in one month (Whr/m2): sum of the daily total for all the days in one month (each day is different, so each month with 31 days is different).

I0 in one year (Whr/m2): sum of the daily total over 12 months.

$$dayAngle = \frac{2 * \pi * (dayNumber - 1)}{365}$$

E0 = 1.00010 + 0.034221 * cos (dayAngle) + 0.001280 * sin (dayAngle)

+ 0.000719 * cos (2 * dayAngle) + 0.000077 * sin (2 * dayAngle);

solarDeclination, δ = (0.006918 - 0.399912 * cos(dayAngle)

+ 0.070257 * sin (dayAngle) - 0.006758 * cos (2 * dayAngle)

+ 0.000907 * sin (2 * dayAngle) - 0.002697 * cos (3 * dayAngle)

+ 0.00148 * sin (3 * dayAngle)

θz = cos-1 [sin(δ) * sin(latitude) + cos(δ) * cos(latitude) * cos(hourA)]

Sunrise Hour Angle = cos-1 (-1 * tan (latitude) * tan(δ))

Sunset Hour Angle = -1 * sunriseHourAngle

Sunrise Time = (12 - (sunriseHourAngle * 180 / PI))

Sunset Time = (12 - (sunsetHourAngle * 180 / PI)

Data Members:

I.	**Access modifier:**	private, public
	Type:	double
	Name:	latitude, Latitude
	Description:	Gives the value of latitude provided by the user
II.	**Access Modifier:**	private, public
	Type:	double
	Name:	ghi_et, GHI_ET
	Description:	holds the calculated value of GHI ET
III.	**Access Modifier:**	private, public
	Type:	double
	Name:	sunriseTime, SunriseTime
	Description:	holds the sunrise time
IV.	**Access Modifier:**	private, public
	Type:	double
	Name:	sunsetTime, SunsetTime
	Description:	holds the sunset time
V.	**Access Modifier:**	public, private
	Type:	double
	Name:	ghiForecastCost, GHIForecastCost
	Description:	To hold the cost of using ForecastGHIET class.

Methods:

I. **Access modifier:** public
Name: ForecastGHIET
Description: Provided a given begin and/or end time for a given latitude, it initializes various data members of the class ForecastGHIET
Parameters: **In:** string beginTime, string endTime, double latitude
Out: None
Return Type: None

II.	**Access modifier:**	public
	Name:	ComputeGHIET
	Description:	Caculates the daily, monthly and yearly values of GHI ET.
	Parameters:	**In:** None
		Out: None
	Return Type:	double, returns the calculated value of GHI ET

6. GC_Distance: There are two overloaded functions; one calculates the distance between two points. The other one calculates distance of a given point with multiple points. It uses Great Circle distance algorithm [Great Circle Distance] to find sites in close proximity to the user provided location.

```
public double GreatCircleDistance(double latitude1, double longitude1,
                                  double latitude, double longitude)
```

By including the following statement the data members and member functions can be accessed by services:

ForecastTools.GC_Distance

<u>**Data Members:**</u>

I.	**Access modifier:**	private, public
	Type:	double
	Name:	latitude, Latitude
	Description:	Gives the value of latitude provided by the user
II.	**Access modifier:**	private, public
	Type:	double
	Name:	longitude, Longitude
	Description:	Gives the value of longitude provided by the user
III.	**Access modifier:**	private
	Type:	double
	Name:	RoE
	Description:	Holds the value of radius of earth (6371 Km.)
IV.	**Access Modifier:**	private, public
	Type:	double
	Name:	distance, Distance

Description: holds the calculated value of minimum distance between sites

V. **Access Modifier:** public, private
Type: double
Name: gcDistanceCost, GCDistanceCost
Description: To hold the cost of using GC_Distance class.

Methods:

I. **Access modifier:** public
Name: GC_Distance
Description: Initializes the values of RoE
Parameters: **In:** None
Out: None
Return Type: None

II. **Access modifier:** public
Name: GreatCircleDistance
Description: Calculates the great circle distance between two points.
Parameters: **In:** double latitude1, double longitude1, double latitude, double longitude
Out: None
Return Type: double, returns the calculated great circle distance

III. **Access modifier:** public
Name: GreatCircleDistance
Description: Calculates the great circle distance between user provided location and an array of closest locations.
Parameters: **In:** double[] latitudes, double[] longitudes, double latitude, double longitude
Out: None
Return Type: double, returns the calculated great circle distance

7. PowerForecast: This class is used to calculated wind power generated through a given wind turbine, when provided with wind speeds for a particular location.

```
public double ForecastPower( List<Tuple<double, double>> powerCurve,
                             List<double> windSpeed )
```

By including the following statement the data members and member functions can be accessed by services:

ForecastTools.PowerForecast

<u>**Data Members:**</u>

I. Access modifier: private, public
Type: double
Name: totalPower, TotalPower
Description: stores the total projected power for a particular site

II. Access Modifier: private
Type: List<<Tuple<double, double>>
Name: powerCurve
Description: holds the power curve as a pair

III. Access Modifier: public, private
Type: double
Name: PowerForecastCost, powerForecastCost
Description: To hold the cost of using PowerForecast class.

<u>**Methods:**</u>

I. Access modifier: public
Name: ForecastPower
Description: forecasts power for a location given the power curve for a manufacturer and the yearly wind speeds for that location.
Parameters: **In:** List<Tupe<double, double>> powerCurve, List<double> windSpeed, list of yearly wind speeds
Out: None
Return Type: double, returns total projected power rounded off to 3 decimal places

II. Access modifier: private
Name: ProjectedPower
Description: Calculates the project power at any given speed (uses the data member powerCurve to project power between two speed values).
Parameters: **In:** double speed
Out: None
Return Type: double, return projected power

III. Access modifier: private
Name: InterpolatePower
Description: used by ProjectedPower to calculate the power at a given speed
Parameters: **In:** double power, double speed, double ratio
Out: None

Return Type:	double, project power at the given wind speed

8. ReadForecastDataFromXML: This class automatically detects the instance calling it (wind or solar) and depending on which instance calls it, this reads the next week's forecasted values of wind speeds or cloud cover from NOAA's XML Document downloaded by other services.

```
public double[] GetForecastData(XmlDocument doc)
public double GetForecastData(XmlDocument doc, double sunriseTime, double sunsetTime)
```

By including the following statement the data members and member functions can be accessed by services:

ForecastTools. ReadForecastDataFromXML

Data Members:

I.	**Access modifier:**	private, public
	Type:	double
	Name:	tcc, TCC
	Description:	Holds the calculated value of TCC
II.	**Access modifier:**	private
	Type:	string
	Name:	typeOfSource
	Description:	Is used to detect if the instance is solar or wind
III.	**Access Modifier:**	private, public
	Type:	double
	Name:	readForecastXMLCost, ReadForecastXMLCost
	Description:	holds the cost for using this service feature

Methods:

I.	**Access modifier:**	public
	Name:	ReadForecastDataFromXML
	Description:	Initializes the typeOfSource to either wind or solar
	Parameters:	**In:** None
		Out: None
	Return Type:	None
II.	**Access modifier:**	public

Name: GetForecastData
Description: Calculates the forecasted wind speeds for the next day.
Parameters: **In:** XmlDocument doc
Out: None
Return Type: double[], returns wind speeds(mph) for entire day, for every three hours

III. Access modifier: public
Name: GetForecastData
Description: reads cloud cover from XMLDocument downloaded by transfer tools and converts it to TCC using sunrise time and sunset time.
Parameters: **In:** XmlDocument doc, double sunriseTime, double sunsetTime
Out: None
Return Type: double, returns the normalized value of TCC

A.4. FOUNDATION TOOLS

9. DailyDataToMonthlyData: This class converts the daily data available in a single array for the entire year to monthly data for the entire year and stores it in the *SolarDataTable*. It takes daily data as an input to its member function and stores them in *SolarDataTable*. It also reads the daily data back from the *SolarDataTable* and returns the monthly data values in a single array with 12 elements (one for each month for each year). This class is used to store the data in data table use the following method and supply the "dailyData" for entire year in a single array

```
public void StoreMonthlyDataInTable(double latitude, double longitude, int year,
                                     string typeOfData, double[] dailyData)
```

To read the data back from table

```
public void ReadMonthlyDataFromTable(double latitude, double longitude, int year,
                                      string typeOfData, out double[] monthlyData)
```

By including the following statement the data members and member functions can be accessed by services:

FoundationTools.DailyDataToMonthlyData

Data Members:

I. **Access Modifier:** private, public
Type: double[]
Name: monthlyData
Description: To hold values of monthly data in a single array.

II. **Access Modifier:** private, public
Type: double
Name: monthlyDataCost, MonthlyDataCost
Description: To hold the cost of using this feature. Sums up the cost of other service features being used by this class.

III. **Access Modifier:** private
Type: TableRead
Name: readFromTable
Description: Object of class TableRead to read monthly data stored in DataTable.

IV. **Access Modifier:** private
Type: TableWrite
Name: writeToTable
Description: Object of class TableWrite to store monthly data in DataTable.

Methods:

I. **Access modifier:** public
Name: DailyDataToMonthlyData
Description: This is a static constructor for the class that initializes the TableRead and TableWrite members of the class.
Parameters: **In:** None
Out: none
Return Type: None

II. **Access modifier:** public
Name: StoreMonthlyDataInTable
Description: Convert the given daily data for an entire year, into monthly data by adding the daily data for number of days in a given month. Stores this data in DataTable
Parameters: **In:** double latitude, double longitude, Int year

	String typeOfData: value has to be one of [SCI, GHI, GHI_ET or GHI_Uncert] Double[] dailyData **Out:** None
Return Type:	None

III.	**Access modifier:**	public
	Name:	ReadMonthlyDataFromTable
	Description:	Returns the monthly data in a single array for entire year.
	Parameters:	**In:** double latitude, longitude, Int year String typeOfData, has to be SCI, GHI, GHI_ET or GHI_Uncert, depending on which type of data do you want to read. **Out:** double[] monthlyData
	Return Type:	void

10. FormShapeDescriptor: Wind speed values are stored as form shape descriptor in tables -

```
public void FormShapeDescriptor(double DC, double[] magnitude, double[] phase)
```

By including the following statement the data members and member functions can be accessed by services:

FoundationTools.FormShapeDescriptor

This class helps create shape using the magnitude and phase of the harmonics, so that it can be stored in the DataTable. The shape descriptor has the following format:

K_DC_HM1.HP1_HM2.HP2_.............

Where,

K = number of harmonics required to be stored in order to be able to regenerate the original data within 2% MAPE.

DC = DC component of the harmonics.*

HM1 = Magnitude of the first harmonic.*

HP1 = Phase of the first harmonic.*

* Decimal point (.) is replaced with a comma (,).

TableWrite and TableRead are used by this service feature to store the harmonics as Shape descriptor in the data table. The data it is stored as shape descriptor (string) and when reading from table, it is read into K, magnitude, phase (int, double, double)

Data Members:

I.	**Access modifier:**	private, public
	Type:	int
	Name:	k, K
	Description:	holds the value of K in shape descriptor
II.	**Access Modifier:**	private, public
	Type:	double[]
	Name:	magnitude, Magnitude
	Description:	holds the value of magnitude used in the shape descriptor.
III.	**Access Modifier:**	private, public
	Type:	double[]
	Name:	phase, Phase
	Description:	holds the value of phase used in the shape descriptor.
IV.	**Access Modifier:**	private, public
	Type:	string
	Name:	shapeDescriptor, ShapeDescriptor
	Description:	holds the value shape descriptor created using the magnitude and phase of the harmonics and the DC component.
V.	**Access Modifier:**	private, public
	Type:	double[]
	Name:	phase, Phase
	Description:	holds the value of phase used in the shape descriptor.
VI.	**Access Modifier:**	public, private
	Type:	double
	Name:	FormShapeCost, forShapeCost
	Description:	To hold the cost of using FormShapeDescriptor class.

Methods:

I.	**Access modifier:**	public
	Name:	FormShapeDescriptor

Description: Constructor for the class, takes values of DC and harmonics and converts them to shape descriptor so that it can be stored in the table.
Parameters: **In:** double DC, hold the value of DC component
Double[] magnitude, holds the values of magnitudes of harmonics.
Double[] phase, holds the values of phases of harmonics.
Out: None
Return Type: Stores the value in public access variable, ShapeDescriptor, to be accessed by the caller.

II. **Access modifier:** public
Name: FormShapeDescriptor
Description: Constructor for the class, used while retrieving the values from the table and converting the harmonics back to magnitudes and phases.
Stores DC at the 0th index of magnitude array.
Parameters: **In:** string shape, takes shape descriptor as the input, as stored in the table.
Out: None
Return Type: Initializes and stores the values in all the public access variables that provides the data from the shape descriptor.

III. **Access modifier:** private
Name: DoubleWithoutDecimalToString
Description: converts a double type of value to string, replaces decimal point with comma.
Parameters: **In:** double value, takes the double type of value as input
Out: None
Return Type: returns the value as string

IV. **Access modifier:** private
Name: StringToDoubleWithDecimal
Description: convert a string previously converted to DoubleWithoutDecimalToString to double type.
Parameters: **In:** string value, takes the Nth harmonic containing magnitude and phase as a string.
Out: double magnitude, returns magnitude as double type.
Double phase, returns phase as double type.
Return Type: void

V. **Access modifier:** private
Name: DataFromShapeDescriptor
Description: used by the constructor to initialize the magnitude and phase of the harmonics using the shape descriptor.

Parameters:	**In:** None
	Out: None
Return Type:	void

11. Harmonics: This class performs the converts the wind speed data from time domain to frequency domain and stores the magnitude and phase of the harmonics in table. It is used by operation contract *HarmonicsToTable* to convert a days wind speed data to harmonics, and calculates number of harmonics required for 2% MAPE, and then stores them in table.

```
private bool HarmonicsToTable(string source, double latitude, double longitude,
                              string date, int height, double maxWindSpeed,
                              string maxWindTime, double[] daysData, int year)
```

By including the following statement the data members and member functions can be accessed by services:

FoundationTools.Harmonics

The DC component is calculated as follows:

$$M_{dc} = X_{H0} + X_{H12}$$

The other harmonics are calculated as follows:

$$M_{Hn} = 2 * \text{sqrt}(XReal_{Hn}^{2} + XImag_{Hn}^{2})$$

$$P_{Hn} = \tan^{-1}(XImag_{Hn}/XReal_{Hn})$$

where,

X_{Hn} is a complex number obtained after performing DFT on nth point., having XReal and XImag as real and imaginary parts.

Mdc, is the magnitude of the DC component

And M_{Hn} and P_{Hn} are the magnitude and phase of the nth harmonic.

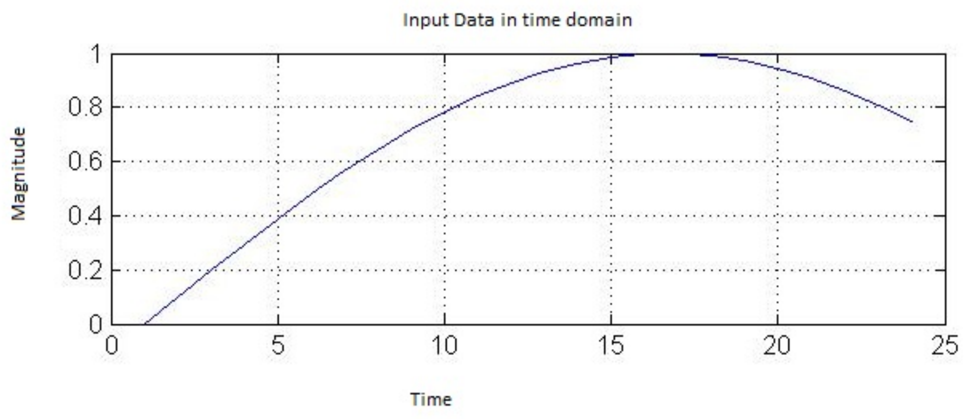


Figure A.1. Data in time domain

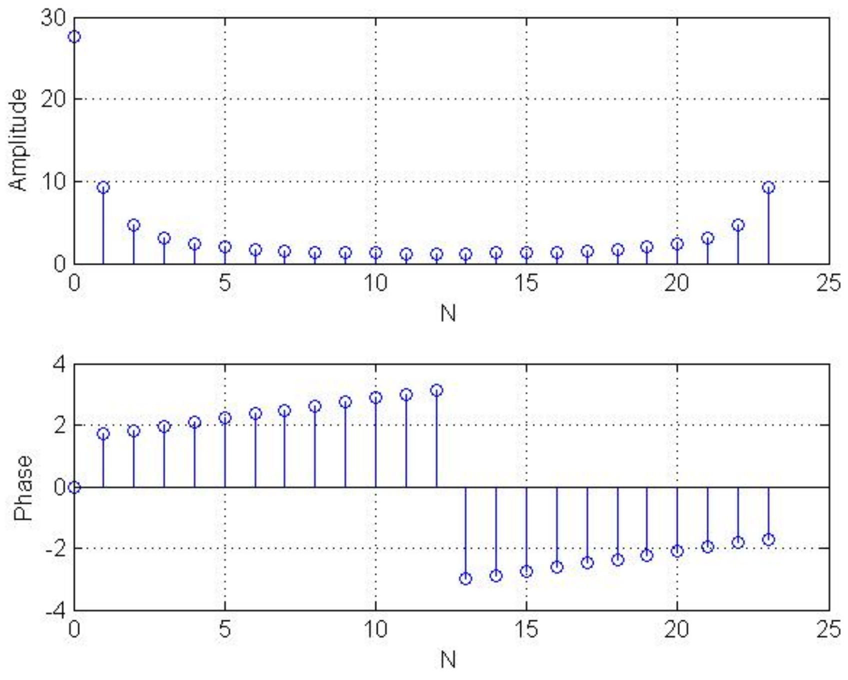


Figure A.2. Data in frequency domain

Data Members:

I. Access modifier: private
Type: int
Name: n
Description: stores the number of harmonics and DC components in the first half of the frequency spectrum.

II. Access Modifier: private, public

Type: double[]
Name: magnitude, Magnitude
Description: holds the values of magnitude of the DC component and harmonics, with DC being at the 0th position in Magnitude array.

III. **Access Modifier:** private, public
Type: double[]
Name: phase, Phase
Description: holds the values of phase of the harmonics.

IV. **Access Modifier:** public, private
Type: double
Name: HarmonicsCost, harmonicsCost
Description: To hold the cost of using Harmonics class.

<u>**Methods:**</u>

I. **Access modifier:** public
Name: Harmonic
Description: Initializes the value of n, number of harmonics and the arrays to store magnitude and phase of the harmonics.
Parameters: **In:** int N, the number of input points on which DFT will be performed to obtain harmonics.
Out: None
Return Type: None

II. **Access modifier:** public
Name: calculateHarmonics
Description: This class calculates the real and imaginary points by performing DFT operation on the given points, and then it calculates the conjugate for all the complex points to finally compute the harmonics.
Parameters: **In:** double[] x, an array of input points to perform DFT
Out: None
Return Type: void

12. HourlyDataToDailyData: This class converts the hourly data available in a single array for the entire year to daily data for each day for the entire year and stores each day’s worth of data as a single entity in the *SolarDataTable*. It takes hourly data as an input to its member function and stores them in *SolarDataTable*. It also reads the daily data back from the *SolarDataTable* and returns it in a single array of 365 (number of days in a year). When supplied with correct parameters, this class creates a primary key, row key and stores the hourly data as a comma separated string for a single day in the table. In case the data to be stored is uncertainty data, sigma's are combined and the stored in table.

```
public void StoreDailyDataToTable(double latitude, double longitude,
                                  string typeOfData, string[] dates,
                                  string[] startTime, string[] endTime,
                                  double[] hourlyData)
```

To read the data back from the table:

```
public void ReadDailyDataFromTable(double latitude, double longitude,
                                  int year, string typeOfData, out string[] date,
                                  out double[] dailyData)
```

By including the following statement the data members and member functions can be accessed by services:

FoundationTools.HourlyDataToDailyData

Data Members:

I. **Access Modifier:** private, public
Type: double[]
Name: dailyPSPData, DailyPSPData
Description: To hold values of daily data in a single array.

II. **Access Modifier:** private, public
Type: double
Name: dailyDataCost, DailyDataCost
Description: To hold the cost of using the service features. Sums up the cost of other service features being used by this class.

III. **Access Modifier:** private
Type: TableRead
Name: readFromTable
Description: Object of class TableRead to read daily data stored in DataTable.

IV. **Access Modifier:** private
Type: TableWrite

Name: writeToTable
Description: Object of class TableWrite to store daily data in DataTable.

<u>**Methods:**</u>

I. **Access modifier:** public
Name: HourlyDataToDailyData
Description: This is a static constructor for the class that initializes the TableRead and TableWrite members of the class.
Parameters: **In:** None
Out: none
Return Type: None

II. **Access modifier:** public
Name: StoreDailyDataInTable
Description: Converts the given hourly data for an entire year, into daily data by adding the hourly data for each hour of the day.
Stores the daily data in DataTable with sunrise (startTime) and sunset time (endTime) and date for each day.
Parameters: **In:** double latitude, double longitude,
String[] date, startTime, endTime
Double[] hourlyPSPData, HourlyGHIData
Int year
String typeOfData has to be SCI, GHI and GHI_ET or if the site belongs to RREDC category, then RREDC:GHI and RREDC:GHI_Uncert
Out: None
Return Type: None

III. **Access modifier:** public
Name: ReadHourlyDataFromTable
Description: Returns the hourly data in a single array for entire year with date, sunset and sunrise times.
Parameters: **In:** double latitude, longitude,
String typeOfData has to be SCI, GHI and GHI_ET or if the site belongs to RREDC category, then RREDC:GHI and RREDC:GHI_Uncert
Int year
Out: Out string[] date
Out double[] dailyData
Return Type: None

13. IDFT: This class takes input of magnitudes and phases of the harmonics and converts them to time domain value of wind speeds. The values have to be read from table by calling function and passed here. NumberOfHarmonicsRequired uses this class in order to calculate MAPE, which is in-turn used by IDRF.

```
public void IDFTFromHarmonics(double[] magnitude, double[] phase)
```

By including the following statement the data members and member functions can be accessed by services:

FoundationTools.IDFT

This class converts data from frequency domain to time domain. Input can be harmonics (currently being used) or a set of real and imaginary points.

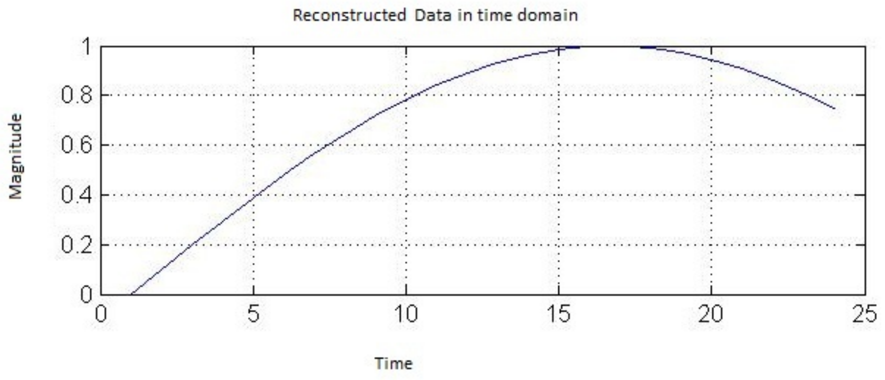


Figure A.3. Data back in time domain

The data is reconstructed in time domain using the cosines of the magnitude and phases of the major harmonics as follows:

$x = M_{dc} + M_{Hn} * \cos (2*PI*0*0/n + P_{Hn}) + M_{Hn-1} * \cos (2*PI*0*0/(n-1) + P_{Hn-1}) + \ldots\ldots$

Data Members:

I. Access modifier:	private
Type:	int
Name:	N
Description:	number of points
II. Access Modifier:	private, public
Type:	double[]

Name: x, X
Description: holds the data regenerated from frequency domain data using harmonics.

III. **Access Modifier:** public, private
Type: double
Name: IDFTCost, idftCost
Description: To hold the cost of using IDFT class.

Methods:

I. **Access modifier:** public
Name: IDFT
Description: initializes the value of n, number of points for which IDFT is to be performed. Allocates memory to x.
Parameters: **In:** int n, number of points to be returned in time domain
Out: None
Return Type: None

II. **Access modifier:** public
Name: IDFTFromHarmonics
Description: converts the data to time domain from frequency domain using the magnitude and phase of the harmonics.
Parameters: **In:** double[] magnitude, takes the magnitude of harmonics, starting with DC component at index 0.
double[] phase, takes the phase of harmonics.
Out:
Return Type: void, assigns the value to public access variable, X

14. MinutelyDataToHourlyData: This class converts the minutely data available in a single array for the entire year to hourly data for the entire year and stores it in the *SolarDataTable*. It takes minutely data as an input to its member function and stores them in *SolarDataTable*. It also reads the minutely data back from the *SolarDataTable* and returns the hourly data values in a single array. The function signatures below shows how the minutely data is converted to daily data, by taking into consideration how many minutes of data there is, what time does the sun rises and set, so that total number of minutes can be calculated. The change of hour is also to be taken into consideration, for the hourly data to be for current hour being considered.

```
public void StoreHourlyDataToTable( double latitude, double longitude,
                                    string typeOfData, int year, string[] date,
                                    string[] startTime, string[] endTime,
                                    double[] data, bool convertMinutelyDataToHourly)
```

To read the data back from table:

```
public void ReadHourlyDataFromTable(double latitude, double longitude,
                                    string typeOfData, int year,
                                     out string[] date, out string[] startTime,
                                     out string[] endTime, out double[] hourlyData)
```

By including the following statement the data members and member functions can be accessed by services:

FoundationTools.MinutelyDataToHourlyData

<u>**Data Members:**</u>

I. **Access Modifier:** private, public
Type: double[]
Name: hourlyData, HourlyData
Description: To hold values of monthly data in a single array returned after reading data from the table.

II. **Access Modifier:** private, public
Type: double
Name: hourlyDataCost, HourlyDataCost
Description: To hold the cost of using the service features. Sums up the cost of other service features being used by this class.

III. **Access Modifier:** private
Type: TableRead
Name: readFromTable
Description: Object of class TableRead to read monthly data stored in DataTable.

IV. **Access Modifier:** private
Type: TableWrite
Name: writeToTable
Description: Object of class TableWrite to store monthly data in DataTable.

Methods:

I. **Access modifier:** public
Name: MinutelyDataToHourlyData
Description: This is a static constructor for the class that initializes the TableRead and TableWrite members of the class.
Parameters: **In:** None
Out: none
Return Type: None

II. **Access modifier:** public
Name: StoreHourlyDataInTable
Description: Convert the given daily data for an entire year, into monthly data by adding the daily data for number of days in a given month.
Stores this data in DataTable
Parameters: **In:** double latitude, double longitude,
String typeOfData, has to be SCI, GHI, GHI_ET
int year,
String[] date, startTime, endTime,
Double[] data,
Bool convertMinutelyDataToHourly, tells whether data is to be converted to hourly values or is it originally being passed as hourly values as sometime are available
Out: None
Return Type: None

III. **Access modifier:** public
Name: StoreHourlyDataInTable
Description: Convert the given daily data for an entire year, into monthly data by adding the daily data for number of days in a given month.
Stores this data in DataTable
Parameters: **In:** double latitude, double longitude,
String typeOfData, has to be GHI_Uncert
int year,
String[] date, startTime, endTime,
Double[] data, double[] uncertaintyPercentage
Out: None
Return Type: None

IV. **Access modifier:** public
Name: ReadHourlyDataFromTable

Description: Returns the hourly data in a single array for each day for entire year.
Parameters: **In:** double latitude, longitude
String typeOfData, has to be SCI, GHI, GHI_ET or GHI_Uncert
int year,
Out: string[] date, startTime, endTime
Double[] hourlyData
Return Type: void

15. MonthlyDataToAnnualData: This class converts the daily data available in a single array for the entire year to monthly data for the entire year and stores it in the *SolarDataTable*. It takes daily data as an input to its member function and stores them in *SolarDataTable*. It also reads the daily data back from the *SolarDataTable* and returns the monthly data values in a single array with 12 elements (one for each month for each year). If the data is uncertainty, it has to be combined to form a new sigma values otherwise simply add the monthly values for each month and store them in table as an annual value. This is how the line to store data above, work:

```
public void StoreAnnualDataInTable(double latitude, double longitude,
                                   int year, string typeOfData, double[] monthlyData)
```

By including the following statement the data members and member functions can be accessed by services:

FoundationTools.MonthlyDataToAnnualData

<u>**Data Members:**</u>

I. Access Modifier: private, public
Type: double
Name: annualPSPData, AnnualPSPData
Description: To hold values of annual power for a given year.

II. Access Modifier: private, public
Type: double
Name: monthlyDataCost, MonthlyDataCost

Description: To hold the cost of using the service features. Sums up the cost of other service features being used by this class.

III. Access Modifier: private
Type: TableRead
Name: readFromTable
Description: Object of class TableRead to read monthly data stored in DataTable.

IV. Access Modifier: private
Type: TableWrite
Name: writeToTable
Description: Object of class TableWrite to store monthly data in DataTable.

<u>**Methods:**</u>

I. Access modifier: public
Name: MonthlyDataToAnnualData
Description: This is a static constructor for the class that initializes the TableRead and TableWrite members of the class.
Parameters: **In:** None
Out: none
Return Type: None

II. Access modifier: public
Name: StoreAnnualDataInTable
Description: Convert the given daily data for an entire year, into monthly data by adding the daily data for number of days in a given month.
Stores this data in DataTable
Parameters: **In:** double latitude, double longitude,
Double[] monthlyData
Int year,
String typeOfData, has to be SCI, GHI, GHI_ET or GHI_Uncert
Out: None
Return Type: None

Return Type: None

16. NumberOfHarmonicsNeeded: This class calculates the minimum number of harmonics needed to be stored in order for the original data to be regenerated within 2% MAPE. This is done by starting with minimum number of harmonics and then calculating them back with IDFT.

```
public int HarmonicsNeeded(double[] input, double[] magnitude, double[] phase)
```

By including the following statement the data members and member functions can be accessed by services:

FoundationTool.NumberOfHarmonicsNeeded

Data Members:

I. Access modifier: private
Type: int
Name: N
Description: holds the number of original points.

II. Access Modifier: private
Type: MAPE
Name: mapeToCalculateError
Description: This object is used to calculate the MAPE between original and regenerated points.

III. Access Modifier: private
Type: IDFT
Name: checkForCurve
Description: This object is used to convert the frequency domain data with a given number of harmonics to check if they are between 2% MAPE.

IV. Access Modifier: public, private
Type: double
Name: NumberOfHarmonicsNeededCost, numberOfHarmonicsNeededCost
Description: To hold the cost of using NumberOfHarmonicsNeeded class.

Methods:

I. Access modifier: public
Name: NumberOfHarmonicsNeeded

Description: This is the default constructor that initializes the object needed to calculate MAPE using the StatisticalTools.

Parameters:	**In:** None
	Out: None
Return Type:	None

II. Access modifier:	public
Name:	HarmonicsNeeded
Description:	calculates the number of harmonics needed for 2% MAPE
Parameters:	**In:** double[] input, holds a set of original input points
	double[] amplitude, holds amplitude of the harmonics
	double[] phase, holds phase of the harmonics
	Out:
Return Type:	int, the number of harmonics required for 2% MAPE

A.5. GENERAL SUPPORT TOOLS

17. Coordinates: This class was created to help convert the latitude and longitude information to a standard form. To convert coordinates from/to different forms of representation

```
public void ConvertCoordinatesToDouble(string latitude, string longitude)
```

By including the following statement the data members and member functions can be accessed by services:

GeneralSupportTools.Coordinates

Data Members:

I. Access modifier:	private, public
Type:	double
Name:	latitude, Latitude
Description:	stores the latitude of the location

II. Access Modifier:	private, public
Type:	double
Name:	longitude, longitude
Description:	stores the longitude of the location

III. **Access Modifier:** public
Type: string
Name: LatitudeAsString
Description: provides the latitude of a location as a string

IV. **Access Modifier:** public
Type: string
Name: LongitudeAsString
Description: provides the longitude of a location as a string

V. **Access Modifier:** public, private
Type: double
Name: CoordinatesCost, coordinatesCost
Description: To hold the cost of using Coordinates class.

Methods:

I. **Access modifier:** public
Name: ConvertCoordinatesToDouble
Description: Converts the latitude and longitude information in any known format to standard format
Parameters: **In:** string latitude, information of latitude as string
string longitude, information of longitude as string
Out:
Return Type: void, stores the changed coordinates in private members of the class

18. CustomDataTime: This class helps convert date and time to a standard format (YYYYMMDD and HH:MM), while reading wind and solar data. All source files provide date and time in different formats, this features was created so that the date and time when stored in tables are standardized. If the source date and time are a single string this converts it into a date string and time string and standardized. If they are different, only standardization is done.

```
public CustomDateTime(string dateTime)
```

By including the following statement the data members and member functions can be accessed by services:

GeneralSupportTools.CustomDataTime

Data Members:

I.	**Access modifier:**	private, public
	Type:	string
	Name:	date, Date
	Description:	holds the value of date
II.	**Access Modifier:**	private, public
	Type:	string
	Name:	time, Time
	Description:	holds the value of time
III.	**Access Modifier:**	public, private
	Type:	double
	Name:	DateTimeCost, dateTimeCost
	Description:	To hold the cost of using DateTime class.

Methods:

I.	**Access modifier:**	public
	Name:	DateTime
	Description:	constructor, takes a single value of date and time and break it into two standard parts of data and time.
	Parameters:	**In:** string dateTime, single value of date time **Out:** None
	Return Type:	None
II.	**Access modifier:**	public
	Name:	DateTime
	Description:	constructor, takes date and time and converts them to standard format
	Parameters:	**In:** string date, accepts the date in any format string time, accepts the time in any format **Out:** None
	Return Type:	None
III.	**Access modifier:**	private
	Name:	DateToStandard

Description: implicitly called by the constructor to convert date to standard format
Parameters: **In:** string date, holds the date
Out:
Return Type: string, returns the date in standard format

IV. **Access modifier:** private
Name: TimeToStandard
Description: implicitly called by the constructor to convert time to standard format
Parameters: **In:** string time, holds the time
Out:

Return Type: string, returns the time in standard format

19. ReportPreparation: This class is used create a file in specified format to in the Blob. There are different methods as the user is mailed information in different formats depending on whether solar or wind data was requested, whether uncertainty data was requested or not. The following code shows how and in what format the file is created for solar and wind for operation data reporting.

```
        public bool CreateReportInBlob(string destinationBlobFileURL, string
dataAsString, double powerData)
```

By including the following statement the data members and member functions can be accessed by services:

GeneralSupportTools.ReportPreparation

Data Members:

I. **Access modifier:** private, public
Type: double
Name: reportPreparationCost, ReportPreparationCost
Description: Initializes the cost for using this class.

II. **Access Modifier:** private
Type: static LocalResource
Name: localResource

Description: points to the location where the file will be created on the virtual
Machine.

III. **Access Modifier:** private
Type: BlobWrite
Name: writeToBlob
Description: creates an object to write the file to the blob.

<u>**Methods:**</u>

I. **Access modifier:** public
Name: ReportPreparation
Description: constructor, Initializes local storage to create file on virtual machine and BlobWrite.
Parameters: **In:** None
Out: None
Return Type: None

II. **Access modifier:** public
Name: CreateReportInBlob
Description: creates a report with given filename in the blob with given data. Used for creating reports for operation data, where just one power data (next days' forecasted power) is to be reported. Can be used for both solar and wind.
Parameters: **In:** string destinationBlobFileURL,
string dataAsString, different parameters related to turbine or panel, separate by comma.
double powerData, Operation Forecast data
Out: None
Return Type: bool, return true if file was successfully created.

III. **Access modifier:** public
Name: CreateReportInBlob
Description: creates a report with given filename in the blob with given data. Used in case of project prospecting without Uncertainity values.
Parameters: **In:** string destinationBlobFileURL,
string dataAsString, different parameters related to turbine or panel, separate by comma.
Double[] powerData, project prospecting data
Out: None
Return Type: bool, return true if file was successfully created.

IV. **Access modifier:** public
Name: CreateSolarReportInBlob

Description: creates a report with given filename in the blob with given data. Used in case of Project prospecting with uncertainty values requested by used for solar forecast.
Parameters: **In:** string destinationBlobFileURL,
string dataAsString, different parameters related to turbine or panel, separate by comma.
Double[] GHIData, double[] GHI_ET_Data
Double annualGHI_Data, double annualGHI_ET_Data
Out: None
Return Type: bool, return true if file was successfully created.

A.6. IO TOOLS

20. BasicOfficeOperations: This class helps performing some of the basic operations on MS Excel (2003), MS Excel (2011), text files and extraction of files from a zip file, untar a tar file, which are then exposed by IDRF as a service.

```
        public static MemoryStream ExtractFileFromStream(Stream zipFile, string
fileToExtract)

        public static string ExtractFileFromStream(Stream zipFile, string fileToExtract,
string nameOfDestinationDirectory)

        public string ExtractFileFromBlobToBlob(string zipFileNameInBlob, string
destinationBlobFileURL, bool isThisSiteSpecificExtraction)
        public static string UntarFile(string tarFilename)
```

By including the following statement the data members and member functions can be accessed by services:

IOTools.BasicOfficeOperations

Data Members:

I. **Access modifier:** private
Type: string
Name: destinationDirectory
Description: provides the name of the directory where the file has been downloaded from the blob to the local disk on the virtual machine.

II. **Access Modifier:** private
Type: string

Name: filename
Description: provides the name of the file which is to be converted to csv or from which the data has to be read and converted to standard form

III. **Access Modifier:** public, private
Type: double
Name: BasicOfficeOperationsCost, basicOfficeOperationsCost
Description: To hold the cost of using BasicOfficeOperations class.

<u>**Methods:**</u>

I. **Access modifier:** public
Name: BasicOfficeOperations
Description: constructor, initializes the name of file and the directory where the file was downloaded from the blob.
Parameters: **In:** string filename, holds the name of the file on which the operation is to be performed
Out: None
Return Type: None

II. **Access modifier:** public
Name: ExtractFileFromStream
Description: this is used for files that are big in size so the required from can be unzipped from a zip file and downloaded as a stream, which can then be uploaded to Blob using BlobWrite by calling service feature or service.
Parameters: **In:** Stream zipFile, a stream of zipFile download from Blob using BlobRead by calling service feature or service.
String fileToExtract, name of the file to extracted
Out: None

III. **Access modifier:** public
Name: static ExtractFileFromStream
Description: this is used for files that are big in size so the required from can be unzipped from a zip file and downloaded to a directory, which can then be uploaded to Blob using BlobWrite by calling service feature or service.
Parameters: **In:** Stream zipFile, a stream of zipFile download from Blob using BlobRead by calling service feature or service.
String fileToExtract, name of the file to extracted
String nameOfDestinationDirectory
Out: None

IV. **Access modifier:** public
Name: ExtractFileFromBlobToBlob

Description: this is used for files that are larger in size so the required from can be unzipped from a zip file and simultaneously uploaded to the blob.
Parameters: **In:** Stream zipFile, a stream of zipFile download from Blob using BlobRead by calling service.
String destinationBlobFilURL
Bool IsThisSiteSpecificExtraction, this is used in case of MISO file extraction as the zip archive created by NREL is in a different format and location in the archive. Not needed other than that.
Out: None

V. **Access modifier:** public
Name: CreateZipFile
Description: this creates a zip file on local storage, which needs to be uploaded to the Blob by calling service.
Parameters: **In:** string sourceDirectory, where all the files are
String zipFilename, desired filename
Out: None
Return Type: None

VI. **Access modifier:** public
Name: UntarFile
Description: Used for Historical data, this enables us to untar a file in .tar format. Only the file that is required (data from 00 hour) is extracted
Parameters: **In:** string tarfilename with directory
Out: None
Return Type: None

VII. **Access modifier:** public static
Name: NumberOfSheets
Description: counts the number of sheets present in a given excel(MS Office 2003) file. The number of sheets is used to create an array of type TurbineData in order to store the data from each of the individual sheets as an object.
Parameters: **In:** string filename, name of the file in which the number of sheets is to be determined.
Out:
Return Type: int, the number of sheets

VIII. **Access modifier:** public
Name: ConvertXLSFileToCSV

Description: converts an xls (MS Office 2003) file to csv file. Does not delete the original file, so must be manually removed from calling function. This function was defined to read the data from the website Illinoiswind.org.
Parameters: **In:** None
Out: None
Return Type: string, the directory + filename where the new file (.csv) had been generated.

21. BlobRead: This class downloads files from the Blob to local disk on the virtual machine, reads the data and deletes the files from the local disk. Other methods also allow file download to a Stream.

```
public string DownloadFileFromBlob(string blobFileURL)
```

By including the following statement the data members and member functions can be accessed by services:

IOTools.BlobRead

<u>**Data Members:**</u>

I. **Access modifier:** private, public
Type: bool
Name: fileDownloadedFromBlob, FileDownloadedFromBlob
Description: indicates if the file was successfully downloaded from the blob or not.

II. **Access Modifier:** private
Type: static string
Name: blobName (=”sitedata”)
Description: points to the name of the container where the files are stored

III. **Access Modifier:** private
Type: string
Name: directoryName
Description: points to the location where files are downloaded from the Blob.

IV. **Access Modifier:** private

Type: LocalResource
Name: localResource
Description: allows access the local storage on the virtual machine

V. **Access Modifier:** private
Type: static CloudStorageAccount
Name: storageAccount
Description: holds the details of the cloud storage account

VI. **Access Modifier:** private
Type: static CloudBlobContainer
Name: blobContainer
Description: holds the container where the files are stored in the blob

VII. **Access Modifier:** private
Type: static CloudBlobClient
Name: blobStorage
Description: allows access to the blob to download files from the blob.

VIII. **Access Modifier:** public, private
Type: double
Name: BrobReadCost, blobReadCost
Description: To hold the cost of using BlobRead class.

Methods:

I. **Access modifier:** static
Name: BlobRead
Description: initializes azure cloud storage details such as, username, password, container, etc.
Parameters: **In:** None
Out: None
Return Type: None

II. **Access modifier:** public
Name: BlobRead
Description: Initializes the localResource to create a local directory for the files from the blob to be downloaded to.
Parameters: **In:** None
Out: None
Return Type: None

III. **Access modifier:** private
Name: DownloadFileFromBlob

Description: downloads a file with given blobFileURL to the local directory on the virtual machine
Parameters: **In:** string blobFileURL
Out: None
Return Type: string, the name of file that was downloaded

IV. **Access modifier:** public
Name: ReadWindSpeedDataFromFile
Description: uses DownloadFileFromBlob to download the file and then extracts the wind speed data from the file using WindSpeedData
Parameters: **In:** string blobFileURL
Out: None
Return Type: WindSpeedData

V. **Access modifier:** public
Name: ReadTurbineDataFromFile
Description: uses DownloadFileFromBlob to download the file and then extracts the TurbinePowerData from the xls file.
Parameters: **In:** string blobFileURL
Out: None
Return Type: TurbinPowerData

22. **BlobWrite**: To store files in Azure Blob -

```
public bool WriteFileToBlob(string destinationBlobURL, Stream stream)
```

By including the following statement the data members and member functions can be accessed by services:

IOTools.BlobWrite

<u>**Data Members:**</u>

I. **Access modifier:** private, public
Type: bool
Name: fileStoredInBlob, FileStoredInBlob
Description: indicates if the file was successfully stored in the blob

II. **Access Modifier:** private
Type: static string
Name: blobName
Description: blobName (=”sitedata”)

III. **Access Modifier:** private
Type: static CloudStorageAccount
Name: storageAccount
Description: holds the details of the cloud storage account

IV. **Access Modifier:** private
Type: static CloudBlobContainer
Name: blobContainer
Description: holds the container where the files are stored in the blob

V. **Access Modifier:** private
Type: static CloudBlobClient
Name: blobStorage
Description: allows access to the blob to download files from the blob.

VI. **Access Modifier:** public, private
Type: double
Name: BlobWriteCost, blobWriteCost
Description: To hold the cost of using BlobWrite class.

<u>**Methods:**</u>

I. **Access modifier:** static
Name: BlobWrite
Description: initializes azure cloud storage details such as, username, password, container, etc.
Parameters: **In:** None
Out: None
Return Type: None

II. **Access modifier:** public
Name: BlobWrite
Description: default constructor.
Parameters: **In:** None
Out: None
Return Type: None

III. **Access modifier:** public
Name: WriteFileToBlob
Description: uploads a file to the blob from an available stream (HTTP, or FTP)
Parameters: **In:** string destinationBlobURL, indicates the destination name of the file in the blob.

Stream stream, has the stream of the file that is to be stored in the blob.
Out:
Return Type: bool, indicates if the file was successfully stored in BLOB.

IV. **Access modifier:** public
Name: AsyncUploadFileToBlob
Description: this was created to enable parallel upload of file streams being downloaded.
Parameters: **In:** Stream sourceStream, string destinationBlobURL
Out: None
Return Type: None

V. **Access modifier:** public
Name: MergeFileInBlob
Description: When a single file is download in parts as stream, this is used to merge the file back together in blob.
Parameters: **In:** Stream str1, str2, string destinationURL
Out: None
Return Type: None

VI. **Access modifier:** public
Name: MergeFileInBlob
Description: When a single file is download in parts to a local directory, this is used to merge the file back together in blob.
Parameters: **In:** Stream str1, str2, string destinationURL
Out: None
Return Type: None

VII. **Access modifier:** public
Name: DeleteFileFromBlob
Description: deletes a file from blob.
Parameters: **In:** string blobFileURL
Out: None
Return Type: None

VIII. **Access modifier:** public
Name: WriteFileToBlob
Description: uploads a file stored on the local storage of the virtual machine on
cloud
Parameters: **In:** string destinationBlobURL, indicates the destination name of the file in the blob.

String sourceFile, contains location and filename of the source file which is to be uploaded to the blob.
Out: None

Return Type: bool, indicates if the file was successfully stored in BLOB.

23. Degribber: (used for historical data download) After the degrib file and tar files have been downloaded to local storage, a name for the file to be decoded is created. The latitude and longitude for which the data is required is also supplied.

```
public string DegribFile(string tarFoder, string fileToDecode, string date, double
latitude, double longitude, string destinationDirectory)
```

By including the following statement the data members and member functions can be accessed by services:

IOTools.Degribber

This service was to decode Historical Solar data and uses .NET API Process to use Degrib.exe available from NDFD to decode the data.

Data Members:

I. **Access modifier:** private, public
Type: double
Name: degribberCost, DegribberCost
Description: initializes the cost of using this service features.

II. **Access Modifier:** private, public
Type: string
Name: degribDirectory, DegribDirectory
Description: indicates the path where Degrib.exe exists.

Methods:

I. **Access modifier:** public
Name: Degribber
Description: constructor, initializes the directory where Degrib.exe exists.
Parameters: **In:** string degribDirectory, initializes the name of the directory

	Out: None
Return Type:	None

II. **Access modifier:** public
Name: DegribFile
Description: decodes the fileToDecode using Degrib.exe in the given folder (tarFolder) to a specified directory. Only data for given date and latitude and longitude is decoded saving time and space. The decoded file is converted to .csv format. To execute any process or .exe file using C#, the following statement is used.

```
Process degribExecutor = new Process();
```

The location of the exe is specified as follow:

```
degribExecutor.StartInfo.FileName = degribDirectory + "degrib.exe";
```

The input arguments to the file must also be specified just like on a command prompt, as below:

```
string exeArguments = " -in "+ inputFile + " -P -pnt 39.74,-105.18 " + "-out " + outputFilename;
```

As soon as the process is started (degribExecutor.Start();) the command is executed and an output file is produced. As the input file is not needed anymore it is deleted and the output file is renamed to ".csv" as the default file is a ".prb" file which is not that useful for a user.

Parameters:	**In:** string tarFolder, String fileToDecode, String date, double latitude, double longitude, String destinationDirectory, **Out:** None
Return Type:	string, decoded filename

24. TableRead: This class reads a given entity from the MetaDataTable or DataTable. For wind, the entity can be of TurbineDataEntry, WindMetaDataEntry or FrequencySpectrumEntry. For solar, the entity can be of type SolarPanelData, SolarMetaDataEntry, MinutelDataEntry, HourlyDataEntry, DailyDataEntry, MonthlyDataEntry, AnnualDataEntry. An entity, source and context (details in Appendix B) have to be created in IOTools, before they can be added to the table.

```
public WindMetaDataEntry GetWindMetaDataEntry(double latitude, double longitude)
public SolarMetaDataEntry GetSolarMetaDataEntry(double latitude, double longitude)
```

By including the following statement the data members and member functions can be accessed by services:

IOTools.TableRead

Data Members:

I. Access modifier:	private
Type:	const string
Name:	metaDataTable (=”MetaDataTable”)
Description:	name of the meta data table
II. Access Modifier:	private
Type:	const string
Name:	dataTable (=”DataTable”)
Description:	name of the table where data is stored
III. Access Modifier:	private
Type:	static WindMetaDataSource
Name:	windMetaDataSource
Description:	object to create a source to add entries in the
MetaDataTable	
IV. Access Modifier:	private
Type:	static TurbineDataSource
Name:	turbineMetaSource
Description:	object to create a source to add turbine entries in
MetaDataTable	
V. Access Modifier:	private
Type:	static FrequencySpectrumSource
Name:	frequencySpectrumSource
Description:	object to create a source to add harmonics in
MetaDataTable	
VI. Access Modifier:	public, private

Type:	double
Name:	TableReadCost, tableReadCost
Description:	To hold the cost of using TableRead class.

Methods:

I.

Access modifier:	public
Name:	GetWindMetaDataEntry
Description:	retrieves the meta data for wind from the MetaDataTable
Parameters:	**In:** double latitude double longitude **Out:**
Return Type:	WindMetaDataEntry, to return a particular entry from the MetaDataTable

II.

Access modifier:	public
Name:	GetManufacturerTurbineData
Description:	
Parameters:	**In:** string manufacturerName, string turbineModelName, **Out:**
Return Type:	TurbineDataEntry

III.

Access modifier:	public
Name:	GetTurbinePowerData
Description:	
Parameters:	**In:** string manufacturerName, string turbineModelName, **Out:**
Return Type:	TurbineDataEntry

IV.

Access modifier:	public
Name:	ReadSolarMetaDataEntry
Description:	
Parameters:	**In:** double latitude, longitude **Out:**
Return Type:	SolarMetaDataEntry

V.

Access modifier:	public
Name:	ReadMinutelyData
Description:	
Parameters:	**In:** double latitude, longitude, String typeOfData, has to be SCI, GHI, GHI_ET or

GHI_Uncert

		Int year
		Out: out string[] date, startTime, endTime
		Double[] hourlyData
	Return Type:	void
VI.	**Access modifier:**	public
	Name:	ReadHourlySolarDataEntry
	Description:	
	Parameters:	**In:** double latitude, longitude,
		String typeOfData, has to be SCI, GHI, GHI_ET or
	GHI_Uncert	
		Int year
		Out: out string[] date, startTime, endTime
		Double[] hourlyData
	Return Type:	void
VII.	**Access modifier:**	public
	Name:	ReadDailySolarDataEntry
	Description:	
	Parameters:	**In:** double latitude, longitude,
		String typeOfData, has to be SCI, GHI, GHI_ET or
	GHI_Uncert	
		Int year
		Out: out string[] date, startTime, endTime
		Double[] hourlyData
	Return Type:	void
VIII.	**Access modifier:**	public
	Name:	ReadMonthlySolarEntry
	Description:	
	Parameters:	**In:** double latitude, longitude,
		String typeOfData, has to be SCI, GHI, GHI_ET or
	GHI_Uncert	
		Int year
		Out: double[] monthlyData
	Return Type:	void
IX.	**Access modifier:**	public
	Name:	GetMonthlySolarDataEntry
	Description:	
	Parameters:	**In:** double latitude, longitude,
		String typeOfData, has to be SCI, GHI, GHI_ET or
	GHI_Uncert	

Int year
Out: None
Return Type: MonthlySolarDataEntry

X. **Access modifier:** public
Name: ReadAnnualSolarDataEntry
Description:
Parameters: **In:** double latitude, longitude,
String typeOfData, has to be SCI, GHI, GHI_ET or GHI_Uncert
Int year
Out: out double annualData
Return Type: void

XI. **Access modifier:** public
Name: GetAnnualSolarDataEntry
Description:
Parameters: **In:** double latitude, longitude,
String typeOfData, has to be SCI, GHI, GHI_ET or GHI_Uncert
Int year
Out: None
Return Type: AnnualSolarDataEntry

XII. **Access modifier:** public
Name: SolarPanelMetaDataEntry
Description:
Parameters: **In:** String manufacturer name, ID
Out: None
Return Type: SolarPanelMetaDataEntry

XIII. **Access modifier:** public
Name: GetHarmonics
Description: retrieves harmonics data from the Dataable, given the number of harmonics that must be retrieved and the year and month pair.
Parameters: **In:** int harmonicsNumber, nth harmonic that must be retrieved
string yearMonth, year and month in YYYYMM
Out: None
Return Type: List<double>, returns the magnitude of the nth harmonic for the given month of a given year.

XIV. **Access modifier:** public

Name: GetValuesOfK
Description: retrives the value of K from the shape descriptor for the given
month of the year
Parameters: **In:** string yearMonth, month of the year in YYYYMM format
Out:
Return Type: List<int>, list of K

25. TableWrite: This class writes a given entity to the MetaDataTable or SolarDataTable and DataTable or SolarDataTable. For wind, the entity can be of TurbineDataEntry, WindMetaDataEntry or FrequencySpectrumEntry. For solar, the entity can be of type SolarPanelData, SolarMetaDataEntry, MinutelDataEntry, HourlyDataEntry, DailyDataEntry, MonthlyDataEntry, AnnualDataEntry. An entity, source and context have to be created in IOTools, before they can be added to the table.

```
public bool StoreWindDataAsHarmonicsInTable(double latitude, double longitude,
                                            string date, int height, double maxWindSpeed,
                                            string maxWindSpeedTime, double DC,
                                            double[] amplitude, double[] phase, int year)
```

By including the following statement the data members and member functions can be accessed by services:

IOTools.TableWrite

Data Members:

I. **Access modifier:** private
Type: const string
Name: metaDataTable (=”MetaDataTable” for wind and “SolarMetaDataTabl for solar”)
Description: name of the meta data table

II. **Access Modifier:** private
Type: const string

Name: dataTable (”DataTable” for wind and “SolarDataTable” for solar)
Description: name of the table where data is stored

III. **Access Modifier:** private
Type: static WindMetaDataSource
Name: windMetaDataSource
Description: object to create a source to add entries in the MetaDataTable

IV. **Access Modifier:** private
Type: static TurbineDataSource
Name: turbineMetaSource
Description: object to create a source to add turbine entries in MetaDataTable

V. **Access Modifier:** private
Type: static FrequencySpectrumSource
Name: frequencySpectrumSource
Description: object to create a source to add harmonics in MetaDataTable

VI. **Access Modifier:** public, private
Type: double
Name: TableWriteCost, talbeWriteCost
Description: To hold the cost of using TableWrite class.

Methods:

I. **Access modifier:** public
Name: TableWrite
Description: static constructor, to initialize the data sources
Parameters: **In:** None
Out: None
Return Type: None

II. **Access modifier:** public
Name: StoreManufacturerTurbineDataToTable
Description: turbine data in metadata table and power curve in data table
Parameters: **In:** TurbinePowerData oe, turbine data as an object
Out: None
Return Type: bool, true if entry successfully added

III. **Access modifier:** public

	Name:	StoreWindDataAsHarmonicsInTable
	Description:	stores wind data in data table as formshape
	Parameters:	**In:** double latitude, longitude,
		String date,
		Int height,
		Double maxWindSpeed
		String maxWindSpeedTime,
		Double DC,
		Double[] amplitude, phase
		Int year
		Out: None
	Return Type:	bool, true if entry successfully added
IV.	**Access modifier:**	public
	Name:	StoreWindMetaDataToTable
	Description:	stores wind meta data in table
	Parameters:	**In:** string metadataAsString
		Out: None
	Return Type:	bool, true if entry successfully added
V.	**Access modifier:**	public
	Name:	UpdateWindMetaDataEntry
	Description:	updates a wind entry in MetaDataTable to update data
	Parameters:	**In:** WindMetaDataEntry entry,
		string fieldToChange,
		string value
		Out: None
	Return Type:	bool, true if entry successfully added
VI.	**Access modifier:**	public
	Name:	GetWindMetaDataEntry
	Description:	Retrieves a wind metadata entry
	Parameters:	**In:** double latitude,
		double longitude,
		Out: None
	Return Type:	WindMetaDataEntry
VII.	**Access modifier:**	public
	Name:	StoreSolarMetaDataToTable
	Description:	Adds a entry to the solar metadata table
	Parameters:	**In:** string metadataAsString
		Out: None
	Return Type:	bool, true if entry successfully added

VIII. **Access modifier:** public
Name: UpdateSolarMetaDataEntry
Description: Updates an entry in the solar meta data table
Parameters: **In:** SolarMetaDataEntry entry, string fieldToChange, string value
Out: None
Return Type: bool, true if entry successfully added

IX. **Access modifier:** public
Name: StoreMinutelyDataInTable
Description: Stores minutely solar data in data table
Parameters: **In:** double latitude, double longitude,
string typeOfData, date, startTime, endTime,
double[] minutelyData
Out: None
Return Type: bool, true if entry successfully added

X. **Access modifier:** public
Name: StoreHourlySolarDataEntry
Description: Stores hourly solar data in data table
Parameters: **In:** double latitude, double longitude, string typeOfData, int year, string date, string startTime, string endTime,
double[] hourlyData
Out: None
Return Type: bool, true if entry successfully added

XI. **Access modifier:** public
Name: StoreDailySolarDataEntry
Description: Stores daily solar data in data table
Parameters: **In:** double latitude, double longitude,
string typeOfData, string date,
double[] dailyData
Out: None
Return Type: bool, true if entry successfully added

XII. **Access modifier:** public
Name: StoreMonthlySolarDataEntry
Description: Stores monthly solar data in data table
Parameters: **In:** double latitude, double longitude, int year,
string typeOfData,
double[] dailyData

Out: None
Return Type: bool, true if entry successfully added

XIII. Access modifier: public
Name: StoreAnnualSolarDataEntry
Description: Stores annual solar data in data table
Parameters: **In:** double latitude, double longitude, int year,
string typeOfData,
double annualData
Out: None
Return Type: bool, true if entry successfully added

XIV. Access modifier: public
Name: UpdateCostMetaDataEntry
Description: Updates cost of service features in metadata table
Parameters: **In:** string nameOfService, double newCost
Out: None
Return Type: bool, true if entry successfully added

XV. Access modifier: public
Name: StoreHarmonicsInTable
Description: stores a day's wind data as a single entity in the DataTable, uses FormShapeDescriptor
Parameters: **In:** string nameOfLocation, holds the name of location
string year,
double maxWindSpeed, maximum wind speed for a particular day
string date,
string maxWindSpeedTime, time at which the maximum wind speed for that day occurs
double DC,
double[] amplitude,
double[] phase
Out: None
Return Type: bool, return true or false depending on whether the entry was successfully added or not.

XVI. Access modifier: pubic
Name: UpdateSolarMetaDataEntry
Description: updates the metadata for wind
Parameters: **In:** WindMetaDataEntry entry, entry that is to be changed

string fieldToChange, name of the property whose value is to be changed
string value, new value of the field
Out: None

Return Type: bool, return true or false depending on whether the entry was successfully updated or not.

26. TurbinePowerData: The operation contract StoreTurbineDataInTables stores the software data from INL website into tables and is used by IDRF. This data is later used to forecast power for a particular wind turbine.

```
public bool StoreTurbineDataInTables(string sourceURL)
```

To obtain number of sheets with data in file:

```
public static int NumberOfSheetsInManufacturerSpreadsheet(string filename)
        {
            return BasicOfficeOperations.NumberOfSheets(filename);
        }
```

To process the data:

```
public TurbinePowerData ReadExcelSheetData(int sheetNumber)
```

By including the following statement the data members and member functions can be accessed by services:

IOTools.TurbinePowerData

This class stores the turbine data software available from the INL website (http://inl.gov/wind/software/). A single object of this class represents a single sheet of data in each of the .xls file. The static method of the class must be used to determine the number of sheets in each file and thus and array of this type must be create with size equal to the half the number of sheets (each file contains equal number of curves, which are not to be part of the data).

Data Members:

I. Access modifier: private
Type: string

Name: filename
Description: points to the directory and the name of the file on the virtual machine

II. **Access Modifier:** private, public
Type: string
Name: manufacturerName, ManufacturerName
Description: stores the name of the manufacturer

III. **Access Modifier:** private, public
Type: string
Name: turbineModelName, TurbineModelName
Description: stores the name of the turbine model

IV. **Access Modifier:** private, public
Type: double
Name: rating, Rating
Description: stores the rating of that particular model of turbine

V. **Access Modifier:** private, public
Type: double
Name: airDensity, AirDensity
Description: stores the air dentsity for that particular model of turbine

VI. **Access Modifier:** private, public
Type: double
Name: turbulence, Turbulence
Description: stores the turbulence at which the reading were made for that particular model of turbine

VII. **Access Modifier:** private, public
Type: double
Name: temperature, temperature
Description: stores the temperatue at which the reading were made for that particular model of turbine.

VIII. **Access Modifier:** private, public
Type: double
Name: elevation
Description: stores the elevation at which the reading were made for that particular model of turbine.

IX. **Access Modifier:** private, public

Type: double
Name: powerCurve, PowerCurve
Description: stores the power curve for a particular rating

X. **Access Modifier:** private, public
Type: double
Name: powerCurveAsString, PowerCurveAsString
Description: stores the power curve in string format for a particular rating.

XI. **Access Modifier:** public, private
Type: double
Name: TurbineDataCost, turbineDataCost
Description: To hold the cost of using TurbinePowerData class.

<u>**Methods:**</u>

I. **Access modifier:** public
Name: TurbinePowerData
Description: constructor, takes an object and copies the values
Parameters: **In:** TurbinePowerData td
Out: None
Return Type: None

II. **Access modifier:** public
Name: TurbinePowerData
Description: initializes the name of the file from which data is to be read.
Parameters: **In:** string filename
Out:
Return Type: None

III. **Access modifier:** public
Name: NumberOfSheetsInManufacturerSpreadsheet
Description: return the number of sheets there are in the spreadsheet
Parameters: **In:** string filename
Out: None
Return Type: None

IV. **Access modifier:** public
Name: ReadExcelSheetData
Description: reads manufacturer information and the table from XLS file
Parameters: **In:** int sheetNumber, for which the data is to be read
Out: None

Return Type: TurbinePowerData, object representing on sheet in the xls file

V. **Access modifier:** private
Name: ReadManufacturerInformation
Description: reads manufacturer information from a single sheet
Parameters: **In:** int sheetNumer,
out bool isSpeedInMeterPerSecond,
out int rowStartIndex,
out in colStartIndex,
out int colEndIndex
Out: None
Return Type: TurbinePower

VI. **Access modifier:** private
Name: ReadTableFromXLSFile
Description: Reads the power curve table in a single sheet
Parameters: **In:** int sheetNumer,
out bool isSpeedInMeterPerSecond,
out int rowStartIndex,
out in colStartIndex,
out int colEndIndex
Out: None
Return Type: void

VII. **Access modifier:** private
Name: ConvertSpeedToMPS
Description: Converts speed from MPH to MPS and calculated corresponding power at this speed(MPS) in KW to store in table. Speeds should always be increments of 1 after the cut-in speed.
Parameters: **In:** None
Out: None
Return Type: None

VIII. **Access modifier:** public
Name: InterpolatePower
Description: interpolates power between two given power and speeds
Parameters: **In:** double speed
Out: None
Return Type: double, projected power at the given speed

27. RawWindSpeedData: The operation contract StoreDataInTable in IIDRF uses this class to read data from files and then uses TableWrite to store this data in tables. This class helps a user read all the wind speed data available in one csv file and store it in different class variables. This works with data from all the web sources.

By including the following statement the data members and member functions can be accessed by services:

IOTools.RawWindSpeedData

Data Members:

I.	**Access modifier:**	private
	Type:	string
	Name:	filename
	Description:	points to the directory and name of the file on the virtual machine
II.	**Access Modifier:**	private, public
	Type:	List<string>
	Name:	date, Date
	Description:	stores the date for which data is being read
III.	**Access Modifier:**	private, public
	Type:	List<string>
	Name:	time, Time
	Description:	stores the time for which wind speed data is being read
IV.	**Access modifier:**	private, public
	Type:	List<double>
	Name:	maxWindSpeed
	Description:	stores the maximum wind speed for a each day of the year
V.	**Access Modifier:**	private, public
	Type:	List<string>
	Name:	maxWindSpeedTime
	Description:	stores the time at which maximum wind speed occurs for a day for each day of year

VI. **Access Modifier:** private, public
Type: List<double>
Name: windSpeeds
Description: stores the values of wind speeds in the whole file (reads speed at defaut height of 100 m).

VII. **Access Modifier:** private, public
Type: List<double>
Name: normalizedWindSpeeds
Description: stored the value of wind speeds normalized by the daily maximum value

VIII. **Access Modifier:** public, private
Type: double
Name: WindDataExtractionCost, windDataExtractionCost
Description: To hold the cost of using WindSpeedData class.

<u>**Methods:**</u>

I. **Access modifier:** public
Name: WindSpeedData
Description: constructor, initializes all the class data members
Parameters: **In:** string filename
Out: None
Return Type: None

II. **Access modifier:** public
Name: GetWindSpeeds
Description: gets wind speeds for a whole year. Maximum wind speed for a day is also calculated and value stored in class variable.
Parameters: **In:** int height, height for which the wind speeds are to be read
Out: None
Return Type: void

III. **Access modifier:** private
Name: WindSpeedAdjustmentByHeight
Description: used to convert wind speeds at a given height to a desired height. Uses the following expression:
$S' = S * (\text{new height} / \text{original height})^{1/7}$
Parameters: **In:** List<double> speedAtGivenHeight, int newHeight
Out: None
Return Type: List<double>, new wind speeds at given height.

28. Untar: The operation contract StoreHistoricalData uses this static method in BasicOfficeOperations to untar a file with historical data. Only the file that has data needed by the user (files with names ending in 00) are untarred.

By including the following statement the data members and member functions can be accessed by services:

IOTools.BasicOfficeOpeartion.Untar

A.7. SOLAR DATA ANALYSIS TOOLS

29. MinutelyData: Stores minutely values of GHI, GHI ET or uncertainty in tables for the entire day. Each minute's data is separated by comma. The beginning and end time of day for which data is collected is also stored, which helps to later recalculate how many number of minutes of data exists. This class is used to store and retrieve minutely SCI data to and from SolarDataTable.

By including the following statement the data members and member functions can be accessed by services:

SolarDataAnalysisTools.MinutelyData

<u>**Data Members:**</u>

I.	**Access modifier:**	private, public
	Type:	double[]
	Name:	minutelySCI, MinutelySCI
	Description:	Holds the values of minutely SCI
II.	**Access Modifier:**	private, public
	Type:	double
	Name:	minutelySCICost, MinutelySCICost
	Description:	Holds the value of using this service feature
III.	**Access Modifier:**	private
	Type:	TableRead

Name:	readFromTable
Description:	TableRead Object to read data from table

IV.	**Access Modifier:**	private
	Type:	TableWrite
	Name:	writeToTable
	Description:	TableWrite object to wrte minutely data to table

Methods:

I.	**Access modifier:**	public
	Name:	StoreMinutelySCIInTable
	Description:	Stores the value of minutely SCI in data table
	Parameters:	**In:** double latitude, longitude, List<string> date, time List<int> numberOfMinutesOfDataInAHour, numberOfHoursOfDataInADay List<double>minutelyPSPData, minutelyGHIData
		Out: None
	Return Type:	double,

II.	**Access modifier:**	private
	Name:	ReadMinutelySCIDataFromTable
	Description:	Reads the value of minutely SCI from data table
	Parameters:	**In:** d
		Out: None
	Return Type:	double, return projected power

A.8. STATISTICAL TOOLS

30. CombiningSigmas: This class combines sigma values of uncertainty to calculate the overall sigma:

```
public double CombineSigma(double[] sigmas, int n)
```

By including the following statement the data members and member functions can be accessed by services:

StatisticalTools. CombiningSigmas

Data Members:

I. Access modifier:	private, public
Type:	double
Name:	sigma, Sigma
Description:	stores the results of overall sigma

II. Access Modifier:	public, private
Type:	double
Name:	CombiningSigmaCost, CombiningSigmaCost
Description:	To hold the cost of using CombiningSigma class.

Methods:

I. Access modifier:	public
Name:	CombiningSigmas
Description:	Default contructor to initialize the value of sigma to zero
Parameters:	**In:** None
	Out: None
Return Type:	None

II. Access modifier:	public
Name:	CombineSigma
Description:	Calculates the overall value of sigma. Also stores the results in private member of the class "sigma".
Parameters:	**In:** double[] sigmas,
	Int n, number of values.
	Out: None
Return Type:	double

31. CreateDistribution: This class returns the distribution of a given se of data, returning the unique occurrences of a data and the number of times it occurs.

```
public void GetDistributionForData()
```

By including the following statement the data members and member functions can be accessed by services:

StatisticalTools.CreateDistribution

Data Members:

I.	**Access modifier:**	private
	Type:	int[]
	Name:	values
	Description:	holds the input set of data
II.	**Access Modifier:**	private
	Type:	int
	Name:	length
	Description:	total number of values in the set
III.	**Access Modifier:**	private, public
	Type:	int[]
	Name:	k, K
	Description:	holds the unique data set
IV.	**Access Modifier:**	private, public
	Type:	int[]
	Name:	frequency, Frequency
	Description:	holds the number of times each of the unique data occurs
V.	**Access Modifier:**	public, private
	Type:	double
	Name:	DistributionCalculationCost, distributionCalculationCost
	Description:	To hold the cost of using CreateDistribution class.

Methods:

I.	**Access modifier:**	public
	Name:	CreateDistribution
	Description:	constructor to initialize the input data and it's length
	Parameters:	**In:** int[] values, the input data set
		Out:
	Return Type:	None

II. **Access modifier:** public

Name: GetDistributionForData

Description: computes how many times a single value of data occurs in the whole set of data and stores the results in private variable of the class.

Parameters: **In:** None
Out: None

Return Type: void

32. ExceedanceProbability: This class calculates the exceedance probability for every month in a year and the yearly exceedance probability, using the previous year data.

```
public void ComputeExceedanceProbability(
                                double[] monthlyAveragePower, double[] monthlySigma,
                                double annualPower, double annualSigma)
```

By including the following statement the data members and member functions can be accessed by services:

StatisticalTools.ExceedanceProbability

Data Members:

I.	**Access modifier:**	private
	Type:	double[]
	Name:	averages
	Description:	holds the average powers for three years of data
II.	**Access Modifier:**	private
	Type:	double[]
	Name:	sigmas
	Description:	holds the standard deviations for three years of data
III.	**Access Modifier:**	private, public
	Type:	double[]
	Name:	m_p_95, M_P_95
	Description:	holds the value of P95 for each month's power for whole year
IV.	**Access Modifier:**	private, public
	Type:	double[]
	Name:	m_p_90, M_P_90
	Description:	holds the value of P90 for each month's power for whole year
V.	**Access Modifier:**	private, public
	Type:	double[]
	Name:	m_p_75, M_P_75

Description: holds the value of P75 for each month's power for whole year

VI. **Access Modifier:** private, public
Type: double[]
Name: m_p_50, M_P_50
Description: holds the value of P50 for each month's power for whole year

VII. **Access Modifier:** private, public
Type: double
Name: y_p_95, Y_P_95
Description: holds the value of P95 for yearly power production

VIII. **Access Modifier:** private, public
Type: double
Name: y_p_90, Y_P_90
Description: holds the value of P90 for yearly power production

IX. **Access Modifier:** private, public
Type: double
Name: y_p_75, Y_P_75
Description: holds the value of P75 for yearly power production

X. **Access Modifier:** private
Type: Averages
Name: averageObj
Description: uses the service feature Average to calculate the average of given list

XI. **Access Modifier:** private
Type: StandardDevaition
Name: sigmaObj
Description: uses the services features StandardDeviation to calculate the sigmas of given list with a given mean.

XII. **Access Modifier:** private, public
Type: double
Name: y_p_50, Y_P_50
Description: holds the value of P50 for yearly power production

XIII. **Access Modifier:** public, private
Type: double

Name: ExceedanceProbabilityCost, exceedanceProbabilityCost
Description: To hold the cost of using ExceedanceProbability class.

<u>**Methods:**</u>

I. **Access modifier:** public
Name: **ExceedanceProbablity**
Description: constructor to initialize the arrays and objects that will be used
Parameters: **In:** None
Out: None
Return Type: None

II. **Access modifier:** public
Name: ComputeExceedanceProbability
Description: computes the exceedance probability data for yearly power and assigns values to the class's public object in order for the caller to get values.
Parameters: **In:** double[] monthlyPower, powers for all the given years as a list
Double[] monthlySigma,
Double annualPower
Double annualSigma
Out: None
Return Type: void

III. **Access modifier:** public
Name: ComputeMonthlyExceedanceProbability
Description: computes the exceedance probability data for monthly power and assigns values to the class's public object in order for the caller to get values.
Parameters: **In:** List<double> monthlyPower, powers for all the given years as a list
Out: None
Return Type: void

IV. **Access modifier:** private
Name: QNorm
Description: calculates the inverse CDF, given the probability, mean and sigma
Parameters: **In: double p**, values of probability
Double mu, mean
Double sigma, standard devaition
Bool lower_tail, is the probability for lower tail?
bool log p, is the probability value in log?

	Out: None
Return Type:	double

33. MAPE: This class helps calculate the mean average percentage error that helps in determining how many harmonics must be stored in the table so that the original data can be retrieved with minimal error or error under 2%.

```
public double CalculateMAPE(double[] originalValue, double[] reconstructedValue)
```

By including the following statement the data members and member functions can be accessed by services:

StatisticalTools.MAPE

Data Members:

I. Access modifier:	private
Type:	double
Name:	error
Description:	holds the value of error calculated for the given data
II. Access Modifier:	public, private
Type:	double
Name:	MAPECost, mapeCost
Description:	To hold the cost of using MAPE class.

Methods:

I. Access modifier:	public
Name:	MAPE
Description:	initializes the value of error to zero
Parameters:	**In:** None
	Out: None
Return Type:	None
II. Access modifier:	public
Name:	CalculateMAPE
Description:	calculates the MAPE, given the original data and the
recovered data	
Parameters:	**In:** double[] originalValue,
	double[] reconstructedValue

Out:

Return Type: double, the error calculated due to the difference beteween the original and reconstructed values

A.9. TRANSFER TOOLS

34. FTPDataDownload: The declaration below shows how by using .NET functionality FtpWebRequest we can download a file from any FTP server to a stream (or file on local disk) and then return the stream for a calling function to store it in the blob.

```
public Stream DownloadFileToStream()
```

By including the following statement the data members and member functions can be accessed by services:

TransferTools.FTPDataDownload

<u>**Data Members:**</u>

I. Access modifier: private
Type: string
Name: ftpURL
Description: holds the ftpURL from where the file is to be downloaded

II. Access Modifier: private
Type: string
Name: destinationDirectory
Description: points to the location where the file will be downloaded on the virtual machine

III. Access Modifier: private
Type: FtpWebRequest
Name: ftpRequest
Description: creates an FTP request to the given server name

IV. Access Modifier: private
Type: FtpWebResponse

Name: ftpResponse
Description: holds the response to the FTP request for a given server name

V. **Access Modifier:** public, private
Type: double
Name: FTPDownloadCost, ftpDownloadCost
Description: To hold the cost of using FTPDownload class.

<u>**Methods:**</u>

I. **Access modifier:** public
Name: FTPDataDownload
Description: constructor, initializes the ftpURL name to the name provided
Parameters: **In:** string URL, provides the name of the resource on a ftp server
Out: None
Return Type: None

II. **Access modifier:** public
Name: FTPDataDownload
Description: constructor, initializes the name of the resource on a FTP server and the name of the destination directory on the virtual machine.
Parameters: **In:** string URL, initializes the name of resource on the ftp server
string destinationDirectory, directory where the file will be downloaded
Out: None
Return Type: None

III. **Access modifier:** public
Name: DownloadFileToStream
Description: sends an FTPRequest at the given resource and receives a Stream as a result. The calling service uploads the stream to be stored in the blobby using BlobWrite.
Parameters: **In:** None
Out: None
Return Type: Stream, stream of the resource file asked for

35. HTTPDataDownload: The following declaration downloads data for 3 years in parallel and stores the files in the Blob. A new http url for source is created every time using metadata (site id) and a destination URL (Blob url) is created and store in the metadata for other services to retrieve

```
private Stream DownloadFile()
```

By including the following statement the data members and member functions can be accessed by services:

TransferTools.HTTPDataDownload

<u>**Data Members:**</u>

I. **Access modifier:** private
Type: string
Name: httpURL
Description: holds the name of the resource to be downloaded over the internet.

II. **Access Modifier:** private
Type: HttpWebRequest
Name: httpRequest
Description: creates an http request to download the given resource

III. **Access Modifier:** private
Type: BlobWrite
Name: writeToBlob
Description: writes the downloaded stream to Blob as a file.

IV. **Access Modifier:** private
Type: HttpWebResponse
Name: httpResponse
Description: creates an http response to receive the response to http request

V. **Access Modifier:** public, private
Type: double
Name: HTTPDownloadCost, httpDownloadCost
Description: To hold the cost of using HTTPDownload class.

Methods:

I. **Access modifier:** public
Name: HTTPDataDownload
Description: initializes the name of the resource to be downloaded using HTTP
Parameters: **In:** string URL
Out: None
Return Type: None

II. **Access modifier:** public
Name: DownloadData
Description: Download the given resource via HTTP to a stream that will be stored in the Blob. Initialization of URL is done while instantiating object.
Parameters: **In:** None
Out: None
Return Type: Stream, returns the requested HTTP resource as a stream

III. **Access modifier:** public
Name: CloseHttpStream
Description: closes the stream after the data has been stored in the blob.
Parameters: **In:** None
Out: None
Return Type: void

IV. **Access modifier:** public
Name: CheckIfFileExists
Description: checks if a file exists at the given http url
Parameters: **In:** string URL
Out: None
Return Type: bool

V. **Access modifier:** private
Name: SetUnsafeHeaderParsing
Description: this is used by DownloadData before downloading solar data.
Parameters: **In:** None
Out: None
Return Type: None

VI. **Access modifier:** public
Name: LinkToDataFiles
Description: reads the name of files on a given html page (for Idaho).
Parameters: **In:** string URL

		Out: None
	Return Type:	string[], names of the files on the html page

VII.	**Access modifier:**	public
	Name:	FindHTTPLinksInHTML
	Description:	reads the name of files on a given html page (for Idaho).
	Parameters:	**In:** string URL
		Out: None
	Return Type:	string[], names of the files on the html page

VIII.	**Access modifier:**	private
	Name:	HTTPPageAsString
	Description:	reads the name of files on a given html page (for Idaho).
	Parameters:	**In:** string URL
		Out: None
	Return Type:	string[], names of the files on the html page

IX.	**Access modifier:**	public
	Name:	FindTextInHTML
	Description:	reads the name of files on a given html page (for Idaho).
	Parameters:	**In:** string URL, int year, string textToFind
		Out: None
	Return Type:	string[], names of the files on the html page

36. XMLDataDownload: The operation contract ForecastPower uses this service feature for operational forecast. The code below shows how this feature is used by other services that extract the XML data and present it in user readable form. This returns the XMLDocument used above to extract daily wind or cloud data and uses the .NET object WebRequest to download data as XML.

```
public XmlDocument DownloadXMLDocument()
```

By including the following statement the data members and member functions can be accessed by services:

TransferTools.XMLDataDownload

Data Members:

I. **Access modifier:** private, public
Type: string
Name: httpURL, HttpURL
Description: holds the name of the URL from where the XML will be downloaded.

II. **Access Modifier:** private
Type: HttpWebRequest
Name: httpRequest
Description: creates an http request to download the given resource.

III. **Access Modifier:** private
Type: HttpWebResponse
Name: httpResponse
Description: creates an http response to receive the response to http request.

IV. **Access Modifier:** private
Type: Stream
Name: responseStream
Description: holds the response (XML) as stream

V. **Access Modifier:** public, private
Type: double
Name: XMLDownloadCost, xmlDownloadCost
Description: To hold the cost of using XMLDownload class.

Methods:

I. **Access modifier:** public
Name: XMLDataDownload
Description: initializes the name of the resource to be downloaded as XML using HTTP. As the URL where the daily data exists, depends on the latitude and longitude values, it creates an HTTP URL specific for that location as follows:

"http://graphical.weather.gov/xml/SOAP_server/ndfdXMLclient.php?whichClient=NDFDgen&lat=" + latitude + "&lon=" + longitude + "&listLatLon=&lat1=&lon1=&lat2=&lon2=&resolutionSub=&listLat1=&listLon1=&listLat2=&listLon2=&resolutionList=&endPoint1Lat=&endPoint1Lon=&endPoint2Lat=&endPoint2Lon=&listEndPoint1Lat=&listEndPoint1Lon=&listEndPoint2Lat=&listEndPoint2Lon=&zipCodeList=&listZipCodeList=&centerPointLat=&centerPointLon=&distanceLat=&distanceLon=&resolutionSquare=&listCenterPointLat=&listCenterPointLon=&listDistanceLat=&listDistanceLon=&listResolutionSquare=&citiesLevel=&listCitiesLevel=&sector=&gmlListLatLon=&featureType=&requestedTime=&startTime=&endTime=&compType=&propertyName=&product=time-

```
series&begin=" + startDate + "T00%3A00%3A00&end=" + endDate +
"T00%3A00%3A00&Unit=e&wspd=wspd&wdir=wdir&sky=sky&Submit=Submit";
```

The start date and end date are always the date tonight midnight and tomorrow midnight respectively. This is because the daily forecast will be for tomorrow, as shown below:

```
    CustomDateTime date = new CustomDateTime(DateTime.Today.ToString());
    //THE FORMAT CHANGE BELOW IS BECAUSE THE HTTP URL ABOVE
EXPECTS IT IN THIS FORMAT
    date.ConvertToSpecifiedFormat("YYYY-MM-DD");
    string startDate = date.NextDate();
    string endDate = CustomDateTime.NextDate(startDate);
```

Parameters:	**In:** double latitude, double longitude
	Out: None
Return Type:	None

II. Access modifier:	public
Name:	DownloadXMLDocument
Description:	Downloads the XML document.
Parameters:	**In:** None
	Out: None
Return Type:	XmlDocument, this contains the cloud cover and wind speed data for tomorrow.

A.10. Development Tools Used

MICROSOFT:

a. Visual Studio 2010 Professional
b. Microsoft .NET framework version 4.0.30319 RTMRel
c. Microsoft Visual C# 2010
d. Microsoft Azure Tools & SDK VS 2010 (Nov. 2011)
e. Microsoft.WindowsAzure.ServiceRuntime
f. Microsoft.WindowsAzure.StorageClient
g. Microsoft Visio 2011 – to prepare flowchart, workflows etc.
h. Microsoft Word 2011 – for documentation

THIRD PARTY SOFTWARES:

a. Azure Storage Explorer version 4.0.0.10 (05.07.2011) by Neudisic
b. Cerabrata Storage Studio (trial version) – to backup cloud data and from cloud
c. Gmail – to mail results to the user.
d. Ionic.zip.dll (v2.0.50727) – To manipulate zip files.
e. ExcelLibrary.dll (v2.0.50727) – to read data from Excel files.
f. Tar.cs (version: 1.4.0.2) – to manipulate tar files.

APPENDIX B – Namespaces Used for Solar & Wind

B.1. List of Namespaces, Classes and Methods used for Wind Project Prospecting

- **EDCF**: Below are the classes created in order to forecast project prospecting:
  - TransferTools.HTTPDataDownload or TransferTools.FTPDataDownload
    - DownloadData or DownloadFileToStream
  - IOTools.BlobWrite
    - WriteFileToBlob
  - IOTools.TableRead
    - GetWindMetaDataEntry
    - GetManufacturerTurbineData
    - GetTurbinePowerData
  - IOTools.TableWrite
    - UpdateWindMetaDataEntry

- **IDRF**: IDRF uses the following classes for project prospecting:
  - GeneralSupportTools.BasicOfficeOperation
    - ExtractFileFromStream
    - ExtractFileFromBlobToBlob
    - NumberOfSheets
    - ConvertXLSFileToCSV
  - IOTools.BlobRead
    - ReadWindSpeedDataFromFile
    - ReadTurbineDataFromFile
  - IOTools.BlobWrite
    - WriteFileToBlob
    - MergeFileInBlob

- IOTools.TableRead
    - GetHarmonics
- IOTools.TableWrite
    - StoreManufacturerTurbineDataToTable
    - StoreWindDataAsHarmonicsInTable
    - StoreWindMetaDataToTable
- IOTools.RawWindSpeedData
    - GetWindSpeeds
- FoundationTools.Harmonics
    - calculateHarmonics
- StatisticalTools.MAPE
    - CalculateMAPE
- FoundationTools.IDFT
    - IDFTFromHarmonics
- FoundationTools.NumberOfHarmonicsNeeded
    - HarmonicsNeeded

➢ **FGF**: FGF uses the following classes for project prospecting:

- IOTools.TableRead
    - GetManufacturerTurbineData
    - GetTurbinePowerData
    - GetHarmonics
- IOTools.BlobRead
    - ReadWindSpeedDataFromFile
    - ReadTurbineDataFromFile
- IOTools.RawWindSpeedData
    - GetWindSpeeds
- ForecastTools.PowerForecast

    - ForecastPower
    - ProjectedPower
    - InterpolatePower

- StatisticalTools.Average
    - CalculateAverage

- StatisticalTools.ExceedanceProbability
    - ComputeExceedanceProbability
    - ComputeMonthlyExceedanceProbability

- StatisticalTools.Sigma
    - CalculateSigma

- GeneralSupportTools.ReportPreparation
    - CreateReportInBlob
- CommunicationTools.EmailServices
    - MailFileToUser

B.2. List of Namespaces, Classes and Methods used for Wind Operational Forecasting

➢ **EDCF**

- TransferTools.XMLDataDownload
    - DownloadXMLDocument

➢ **IDRF**

- ForecastTools.ReadForecastDataFromXML
    - GetForecastData

➢ **FGF**

- IOTools.BlobWrite
    - WriteFileToBlob

- GeneralSupportTools.ReportPreparation

  - CreateReportInBlob

- CommunicationTools.EmailService
  - MailFileToUser

B.3. List of Namespaces, Classes and Methods used for Solar Project Prospecting

➢ **EDCF**: EDCF uses the following service features for Project prospecting:

- TransferTools.HTTPDataDownload
  - DownloadData
- IOTools.BlobWrite
  - WriteStreamToBlob
- IOTools.UpdateMetaData
  - UpdateSolarMetaDataEntry

➢ **IDRF**: For Project prospecting following service features are used:

- TranferTools.BlobWrite
  - WriteFileToBlob
- IOTools.RawSolarData
  - GetSolarData
- SolarDataAnalysis.Minutelydata
  - StoreMinutelySCIInTable
- FoundationTools.MinutelyDataToHourlyData
  - StoreHourlyDataInTable
  - ReadHourlyDataFromTable
- FoundationTools.HourlyDataToDailyData
  - StoreDailyDataInTable

    - ReadDailyDataFromTable

- FoundationTools.DailyDataToMonthlyData
    - StoreMonthlyDataInTable
    - ReadMonthlyDataFromTable
- FoundationTools.MonthlyDataToAnnualData
    - StoreAnnualDataInTable
    - ReadAnnualDataFromTable

➢ **FGF**: Given below are service features used by FGF for project prospecting

- IOTools.TableRead
    - ReadMinutelySCIDataFromTable
- FoundationTools.DailyDataToMonthlyData
    - ReadMonthlyDataFromTable
- FoundationTools.MonthlyDataToAnnualData
    - ReadAnnualDataFromTable
- StatisticalTools.Average
    - CalculateAverage
- StatisticalTools.Sigma
    - CalculateSigma
- StatisticalTools.ExceedanceProbability
    - ComputeExceedanceProbability
    - ComputeMonthlyExceedanceProbability
- GeneralSupportTools.ReportPreparation
    - CreateSolarReportInBlob
- CommunicationTools.EmailServices
    - MailFileToUser

B.4. List of Namespaces, Classes and Methods used for Solar Operational Forecasting

- **EDCF**
  - TransferTools.XMLDataDownload
    - DownloadXMLDocument
- **IDRF**
  - ForecastTools.ReadForecastDataFromXML
    - GetForecastData
- **FGF**: Given below are the service features used by FGF for project prospecting
  - GeneralSupportTools.CustomDateTime
    - ConvertToSpecifiedFormat
    - NextDate
  - ForecastTools.ForecastGHIET
    - ComputeGHIET
  - ForecastTools.ComputeKD
    - ComputeKD
  - GeneralSupportTools.ReportPreparation
    - CreateSolarReportInBlob
  - CommunicationTools.EmailService
    - MailFileToUser

APPENDIX C – Implementation Details

C.1. Data Organization in Blob and Tables

The data organized for any forecast is divided in tables and blobs. Fig. C.1 shows the names of the tables for metadata and data.

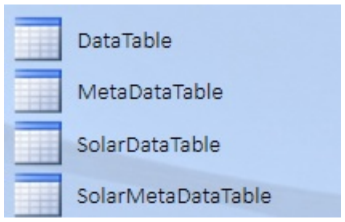


Figure C.1. Names of metadata and data tables for wind and solar

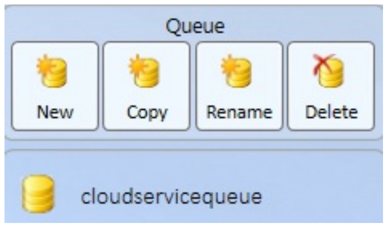
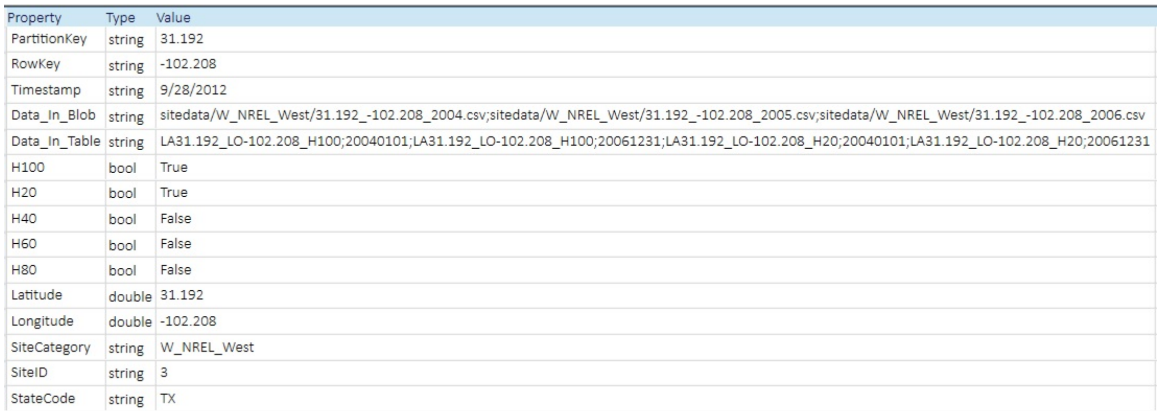


Figure C.2. An Azure data storage queue

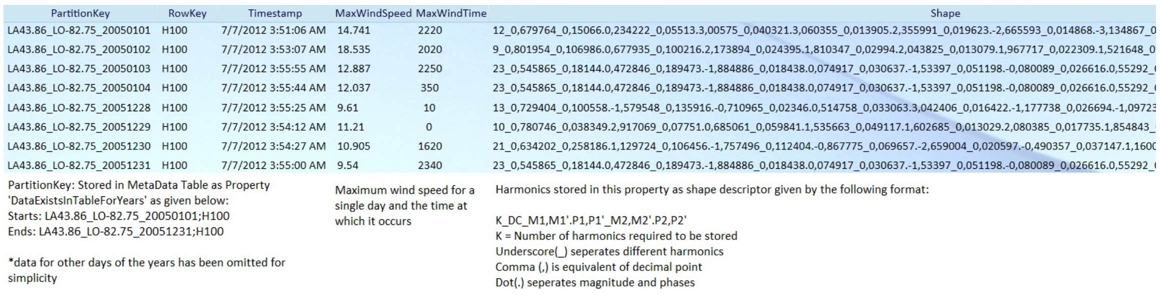
C.1.1. Wind Data Organization

Fig. C.3 shows a metadata entry for a wind resource with location coordinates (31.192, -102.208). 'Data_In_Table' and 'Data_In_Blob' contain link to the location of data in tables and blobs as shown in Fig C.4 and C.5 respectively.



Property	Type	Value
PartitionKey	string	31.192
RowKey	string	-102.208
Timestamp	string	9/28/2012
Data_In_Blob	string	sitedata/W_NREL_West/31.192_-102.208_2004.csv;sitedata/W_NREL_West/31.192_-102.208_2005.csv;sitedata/W_NREL_West/31.192_-102.208_2006.csv
Data_In_Table	string	LA31.192_LO-102.208_H100;20040101;LA31.192_LO-102.208_H100;20061231;LA31.192_LO-102.208_H20;20040101;LA31.192_LO-102.208_H20;20061231
H100	bool	True
H20	bool	True
H40	bool	False
H60	bool	False
H80	bool	False
Latitude	double	31.192
Longitude	double	-102.208
SiteCategory	string	W_NREL_West
SiteID	string	3
StateCode	string	TX

Figure C.3. Metadata entry for wind resource in *MetaDataTable*



PartitionKey	RowKey	Timestamp	MaxWindSpeed	MaxWindTime	Shape
LA43.86_LO-82.75_20050101	H100	7/7/2012 3:51:06 AM	14.741	2220	12_0,679764_0,15066.0,234222_0,05513.3,00575_0,040321.3,060355_0,013905.2,355991_0,019623.-2,665593_0,014868.-3,134867_0
LA43.86_LO-82.75_20050102	H100	7/7/2012 3:53:07 AM	18.535	2020	9_0,801954_0,106986.0,677935_0,100216.2,173894_0,024395.1,810347_0,02994.2,043825_0,013079.1,967717_0,022309.1,521648_0
LA43.86_LO-82.75_20050103	H100	7/7/2012 3:55:55 AM	12.887	2250	23_0,545865_0,18144.0,472846_0,189473.-1,884886_0,018438.0,074917_0,030637.-1,53397_0,051198.-0,080089_0,026616.0,55292_
LA43.86_LO-82.75_20050104	H100	7/7/2012 3:55:44 AM	12.037	350	23_0,545865_0,18144.0,472846_0,189473.-1,884886_0,018438.0,074917_0,030637.-1,53397_0,051198.-0,080089_0,026616.0,55292_
LA43.86_LO-82.75_20051228	H100	7/7/2012 3:55:25 AM	9.61	10	13_0,729404_0,100558.-1,579548_0,135916.-0,710965_0,02346.0,514758_0,033063.3,042406_0,016422.-1,177738_0,026694.-1,0972
LA43.86_LO-82.75_20051229	H100	7/7/2012 3:54:12 AM	11.21	0	10_0,780746_0,038349.2,917069_0,07751.0,685061_0,059841.1,535663_0,049117.1,602685_0,013029.2,080385_0,017735.1,854843_
LA43.86_LO-82.75_20051230	H100	7/7/2012 3:54:27 AM	10.905	1620	21_0,634202_0,258186.1,129724_0,106456.-1,757496_0,112404.-0,867775_0,069657.-2,659004_0,020597.-0,490357_0,037147.1,160
LA43.86_LO-82.75_20051231	H100	7/7/2012 3:55:00 AM	9.54	2340	23_0,545865_0,18144.0,472846_0,189473.-1,884886_0,018438.0,074917_0,030637.-1,53397_0,051198.-0,080089_0,026616.0,55292_

PartitionKey: Stored in MetaData Table as Property
'DataExistsInTableForYears' as given below:
Starts: LA43.86_LO-82.75_20050101;H100
Ends: LA43.86_LO-82.75_20051231;H100

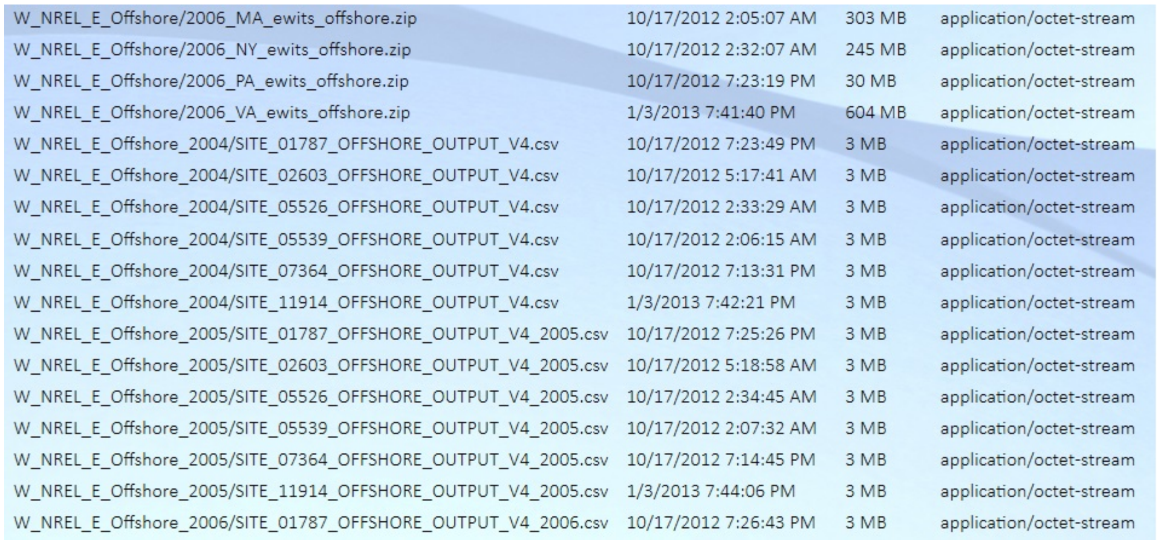
*data for other days of the years has been omitted for simplicity

Maximum wind speed for a single day and the time at which it occurs

Harmonics stored in this property as shape descriptor given by the following format:

K_DC_M1,M1'.P1,P1'_M2,M2'.P2,P2'
K = Number of harmonics required to be stored
Underscore(_) seperates different harmonics
Comma (,) is equivalent of decimal point
Dot(.) seperates magnitude and phases

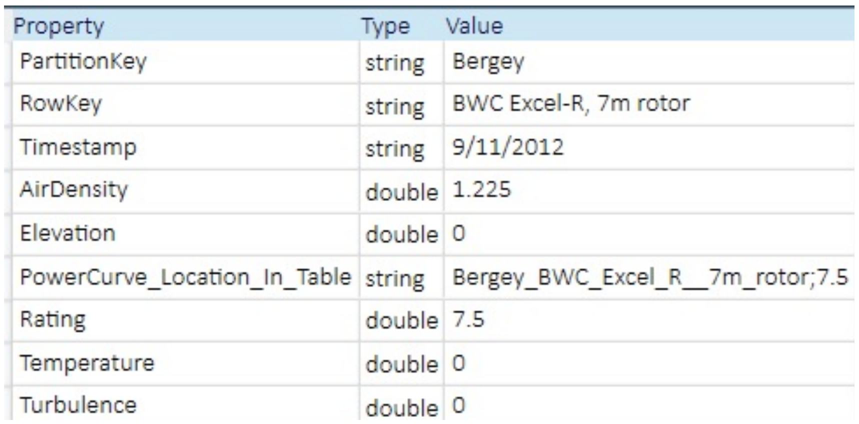
Figure C.4. Multiple metadata entries for wind sites in *MetaDataTable*



W_NREL_E_Offshore/2006_MA_ewits_offshore.zip	10/17/2012 2:05:07 AM	303 MB	application/octet-stream
W_NREL_E_Offshore/2006_NY_ewits_offshore.zip	10/17/2012 2:32:07 AM	245 MB	application/octet-stream
W_NREL_E_Offshore/2006_PA_ewits_offshore.zip	10/17/2012 7:23:19 PM	30 MB	application/octet-stream
W_NREL_E_Offshore/2006_VA_ewits_offshore.zip	1/3/2013 7:41:40 PM	604 MB	application/octet-stream
W_NREL_E_Offshore_2004/SITE_01787_OFFSHORE_OUTPUT_V4.csv	10/17/2012 7:23:49 PM	3 MB	application/octet-stream
W_NREL_E_Offshore_2004/SITE_02603_OFFSHORE_OUTPUT_V4.csv	10/17/2012 5:17:41 AM	3 MB	application/octet-stream
W_NREL_E_Offshore_2004/SITE_05526_OFFSHORE_OUTPUT_V4.csv	10/17/2012 2:33:29 AM	3 MB	application/octet-stream
W_NREL_E_Offshore_2004/SITE_05539_OFFSHORE_OUTPUT_V4.csv	10/17/2012 2:06:15 AM	3 MB	application/octet-stream
W_NREL_E_Offshore_2004/SITE_07364_OFFSHORE_OUTPUT_V4.csv	10/17/2012 7:13:31 PM	3 MB	application/octet-stream
W_NREL_E_Offshore_2004/SITE_11914_OFFSHORE_OUTPUT_V4.csv	1/3/2013 7:42:21 PM	3 MB	application/octet-stream
W_NREL_E_Offshore_2005/SITE_01787_OFFSHORE_OUTPUT_V4_2005.csv	10/17/2012 7:25:26 PM	3 MB	application/octet-stream
W_NREL_E_Offshore_2005/SITE_02603_OFFSHORE_OUTPUT_V4_2005.csv	10/17/2012 5:18:58 AM	3 MB	application/octet-stream
W_NREL_E_Offshore_2005/SITE_05526_OFFSHORE_OUTPUT_V4_2005.csv	10/17/2012 2:34:45 AM	3 MB	application/octet-stream
W_NREL_E_Offshore_2005/SITE_05539_OFFSHORE_OUTPUT_V4_2005.csv	10/17/2012 2:07:32 AM	3 MB	application/octet-stream
W_NREL_E_Offshore_2005/SITE_07364_OFFSHORE_OUTPUT_V4_2005.csv	10/17/2012 7:14:45 PM	3 MB	application/octet-stream
W_NREL_E_Offshore_2005/SITE_11914_OFFSHORE_OUTPUT_V4_2005.csv	1/3/2013 7:44:06 PM	3 MB	application/octet-stream
W_NREL_E_Offshore_2006/SITE_01787_OFFSHORE_OUTPUT_V4_2006.csv	10/17/2012 7:26:43 PM	3 MB	application/octet-stream

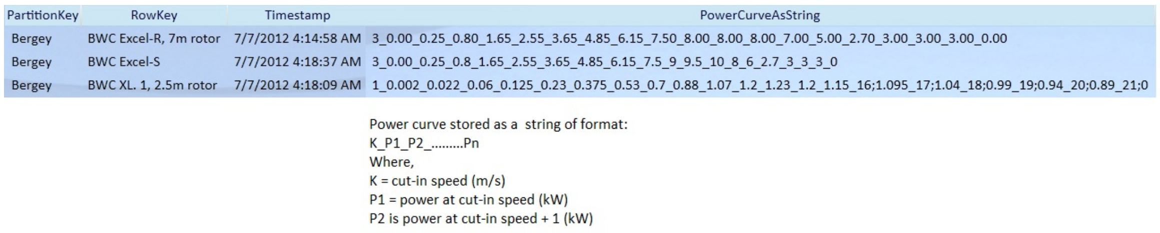
Figure C.5. Blob data for wind resources in container, *sitedata*

Fig C.6 shows metadata entry for wind turbine data. The field 'PowerCurve_Location_In_Table' points to the location of the power curve in data table as shown in Fig. C.7.



Property	Type	Value
PartitionKey	string	Bergey
RowKey	string	BWC Excel-R, 7m rotor
Timestamp	string	9/11/2012
AirDensity	double	1.225
Elevation	double	0
PowerCurve_Location_In_Table	string	Bergey_BWC_Excel_R__7m_rotor;7.5
Rating	double	7.5
Temperature	double	0
Turbulence	double	0

Figure C.6. Metadata entry for wind turbine data in *MetaDataTable*



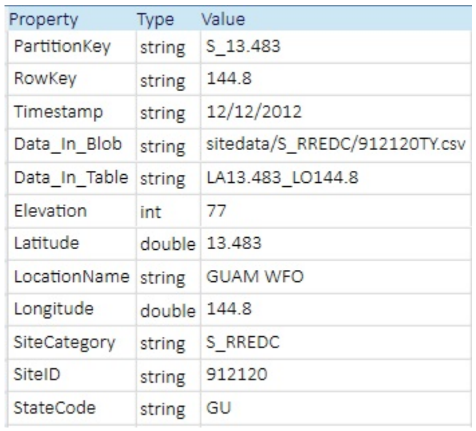
PartitionKey	RowKey	Timestamp	PowerCurveAsString
Bergey	BWC Excel-R, 7m rotor	7/7/2012 4:14:58 AM	3_0.00_0.25_0.80_1.65_2.55_3.65_4.85_6.15_7.50_8.00_8.00_8.00_7.00_5.00_2.70_3.00_3.00_3.00_0.00
Bergey	BWC Excel-S	7/7/2012 4:18:37 AM	3_0.00_0.25_0.8_1.65_2.55_3.65_4.85_6.15_7.5_9_9.5_10_8_6_2.7_3_3_3_0
Bergey	BWC XL. 1, 2.5m rotor	7/7/2012 4:18:09 AM	1_0.002_0.022_0.06_0.125_0.23_0.375_0.53_0.7_0.88_1.07_1.2_1.23_1.2_1.15_16;1.095_17;1.04_18;0.99_19;0.94_20;0.89_21;0

Power curve stored as a string of format:
K_P1_P2_.........Pn
Where,
K = cut-in speed (m/s)
P1 = power at cut-in speed (kW)
P2 is power at cut-in speed + 1 (kW)

Figure C.7. Power curve data entry for wind turbine in *DataTable*

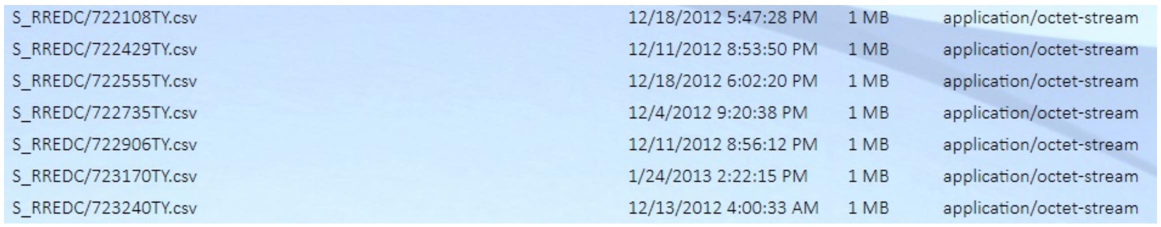
C.1.2. Solar Data Organization

Fig C.8 shows a metadata entry for a solar resource with 'Data_In_blob' shown in Fig B.9 and 'Data_In_Table' shown in Fig C.10 through C.14.



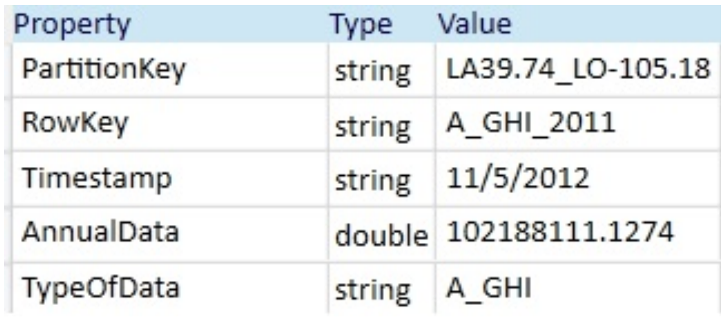
Property	Type	Value
PartitionKey	string	S_13.483
RowKey	string	144.8
Timestamp	string	12/12/2012
Data_In_Blob	string	sitedata/S_RREDC/912120TY.csv
Data_In_Table	string	LA13.483_LO144.8
Elevation	int	77
Latitude	double	13.483
LocationName	string	GUAM WFO
Longitude	double	144.8
SiteCategory	string	S_RREDC
SiteID	string	912120
StateCode	string	GU

Figure C.8. A typical metadata entry for a solar resource, *SolarMetaDataTable*



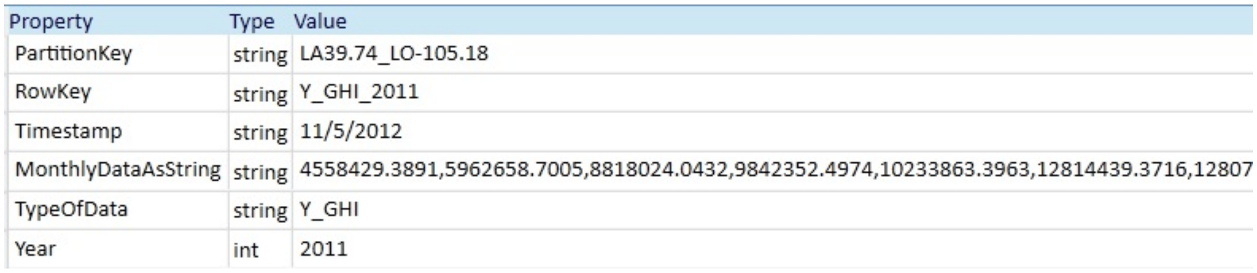
S_RREDC/722108TY.csv	12/18/2012 5:47:28 PM	1 MB	application/octet-stream
S_RREDC/722429TY.csv	12/11/2012 8:53:50 PM	1 MB	application/octet-stream
S_RREDC/722555TY.csv	12/18/2012 6:02:20 PM	1 MB	application/octet-stream
S_RREDC/722735TY.csv	12/4/2012 9:20:38 PM	1 MB	application/octet-stream
S_RREDC/722906TY.csv	12/11/2012 8:56:12 PM	1 MB	application/octet-stream
S_RREDC/723170TY.csv	1/24/2013 2:22:15 PM	1 MB	application/octet-stream
S_RREDC/723240TY.csv	12/13/2012 4:00:33 AM	1 MB	application/octet-stream

Figure C.9. Metadata entries for solar resources, *SolarMetaDataTable*



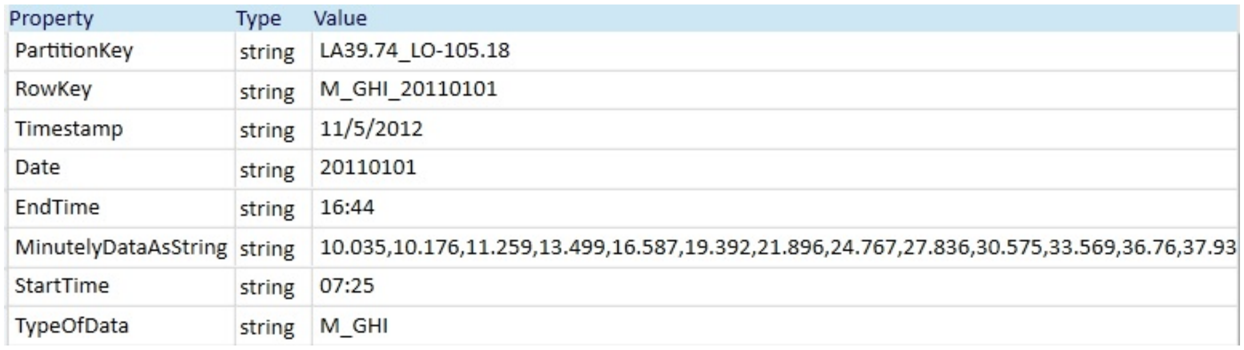
Property	Type	Value
PartitionKey	string	LA39.74_LO-105.18
RowKey	string	A_GHI_2011
Timestamp	string	11/5/2012
AnnualData	double	102188111.1274
TypeOfData	string	A_GHI

Figure C.10. Annual solar data stored in the data table, *SolarDataTable*



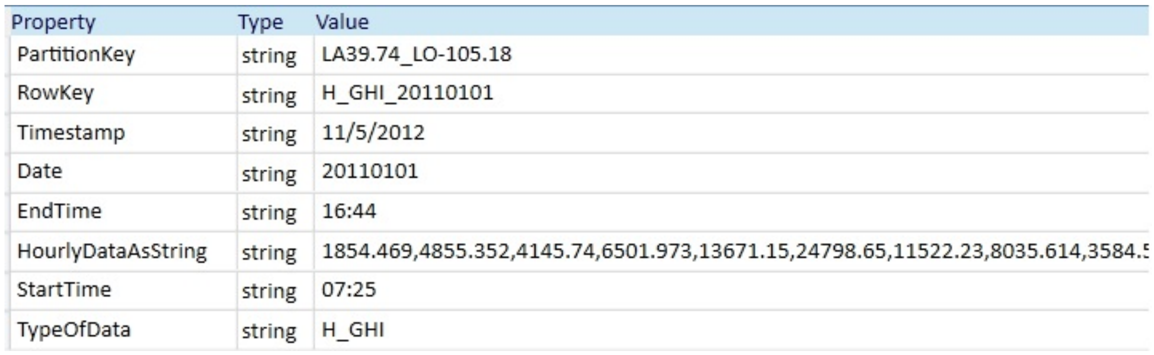
Property	Type	Value
PartitionKey	string	LA39.74_LO-105.18
RowKey	string	Y_GHI_2011
Timestamp	string	11/5/2012
MonthlyDataAsString	string	4558429.3891,5962658.7005,8818024.0432,9842352.4974,10233863.3963,12814439.3716,12807
TypeOfData	string	Y_GHI
Year	int	2011

Figure C.11. Monthly solar data for the year 2011 in the data table, *SolarDataTable*



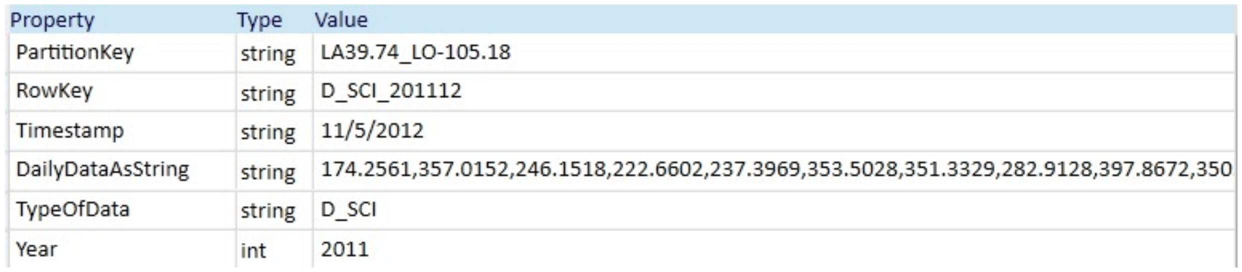
Property	Type	Value
PartitionKey	string	LA39.74_LO-105.18
RowKey	string	M_GHI_20110101
Timestamp	string	11/5/2012
Date	string	20110101
EndTime	string	16:44
MinutelyDataAsString	string	10.035,10.176,11.259,13.499,16.587,19.392,21.896,24.767,27.836,30.575,33.569,36.76,37.93
StartTime	string	07:25
TypeOfData	string	M_GHI

Figure C.12. Daily solar data in the data table, *SolarDataTable*



Property	Type	Value
PartitionKey	string	LA39.74_LO-105.18
RowKey	string	H_GHI_20110101
Timestamp	string	11/5/2012
Date	string	20110101
EndTime	string	16:44
HourlyDataAsString	string	1854.469,4855.352,4145.74,6501.973,13671.15,24798.65,11522.23,8035.614,3584.
StartTime	string	07:25
TypeOfData	string	H_GHI

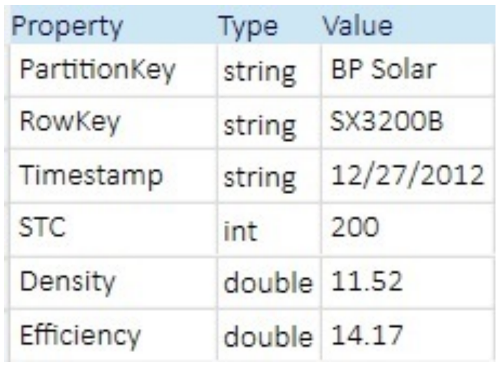
Figure C.13. Hourly solar data in the data table, *SolarDataTable*



Property	Type	Value
PartitionKey	string	LA39.74_LO-105.18
RowKey	string	D_SCI_201112
Timestamp	string	11/5/2012
DailyDataAsString	string	174.2561,357.0152,246.1518,222.6602,237.3969,353.5028,351.3329,282.9128,397.8672,350
TypeOfData	string	D_SCI
Year	int	2011

Figure C.14. Minutely solar data in the data table, *SolarDataTable*

The following figure shows how a solar panel data entry looks in *SolarMetaDataTa*ble.



Property	Type	Value
PartitionKey	string	BP Solar
RowKey	string	SX3200B
Timestamp	string	12/27/2012
STC	int	200
Density	double	11.52
Efficiency	double	14.17

Figure C.15. A typical entry for solar panel metadata, *SolarMetaDataTable*

C.2. Detailed Algorithm to Standardize Data (IDRF) for Wind Forecasting

STEP 1: If the downloaded file is a zip file, extract the file using *unzip* service feature to obtain a .csv file and update *MetaDataTable* with a link to the unzipped file.

STEP 2: Read the required information from this file. For wind, the service should read the date, time, and wind speeds for 100m height using service feature *RawWindSpeedData*.

STEP 3: Convert the wind speed data to the user requested height using the formula:

$$\text{NewSpeed} = \text{GivenSpeed} * (\text{newHeight}/\text{OldHeight})^{1/7}$$

STEP 4: How many wind speed values are there for a given day? This information is needed to perform N-point DFT on the wind speed data.

STEP 5: Daily maximum wind speed values and the time at which they occur must also be recorded.

STEP 6: For each day of the year, copy the wind speed values for a single day, normalize them by maximum daily wind speed value and calculate their harmonics using service features *Harmonics*.

STEP 7: After obtaining the harmonics, calculate the minimum number of harmonics required to obtain the data back within 2% MAPE by reconstructing the data back in time-domain by using *MAPE* service feature.

$$MAPE = \frac{1}{N}\Sigma\frac{|At - Ft|}{At}$$

STEP 8: Create a Form Shape Descriptor as follows (uses *FormShapeDescriptor* service feature):
Pattern description = (MPeak, S)
Where,
MPeak is the peak value of the period and S is the Shape descriptor.
S = K – d – i.j – m.n - ……….. p.q -> S
= (Index K, MDC, (M1, P1), (M2, P2), …….(M3, P3))
The value of K indicates how much information will follow.

MDC, M1, M2,…., MK are the magnitudes of the harmonics normalized by MPeak.
P1, P2,…., PK are the phase angles of the harmonics.

STEP 9: Repeat Step 1 to 8 for all the years of data that is available.

STEP 10: Download turbine software files from www.inl.gov to obtain turbine data for different manufacturers. Store the power curve information and model details, etc. in a file or table using *TurbinePowerData* service feature. This information will be used later to forecast power by FGF.

C.3. Detailed Algorithm to Standardize Data (IDRF) for Solar Forecasting

An algorithm is required to read the information from the files in the blob.

STEP 1: For solar, the algorithm should be able to read the date, time, PSP and GHI data. It should also record how many number of minutes of data is given for a given hour, and how many hours of data is given for a day, for each day of the year. IDRF uses service feature *RawSolarData* to read the required details from a file.

STEP 2: If the data is minutely, it needs to be converted to hourly data and stored in *SolarDataTable*. IDRF does so by using service feature *MinutelyDataToHourlyData.*

STEP 3: The hourly data was converted to daily data and stored in *SolarDataTable*. Service feature *HourlyDataToDailyData* is used by IDRF to perform such operations.

STEP 4: Convert daily data from above to monthly data and stored in *SolarDataTable*. Service feature *DailyDataToMonthlyData* is used by IDRF to perform such operations.

STEP 5: Convert monthly data from above to annual data and stored in *SolarDataTable*. Service feature *MonthlyDataToAnnualData* is used by IDRF to perform such operations.

STEP 6: Minutely SCI was calculated by taking ratio of the 1-minute global PSP data and the corresponding 1-minute GHI-ET and stored it in *SolarDataTable*. Service feature *MinutelyData* was used by IDRF for this purpose.

STEP 7: Repeat Step 2 to 6 for all years of data available.

C.4. Detailed Algorithm used by FGF for Wind Project Prospecting

STEP 1: Read the turbine power curve using *TurbinePowerCurveEntry* for the turbine specified by the user.

STEP 2: Read the wind speed data and other details from the csv file using *RawWindSpeedData*.

STEP 3: Using the power curve for turbine from Step 1, the power produced by the turbine at the given speed was calculated. Interpolation must be used to obtain precise power values using wind speeds.

STEP 4: The daily power values were added to obtain monthly power produced.

STEP 5: Repeat step 2 to 4 for each month.

STEP 6: Repeat Step 2 to 5 for each year of data available.

STEP 7: Three-year average of the monthly power produced for each year was calculated.

STEP 8: A report was generated using *ReportPreparation* and emailed to the user using EmailServices.

C.5. Algorithm used by FGF for Wind Project Prospecting with Uncertainty Quantification

For uncertainty quantification, all steps in C.4 are required, in addition to the *ExceedanceProbability* service feature to calculate the uncertainty data as described below:

To find monthly uncertainty values for NREL data:

STEP 1: Find monthly total production for each month of each year for three years.

STEP 2: Use the 36 points to find μ_m and σ_m using service feature *Average* and *Sigma*.

STEP 3: For each month, use the three data points, one from each year for that month, to find μ_i.

STEP 4: Use the cumulative distribution function of Normal Distribution (with μ_i and σ_m) to find P95, P90, P75 and P50 through service feature *ExceedanceProbability*.

STEP 5: Repeat steps 3-4 for each month of the year (each month has a different μ_i but same σ_m.

To find annual uncertainty values for NREL data, steps 1-4 are implemented in service feature *Averages*:
STEP 1: Find annual total production for the first 12 months. This is ATP1.

STEP 2: Advance one month to February. Find the annual total production for the next 12 months. This is ATP2.

STEP 3: Repeat step 2 until the end of third year is reached. That will be ATP 25 (24+1).

STEP 4: Used the 25 data points to fund μATP and σATP

STEP 5: Used the cumulative distribution function of Normal Distribution (with μATP and σATP) to find P95, P90, P75 and P50.

C.6. Detailed Algorithm used by FGF for Solar Project Prospecting

STEP 1: Read solar panel information from *SolarMetaDataTable* using *SolarPanelEntity*. Read *Data_In_Table* field to obtain location of data in *SolarDataTable*.

STEP 2: Read the monthly GHI, GHI-ET and SCI data for the location from table *SolarDataTable*.

STEP 3: Read the annual GHI, GHI-ET and SCI data for the location from table *SolarDataTable*.

STEP 4: Repeat Step 1 to 3 for each year.

STEP 5: Calculated three-year average for monthly GHI and GHI-ET.

STEP 6: Calculated the product of solar panel efficiency and monthly averages.

STEP 7: A report was generated using *ReportPreparation* and emailed to the user using EmailServices.

C.7. Detailed Algorithm used by FGF for Solar Project Prospecting with Uncertainty Quantification

For uncertainty quantification, all the above steps in C.6 are required in addition to *ExceedanceProbability* service feature to calculate the uncertainty data.

To find monthly uncertainty values for NREL data (3 years solar data):
STEP 1: Find monthly total production for each month of each year for three years from *SolarDataTable* using *DailyDataToMonthlyData* to read monthly data.

STEP 2: Use the 36 points to find μ_m and σ_m using service features *Averages* and *Sigmas*, respectively.

STEP 3: For each month, use the three data points, one from each year for that month, to find μ_i using service feature *Averages*.

STEP 4: Use the cumulative distribution function of Normal Distribution (with μ_i and σ_m) to find P95, P90, P75 and P50 through service feature *ExceedanceProbability*.

STEP 5: Repeat steps 3-4 for each month of the year (each month has a different μ_i but same σ_m.

To find annual uncertainty values for NREL data (3 years solar data), steps 2-4 are implemented in service features *Averages* and *ConbiningSigmas*:

STEP 1: Find annual total production for the first 12 months from *SolarDataTable* using *DailyDataToMonthlyData* to read monthly data. This is ATP1.

STEP 2: Advance one month to February. Find the annual total production for the next 12 months. This is ATP2.

STEP 3: Repeat step 2 until the end of third year is reached. That will be ATP 25 (24+1).

STEP 4: Use the 25 data points to fund μ_{ATP} and σ_{ATP}.

STEP 5: Use the cumulative distribution function of Normal Distribution (with μATP and σATP) to find P95, P90, P75 and P50 through *ExceedanceProbability* service feature.

CombiningSigmas service feature has the functionality that given the uncertainty in the measurement of hourly data for a given day it can return the uncertainty (sigma) value for the entire day. The following equation were be implemented in order to achieve this:

$$\sigma_X = \sqrt{\sum_i \sigma^2_{X_i} + \sum_{i,j} \text{cov}(X_i, X_j)}$$

As the correlation between hourly data for a given day is not zero, i.e. next hour's data is in to previous hour's data. The GHI data was used along with the sigma values to compute a day's sigma, by using the equations below. If not, COV = 0.

$$\mu_X = \frac{\sum_i N_{X_i}\mu_{X_i}}{\sum_i N_{X_i}}$$

$$\sigma_X = \sqrt{\frac{\sum_i N_{X_i}(\sigma^2_{X_i} + \mu^2_{X_i})}{\sum_i N_{X_i}} - \mu^2_X} = \sqrt{\frac{\sum_i N_{X_i}\sigma^2_{X_i}}{\sum_i N_{X_i}} + \frac{\sum_{i<j} N_{X_i}N_{X_j}(\mu_{X_i} - \mu_{X_j})^2}{\left(\sum_i N_{X_i}\right)^2}}$$

In case of combining sigma's for a given day to form a month's sigma, it can be assumed that the GHI data are fairly independent and so are the sigma values, as a day's GHI might not affect next day's GHI. Therefore, we can assume

$$\mathrm{cov}(X_i, X_j) = 0, \quad \forall i < j$$

$$\Rightarrow \sigma_X = \sqrt{\sum_i \sigma_{X_i}^2}.$$

Similarly, each month's GHI can be assumed to be independent of last month's GHI, so the above equation would still hold true.

C.8. Detailed Algorithm to obtain Wind Operational Forecast

External Data Capture Framework -

STEP 1: Operational data is obtained using XML for any given location using service feature *XMLDataDownload*.

STEP 2: After downloading the XML document for the next day (from midnight to midnight) for the identified NDFD locations, wind speeds must be read from this document and stored temporarily.

Internal Data Storage Framework -

STEP 3: Power curve values must be read for the specified turbine using *TurbinePowerCurveEntry*.

STEP 4: Using the XML *doc* obtained by EDCF, compute the daily total using the NDFD forecast (if forecast is made in 3-Hour block, multiply those values by 3 when computing the daily total).

Forecast Generation Framework -

STEP 5: Using interpolation, power generated must be calculated for the next day.

STEP 6: A report was generated using *ReportPreparation* and emailed to the user using EmailServices.

C.9. Detailed Algorithm to Calculate Solar Operational Forecast

External Data Capture Framework-

STEP 1: Operational data is obtained using XML for any given location using service feature *XMLDataDownload*.

STEP 2: After downloading the XML document for the next day (from sunrise to sunset) for the identified NDFD locations, cloud cover must be read from this document and stored temporarily.

Internal Data Storage Framework -

STEP 3: Using the XML *doc* obtained by EDCF, compute the daily total cloud cover (TCC) and limit its value such that TCC ≤ 0.9009.

Forecast Generation Framework -

STEP 4: Convert the forecasted daily average cloud cover TCC into forecasted daily K_D using the following equation:

$$KD = \frac{sin^{-1}(\frac{TCC}{0.9009}) - 1.277}{2.335}$$

STEP 5: Compute extra-terrestrial solar irradiation using the formula below

θ_z = $\cos^{-1}$ [sin(δ) * sin(latitude) + cos(δ) * cos(latitude) * cos(hourAngle)]
Sunrise Hour Angle = $\cos^{-1}$ (-1 * tan(latitude) * tan(δ))
Sunset Hour Angle = -1 * sunriseHourAngle
Sunrise Time = (12 - (sunriseHourAngle * 180 / PI))
Sunset Time = (12 - (sunsetHourAngle * 180 / PI)

I_0 one hour: $I_0 = 1367 * E_0 * cos\theta_z$

I_0 in one day (Wh/m^2): sum of the hourly total from sunrise hour to sunset hour for that day (each day is different).

I_0 in one month (Whr/m^2): sum of the daily total for all the days in one month (each day is different, so each month with 31 days is different).

I_0 in one year (Whr/m^2): sum of the daily total over 12 months.

$$dayAngle = \frac{2 * \pi * (dayNumber - 1)}{365}$$

E_0 = 1.00010 + 0.034221 * cos (dayAngle) + 0.001280 * sin (dayAngle)
+ 0.000719 * cos (2 * dayAngle) + 0.000077 * sin (2 * dayAngle);

solarDeclination, δ = (0.006918 - 0.399912 * cos(dayAngle)
+ 0.070257 * sin (dayAngle) - 0.006758 * cos (2 * dayAngle)
+ 0.000907 * sin (2 * dayAngle) - 0.002697 * cos (3 * dayAngle)
+ 0.00148 * sin (3 * dayAngle)

STEP 6: A report was generated using *ReportPreparation* and emailed to the user using EmailServices.

C.10. Great Circle Distance Algorithm

As the user would not know the exact co-ordinates of the location in metadata table, an algorithm is in place to predict a site close to the location specified by the user. The algorithm calculates the great circle distance between various locations that are close to the user provided location. The site which has minimum distance from the location specified by the user is automatically selected. The user is informed about the location where the data will be coming from and how far that location is from the specified location (in Km.). The great circle distance is calculated as follows:

$$\Delta\widehat{\sigma} = \arccos\big(\sin\phi_s\sin\phi_f + \cos\phi_s\cos\phi_f\cos\Delta\lambda\big).$$
$$d = r\,\Delta\widehat{\sigma}.$$

Where,

$\phi_s, \lambda_s;\ \phi_f, \lambda_f$ are the latitude and longitude of the two points.

The following code implements the great circle distance functionality:

```
double dLat = (latitudes[i] - latitude) * Math.PI / 180;
double dLon = (longitudes[i] - longitude) * Math.PI / 180;
```

```
var a = Math.Sin(dLat / 2) * Math.Sin(dLat / 2) +
   Math.Sin(dLon / 2) * Math.Sin(dLon / 2) * Math.Cos(latitudes[i]) *
Math.Cos(latitude);

var c = 2 * Math.Atan2(Math.Sqrt(a), Math.Sqrt(1 - a));
dMin[i] = RoE * c;
```

where RoE = 6371

The following link was used to validate the algorithm:

http://www.movable-type.co.uk/scripts/latlong.html

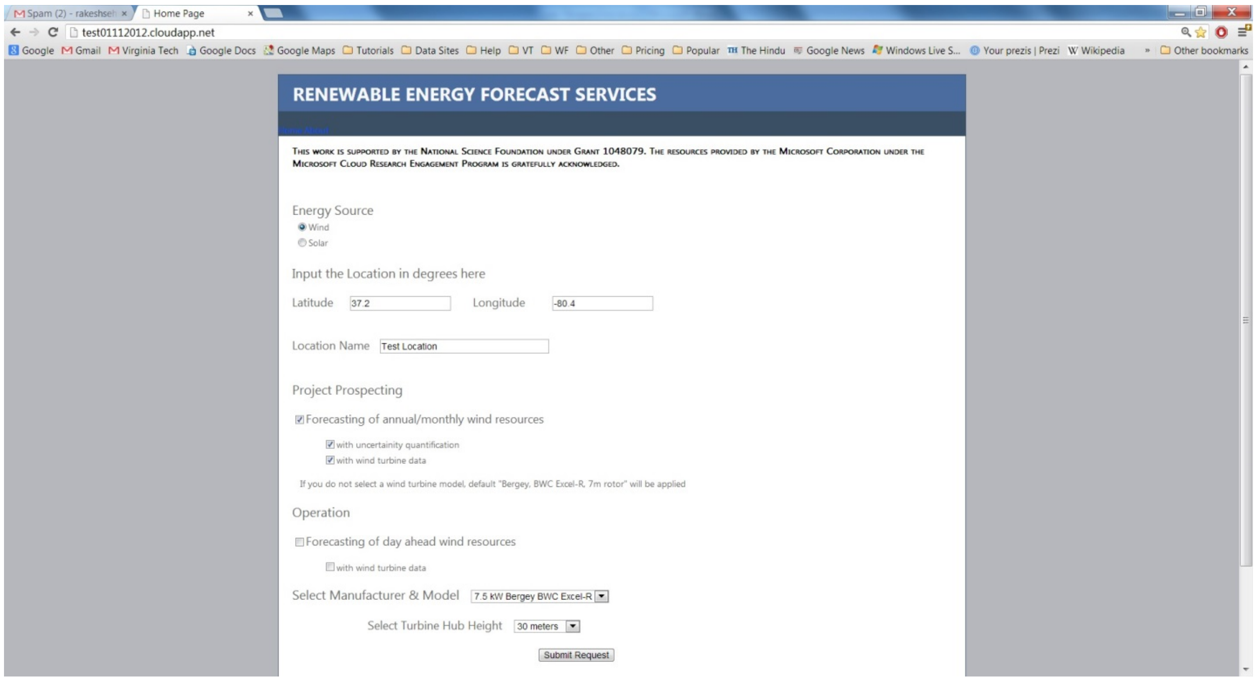
APPENDIX D - Web Interfaces

The webpage at this link http://test01112012.cloudapp.net/ (the URL does not work anymore as the contract with Microsoft ended May 2013) provides access to all the forecast services. The forecast services are available for wind and solar, both of which can be further used to forecast project prospecting or operational forecast data.

D.1. Wind Forecasting

Project Prospecting

By selecting the radio button on the webpage as below, the user can request wind (shown below) or solar forecast.



THIS WORK IS SUPPORTED BY THE NATIONAL SCIENCE FOUNDATION UNDER GRANT 1048079. THE RESOURCES PROVIDED BY THE MICROSOFT CORPORATION UNDER THE MICROSOFT CLOUD RESEARCH ENGAGEMENT PROGRAM IS GRATEFULLY ACKNOWLEDGED.

Energy Source
[✓ Wind]
Solar

Input the Location in degrees here

Latitude: 37.2 Longitude: -80.4

Location Name: Test Location

Project Prospecting

[✓] Forecasting of annual/monthly wind resources
- [✓] with uncertainity quantification
- [✓] with wind turbine data

If you do not select a wind turbine model, default "Bergey, BWC Excel-R, 7m rotor" will be applied

Operation

[ ] Forecasting of day ahead wind resources
- [ ] with wind turbine data

Select Manufacturer & Model: 7.5 kW Bergey BWC Excel-R

Select Turbine Hub Height: 30 meters

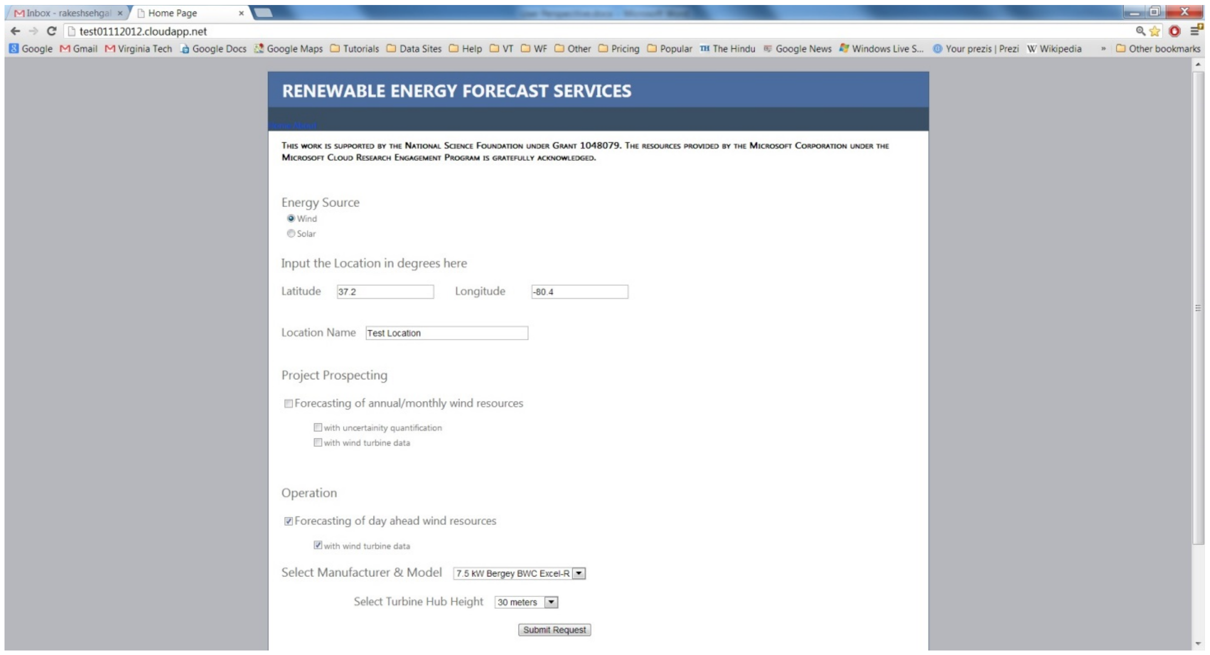
Submit Request

Figure D.1 Shows wind project prospecting forecast made with uncertainty data and for the shown turbine model

After the user submits the request, Fig D.5 shows the next page that is available for user's information and input.

Wind Operational Forecasting

If a user wants to request operation data for the next day, they can select the "Operation" option and select if they want to choose any particular turbine model. Fig. D.2 shows such a page and Fig. D.5 will be the next page to be displayed after the user submits the request.



This work is supported by the National Science Foundation under Grant 1048079. The resources provided by the Microsoft Corporation under the Microsoft Cloud Research Engagement Program is gratefully acknowledged.

Energy Source
[✓ Wind] Solar

Input the Location in degrees here

Latitude 37.2 Longitude -80.4

Location Name Test Location

Project Prospecting

☐ Forecasting of annual/monthly wind resources
☐ with uncertainity quantification
☐ with wind turbine data

Operation

☑ Forecasting of day ahead wind resources
☑ with wind turbine data

Select Manufacturer & Model 7.5 kW Bergey BWC Excel-R

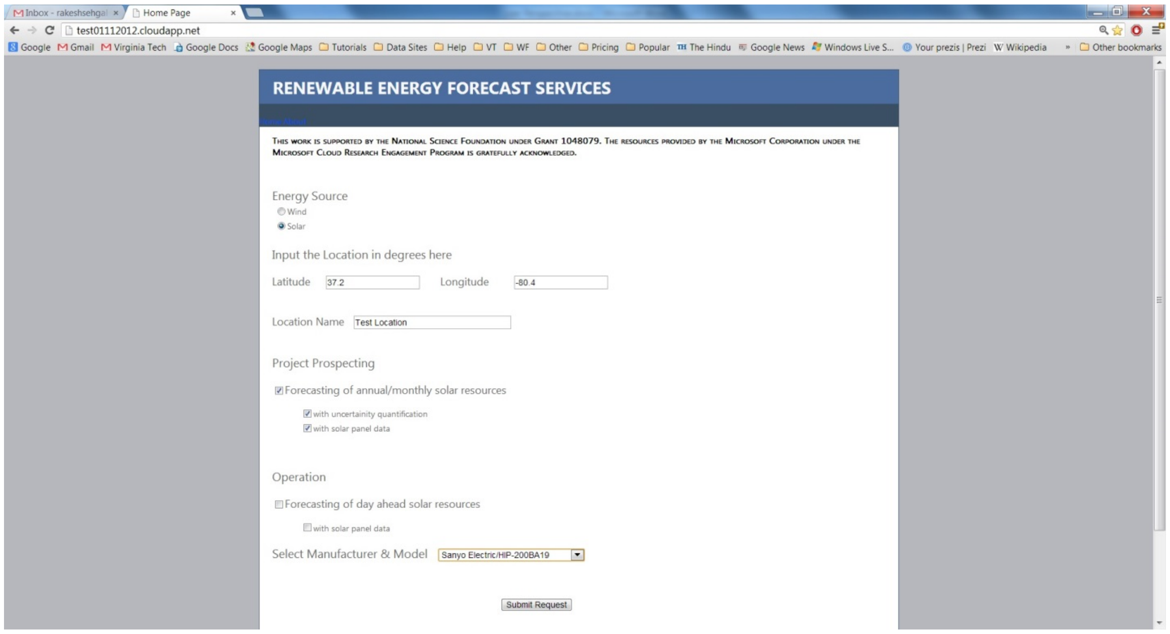
Select Turbine Hub Height 30 meters

Figure D.2 Webpage where a user submits request

D.2. Solar Forecasting

Project Prospecting

Fig D.3 below shows a request for solar data for project prospecting. In this request, the use request for a particular model of solar panel and data with uncertainty information. Fig D.5 will be the next screen to be shown to the user.


RENEWABLE ENERGY FORECAST SERVICES

THIS WORK IS SUPPORTED BY THE NATIONAL SCIENCE FOUNDATION UNDER GRANT 1048079. THE RESOURCES PROVIDED BY THE MICROSOFT CORPORATION UNDER THE MICROSOFT CLOUD RESEARCH ENGAGEMENT PROGRAM IS GRATEFULLY ACKNOWLEDGED.

Energy Source
Wind [✓ Solar]

Input the Location in degrees here

Latitude: 37.2 Longitude: -80.4

Location Name: Test Location

Project Prospecting

[✓] Forecasting of annual/monthly solar resources
- [✓] with uncertainity quantification
- [✓] with solar panel data

Operation

[ ] Forecasting of day ahead solar resources
- [ ] with solar panel data

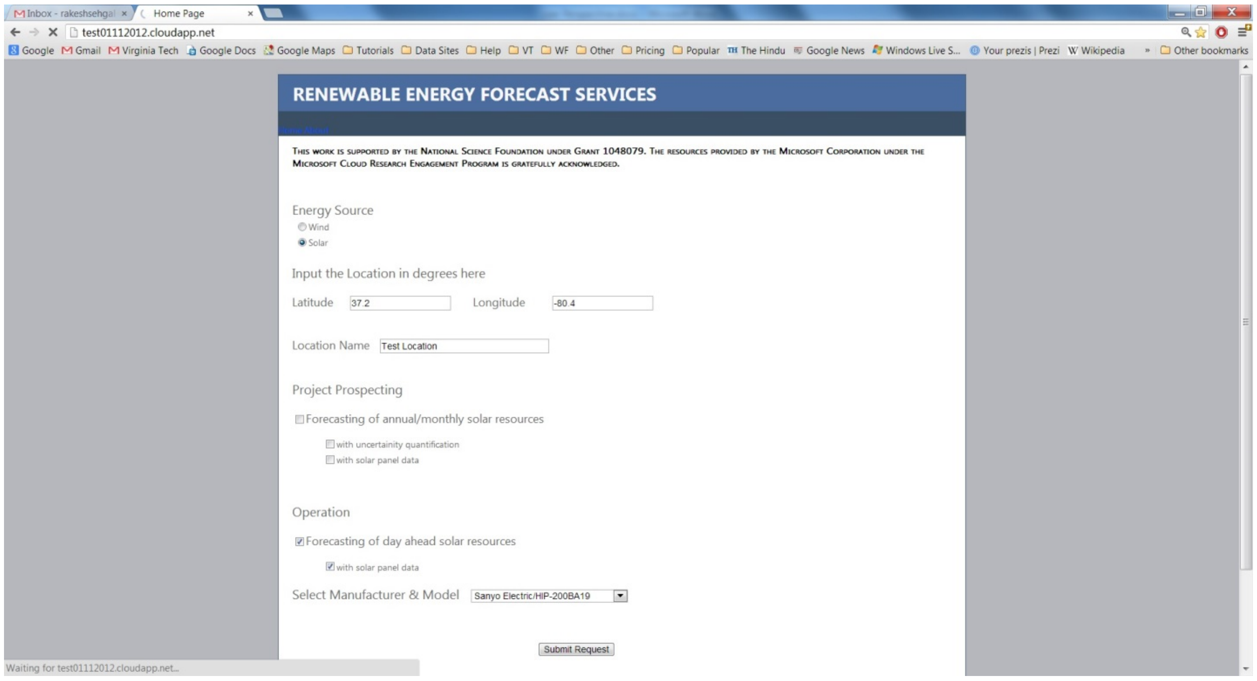
Select Manufacturer & Model: Sanyo Electric/HIP-200BA19

Submit Request

Figure D.3 Webpage for project prospecting request for solar

Solar Operational Forecasting

Fig D.4 shows a request for operation solar data.


RENEWABLE ENERGY FORECAST SERVICES

THIS WORK IS SUPPORTED BY THE NATIONAL SCIENCE FOUNDATION UNDER GRANT 1048079. THE RESOURCES PROVIDED BY THE MICROSOFT CORPORATION UNDER THE MICROSOFT CLOUD RESEARCH ENGAGEMENT PROGRAM IS GRATEFULLY ACKNOWLEDGED.

Energy Source
Wind [✓ Solar]

Input the Location in degrees here

Latitude: 37.2 Longitude: -80.4

Location Name: Test Location

Project Prospecting

☐ Forecasting of annual/monthly solar resources
☐ with uncertainity quantification
☐ with solar panel data

Operation

☑ Forecasting of day ahead solar resources
☑ with solar panel data

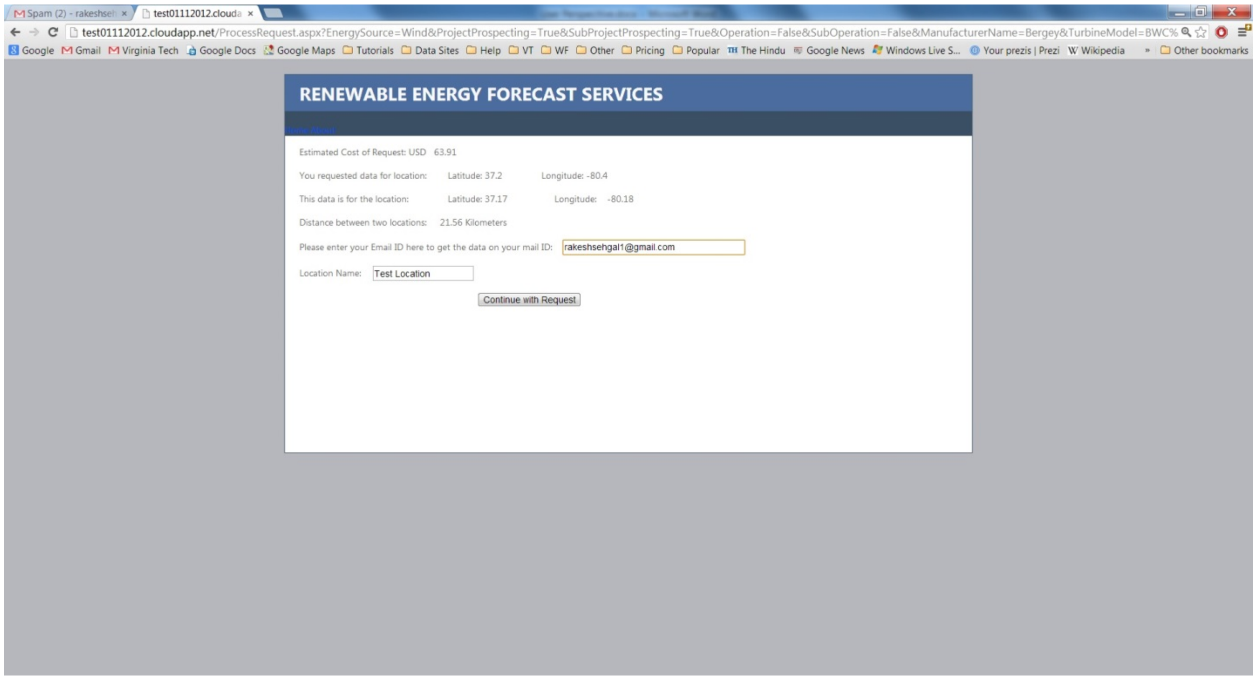
Select Manufacturer & Model: Sanyo Electric/HIP-200BA19

Submit Request

Figure D.4 Webpage for operational forecast request for solar

After the user submit an operation or project prospecting request, Fig D.5 is encountered by the user next. It shows the user how much is the cost of using the services, according to the options selected by them. It also shows them what co-ordinates they entered.

Also, as the data for the exact co-ordinates entered by the user, might not always be available, it shows where exactly will the data be coming from. It also, shows the distance (calculated using the Great Circle Distance) from their requested co-ordinates in Kilometers. If the user is ok with that location, they have to enter their email address, and click on “Continue with Request” button to have the desired results emailed to them.


Spam (2) - rakeshseh
test01112012.clouda
test01112012.cloudapp.net/ProcessRequest.aspx?EnergySource=Wind&ProjectProspecting=True&SubProjectProspecting=True&Operation=False&SubOperation=False&ManufacturerName=Bergey&TurbineModel=BWC%
Google Gmail Virginia Tech Google Docs Google Maps Tutorials Data Sites Help VT WF Other Pricing Popular The Hindu Google News Windows Live S... Your prezis | Prezi Wikipedia Other bookmarks

RENEWABLE ENERGY FORECAST SERVICES

Estimated Cost of Request: USD 63.91

You requested data for location: Latitude: 37.2 Longitude: -80.4

This data is for the location: Latitude: 37.17 Longitude: -80.18

Distance between two locations: 21.56 Kilometers

Please enter your Email ID here to get the data on your mail ID: rakeshsehgal1@gmail.com

Location Name: Test Location

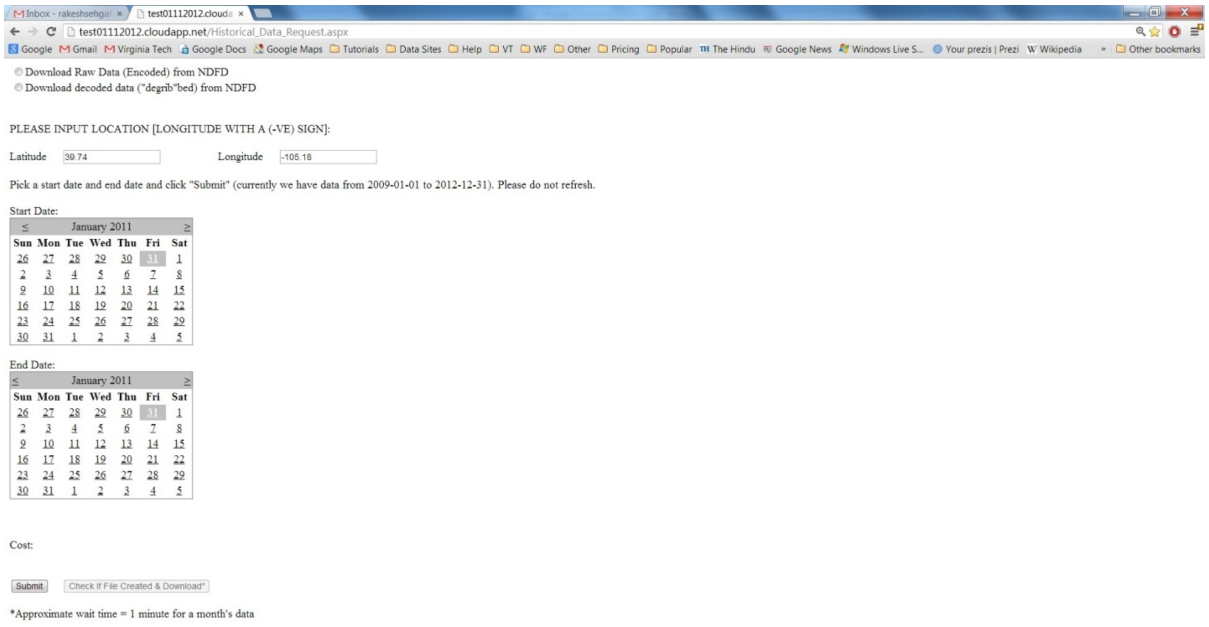
Figure D.5 Shows the intermediate page, where the user enters their email address if they want to continue with the request

When the user submits the request, they are shown a message that reads, "`Your data will be emailed to you! Please make sure to add us to your address book and check the emails in Junk Folder`" and depending on the services selected by them, they receive an email with the results.

APPENDIX E – Reuse & Extension Details

E.1. Reuse of FaaS for Historical Data Download Service

The webpage at the link http://test01112012.cloudapp.net/Historical_Data_Request.aspx lets user request historical data in raw format (.tar file) or decoded (*degribbed*) file as a zip file. Fig. E.1 is the default page that opens up when one clicks on the above link. No radio button is selected by default, so when a user click on one of the radio button options, it displays the cost of using that service. The calendar provided, lets user selected a date or a range of dates for which they want data.



test01112012.cloudapp.net/Historical_Data_Request.aspx

○ Download Raw Data (Encoded) from NDFD
○ Download decoded data ("degrib"bed) from NDFD

PLEASE INPUT LOCATION [LONGITUDE WITH A (-VE) SIGN]:

Latitude 39.74 Longitude -105.18

Pick a start date and end date and click "Submit" (currently we have data from 2009-01-01 to 2012-12-31). Please do not refresh.

Start Date:

≤			January 2011			≥
Sun	Mon	Tue	Wed	Thu	Fri	Sat
26	27	28	29	30	31	1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30	31	1	2	3	4	5

End Date:

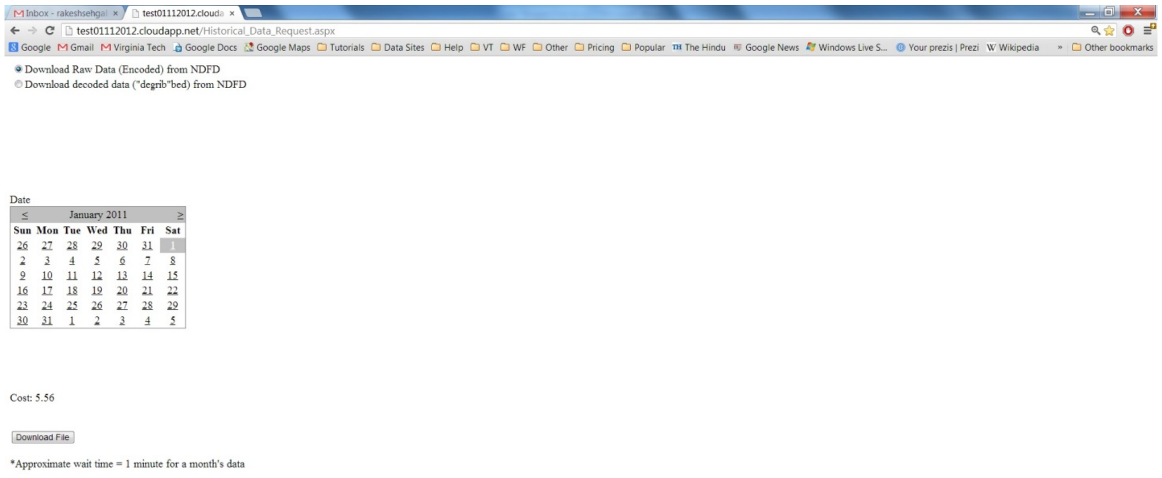
≤			January 2011			≥
Sun	Mon	Tue	Wed	Thu	Fri	Sat
26	27	28	29	30	31	1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30	31	1	2	3	4	5

Cost:

*Approximate wait time = 1 minute for a month's data

Figure E.1 Shows the default historical data webpage

The figure below shows that the user selected "Raw Data" option and its corresponding cost. When the user clicks on "Download File" button, a tar file is downloaded.



test01112012.cloudapp.net/Historical_Data_Request.aspx

● Download Raw Data (Encoded) from NDFD
○ Download decoded data ("degrib"bed) from NDFD

Date

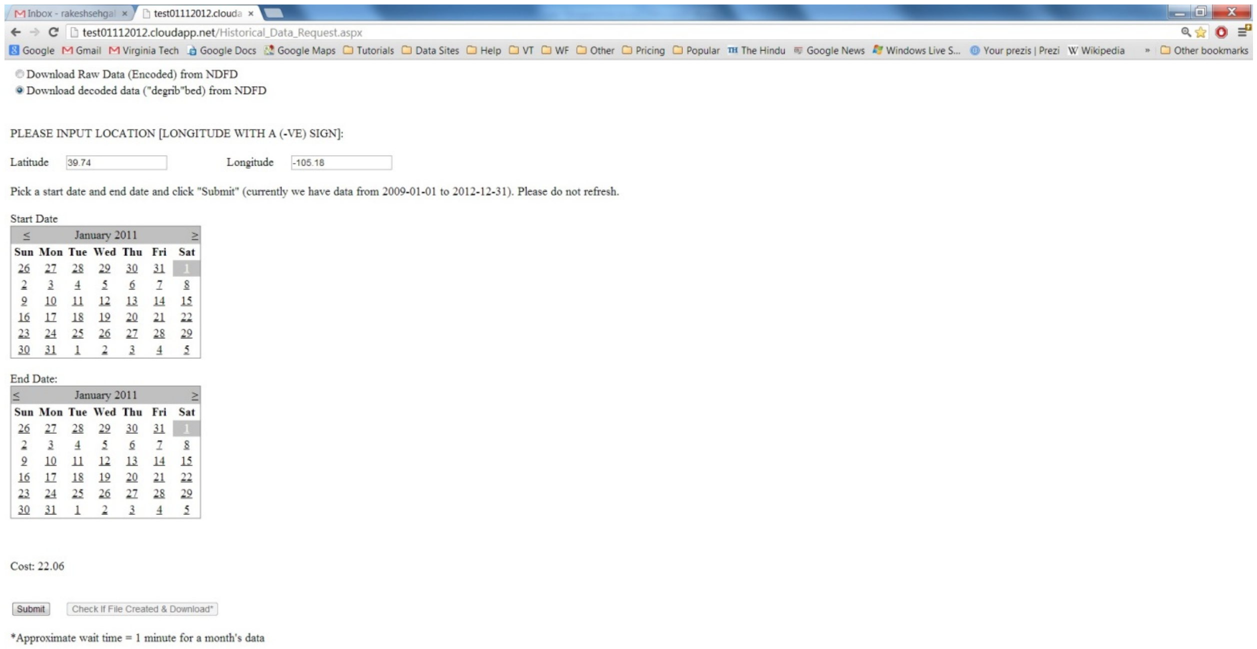
<			January 2011			>
Sun	Mon	Tue	Wed	Thu	Fri	Sat
26	27	28	29	30	31	1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30	31	1	2	3	4	5

Cost: 5.56

Download File

*Approximate wait time = 1 minute for a month's data

Figure E.2 Shows the user selected "Raw Data" option



test01112012.cloudapp.net/Historical_Data_Request.aspx

○ Download Raw Data (Encoded) from NDFD
● Download decoded data ("degrib"bed) from NDFD

PLEASE INPUT LOCATION [LONGITUDE WITH A (-VE) SIGN]:

Latitude 39.74 Longitude -105.18

Pick a start date and end date and click "Submit" (currently we have data from 2009-01-01 to 2012-12-31). Please do not refresh.

Start Date

<			January 2011			>
Sun	Mon	Tue	Wed	Thu	Fri	Sat
26	27	28	29	30	31	1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30	31	1	2	3	4	5

End Date:

<			January 2011			>
Sun	Mon	Tue	Wed	Thu	Fri	Sat
26	27	28	29	30	31	1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30	31	1	2	3	4	5

Cost: 22.06

Submit Check If File Created & Download*

*Approximate wait time = 1 minute for a month's data

Figure E.3 Shows decoded data option selected by the user

What's going on inside?

As soon as the user submits the request, the web role puts the request in a queue (name of the queue container is, *cloudservicequeue*), and the worker role (through

FaaSController) picks up the request and decodes the data in request. FaaS Controller then uses services, EDCF and IDRF to process the request.

This service exists only for Solar instance. As the original tar files are available for download on request and only for a week from NDFD, the files have to be downloaded by a developer to the blob using EDCF (the files currently exist for 4 years (2009-2012)). Using the web page, shown in above figures, the user can at any time request processed or raw data.

a. **Raw Data**: As soon as the user clicks the "Download File" button, a tar file is automatically downloaded to the default download directory of the user from the blob.
b. **Decoded Data**: After waiting for an appropriate time (approximate wait time of 1 minute for a month's decoded data), when the user clicks "Check if file created & Download" button, the browser starts to download a zip file that contains ".csv" file with the requested data. Multiple files exist for each of data the user requested.

As the HTTPDownload functionality already existed in the tools, only a new operation contract had to be defined with http source URL as a data member to point to a location of the data of the resource on the Internet. After the data is downloaded to the blob, IDRF uses custom services features (BasicOfficeOperations.*UntarFile* , *Degribber*, and BasicOfficeOperations.*ExtractFileFromStream*) to convert the data from NWS encoded format to a text file and store it in the blob. After the data is converted, the user can download the data by clicking on the button on the web page. The following code shows the implementation of the operation contract *StoreHistoricalData* from service contract IIDRF:

```
public void StoreHistoricalData(string startDate, string endData, double latitude, double
                                                                                   longitude)
{
bool lastDay = false;
CustomDateTime startDateInFormat = new CustomDateTime(startDate);
CustomDateTime endDateInFormat = new CustomDateTime(endData);
string nextDate = startDateInFormat.Date;
string zipFilenameInBlob = "sitedata/HistoricalData/" + nextDate + "_" + endDateInFormat.Date
+ ".zip";
string zipFileDirectory = localResource.RootPath + startDateInFormat.Date + "_" +
endDateInFormat.Date + "\\";

//CREATE A FOLDER FOR STARTDATE, THIS IS WHERE ALL THE DECODED FILES
WILL BE AND THEN ZIPPED
Directory.CreateDirectory(zipFileDirectory);

//DOWNLOAD DEGRIB.ZIP (degrib has been zipped as .exe files can't be stored in the blob)
Stream str = readFromBlob.ReadFileToStream("sitedata/degrib.zip");

//EXTRACT DEGRIB.EXE
BasicOfficeOperations.ExtractFileFromStream(str, "degrib.exe", zipFileDirectory);

while(true)
{
CustomDateTime dateInFormat = new CustomDateTime(nextDate);
string blobFileURL = "sitedata/HistoricalData/9959_NDFD_YAN_" + nextDate + ".tar";
//DOWNLOAD TAR FILE FROM THE BLOB
string tarBallName = readFromBlob.DownloadFileFromBlob(blobFileURL, zipFileDirectory);

//IF FILE DOES NOT EXISTS IN BLOB, tarBallName = null, do nothing
        if (tarBallName != null)
```

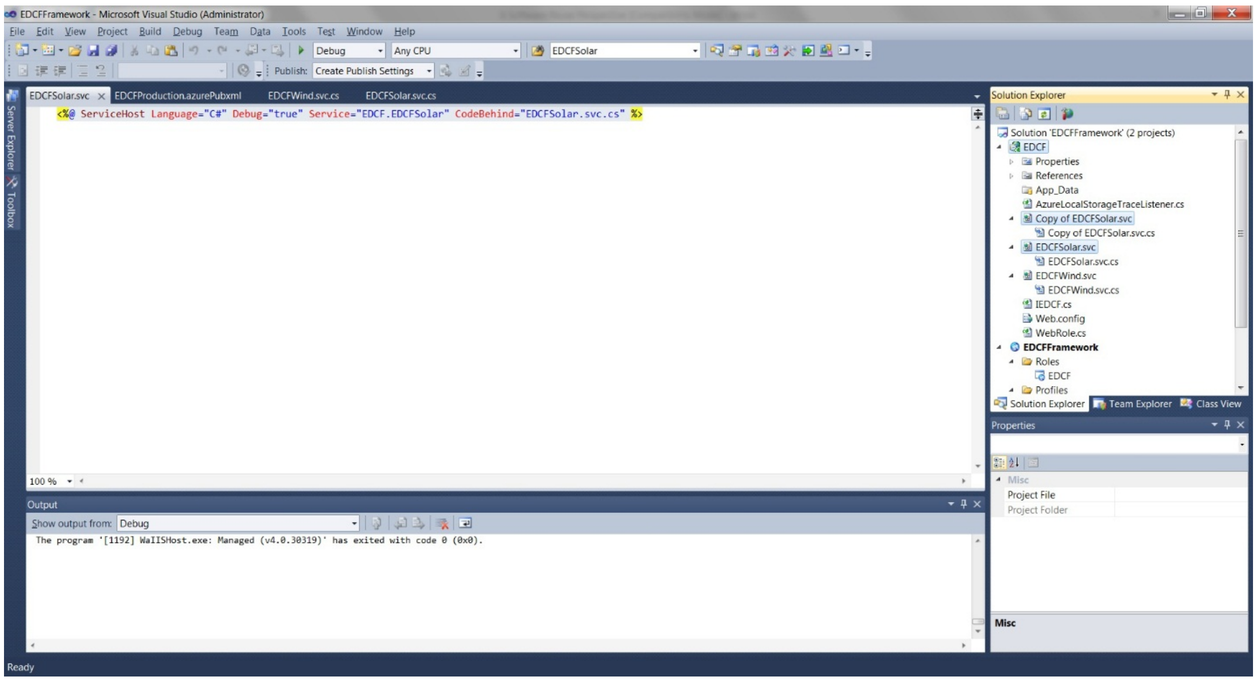
```
                {
                    //FORM THE NAME WHERE FILE WILL BE UNTARRED
                    var parts = blobFileURL.Split('/');
                    string foldername = parts[parts.Length - 1];
string untarredFolder = zipFileDirectory + foldername.Substring(0, foldername.IndexOf('.')) + "\\";

                    //UNTAR A SINGLE FILE FROM THE TAR FILE
                    string untaredFile = BasicOfficeOperations.UntarFile(tarBallName);

//IF THE DATA DOESN'T EXISTS FOR TIME BEGINNING 00 (I.E. AT MIDNIGHT)
                    if (untaredFile != null)
                    {
                        //DECODE FILE USING DEGRIB
                        Degribber decodeFile = new Degribber(zipFileDirectory);
                        string outputFilename = decodeFile.DegribFile(untarredFolder, untaredFile,
nextDate, latitude, longitude, zipFileDirectory);

                        //CLEAN UP
                        Directory.SetCurrentDirectory(localResource.RootPath);
                        Directory.Delete(untarredFolder, true);
                    }
                }

                //IF A TAR FILE WAS CREATED, WITH SIZE 0 BYTES, DELETE IT!
                File.Delete(zipFileDirectory + "9959_NDFD_YAN_" + nextDate + ".tar");

                //GET THE NEXT DAY'S DATE HERE.
                if (startDate.Equals(endData))
                    break;
                else if (lastDay == false)
                {
```

```
                nextDate = dateInFormat.NextDate();
                if (nextDate.Equals(endDateInFormat.Date))
                {
                    lastDay = true;
                }
            }
            else if (lastDay == true)
                break;
        }
        File.Delete(zipFileDirectory + "degrib.exe");

//ZIP ALL THE DECODED FILES TOGETHER
string zipFilename = BasicOfficeOperations.CreateZipFile(zipFileDirectory,
startDateInFormat.Date);
writeToBlob.WriteFileToBlob(zipFilenameInBlob, zipFilename);

//CLEAN UP
Directory.Delete(zipFileDirectory, true);
}
```

E.2. Extension of the FaaS for Electrical Load Forecasting

A developer can use the platform as a service to create new services by adding a service reference to the existing frameworks. This can be done a many ways and one such procedure is shown below:

1. **EDCF**: Open *EDCFFramework* with admin rights. Create a new file named *EDCFElectric*.svc. The project should now look similar to the one shown in fig. E.4.



```
<%@ ServiceHost Language="C#" Debug="true" Service="EDCF.EDCFSolar" CodeBehind="EDCFSolar.svc.cs" %>
```

```
The program '[1192] WaIISHost.exe: Managed (v4.0.30319)' has exited with code 0 (0x0).
```

Figure E.4. Shows a screenshot of VS 2010 project

Right click on file, click view marker, in the file that opens up and name the Service field to "EDCF.EDCFElectric" if it isn't already. The operation contracts in *IEDCF*.cs must be redefined, by using the following class derivation:

```
public class EDCFElectric : IEDCF, IEDCFServiceCost
{
   //DEFINE CONTRACTS HERE
 }
```

As EDCF uses metadata entry to download data, a source, context and entity must be added in *IOTools.Data* namespace, similar to the one for solar or wind with custom metadata information. If a new source is being added, another folder (namespace) must be added and these three files, *ElectricalLoadMetaDataEntry*.cs, *ElectricalLoadMetaDataContext*.cs and *ElectricalLoadMetaDataSource*.cs be created.

Within EDCF, there must be logic (that can be derived from Meta data table too) to generate the HTTP or FTP or XML link to download files from the Internet and use the respective service feature in order to download files from the Internet. EDCF must also generate a logical filename for link to blobs where the download files will be stored and update these links in the metadata table.

2. **IDRF**: Same procedure as in 1, must be used to create *IDRFElectric*.svc

The operation contract *CreateMetaDataTable* must be created and requested using a client (this service feature must not be accessible to a user, but only a developer of service). The metadata file from any new source must be copied to the blob manually and a service feature like *LocationDescription* can be used to read data from the Metadata file and extract information from it and store in a new metadata table for electric load.

As IDRF reads the data from download files in the blob and stores the standardized data in data table, it should update metadata entries to point to the data in Data table.

Depending on how data is to be stored in Data table, service features from *FoundationTools given in Appendix A and B*, like DFT, IDFT, MAPE, etc. can be reused to store the data in time or frequency domain.

If service features like unzip, reading excel files, etc. need to be used in order to read data from the blob, the features from *BasicOfficeOperations* can be used.

3. **FGF**: Same procedure as in 1, must be used to create FGFElectric.svc

Redefining the operation contracts in *IFGF* and implementing them in *FGFElectric* can extend the use of FGF extended to electrical load forecasting. After reading the standardized data (by IDRF) from tables, FGF should use service features to prepare daily, monthly and annual data in addition to the exceedance probability data. Then, FGF should use the service features from *IOTools* and *CommunicationTools* to generate a report in the blob and finally email the generated file to the user.

4. **FaaSController**: Create a new file name *FaaSControllerElectric*.svc. As the FaaS Controller is a composite service that is based on EDCF, IDRF and FGF, the new endpoints to these should be added in the file *FaaSControllerElectric*.svc as follows:

```
private static string edcfEndpointURLElectric =
                                        "http://127.0.0.1:81/EDCFElectric.svc?wsdl";
private static string fgfEndpointURLElectric =
                                       "http://127.0.0.3:81/FGFElectric.svc?wsdl";
private static string idrfEndpointURLElectric =
                                       "http://127.0.0.2:81/IDRFElectric.svc?wsdl";
```

In order to instantiate the correct instance of each service, these endpoints should also be added to as a service reference to the project. A client must be created to instantiate the correct instance and call the right operation contracts for electric load as follows:

```
        private static EDCFClient edcfClientElectric;
        private static EDCFServiceCostClient edcfCostClientElectric;

//TO GET COSTS
double cost = edcfCostClientElectric.DownloadDataToBlobCost(…, …);
```

```
//EDCF
edcfClientElectric.DownloadDataToBlob(…, …);

//IDRF
idrfClientElectric.StoreDataInTable(…, …);

//FGF
fgfClientElectric.ForecastPower(…, …, …, ….);
```

5. **WorkerRole**: The new endpoint of the FaaS Controller must be added to the worker role as follows

```
private static string faasControllerElectricURL =
              "http://faascontoller.cloudapp.net/FaaSControllerElectric.svc?wsdl";
```

This endpoint must also be added as a service reference in order for the services of the FaaS Controller to be consumed.

6. WebRole (if required): A new .aspx page has to be added or Default.aspx has to be modified in order to accommodate the new data source. The worker role will need modification to create a new workflow in order to forecast electric load. The worker role, must consume services of the FaaSController to use EDCF, IDRF, and FGF on its behalf. The worker role waits for the download to finish (EDCF), the data to be stored in data tables (IDRF) and then finally forecast (FGF) the data.

APPENDIX F – Costing Details

F.1. Cost of Service Features in FaaS

Service Feature Name	Total Man-Hour in Design & Development	Labor Cost	Total Indirect Cost	Total Cost	Expected Number of Users	Price of single usage
CreateWindMetaData	15	2250	1575	3825	2000	1.9125
CreateSolarMetaData	10	1500	1050	2550	2000	1.275
FoundationTools						
DFT	15	2250	1575	3825	2000	1.9125
FormShapeDescriptor	15	2250	1575	3825	2000	1.9125
Harmonics	20	3000	2100	5100	2000	2.55
IDFT	7.5	1125	787.5	1912.5	2000	0.95625
NumberOfHarmonicsNeeded	7.5	1125	787.5	1912.5	2000	0.95625
MinutelyDataToHourlyData	15	2250	1575	3825	2000	1.9125
MonthlyDataToAnnualData	6	900	630	1530	2000	0.765
HourlyDataToDailyData	10	1500	1050	2550	2000	1.275
DailyDataToMonthlyData	10	1500	1050	2550	2000	1.275
TransferTools						
FTPDataDownload	20	3000	2100	5100	2000	2.55
HTTPDataDownload	32	4800	3360	8160	2000	4.08
XMLDataDownload	10	1500	1050	2550	2000	1.275
CommunicationTools						
EmailServices	10	1500	1050	2550	2000	1.275
ForecastTools						
PowerForecast	20	3000	2100	5100	2000	2.55
ComputeKD	5	750	525	1275	2000	0.6375
ForecastGHIET	20	3000	2100	5100	2000	2.55
GC_Distance	5	750	525	1275	2000	0.6375
ReadForecastDataXML	10	1500	1050	2550	2000	1.275

IOTools

BasicOfficeOperations	30	4500	3150	7650	2000	3.825
BlobRead	7.5	1125	787.5	1912.5	2000	0.95625
BlobWrite	11.5	1725	1207.5	2932.5	2000	1.46625
TableRead	20	3000	2100	5100	2000	2.55
TableWrite	19	2850	1995	4845	2000	2.4225
TurbinePowerData	20	3000	2100	5100	2000	2.55
RawWindSpeedData	60	9000	6300	15300	2000	7.65
RawSolarData	76	11400	7980	19380	2000	9.69
Untar	20	3000	2100	5100	2000	2.55
Degribber	60	9000	6300	15300	2000	7.65

StatisticalTools

Averages	5	750	525	1275	2000	0.6375
CoefficientOfVariation	5	750	525	1275	2000	0.6375
CombineSigmas	5	750	525	1275	2000	0.6375
CreateDistribution	5	750	525	1275	2000	0.6375
MAPE	5	750	525	1275	2000	0.6375
StandardDeviation	5	750	525	1278.5	2000	1.2785
ExceedanceProbability	15	2250	1575	3825	2000	1.9125

SolarDataAnanlysis

MinutelySCIData	10	1500	1050	2550	2000	1.275

GeneralSupportTools

ReportPreparation	20	3000	2100	5100	2000	2.55
Coordinates	15	2250	1575	3825	2000	1.9125
DateTime	20	3000	2100	5100	2000	2.55
LocationDescription	5	750	525	1275	2000	0.6375

F.2. Cost of Infrastructure for FaaS

Bandwidth Cost

Cost per 6 GB per month	Cost per year
0.12	1.44

BlobStorage Cost

Total number of sites	File size for each year (MB)	Number of year of data available	Total storage size required for blob at full capacity (GB	Storage cost per month	Total cost for storing data in blob per year	Expected Number of Users	Charge user for storin 3 file for a year
40000	3	3	360	45	540	1000	1.62

TableStorage Cost

Total number of sites	Number of entities per year	Number of years of data available	Total number of entries for all locations	Transaction cost per day	Total cost per month assuming 1 user per day		
40000	365	3	43800000	1000	0.03		1.08

Compute Cost

Cost of 1 web and worker role per month	Number of roles required for all services	Total cost per month			Total cost for 1 year	Expected Number of Users	
14.4	7	100.8			1209.6	1000	3.6288

F.3. Assumptions For Cost Calculations

1. The cost of cloud infrastructure remains the same for the period of calculation (3 years).
2. The factors have a linear relationship with cost. This assumption states that the cost will vary linearly, i.e. if one of the factor increases, it has direct relation to the cost.
3. The size of files for each location is same. The cost of storing data is not much for our use, but the calculations assume that each file will be approximately the same size (3MB).
4. Number of entities created for each site is the same. Although table transactions do not have much cost attached to them, the calculations assume that each location with have data for 365 days and for three years for project prospecting.

5. The number of users per year will at least be 2000. These many users will use each service, so that a return on investment cost of development and infrastructure is repaid.

Glossary

Service-Orientation is a design paradigm intended for the creation of solution logic units that are individually shaped so that they can be collectively and repeatedly utilized in support of the realization of the specific strategic goals and benefits associated with SOA and service-oriented computing.

SOA is a distributed technology architectural model for service-oriented solutions with distinct characteristics in support of realizing service-orientation and the strategic goals of service-oriented computing. It is defined via a series of concrete characteristics that allow a given application to be comprised of multiple services that can be repeatedly composed and evolved in response to on-going business change.

Services: A service is a unit of logic to which service-orientation has been applied to a meaningful extent.

Service as Components: A component is a software program designed to be part of a distributed system. It provides a technical interface comparable to a traditional application-programming interface (API) through which it exposes public capabilities as methods, thereby allowing it to be explicitly invoked by other programs.

Services as Web Service: a Web service is a body of solution loic that provides a physically decoupled technical contract consisting of a WSDL definition and one or more XML schema definitions ans also possible WS-Policy expressions.

Services as REST services: A REST services architecture focuses on the resource as the key element of abstraction, with an emphasis on simplicity, scalability and usability.

Service Models: A service model is a classification used to indicate that a service belongs to one of several predefined types based on the nature of the logic it encapsulates, the reuse of potential of this logic, and how the service may relate to domains within its enterprise.

Service Composition is an aggregate of services collectively composed to automate a particular task or business process.

Service Contract is comprised of one or more published documents that express meta information about a service. The fundamental part of a service contract consists of the documents that express its technical interface.

XML is the definition language that can accompany information. It tells a computer program what that information actually is.

SOAP: A standard that uses XML to describe messages sent from one program to another. A program uses SOAP to request a service from another program and then pass it related data.

WSDL: A standard based on XML. Programmers use WSDL to create an XML document that describes a Web Service and how to access it.

REST: A simple interface for transferring data over HTTP without the added complications of a messaging layer such as SOAP or the use of session tracking via HTTP cookies.