Cloud Computing for Mission Design and Operations

Juan Arrieta[*] Robert Beswick[†] Dimitrios Gerasimatos[†]

Jet Propulsion Laboratory, California Institute of Technology

4800 Oak Grove Drive, Pasadena, California 91109 USA

The space mission design and operations community already recognizes the value of cloud computing and virtualization. However, natural and valid concerns, like security, privacy, up-time, and vendor lock-in, have prevented a more widespread and expedited adoption into official workflows. In the interest of alleviating these concerns, we propose a series of guidelines for internally deploying a resource-oriented hub of data and algorithms. These guidelines provide a roadmap for implementing an architecture inspired in the cloud computing model: associative, elastic, semantical, interconnected, and adaptive. The architecture can be summarized as *exposing data and algorithms as resource-oriented Web services, coordinated via messaging, and running on virtual machines*; it is simple, and based on widely adopted standards, protocols, and tools. The architecture may help reduce common sources of complexity intrinsic to data-driven, collaborative interactions and, most importantly, it may provide the means for teams and agencies to evaluate the cloud computing model in their specific context, with minimal infrastructure changes, and before committing to a specific cloud services provider.

I. Introduction

The NIST[a] has defined cloud computing as "*a model for enabling ubiquitous, convenient, on-demand access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction*[1]". More than a *new* technology, cloud computing is a *model* for using existing technology. The model is based on *abstraction*, meaning that it favors the *what* over the *how*; concepts over concrete devices (e.g., *storage* over *hard drive*, or *computing power* over *workstation*).

Abstraction is attained by coordinating the operation of different physical devices using software. The coordination is such that the total capacity of the underlying hardware can be offered as a continuum, behind a common interface, and at any level of granularization. A portion of this continuum represents a *virtual* service because such portion does not correspond to a physical device, even though it behaves as one. The process of using software to join together physical devices into a virtual continuum is called *virtualization.* In this sense, it is possible to offer an alternative, simpler definition for *the cloud*: it is simply *highly virtualized infrastructure*.[2]

Virtualization can increase datacenter throughput because it leverages the total capacity of the installed hardware. Whereas traditionally three users would use three workstations with an average load of 50%, those same three users could use two workstations joined as one, with an average load of 75%, and a third workstation standing by, ready to handle spikes in demand. Virtualization can also increase reliability, because it makes it possible to separate *information* from specific hardware. This separation makes the latter immediately replaceable with little or no downtime.

The space mission design and operations community already recognizes the value of the cloud computing model and virtualization in general.[3,4,5,6] In fact, some space missions have already included cloud computing components into their official operational workflow.[7] However, natural and valid concerns have prevented a more widespread and expedited adoption. These include vendor lock-in, security, privacy, online

[*] Navigation Engineer, Flight Path Control Section, M/S 230-205, AIAA Member.

[†] Senior Member of the Technical Staff, Navigation and Mission Design System Engineering Section, M/S 301-121, AIAA Senior Member.

[a] The National Institute for Standards and Technology is an agency of the United States Department of Commerce

copyright, liability, and policy enforcement.[8] As with any new technological trend, organizational inertia and user resistance also delay adoption.

Independently of the timetable for adopting cloud computing in a given team or agency, we believe that it is possible to immediately benefit from the model's philosophy, in a manner which does not conflict with internal policies or security restrictions, and with little or no infrastructure change.

In concrete, we propose to map current mission design and operation workflows into well-defined *resources* and *actions*, and expose the corresponding data and algorithms as resource-oriented Web services running on virtual servers. The proposed architecture may help alleviate common sources of complexity intrinsic to data-driven, collaborative interactions like:

- **Proliferation of Systems**
  It is natural that highly interdisciplinary activities rely on vastly different computing systems. For example, the supported operating system for the science team software may be different from the one running that of the operations team; different operating systems normally translate into different physical workstations, often in different networks.

- **Proliferation of Formats**
  As the number of teams and systems involved in a workflow increases, so does the number of data formats. This proliferation leads to the sprawling of ad-hoc data parsers, which often produce output in even more data formats.

- **Proliferation of Protocols**
  Different teams and systems produce or consume data at different workflow stages and levels of aggregation. The different interfacing requirements lead to the creation of different data exchange protocols: while one team lead may request a phone call once a product has been created, another may require a complex database transaction on a given server, or copying files to a common filesystem and notifying stakeholders via email.

- **Siloing of Datasets**
  Serial workflow activities often interface only at the end. Before the interfacing, datasets within a given serial component normally remain hidden from the other components. For example, an orbit determination team may leverage a trajectory covariance analysis to publish an uncertainty plot. In the long run, the team will have produced a rich covariance dataset, probably unknown to other teams.

In addition to what we perceive as immediate, day-to-day benefits in the design and operation of space missions, we believe that the proposed architecture may help teams and agencies evaluate the cloud computing model in their specific context, with minimal infrastructure changes, and before committing time or resources towards the adoption of a specific cloud services provider.

II. The Vision

At the most general level, there are three stages in the lifecycle of space missions: development, operations, and decommissioning. From a datacenter perspective, they are characterized as follows:

The *development* stage is centered around computer power and requirements management; it is a race between engineering design and regulatory compliance. Demand for computing services fluctuates from relatively slow periods to complex simulations that are fed from multiple data sources and generate vast amounts of scratch data.

The *operations* stage is characterized by real-time communication and the coordinated interaction of dozens of teams; its central concerns are timeliness and accountability, where decision-making is governed by well established workflows, centered around specific products. Mission data needs to be processed rapidly, and is released to a worldwide network of collaborators, or the general public.

The *decommissioning* stage is governed by data storage, classification, indexing, and distribution. It recognizes that the analysis of space mission data can span decades, and even the most unexpected data sources can enable or explain discoveries.[9] In fact, data-driven decision-making plays an increasingly important role, and a number of agencies are recognizing *Big Data* as an strategic asset.[10]

And to keep up with the increasing complexity of space missions, we believe that the next-generation datacenter will be based on three components of the cloud computing model:

- **Associative Elasticity**
  From a user perspective, there will be an unlimited amount of readily available computing services, billed only for actual usage (i.e., no charges for idle-time). Metering will involve such small unit increments (like computing power in core-minutes, or storage in megabyte-minutes) that it will be practically continuous.

  The provisioning and pricing model will be *associative*.[11] For example, procuring 1,000 computing cores for one hour will involve the same effort and expense as procuring one computing core for 1,000 hours; storing one terabyte of data for one minute will be no different than storing two megabytes for one year.

- **Semantical Hyperdata**
  Data will be integrated with its metadata, links connecting it to other data or products, and information about the nature of such connection; access control will be granular to any extent.

  A given dataset will be available in different representations: while one user may request a plain-text tabulation of a given thrust profile, with time in seconds from t_0 and force in Newtons, another may request a plot of the same data, with horizontal ticks in minutes, vertical ticks in tenths of one Newton, and a footnote indicating the source telemetry product.

  The semantic relationship of datasets, and their availability in any desired format will compound the information content; in the previous example, either user could have requested a weather report for the geographical area where telemetry was received.

- **Living Workflows**
  When policy or preference dictate that a given activity should evolve in accordance to a given workflow, it will be possible to monitor the real-time status at any level of aggregation. For example, while a maneuver analyst may be interested in receiving immediate notification when the latest orbit determination is available, a mission manager may wish to concentrate on the overall go/no-go decision on a given activity.

  Notifications will follow a *push* mechanism[b], where well-defined triggers issue relevant status notifications in real time; it will contrast with the more common *poll* mechanism, where users repeatedly inquire for status updates.

  It will be possible to incorporate or remove arbitrary *units of work* from the workflow, in real time, and without system downtime.

  While not strictly necessary for the design of adaptive, automated workflows, cloud computing does provide a practical abstraction layer which makes the units of work easier to identify, connect/disconnect, and monitor.

III. REST and the Resource-Oriented Architecture

The proposed architecture is based on a more general architectural style for distributed hypermedia systems called Representational State Transfer (REST), which provides a set of constraints that, when applied as a whole, emphasize scalability of component interactions, generality of interfaces, independent deployment of components, and intermediary components to reduce interaction latency, enforce security, and encapsulate legacy systems.[12] There are five fundamental properties of a REST architecture:

- **Client-Server**
  The architecture provides a separation of concerns, where data and algorithms are provided and maintained in the server, and offered to clients over a network connection.

- **Stateless**
  Each request contains all of the information necessary for a connector to understand the request, independent of any requests that may have preceded it. Whether clients maintain state across requests is immaterial to the server.

[b] Known in computer programming as the Hollywood Principle: *don't call us, we'll call you.*

- **Cacheable**
  Responses can be stored in a ready-to-deliver manner, and be transmitted to clients without further processing. Cached responses can dramatically reduce server workload and response latency.

- **Layered**
  A client cannot distinguish whether it is connected directly to the master, or to an intermediary along the way. Layering can be used for load-balancing (a load-balancer receives the request, and routes it to the least loaded node) and request proxying, where a secondary server responds to specific requests, thus providing a layer between clients and the master server.

- **Uniform**
  Interaction with resources follows the same protocol, regardless of the underlying data. In REST, this protocol is strictly HTTP.

The key abstraction of information in REST is a *resource*; any information that can be named can be a resource: a document, image, dataset, or temporal service (e.g., the current position of a spacecraft).

Resources are identified by their Uniform Resource Locator, or URL,[13] a string which provides an abstract identification of the resource location. Guidelines have been established for the generic syntax of such strings,[14] and best practices for implementing such guidelines have been identified,[15, 16] including:

- use forward slash (`/`) to indicate a hierarchical relationship
- use comma (`,`) and semicolon (`;`) to indicate a non-hierarchical relationship
- use hyphen (`-`) and underscore (`_`) to improve readability
- use ampersand (`&`) to separate parameters

For example, the URL:

```
http://example.com/ephemeris/position/saturn;sun/
```

could denote a resource which contains the current EME-2000 position of Saturn with respect to the Sun. Notice that `position` is more specific than `ephemeris`, and `ephemeris/position` indicates this relationship. On the other hand, the semicolon between `saturn` and `sun` denotes that the relationship between body and center is non-hierarchical. Additional parameters could be provided in the URL to specify times or frames:

```
[...]/position/saturn;sun/parameters?utc-epoch=2012-121T00:00:00&frame=emo2000
```

where the dashes help improve readability and the parameters are clearly separated by ampersand.

Somewhat paradoxically, identifying resources and endowing them with appropriate URLs is simultaneously the simplest and most complex part of the proposed architecture. It is the simplest because it requires only a piece of paper upon which to write down the resources and their proposed URLs. And it is the most complex because identifying resources, their hierarchy, and parameterization can be a daunting task. The following guidelines can be helpful during the resource identification process:

- **Map CRUD operations to resources**
  Any unit of information subject to a CRUD operation (Create, Read, Update, Delete) is a candidate resource. As in object-oriented programming, a practical way of identifying resources is to enunciate the workflow verbally. The grammatical *objects* (what or whom the verbs are acting upon) tend to correspond to resources.

- **Separate data from views and algorithms**
  It is common that different teams interact with the same data in different ways. For example, one team may require a second-by-second tabulation of propellant mass flowrate, while another may simply require the average. In this case the tabulation corresponds to a *view* and the average corresponds to an *algorithm* on the *mass flowrate* resource.

- **Choose grouping and granularity based on the underlying workflow**
  If the underlying workflow is mostly separated by subsystem, it makes sense to reflect such separation in the resource grouping (`/power/`, `/propulsion/`, `/navigation/`, etc.) The workflow action demanding the most granularity will determine the granularity of the underlying resource.

The core idea behind REST is to leverage HTTP and work *in* it as opposed to *around* it. By design, HTTP provides a rich set of mechanisms to interact with well-defined resources, and REST provides the architectural means to offer data and services precisely as resources. An HTTP *request* is made by a client to a server, and is composed of four major parts:

- **Method**
  The HTTP method is a *verb* which indicates *what* interaction is desired with a resource. The five most important ones are: `GET` (read a resource), `POST` (create a resource), `PUT` (update a resource), `DELETE` (delete a resource), and `HEAD` (obtain resource metadata).
- **URL**
  The URL specifies the resource on which to apply the method. As explained above, the URL uniquely identifies a resource.
- **Headers**
  Any additional information required to uniquely refine the request can be provided via headers: a collection of key, value pairs provided in the request.
- **Body**
  The contents of the request, like an image or a JSON document.

An HTTP *response* is returned from a server to a client, and is almost identical to a request. It contains a response **status code** which indicates the outcome of the request, **headers** providing resource metadata, and **body** containing the resource representation. The following is a complete HTTP request/response interaction: The client's request:

```
GET /ephemeris/position/saturn;sun/ HTTP/1.1
Host: example.com
Accept: text/csv
```

The server's response:

```
HTTP/1.1 200 OK
Date: Mon, 30 Apr 2012 00:00:00 GMT
Last-Modified: Fri, 27 Apr 2012 05:43:21 GMT
Content-Length: 15000
Content-Type: text/csv

2456047.50, -1.301772228641044E+09, -6.195569431611598E+08, -1.998658771522971E+08
2456047.54, -1.301758617744085E+09, -6.195855778417170E+08, -1.998782903024700E+08
[and more entries; 15,000 bytes of data]
```

In this interaction, the client requests the current heliocentric position of Saturn as a comma-separated plain text file (`Accept` headers). The server successfully processes the request (status code `200 OK`), indicates the date when the request was received (`Date`), the last time the underlying resource was modified (`Last-Modified`), the content type provided (`Content-Type`), the amount of data provided (`Content-Length`), and the actual data.

Because HTTP is standardized, widely supported, and designed around the concept of *resources*, it can be used effectively to map data-driven workflows into a computer system. In the remainder of this paper we will provide specific architectural elements which can help bring current infrastructure closer to this vision, and propose a tiered approach to adoption and implementation based on specific recommendations.

IV. Architecture Components

In line with the cloud philosophy, the proposed architecture is simple, and based on widely adopted standards, protocols, and tools. It can be summarized as follows: *expose data and algorithms as resource-oriented Web services, coordinated via messaging, and running on virtual machines.* The different design and operation workflows are then mapped to actions on these Web resources. The building blocks of the architecture are described below.

A. Virtual Machine

A virtual machine is a unit of a physical server that has been divided into multiple virtual servers, controlled by software. Each owns a share of the CPU and other physical resources and is supervised by a shared *hypervisor*, which manages calls for hardware services and resolves conflicts. Figure 1 depicts an example where one physical workstation is running three different operating systems simultaneously, and has capacity for running a fourth one. In addition to taking advantage of every computing cycle available, deploying virtual environments may enable the permanent support of legacy applications which function appropriately only on deprecated operating systems. Instead of incurring the difficulty, risk, or expense of porting the legacy application to a new operating system, it could be deployed in the appropriate virtual environment and supplied with appropriate hardware resources. The underlying host machine—which would also host other operating systems and applications—could be upgraded or replaced without disrupting the availability of the legacy application. Live migration—unthinkable less than ten years ago—is possible because the hypervisor can determine the part of an application that the virtual machine is about to use, and leave that part running on the existing server, while moving all the other parts and data to a different physical server. Upon completing the transfer, the hypervisor suspends the original virtual machine, and transfers the remaining application code to the new location and resumes operation. Lasting a fraction of a second, the whole process is imperceptible to most applications.[2]

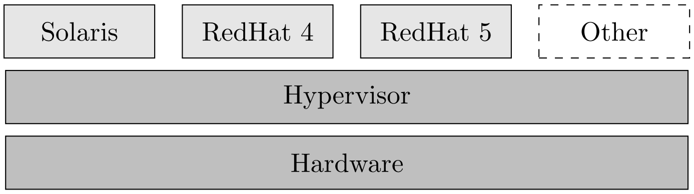


Figure 1. A hypervisor is a layer of software running directly on computer hardware. By replacing the operating system, it allows the computer hardware to run multiple *guest* operating systems concurrently.

A wide range of providers offer reliable hypervisors. In particular, the Xen hypervisor (`http://xen.org`) has been widely adopted by major cloud providers, like Amazon EC2 and Rackspace. Converting a workstation to a virtual host is a straightforward process, and recent versions of popular operating systems contain hypervisors in their standard distribution.

B. HTTP Server

The Hypertext Transfer Protocol (HTTP) is an application-level protocol for distributed, collaborative, hypermedia information systems;[17] it drives Internet communication. HTTP server technology is well known, and reliable products are available that are normally included in the standard distribution of popular operating systems. Apache (`http://httpd.apache.org`) and nginx (`http://nginx.org`), which together power almost 75% of the Internet hosts, offer ready-to-deploy configurations for a vast array of Web service configurations.

For internal applications, it is possible to deploy the HTTP server in the Local Area Network, behind the institutional firewalls, and leverage existing user authentication services (e.g., LDAP).

C. Database

While any organized collection of data can be described as a database, we use the term to refer specifically to a data management system. Traditional solutions include relational databases, like MySQL (`http://mysql.com`) and SQLite (`http://sqlite.org`), which provide strict ACID[c] compliance. These table-oriented databases rely on predefined, strongly typed *schemas*, and are thus suited for strongly typed applications, where data structure is constant or changes slowly, and is known a priori.

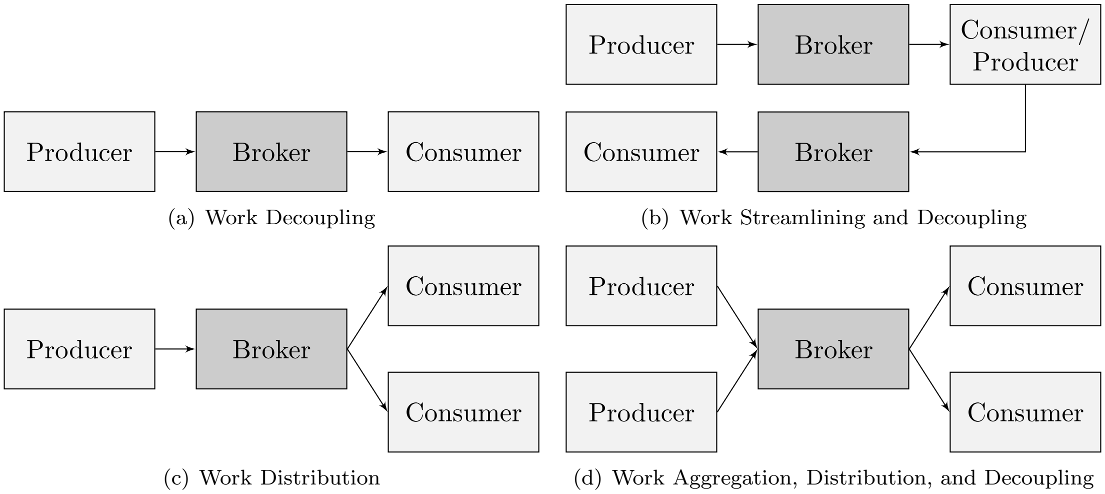
Recently there has been a surge in the development of non-relational, schema-free, document-oriented databases, like MongoDB (`http://mongodb.org`) and CouchDB (`http://couchdb.apache.org`). These are normally suited for applications with dynamic typing, where structure evolves constantly, and scalability

[c] Atomicity, Consistency, Isolation, and Durability are a set of properties which guarantee that database operations are *transactional*: a logical operation either succeeds in full, or is not executed at all.

and fast read/write operations are critical. Instead of full ACID compliance, they offer *eventual* consistency, and atomicity only at the *document* level (what would be a *row* in a table-oriented database).

D. Message Broker

A message broker is middleware which routes information in an orderly, predictable manner. It provides a common layer that insulates the user or application developer from the details of the operating system or software implementation. This layer is useful for reducing dependencies and complexity among heterogeneous applications. The broker normally runs on a dedicated machine, and stands ready to route messages among applications, which are producers or receivers of messages (or both). As presented in Figure 2, this separation enables multiple workflow patterns.



(a) Work Decoupling

(b) Work Streamlining and Decoupling

(c) Work Distribution

(d) Work Aggregation, Distribution, and Decoupling

Figure 2. Different message broker workflow arrangements.

A message broker also serves as the central component of a scalable push-based notification and automation system; it enables individual applications to issue messages based on arbitrary criteria (like reaching a processing milestone). The message is broadcasted to any listening application, which in turn can notify users, log the event, start postprocessing, or execute any action with the incoming message (cf. Figure 3). Applications do not need to be aware of their data producers or consumers.

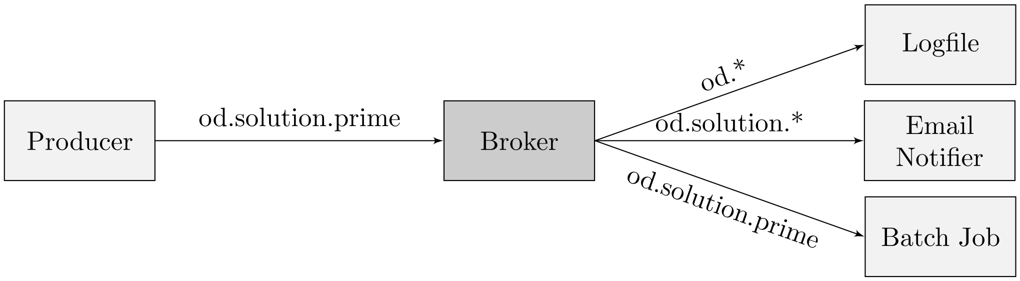


Figure 3. In this sample message broadcasting using a broker, the application (Producer) finishes producing an orbit determination solution for a prime maneuver window; it issues a message with all relevant information to the exchange `od.solution.prime`. An application listening on the `od.*` channel will log any Orbit Determination event. An application listening specifically for Orbit Determination Solutions is listening on `od.solution.*` and will send an email when a solution is ready. Finally, an application which listens specifically for Orbit Determination solutions for the prime maneuver window will trigger a batch job.

Enterprise-grade messaging systems based on the standard *Advanced Message Queuing Protocol*, or AMQP, are widely available via open-source projects, like RabbitMQ (`http://rabbitmq.com`) and Apache Qpid (`http://qpid.apache.org`). In addition, development is underway for implementations based on HTTP (`http://restms.org`).

E. Data Serialization

Space mission design and operations rely on a variety of data formats. While it is not pragmatical or effective to rewrite all tools to natively support *one format*, it is indeed possible to provide facades to the heterogeneous data in a variety of common serialization formats. For example, suppose that the output of a legacy application is provided in the following plain-text format:

```
EPOCH: 18-Dec-2012 00:12:34.567891 UTC
FRAME: EME2000
POSITION: 1.234567891D+06 -7.6543210D+06 3.4567891D+02
```

It is likely that every team which uses such legacy application will write a custom data parser to extract the relevant information. And even this simple example presents many challenges: most modern languages will not recognize `D` as the token indicating exponent[d] (most recognize only `E` or `e`). The output also contains a date string, and parsing dates is a well known source of bugs in user scripts. Most users will likely rely on text substitutions and regular expressions; one change to the legacy application output format could result in breaking the workflow of dozens of teams *in different ways*.

On the other hand, a single team could be responsible for producing and maintaining a utility capable of providing the same information, but in a homogeneous data serialization format. For example, the previous legacy output could be serialized as a JSON (`http://json.org`) document:

```
{
 "epoch":{
   "day":18, "month":12, "year":2012,
   "hour":0, "minute":12, "second":34.567891,
   "iso":"2012-12-18T00:12:34.567891",
   "clock":"utc"
   },
 "frame":"eme2000",
 "position":[1.234567891e6, -7.6543210e6, 3.4567891e2]
}
```

Libraries for parsing JSON documents are available for all major programming language, and obtaining the data becomes a simple statement. In Python, for example (assuming the document is called `data.json`):

```
import json # this module is part of the Python Standard Library (version >= 2.6)
with open("data.json", "rb") as fp:
    data = json.load(fp)
# data is now available as a native data type (in this case, a Python dictionary)
print(data["position"])
[1234567.8910000001, -7654321.0, 345.67890999999997]
from datetime import datetime # Standard Python Library
ISO_FORMAT = "%Y-%m-%dT%H:%M:%S.%f" # timestamp format according to ISO 8601
print(datetime.strptime(data["epoch"]["iso], ISO_FORMAT))
2012-12-18 00:12:34.567891
```

Other formats, like XML (`http://w3.org/XML`) and YAML (`http://yaml.org`), can be used for plain-text serialization. Binary serialization formats are also available that provide all benefits of plain-text serialization (except being *human readable*) and can dramatically enhance performance. BSON (`http://bsonspec.org`)—the binary counterpart to JSON—has been adopted as the underlying format for MongoDB, and has been used successfully in other data-intensive applications.[18]

F. Web Applications

A Web application is a computer program which can respond to HTTP requests. There are dozens of frameworks for their development, like those based on popular dynamic languages like Python (`http://python.org`),

[d] Common in the output of Fortran programs.

Ruby (http://ruby-lang.org), and PHP (http://php.net), which have been favored by the major Internet players.

The Python programming language offers a simple and universal interface between Web servers and Web applications (the Web Server Gateway Interface, or WSGI[19]); Apache and nginx support the interface specification. As for other popular programming languages, powerful and widely available libraries exist that can connect Python applications to a variety of databases and message brokers. The Python source code for the canonical WSGI application is as simple as follows:

```
def application(environ, start_response):
    status = "200 OK"
    headers = [("Content-type", "text/plain")]
    start_response(status, headers)
    return [u"Hello, World!\n"]
```

G. Web Clients

A Web client is a computer program which can interact with a Web application. The most widely deployed (and most familiar) client is the Web browser, which generally retrieves Web pages for visual navigation. However, a wide variety of clients can be easily built with standard libraries that can interact with Web services in a variety of formats. For example, if a JSON data source were available at http://example.com/data/, a command line application capable of retrieving it could be as simple as:

```
import httplib # Python Standard Library
import json    # Python Standard Library
connection = httplib.HTTPConnection("example.com") # open server connection
headers = {"Accept":"application/json"} # specify JSON as the desired format
connection.request("GET", "/data/", headers=headers)
data = json.loads(connection.get_response().read())
# do something with ‘‘data‘‘...
```

V. Tiered Path to Implementation

We outline the implementation of the proposed architecture following a series of guidelines:

1. Adopt a common data-interchange format

Shared, annotated data and methods offer new discovery opportunities and prevent unnecessary repetition of work.[20] In line with this view, the main concern in the architecture is to make data usable. Whether data comes from a legacy application with frozen development, or as the output of the most recently engineered software tool, it should be provided in a common format. And providing homogeneous formats does not necessarily imply modifying the underlying application; simple parsers and data wrappers built for legacy applications can often provide all required functionality.

JSON is particularly well suited for structured datasets with heterogeneous types. It is simpler, more flexible, and more compact than XML, yet it retains human readability and a rich offering of reliable libraries for all major languages.

2. Evolve raw data to semantical hyperdata

Upon adopting a common data format, it becomes possible to incorporate a common meta-information field. Common meta-information includes authorship (user, date, time, software version), data sources (telemetry product, planetary constants), intended purpose (mission, instrument calibration, maneuver implementation), expiration (time span), and cross-check tags (whether a product has been reviewed by key stakeholders).

Linked data compounds information content, and simplifies cross-checking critical products. For example, upon detecting a bug, it becomes simpler to recall all products made with the faulty software version. In practice, JSON data can be annotated with a _meta key.

3. Decompose workflows into simple actions

While space missions tend to evolve according to strictly defined workflows, it is nevertheless possible to further divide activities into simple actions. For example, a unit of work which receives an orbit determination and produces a maneuver design can be further divided into *retrieving* the orbit determination, *running* a trajectory optimizer, *producing* design data, *integrating* data into specific products, and *delivering* the maneuver design.

Simpler actions tend to map naturally as resources, and enable software to emit alerts upon start or completion in a more granular level. This simplification enables the real-time status monitoring, and simplifies root cause analysis in the event of workflow anomalies.

4. Categorize data and algorithms as resources

As explained in Section III, the identification of resources can be simplified by mapping as resources those entities which undergo CRUD operations. Algorithms and workflow stages should also be mapped as resources. Adhering to the URL specification[13] and following best practices[15] can simplify the development of clients and servers, and provide a solid foundation upon which to build complex workflows oriented to specific resources with transparency.

5. Provide a common interface to resources via HTTP

The interaction between users and data should be available via a unified interface, capable of receiving simple requests from generic clients. HTTP accommodates this mechanism via the standard verbs, which map to operations on resources as follows:

Verb	**Operation**
`HEAD`	read the metadata provided in a resource's headers
`GET`	read a resource in a specified representation
`POST`	create a resource by providing a specific representation
`PUT`	update a resource in whole or in part
`DELETE`	delete a resource

6. Coordinate the system interaction via messaging

The separation of concerns provided by message brokers constitutes a flexible model for growth, adaptation, and reliability. The basic unit of work then becomes:

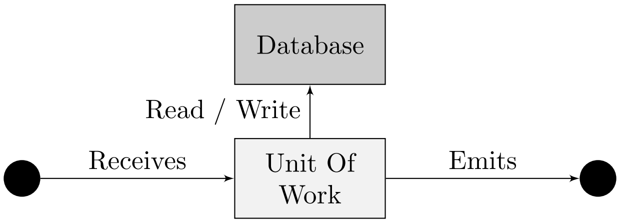


Figure 4. The unit-of-work in the architecture

The model allows for arbitrary units of work to be added or removed from the workflow. It also provides the means for monitoring in real time the evolution of activities with push notifications.

7. Deploy worker and data nodes and applications in virtual machines

Virtual machines provide the foundation to an elastic system, capable of supporting multiple operating systems, and configurable in real time. And this benefit can be captured without the need of multiple workstations.

A typical approach to virtualization is to deploy a fresh install of the desired operating system as a virtual machine, install only the required components to provide a well-defined functionality, and save and store the resulting *image*.

Creating and maintaining an inventory of images provides unprecedented advantages, because it enables teams and agencies to deploy an arbitrary number of working nodes delivering a specific functionality both in the internal datacenter and in the infrastructure of commercial cloud services providers.

VI. Conclusions

In the interest of expediting the adoption of the cloud computing model within the space mission design and operations community, we propose an architecture based on widely adopted standards, protocols, and tools. We describe specific implementations of tools that can provide specific functionalities, and have been field-tested by the Internet and cloud computing communities at large.

We propose a path to implementation based on a set of specific guidelines which can be applied incrementally, with little or no infrastructure changes, and converge toward a vision of elasticity, homogeneity, and workflow flexibility that can provide access to living systems interchanging highly semantical data.

We believe that the proposed architecture can enable current teams and agencies to evaluate the cloud computing model in their specific context and, independently of their timetable for eventual adoption or selection of a cloud provider, these teams and agencies may capture immediate benefits.

VII. Acknowledgments

The research was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement by the United States Government or the Jet Propulsion Laboratory, California Institute of Technology.

References

[1] Mell, P. and Grance, T., "The NIST definition of cloud computing," National Institute of Standards and Technology Special Publication 800-145, Sep 2011.

[2] Babcock, C., *Management strategies for the cloud revolution*, McGraw-Hill, 2010.

[3] Lankford, K., Pruitt, R., Pitts, R., and Felton, L., "Utilization of virtual server technology in mission operations," *SpaceOps 2010 Conference*, 2010.

[4] Prieto, J., Feiteirinha, J., Bizarro, P., Gómez, E., and Pecchioli, M., "Use of virtualisation techniques for ground data systems," *SpaceOps 2008 Conference*, 2008.

[5] Schmidhuber, M., Kretschel, U., and Singer, T., "Virtualizing monitoring and control systems: first operational experience and future applications," *SpaceOps 2010 Conference*, 2010.

[6] Zimmer, T., "Modular application design for heterogeneous operational environments," *SpaceOps 2010 Conference*, 2010.

[7] Joswig, J. C. and Shams, K. S., "Redefining tactical operations for MER using cloud computing," *Proceedings of the 2011 IEEE Aerospace Conference*, AERO '11, IEEE Computer Society, Washington, DC, USA, 2011, pp. 1–7.

[8] Nelson, M. R., "Building an open cloud," *Science*, Vol. 324, No. 5935, 2009, pp. 1656–1657.

[9] Turyshev, S. G., Toth, V. T., Ellis, J., and Markwardt, C. B., "Support for temporally varying behavior of the Pioneer Anomaly from the extended Pioneer 10 and 11 Doppler data sets," *Phys. Rev. Lett.*, Vol. 107, Aug 2011, pp. 081103.

[10] Mervis, J., "Agencies rally to tackle Big Data," *Science*, Vol. 336, No. 6077, 2012, pp. 22.

[11] Fox, A., "Cloud computing: what's in it for me as a scientist?" *Science*, Vol. 331, No. 6016, 2011, pp. 406–407.

[12] Fielding, R. T., *Architectural styles and the design of network-based software architectures*, Ph.D. thesis, University of California, Irvine, 2000.

[13] Berners-Lee, T., Masinter, L., and McCahill, M., "Uniform resource locators (URL)," RFC 1738, Dec. 1994.

[14] Berners-Lee, T., Fielding, R., and Masinter, L., "Uniform resource identifiers (URI): generic syntax," RFC 2396, Aug. 1998.

[15] Allamaraju, S., *RESTfull Web services cookbook*, O'Reilly Media, 2010.

[16] Richardson, L. and Ruby, S., *RESTful Web services*, O'Reilly Media, 2007.

[17] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and Berners-Lee, T., "Hypertext transfer protocol – HTTP/1.1," RFC 2616, June 1999.

[18] Morrow, A., "Low latency event logging with BSON," Video from the MongoDB Silicon Valley Conference, 2010.

[19] Eby, P. J., "Python web server gateway interface v1.0.1," Python Enhancement Proposal No. 3333, Sep 2010.

[20] Sansone, S.-A. et al., "Toward interoperable bioscience data," *Nature Genetics*, Vol. 44, No. 2, 2012, pp. 121–126.

VIII. List of Abbreviations

ACID	Atomicity, Consistency, Isolation, and Durability
AMQP	Advanced Message Queuing Protocol
BSON	Binary JSON
CRUD	Create, Read, Update, and Delete
EC2	Elastic Compute Cloud
HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation
LDAP	Lightweight Directory Access Protocol
REST	Representational State Transfer
URL	Uniform Resource Locator
WSGI	Web Server Gateway Interface
XML	Extensible Markup Language
YAML	YAML Ain't Markup Language