Brown Dog: Leveraging Everything Towards Autocuration

Smruti Padhy, Greg Jansen, Jay Alameda, Edgar Black, Liana Diesendruck, Mike Dietze, Praveen Kumar, Rob Kooper, Jong Lee, Rui Liu, Richard Marciano, Luigi Marini, Dave Mattson, Barbara Minsker, Chris Navarro, Marcus Slavenas, William Sullivan, Jason Votava, Kenton McHenry
National Center for Supercomputing Applications
University of Illinois at Urbana-Champaign
Email: {spadhy, mchenry}@illinois.edu

***Abstract*—We present Brown Dog, two highly extensible services that aim to leverage any existing pieces of code, libraries, services, or standalone software (past or present) towards providing users with a simple to use and programmable means of automated aid in the curation and indexing of distributed collections of uncurated and/or unstructured data. Data collections such as these encompassing large varieties of data, in addition to large amounts of data, pose a significant challenge within modern day "Big Data" efforts. The two services, the Data Access Proxy (DAP) and the Data Tilling Service (DTS), focusing on format conversions and content based analysis/extraction respectively, wrap relevant conversion and extraction operations within arbitrary software, manages their deployment in an elastic manner, and manages job execution from behind a deliberately compact REST API. We describe both the motivation and need/scientific drivers for such services, the constituent components that allow for arbitrary software/code to be used and managed, and lastly an evaluation of the systems capabilities and scalability.**

***Index Terms*—digital preservation, unstructured data, web services**

I. Introduction

Over the past decades we have seen exponential growth in the amount of digital data [1] with the growth only increasing as it continues to become cheaper and easier to create data digitally. This continuing shift away from physical/analogue representations of information to digital forms has created a number of social, policy, and practical problems that must be addressed in order to ensure the availability of these digital assets [2, 3, 4]. One aspect of these problems are that of the storage, movement, and computation on large datasets, what most think of when one hears the term Big Data, i.e. problems involving large quantities of data. A significant amount of research and development has gone into addressing these issues from computation [5, 6, 7], to data repositories and replication [8, 9], data transfer [10, 11], visualization [12, 13], to commercial as well as academic [14] storage solutions.

Another aspect involves that of indexing and finding data as well as accessing the contents of data long term, a problem involving large amounts of data but further hindered by problems involving large varieties of data. This latter problem is a significant issue for several reasons including the rapid evolution of technology, relatively short lifespans of software, commercial interests, and the ease and reward towards creating data versus curating data. As digital software and digital data have become key elements in just about every domain of science the preservability of data has become a major concern within the scientific community with regards to ensuring the reproducibility of results. This has become a particular concern for what is often referred to as the "Long-Tail" of science, spanning the vast majority of grants involving one or more graduate students and little funds for a significant data management effort (most especially post-award) [15]. Research and development addressing this second aspect has focused on preserving the execution provenance trail [16, 17], building repositories for scientific code/tools [18, 19], developing user friendly content management systems [20, 21, 22], dealing with format conversions and information loss [23, 24, 25, 26], building test suites [27], as well as efforts within the artificial intelligence and machine learning communities such as computer vision [28, 29, 30] and natural language processing [31, 32].

We focus on two of the lower level problems involved with this latter category, a lack of appropriate metadata describing the contents of files, needed to find information of interest within large collections of data, and the lack of format specifications describing how the data is laid out within a file so that one can get at its contents (e.g. 3D/depth data, pixels, text, waveforms, etc.) independent of how it is represented on the storage medium/file system. With regards to each of these there are a number of efforts, tools, and frameworks that have been built to help users curate their own data [8, 33] and access file contents or convert to a format that can then be accessed[1,2,3]. More accurately subsets of functionality towards this exists across a wide variety of software. For example with regards to file formats, conversion capabilities exists across a heterogeneous set of libraries and software (from command line tools such as the popular ImageMagick to the GUI driven software that we use every day). With regards to metadata a similar argument can be made if we for the moment relax the typical use of the term to be solely that of data describing data, data useful for searching/indexing collections of data

[1] http://www.opendocumentformat.org/
[2] https://cloudconvert.com/
[3] https://www.ps2pdf.com/

and its contents. In this context a wide variety of tools exist that take data and analyze it for some higher level piece of derived information that is then produced (e.g. machine learning classifiers, models of all kinds, statistical software, and actual metadata extractors).

Needs for such tools permeate the day to day workflows of just about everyone. Use cases span ecology, biology, civil environmental engineering, hydrology, oceanography, material science, library & information science, social science, and so forth to including the public at large. For example many communities utilize a wide variety of models to predict/simulate events, e.g. plant growth at various areas over different time frames. These models typically support data in unique formats. Efforts such as PEcAn [34], one of our use cases, aims to make it easier to connect data sources to models by providing these conversion capabilities for an extensible number of data sources (e.g. Ameriflux[4], NARR). Similarly with navigating large collections of unstructured data. For example in biology automatically classifying microscopy images of fossilized pollen [35] (in addition to converting out of proprietary microscopy formats), in entymology the tracking of bees within a colony, identifying plant phenology [36], deriving data from digitized handwritten documents, in medicine classifying/tracking cells in microscopy images, extracting data from publications (i.e. tables, figures) as no other source of the data may be available, extracting data from spreadsheets of all kinds (with different internal layouts and naming conventions), labelling land coverage in satellite and Lidar data, in material science identifying failed fabrication experiments in SEM data, finding and tracking people in social science experiments [37], and so on. This applies to the public at large as well with the format conversion needs we deal with regularly (e.g. between document formats, image formats, video formats, etc.) and indexing needs such as finding desired files in our photo/video collections, or smarter ways of searching a folder of documents (e.g. via NLP techniques, etc). Tools to help with portions of these are everywhere. We aim to leverage all of them whether they exists as libraries, command line tools, GUI applications, or web services, and make these capabilities as trivial to users/applications as possible while simultaneously combining their abilities, and perhaps preserving these tools at the same time.

In the sections below we outline the Data Access Proxy (DAP) for file format conversions and the Data Tilling Service (DTS) for metadata extraction which make up the Brown Dog[5] effort. Both exist as services/frameworks that aim to make it as easy as possible to incorporate arbitrary conversion and extraction capabilities from 3rd party software and services, connect them to obtain the union of their capabilities, scale them dynamically to meet the demands of the service, and manage them towards a robust service. Both the DAP and DTS are modeled so as to fit a role within the internet analogous to a DNS service in terms of setup and usage by other applications.

[4]http://ameriflux.lbl.gov/

[5]Playing on the notion of a mutt.

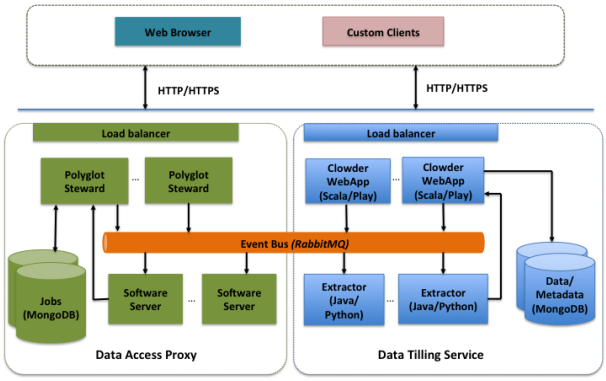

Figure 1. Architecture of the Brown Dog services. Both the DAP and DTS exist as web services behind a load balancer coordinating between a potentially distributed number of Clowder or Polyglot instances respectively. Each Clowder or Polyglot instance in turn manages a number of distributed extractors and/or Software Servers respectively which handle elements of a job.

In the sections below we describe the architecture of the two services, how they interact with arbitrary software, their scalability and extensibility, an evaluation of the two services, and a number of prototype client applications.

II. Architecture

The DAP built off of the Polyglot framework [23, 38] and the DTS built on top of the Clowder framework [22, 33] are architected to manage a number of heterogeneous tools, specifically conversion and extraction tools respectively, distributed across the web and provide access to the union of their capabilities via a fairly compact REST interface (Figure 1). These tools, essentially black boxes of code/functionality, are tracked and managed by the head node services and elastically grown/shrunk to accommodate user demands. Both emphasize extensibility in the sense of allowing new converters/extractors to be added and deployed across a DAP/DTS instance as trivially as possible. A typical workflow might involve calling the DTS to index and find relevant data according to some criteria within a collection of data and using the DAP to convert the files in that collection to a format that can be processed.

A. Data Tilling Service

Clowder is an open source web based content management system which allows users to upload files, create datasets and collections, socially curate data by assigning tags, metadata, and leaving comments, then publishing their data to a long term archive for preservation once work with the data has been completed [33]. In addition to the social curation capabilities Clowder provides it also emphasizes an element of aided auto-curation through a suite of extensible extractors that are automatically triggered and executed when files of an appropriate type are uploaded into a given instance of the system. These extractors can do anything from pulling metadata within the file, analyzing the file's contents and tagging it according to some specific classification or criteria, towards the generation new data such as metadata, previews,

Figure 2. An example of the type of metadata returned by the DTS, in this case given an image file. The DTS returns a breadth of extracted information in JSON such as EXIF metadata associated with the image; computer vision derived data from the image contents such as faces, eyes, and OCR text; as well as more domain specific information available within the instance such as the green index and human preference score used by our Green Infrastructure use case.

sections (i.e. areas of interest), etc. Typically triggered by a file's mime type, Clowder's extractors exist within a cloud environment distributed across any number of physical or virtual machines and listen to a distributed messaging bus for new files to the system. The Brown Dog Data Tilling Service (DTS)[6] builds on top of Clowder, emphasizing its REST interface towards allowing other applications to leverage its extraction capabilities, making it easier to create and deploy new extractors, building up an extensive catalogue of extractors, hardening the representation of derived data/metadata, enhancing the scalability and adding an elastic component to grow and shrink capabilities intelligently and dynamically based on user demand.

The DTS serves as a web based service where client applications or users can pass in one or more files or URLs and get back JSON or JSON-LD containing a number of derived products from tags, metadata, or other derived files that are typically higher level than the original data and/or holding some semantic information. Given this derived information applications can then use it to index, compare, and/or further analyze collections of data, in particular uncurated and/or unstructured data collections (Figure 2). As stated previously the main interface to the DTS is its REST API (Table 1). Through the REST interface a user/application can upload/point to a file for processing, list available extractors, check the status of extractors that are running on it, and download derived metadata produced thus far.

At the heart of the DTS is a distributed messaging bus (Figure 1), specifically RabbitMQ[7]. A widely used and hardened framework, RabbitMQ can be used to reliably distribute

[6]We use data tilling, like in farming, to emphasize a notion of churning data in various ways towards enhancing its usability via the uncovering of various higher level data products useful for indexing or otherwise using the data.

[7]https://www.rabbitmq.com/

GET	/api/extractions/inputs	Lists the input file format supported by currently running extractors
POST	/api/extractions/upload	Uploads a file for extraction of metadata and returns file id
GET	/api/extractions/upload	Uploads a file for extraction using the file's URL
GET	/api/extractions/{id}/status	Checks for the status of all extractors processing the file with id
GET	/api/files/{id}/metadata	Gets tags, technical metadata, and content based signatures extracted for the specified file
GET	/api/extractions/servers	Lists servers IPs running the extractors
GET	/api/extractions/extractors	Lists the currently running extractors
GET	/api/extractions/extractors/details	Lists the currently details running extractors

Table 1. The DTS REST API for metadata, tags, and signature extraction.

and manage job execution in a cloud environment by placing messages in a queue for each extractor, taking them off when jobs are completed, and automatically resubmitting messages should a job fail. A cloud setup is particularly well suited for our application, versus other high performance oriented infrastructures, as we are building a framework that can leverage potentially any other code/tool as extractors, essentially black boxes, which typically can't be optimized further internally. Towards further reliability the RabbitMQ service itself can further be distributed should one instance of it fail as well. The DTS Clowder head node handles all messages to and from the distributed queue and further manages the moving and storing of intermediary files. Files uploaded to the DTS for processing are either stored or referenced and given a unique ID. At this time a message is put on the bus with the file ID along with a key based off of the files mime-type. Any extractor capable of handling that type of file takes the message off of the queue, uses the ID to obtain the file for further processing, processes the job, then returns any derived data back to Clowder associating it with the file's ID. Data and metadata stored within Clowder can be placed into one of a number of extensible storage options such as MongoDB, iRODs [9], or the local filesystem. By default we use MongoDB which is convenient in that all communication between the various Clowder components uses JSON and MongoDB is JSON document based. The underlying mongo database can further be sharded for added capacity, performance, and reliability. Lastly, the DTS Clowder head node itself is designed so as to be stateless itself allowing it to be replicated and placed behind a load balancer, e.g. NGINX, in order to increase performance and reliability.

Extractors can be written in any language so long as it is capable of interacting RabbitMQ and HTTP in order to access the Clowder REST interface (e.g. Java, C/C++, Scala, Python, Ruby, etc). In addition to carrying out some sort of analysis of the data, often times by wrapping some external piece of code/software, an extractor requires a relatively small amount of code in order to interact with the rest of the system. Specifically, it must register itself with the RabbitMQ bus and specify what keys it will respond too, listen for incoming messages and pick up those that it can process, and lastly return derived tags, metadata, etc. to Clowder. To further simplify the creation of extractors we have written a

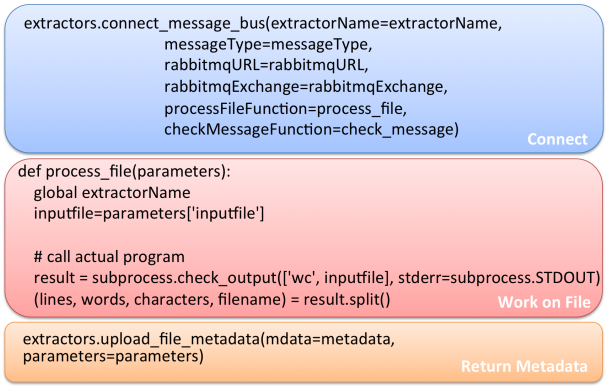

Figure 3. Creating an extractor for deployment within the DTS is simplified through the use of various language specific libraries. For example the pyClowder library allows one to create python extractors with essentially the three pieces of code shown here which: connects to the distributed queue, carries out or calls the analysis code, then lastly returns the derived data.

library, pyClowder[8], that reduces this setup to a handful of boilerplate lines of code (Figure 3). Additional libraries are in development for commonly used languages in research such as R[9] and Matlab[10].

Three types of metadata are differentiated by the system: technical metadata, versus metadata, and previews. Technical metadata is automatically generated, derived data, by the extractors, e.g. obtaining text contents within an image, classification of an object, a Greeness Index, coordinates of the specific sections of a file, etc. Versus metadata, obtained from an extractor leveraging the Versus framework [39], are the signatures extracted from a file's contents. These signatures, effectively a hash for the data, are typically numerical vectors, which capture some semantically meaningful aspect of the content so that two such signatures can then be compared using some distance measure (capturing either similarity or dissimilarity). These signatures along with Versus allow content based retrieval [30] tools to be included and produced by the DTS towards yet another means of comparing and sifting through data. The final type, previews, as well as other types of derived file products, are also generated by the extractors. These can include things such as thumbnails, image pyramids, sections/clips, maps, spreadsheets, etc. These derived products again aid in the navigation of data, this time in the manual human sense, perhaps providing a visualization of the data or collection of data. This may also be used as an intermediary in a chained extraction process where some other extractor will then be triggered to extract something more meaningful from this new data (e.g. generating a signature from a key frame extracted from a video). For each type of metadata the system also keeps track of the source extractor that created it for provenance purposes.

Again the metadata returned from the extraction process is in the JSON (JavaScript Object Notation) format which is popular in REST clients within web applications. The technical

[8]https://opensource.ncsa.illinois.edu/stash/projects/CATS/repos/pyclowder/
[9]http://www.r-project.org/
[10]http://www.mathworks.com/products/matlab/

GET	/api/conversions/outputs	Lists all output formats that can be reached
GET	/api/conversions/inputs	List all input formats that can be accepted
GET	/api/conversions/inputs/ input format	List all output formats that can reach the specified input format
GET	/api/conversions/outputs/ output format	List all input formats that can reach the specified output format
GET	/api/conversions/convert/ output format/file URL	Convert the specified file to the requested output format
POST	/api/conversions/convert/ output format	Convert the uploaded file to the requested output format
GET	/api/conversions/software	List all available conversion software
GET	/api/conversions/servers	List all currently available Software Servers

Table 2. The DAP REST API for format conversions.

and versus metadata are represented in JSON format and often have a nested structure with more arbitrary content. Previews are represented also in JSON as well with just the URL to the preview, file to which they are associated with and the extractors that generated it. To provide more structure to the metadata extracted and making it easier for external clients to consume the metadata produced by the DTS, we added initial support for linked data and JSON-LD[11]. This is also an important step towards making the information extracted by the Brown Dog services more easily accessible to the other information sources using core standards for structured data developed by the World Wide Web consortium[12]. Even for cases where the client of the Brown Dog service is not particularly interested in the need for such standards, the linked data approach provides best practices for data on the Web that help make the services easier to access. More specifically, the use of JSON-LD provides a lightweight overlay onto the raw JSON that can be safely ignored by clients that are not interested in the semantic web aspects of linked data.

B. Data Access Proxy

The Brown Dog Data Access Proxy (DAP) handles conversions between formats, ideally to one that is more readily accessible for the user. Unlike the DTS which takes no parameters but triggers any extractor that will fire based on the file type, the DAP takes a single parameter specifying the desired output format. Like the DTS, the DAP provides a compact REST interface allowing users and applications to call its capabilities (Table 2).

Built on top of the the Polyglot framework the DAP is designed to support the inclusion of any piece of code, library, software, or service into its ecosystem of conversion tools. A component tool called a Software Server [38] utilizes one or more very light weight wrapper scripts to automate specific capabilities within arbitrary code and then provide access to it via a consistent REST interface:

`https://<host>/software/:application/:output/:file`

Applications can then call, program against, these capabilities within the software as easily as they would a library. When GUI applications are involved scripting languages such as AutoHotKey[13] or Sikuli [40] are used to wrap needed open/save

[11]http://json-ld.org/
[12]http://www.w3.org/
[13]http://www.autohotkey.com/

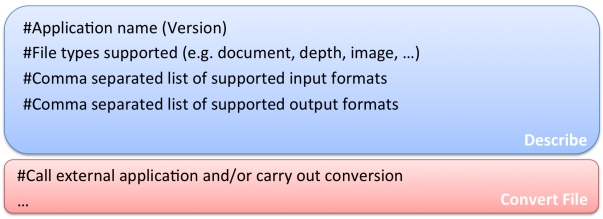

Figure 4. Adding converters to the DAP is done by simply annotating a wrapper script for the software in the scripts comments. The first few comments tell the DAP what the software is, what types of data it works on, and what inputs and outputs it supports. This is sufficient for the DAP to manage and delegate conversion jobs to that software. The remainder of the script either carries out or calls another application to do the conversion.

functionality. Any text based scripting language may be used for these scripts so long as it follows certain conventions in its comments indicating the name and version of the software, the types of data it handles (e.g. documents, 3D), and possesses a list of accepted input and output formats (Figure 4). This is similar to the idea in the YesWorkflow system [41] where the comments are used to annotate parts of the code. The DAP also supports the use of Data Format Definition Language (DFDL) schemas [42], a relatively recently standardization of a machine readable language for format specifications, through the use of a specialized wrapper script for Daffodil, an open source implementation of DFDL. New schemas can be incorporated into the DAP by making a minor modification to this template script. DFDL schemas, generating XML representations from data contents for a given format, possibly mapping elements to standardized ontologies, provide a long term/preservable means of capturing the layout of file contents, particularly important for the many ad hoc formats scientists/graduate students utilize in their work (e.g. for tabular, spreadsheet like data, or other representations for data).

When a Software Server comes online, each potentially hosting one or more applications, it will attempt to connect to a specified RabbitMQ bus and then listen to one queue per software that it controls. The DAP Polyglot head node instance monitors the RabbitMQ bus, specifically the queues and the consumers of the queues, leveraging it as a discovery service for new Software Servers, software, and conversion capabilities. For each Software Server found, it queries it for the applications provided and the input and output formats each in turn supports. From this a graph is constructed, referred to as an input/output graph or I/O-graph, with formats as the vertices and applications as directed edges between vertices indicating conversions they are capable of carrying out. Given an input format and a desired output format the DAP will search this graph for a shortest path between the source and target format, allowing conversions to occur that would require multiple applications, possibly running on different machines. The constructed job is stored in a MongoDB instance, where one or more DAP Polyglot instances will monitor it and move it across the path, placing portions of the job on the appropriate application queues as need be. Since all information about the job is stored in a shared MongoDB instance the Polyglot head nodes are also stateless as in the case of the DTS and can be placed behind a load balancer for added performance and reliability. Software Servers, essentially pilots [43], will monitor the relevant software queues, pull off jobs, execute them on the local hardware, and return a link to the resulting output file back to the DAP head node. All files are preferably passed between the DAP, each Software Server, and even the external source, as URLs in order to minimize file transfers. This saves transfers in a number of instances, e.g. returning the output of the last Software Server called to the DAP head node which then returns it to the user, or should a Software Server possess the needed applications for two parts of a conversion path.

As a cloud based service it is expected that the applications and Software Servers are elastic, i.e. new ones will come on line on occasion, and current ones will go offline on occasion. As such the I/O-graph must be updated continuously so that all currently valid conversion paths are represented. A thread within each DAP Polyglot instance will continuously poll the consumers on the RabbitMQ bus. When new Software Servers are found their vertices and edges are simply added to the graph, which represents the union of capabilities among all discovered Software Servers. When a found Software Server no longer responds, for whatever reason, the constituent edges for its applications are pruned from the graph. In order to make the traversal and the appending of new applications to the graph efficient the I/O-graph is represented as an adjacency list in memory. This however, makes the removal of edges and vertices somewhat costly, especially for large graphs, thus pruning is done sparingly and limited only to the edges which is sufficient to eliminate invalid conversion paths.

C. Elasticity

The two building blocks of Brown Dog, the DTS and DAP, which are built to utilize arbitrary code/software as extractors/converters, need to have the ability to handle heavy loads, adapt to spikes in requests, handle a mix of long running and short running jobs, and support heterogeneous architectures. Specifically, the two services need to be able to auto scale based on the demand of the system.

There are two approaches [44] to scale a system: vertical scaling, i.e. increasing the resources allocated to the nodes in the system such as CPU, memory, storage; and horizontal scaling, i.e. by adding nodes to the system. Cloud computing services modeled as Infrastructure-as-a-Service (IaaS)[14] [11] provide features such as elasticity, i.e., automatic resource provisioning and de-provisioning and allows horizontal scaling through an inherent ease at starting new VMs during variable workloads. We aim to leverage cloud computing IaaS for achieving the autoscaling of the Brown Dog services. We designed and implemented a Brown Dog Virtual Machine (VM) elasticity module that focuses on auto-scaling the DTS's extractors and DAP's Software Servers. Specifically, the module starts or uses more extractors when certain criteria are met, such as the number of outstanding requests exceeding a

[14]http://aws.amazon.com/

certain threshold - this is called scaling up, and suspends or stops extractors when other criteria are met - this is called scaling down. Based on extractor types, the module needs to support multiple operating system (OS) types, including both Linux and Windows, for its proper execution. Further we wish to support a variety of VM/container frameworks to allow extractors and Software Servers to be deployed on a variety of different resources.

We considered a number of cloud computing software platforms, both open source such as OpenStack, Cloud Stack, and Eucalyptus; and commercial such as Amazon Web Services (AWS), Microsoft Azure, and VMWare; as well as other related technologies such as Olive [45], OpenVZ[15] and Docker. AWS is relatively mature but can be costly as when CPU and memory usage go up the cost of using AWS also goes up. In the open source space, OpenStack is mature, widely adopted, and supports both Linux and Windows. For our initial setup we chose OpenStack as the top level VM technology, Unix system services as the low level technology, and Docker as an intermediary level technology candidate with regards to the level of granularity by which we control the elasticity (e.g. Docker has a smaller resource usage overhead and faster VM/container startup time compared with OpenStack).

1) Elasticity Module Design: The work here focuses on the scaling of extractors and converters within Software Servers (SS) but could be extensible to other services as well with some modifications. We make the following assumptions in our design:

- An extractor or a SS is installed as a service on a VM. Extractors of the same type, i.e. requiring the same execution environments, can be deployed in the same VM. So when a VM starts, all the extractors that the VM contain as services will start automatically and successfully.
- The resource limitation of using extractors/SS to process input data is CPU processing, not memory, disk I/O, or network I/O, so the design is only for scaling for CPU usage. However, in the future we aim to consider other resource limitations.
- The system uses RabbitMQ as the messaging technology.

There have been several works on elasticity in the cloud computing context [46, 47, 44, 48] and references there in. Lots of policies discussed in these works, and also the ones provided by the cloud providers allow scaling based on the VM internal information such as CPU usage, memory, etc; and based on prediction of workload pattern. We provide an auto scaling solution that is based on the queue lengths at the message queues for extractors and SS. Our web application/service is a loosely coupled publish/subscribe system. As we use RabbitMQ as the messaging technology which acts as a broker, we get the added advantage of getting finer grained details such as queue lengths, channel activities, connection details, consumer details etc. using it's management HTTP API. As extractors and SS are written such that they could potentially use any existing tool/software/web service the performance and processing of requests may be limited by the tool being used under the hood. Thus, each extractor/Software Server is configured as a service to fetch one message from the queue at a time which loaded balances the job requests among the services. We also present scaling at two levels: service-level and VM-level. At the service-level, an extra instance of the service is deployed in the VM already running the same type of services while at the VM-level, a new VM or a suspended VM containing the service is started. In our algorithm, the module will monitor the message queue and take scaling actions accordingly using the Openstack API to start/suspend/resume/terminate a VM.

[15]http://openvz.org

Algorithm 1 BDMonitor()

```
1: Read the control parameters' values from the config file
2: while TRUE do
3:     Build OpenStack Server Map
4:     Build Run-time Mapping between Services and running VMs, service2RunningVMMap
5:     Obtain VMs usage information, e.g., loadAverage and numvCPU and build vmInfoMap
6:     ScaleUp()
7:     ScaleDown()
8: end while
```

Taking a modular approach, we define a separate module for: i) monitoring the queues for extractors/SS in the message bus, ii) obtaining information such as load average from the VMs deployed in the cloud, iii) setting criteria for scaling up/down decisions, and iv) providing scaling actions using cloud specific APIs. Algorithm 1 shows the pseudo code for the auto scaling that we use in our implementation which periodically checks if scaling up/down is required for the service. It reads all the parameters needed from a configuration file. Algorithm 2 and Algorithm 3 show the psuedo code for scale up and scale down, respectively. In Algorithm 2 Line 1, the services scale up candidate list is obtained by following the criteria: i) the length of the RabbitMQ queue for a service is greater than a pre-defined threshold, such as 100 or 1000, or ii) the number of consumers (extractors/Software Servers) for the queue is less than the minimum number configured. In Line 3-22, the algorithm iterates over the candidate lists, obtains the VM information of each service type, checks for the condition in Line 7, i.e. number of virtual CPUs in the VM is greater than the load average plus the CPU buffer, if true then start an instance of the service type, else resume a previously suspended VM that contains the service type or start a new VM that contains the service type. In Algorithm 3, Line 1 obtains the service instance scale down list by following the criteria: i) idle queues (no data /activity for a configurable amount of time), ii) idle VMs by using the channels idle time taking care of multiple channels on the same same VM. Lines 2-9 are self-explanatory.

2) Implementation: The elasticity module is a stand-alone program written in Python. The statistics obtained from RabbitMQ and Openstack, as well as the scaling actions information, are written into a MongoDB database so these values can later be analyzed/visualized. As a monitoring component in Brown Dog this module is important to ensure DTS and

Algorithm 2 ScaleUp()

```
Get service scale up candidate list, sList
for sName in sList do
    if sName in service2RunningVMMap then
        Get the running vmList for the sName
        Sort vmList based on VM's cpuLoadRoom, i.e. VM.numvCPUs − VM.loadAverage
        for each VM in vmList do
            if numvCPUs > (loadAverage + cpuBuffer) then
                Add a service instance to the VM
                break
            end if
        end for
        continue
    end if
    Resume a VM containing the service sName
    if Resume is successful then
        continue
    else if a VM is started containing the service sName within the ScaleUpAllowanceTime then
        Skip starting a new VM
    else
        Start a new VM instance containing the service sName
    end if
end for
```

Algorithm 3 ScaleDown()

```
Get service instance scale down candidate list, sList
for sName in sList do
    Get the vmList for the service sName
    for each VM in vmlist do
        Stop and remove the idle service exName instances from the VM maintaining minimum number of sName instances
    end for
end for
Get Idle VM scale down candidates list
Suspend the idle VMs from the list based on its channel's idle time in the message bus while maintaining minimum number of service instances
```

DAP's performance, robustness, and availability. We run this module as a service, so that the OS watches and ensures that this module is up and running.

We used a configuration file to specify the information for RabbitMQ access, OpenStack access, OpenStack VM images, and the scaling parameters such as checking interval, queue length threshold and CPU buffer room. The implementation obtains run-time RabbitMQ queue and VM information and idle time using the RabbitMQ management API; obtains run-time VM information such as load average and #vCPUs and started/stopped extractor instances through executing SSH commands on the VMs; and uses the OpenStack Python API to suspend, resume, and start VMs.

III. Extensibility

As the name suggests, a mutt of software, the Brown Dog services, in particular via Software Servers, are designed to use and manage potentially any piece of software to carry out conversions and extractions. In a manner similar to Apple's App Store, Galaxy's Tool Shed [19], or other such repositories for applications[16] [18, 45] we build a Tools Catalog which allows users to add new tools and their capabilities to the two services. As the DAP and DTS can utilize arbitrary code to carry out operations the Tools Catalog doesn't actually store the actual tools (i.e. code, software), but instead references them typically via a URL to a website or source code repository. What is stored within the repository is information needed to both call these referenced tools and to give credit to their creators so as to motivate the addition of new tools. Control of the tool is done through the wrapper scripts mentioned previously (Figure 3 and Figure 4), and is designed to be as simple and straight forward as possible. Credit for the time being is done via providing citation information, such as a relevant paper, or to the software directly[17]. This can be done as simply as providing a Digital Object Identifier (DOI) and will become more and more important as the scientific community moves towards providing as much credit for software and data as it does articles [3, 4].

Like the Apple App store the Brown Dog Tools Catalog operates under a tool approval process. Tools added to the Tools Catalog are not visible or available until an administrator reviews and approves it. Once approved the tool is visible and available to others and may also be directly deployed to one of the main DAP[18] and DTS[19] instances.

IV. Evaluation

With regards to performance of the proposed infrastructure we focus on two measures, scalability, specifically with regards to the very heterogeneous tools that make up the two services, and capabilities, what types of data are we currently able to support. Though the deliberate extensibility of the system plays into the latter criteria it is still not trivial as elements within the system, e.g. the I/O-graph utilized by the DAP, allow for combinations of tools to present additional capabilities (e.g. a chain of conversions to reach a desired target format). Below we describe our results with regards to the elastic scaling of the system as well as a framework we are building towards the continuous evaluation of the capabilities of the system.

A. Elasticity

The experimental evaluation is designed to study the effectiveness of our approach and to get insights into the autoscaling behavior of the monitored services. Our experimental setup comprises of:

- Two extractor types - OpenCV based [49] and OCR [50], both extractors triggered off of images. Specifically we consider four OpenCV extractors to detect faces, eyes, profiles and closeups of faces, and one OCR extractor.
- Created two VM images based on Ubuntu Trusty (14.04) with 1 GB RAM, 1vCPU, 10GB disk space for the two extractor types. One image is configured to have one instance of four OpenCV extractors (faces, eyes, profiles, closeups) as services with OpenCV already installed. The second image is configured to have one OCR extractor.
- Single instance of DTS and RabbitMQ server.
- Sixty test image files in the PNG format with varying sizes between 10KB-4MB . The image contents have a

[16] https://www.docker.com/

[17] http://zenodo.org/

[18] https://dap.ncsa.illinois.edu

[19] https://dts.nsca.illinois.edu

varied number of faces, eyes, closeups, profiles, and text. Each of the images has been tested against the extractors to make sure the extraction process works correctly.
- The queue length threshold is thirty and Idle Time for VM before it could be suspended is fifteen minutes. Minimum number of extractor instances required is two for all extractors.

For our experiments we utilize an OpenStack cloud run out of the NCSA Innovative Systems Lab (ISL).

We wrote a Python script to submit 1200 job requests using the DTS API, uploading 1200 images (20 times each image in the dataset) for extraction, to observe the autoscaling behavior. As all OpenCV and OCR extractors process images, each extractor receives all 1200 files for extraction. We observe the queue lengths for each extractor, number of extractor service instances for each queue, load average of each VM, and number of VMs. Figure 6 shows the plot of the number of extractor instances started for the observed queue length over the time. The total time taken to serve all the requests by all the extractors including the scaling down to suspend idle VMs was approximately seventy minutes. As seen in the figure, we see the queue lengths of all the five extractor types grow to 1200 (approximately in three minutes). The new VMs and extractors are started as the queue lengths cross the threshold of 30 requests. Once an action to start a VM taken, we wait at least 3-5 mins, which is the usual startup time for a VM, before starting a new VM. In the plot we see the number of extractor instances is increasing unit stepwise. During the scaling up the total number of new VMs started for the OCR extractor is three with the total number of OCR extractor instances nine; and for OpenCV extractor 10 VMs with the number of extractor instances for faces, eyes, closeups, and profiles started 14, 14, 11, and 20 respectively. The second plot in Figure 6 shows the load average over the scaling up time for some of the extractors VMs. As we see in the figure, opencv-27 and ocr-10 are the initial VMs. After approximately 3-5 minutes, when a new VM started, the load average for ocr-11 and opencv-28 begin to go up. As new VMs are started, the load average becomes almost equal for each of those VMs indicating the load balancing of the tasks.

We then ran the same experiment with the VMs suspended from the first experiment. The third plot in Figure 6 shows the plot of the number of extractor instances over time with no suspended VMs (from the first experiment) and with suspended VMs at the beginning of the experiment. It is observed that due to long start up time of a new VM the scaling up is slow in the first experiment while in the second experiment, as the suspended VMs from the first experiment are resumed, takes usually 30-60 seconds. It is also observed that the number of extractor instances deployed in the second experiment is more than the first experiment. The final plot in Figure 6 shows the queue lengths for all extractors over time for both experiments. We can clearly see the queue length almost reaches 1200 for all five extractors in the first experiment while in the second experiment, the maximum queue lengths reached is approximately 700. OCR and Closeup

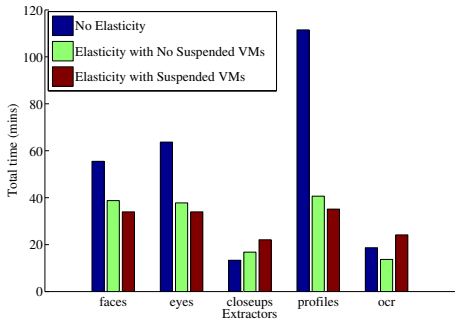

Figure 5. Comparison of total time to serve 1200 job requests by each extractor with no elasticity module, elasticity module with no suspended VMs, and with elasticity module with suspended VMs at the beginning of the experiment.

extractors are computationally less expensive as compared to the other three extractors. This is also reflected in the total time to serve the 1200 requests. Figure 5 shows the comparison of total time to complete 1200 requests with/without the elasticity module.

B. Curation and the CI-BER Testbed

In order to test and report on the application of Brown Dog to archival collections, we attempt to simulate archival processing using the CI-BER[20] [27] collection housed at the UMD Digital Curation Innovation Center. CI-BER contains 52 terabytes, 72 million objects spanning a wide variety of file formats, scientific datasets, and organizational records. Through scripted interactions with the services we gather metrics and identify gaps in function. The scripts are run over and over again and seeded with different sample data files. The technology we use for simulation is a testing framework called Gatling.io[21]. Gatling is based on the Scala functional programming language, which lends itself to terse expressions of complex functional patterns, such as with a simulation. Gatling adds a domain-specific language (DSL) for constructing web-based tests. These scenarios may be run as single user tests or they may be run in parallel as realistic load tests, simulating the same scenario for hundreds of users at the same time. Gatling produces detailed metrics on service performance under load, which can be used by developers to isolate issues. Gatling also allows custom validation patterns that our simulations use to verify specific outcomes. We write our simulation scripts in the Gatling DSL, seeding simulations with a randomized set of sample files. After each simulation runs we gather up results from Gatling log files and other sources to build a database in MongoDB of simulation results.

1) Archival Processing Simulations: The initial approach was to employ Brown Dog for data format conversions typical of digital preservation systems, namely the conversion of legacy materials into current file formats for preservation and access. Twice a day we take a random sample of modern and legacy office files from the CI-BER collection, attempting

[20]https://ciber.umd.edu
[21]http://gatling.io/

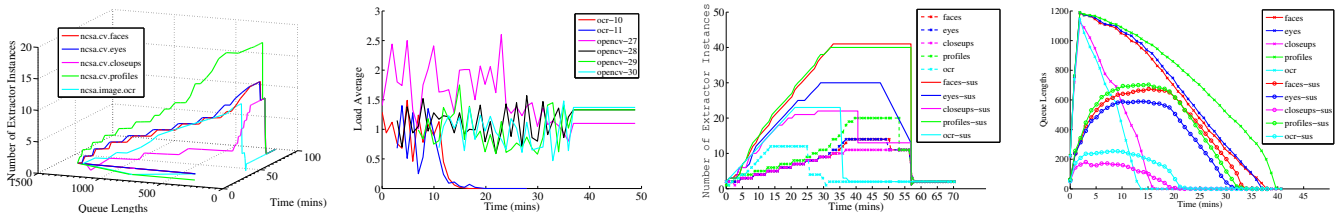

Figure 6. Experimental evaluation of Brown Dog elasticity module. **Left** to **right:** queue lengths vs time vs number of extractors, load average vs time for a set of VMs, number of extractor instances over time, and queue lengths over time.

to convert them all into PDF using the DAP. Any office documents that have no conversion path are flagged as a problem (this can change over time as this is an elastic cloud based system). We periodically analyze missing conversion paths and make these into feature requests for the Brown Dog development team. The Gatling test framework records precise timestamps at various points in the HTTP interaction. For instance we can measure conversion time as the time between when the last byte of the HTTP request was sent, until the time the first byte of the HTTP response is received. Figure 7 shows the performance of the office document conversion over time for 100 file conversions. Each simulation consists of either 100 or 1000 users, each running one conversion. The formats used included: DOC, DOCX, ODF, RTF, WPD, WP, LWP, and WSD. We perform a similar file conversion test for image file formats. Formats currently under test include: TARGA, PICT, WMF, BMP, PSD, TGA, PCT, EPS, MACPAINT, MSP, and PCX. These are converted to the TIFF format and any exceptions are reported to the development team.

While TIFF and PDF are used as preservation formats by some archives, we do not assume that TIFF and PDF are the most desirable formats for all archives. Instead, the TIFF and PDF conversions ensure that the content is not trapped in the original format, that Brown Dog can open the original file and get content out of it. Thus far we have identified numerous samples that are trapped in WordPerfect (WP and WPD), Photoshop (PSD), and Windows MetaFile (WMF) formats. A more robust simulation is also being developed through a policy-based format migration which will sample files randomly from the CI-BER collection and conult a lookup table to obtain the preferred preservation format.

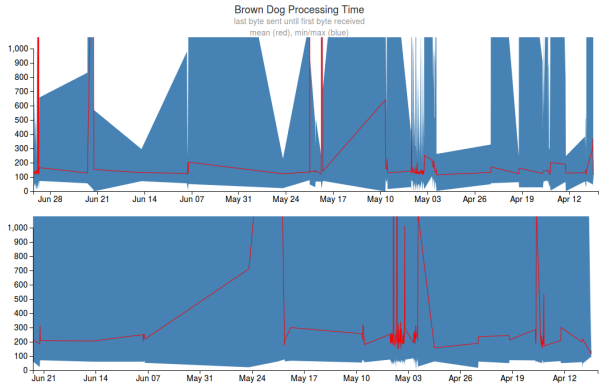

Figure 7. **Top:** Conversion simulation results for document conversions to PDF. **Bottom:** Conversion simulation results for image conversions to TIFF.

The simulation will report missing migration paths, as well as missing migration policies, i.e. data files or formats for which no preservation format has been recommended.

V. Conclusion

We have deployed an alpha release of the two services and begun incorporating tools in support of a number of our use cases (e.g. supporting ecological model conversion via PEcAn, supporting Lidar analysis for our hydrology use case, supporting human preference modeling for our green infrastructure use case, as well as other capabilities suited for more general usage). Further, towards supporting the wide range of users across our use cases we have begun developing a number of client interfaces[22] to leverage the DAP and DTS. These include language specific libraries, a bookmarklet interface that can be used to call the services on arbitrary web pages, a Google Chrome extension, a command line interface, incorporation into a scientific workflow system [51], and incorporation back into Clowder. Efforts moving forward aim to add additional cloud infrastructure support to the elasticity module, refine the level of granularity considered during scaling by deploying Docker instances of converters/extractors within a single VM, exploring how we might optimize data movement so as to as efficiently as possible handle large data collections, and incorporating/using additional information relevant to provenance such as the estimates of information loss incurred during specific conversions described in [23].

Acknowledgements

This research & development has been funded through National Science Foundation Cooperative Agreement ACI-1261582.

References

[1] P. Lyman *et al.*, "How much information 2003?" 2003. [Online]. Available: http://www2.sims.berkeley.edu/research/projects/how-much-info-2003/

[2] N. webmaster, "Vision for cyberinfrastructure framework for 21st century science and engineering," 2013. [Online]. Available: http://www.nsf.gov/od/oci/cif21/CIF21Vision2012current.pdf

[3] L. Lannom and F. Berman, "Special issue on the research data alliance," *D-Lib Magazine*, 2014. [Online]. Available: http://www.dlib.org/dlib/january14/01contents.html

[4] E. Seidel, "The national data service - a vision for acceleration discovery through data sharing," 2015. [Online]. Available: http://www.nationaldataservice.org

[22] http://browndog.ncsa.illinois.edu/blog.html

[5] M. Zaharia *et al.*, "Resilient distributed datasets: A fault-tolerant abstraction of in-memory cluster computing," *9th USENIX Symposium on Networked Systems Design and Implementation*, 2012.

[6] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Symposium on Operating System Design and Implementation*, 2004.

[7] K. Jeon *et al.*, "Pigout: Making multiple hadoop clusters work together," *IEEE BigData*, 2014.

[8] W. Michener *et al.*, "Participatory design of dataone - enabling cyberinfrastruture for the biological and environmental sciences," *Ecological Informatics*, 2012.

[9] A. Rajasekar, M. Wan, W. Schroeder, and R. Moore, "From srb to irods: Policy virtualization using rule-based data grids," 2005.

[10] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," *IEEE Symposium on Mass Sotrage Systems and Technologies*, 2010.

[11] I. Foster, "Globus online: Accelerating and democratizing science through cloud-based services," *IEEE Internet Computing*, 2011.

[12] L. Bavoil *et al.*, "Vistrails: Enabling interactive multiple-view visualizations," *IEEE Visualization*, 2005.

[13] M. Turk *et al.*, "yt: A multi-code analysis toolkit for astrophysical simulation data," *The Astrophysical Journal Supplement*, 2011.

[14] J. Towns *et al.*, "Xsede: Accelerating scientific discovery," *Computing in Science Engineering*, 2014.

[15] R. Heimann, "Big social data: The long tail of science data," *Imaging Notes*, 2013.

[16] B. Ludascher *et al.*, "Scientific workflow management and the kepler system," *Concurrence and computation: Practice and Experience, Special Issue on Scientific Workflows*, 2006.

[17] E. Deelman *et al.*, "Pegasus: A framework for mapping complex scientific workflows onto distributed systems," *Scientific Programming Journal*, 2005.

[18] G. Klimeck *et al.*, "nanohub.org: Advancing education and research in nanotechnology," *Computing in Science and Engineering*, 2008.

[19] J. Goecks, A. Nekrutenko, and J. Taylor, "Galaxy: A comprehensive approach for supporting accessible, reproducble, and transparent computation research in the life sciences," *Genome Biology*, 2010.

[20] C. Lagoze, S. Payette, E. Shin, and C. Wilper, "Fedora: An architecture for complex objects and their relationships," *Journal of Digial Libraries*, 2005.

[21] M. Smith *et al.*, "Dspace: An open source dynamic digial repository," *D-Lib Magazine*, 2003.

[22] L. Marini *et al.*, "Medici: A scalable multimedia environment for research," *The Microsoft e-Science Workshop*, 2010.

[23] K. McHenry, R. Kooper, and P. Bajcsy, "Towards a universal, quantifiable, and scalable file format converter," *The IEEE Conference on e-Science*, 2009.

[24] W. Underwood, "Grammar-based specification and parsing of binary file formats," *International Journal of Digital Curation*, 2012.

[25] W. Reggli, J. Kopena, and M. Grauer, "On the long-term retention of geometry-centric digital engineering artifacts," *Computer Aided Design*, 2010.

[26] F. Soper, "The pronom file format registry," *Experts Workgroup on the Preservation of Digital Memory*, 2004.

[27] J. Heard and R. Marciano, "A system for scalable visualization of geographic archival records," *IEEE Symposium on Large Data Analysis and Visualization*, 2011.

[28] T. Rath and R. Manmatha, "Word spotting for historical documents," *International Journal on Document Analysis and Recognition*, 2007.

[29] L. Diesendruck, R. Kooper, L. Marini, and K. McHenry, "Using lucene to index and search the digitized 1940 us census," *Concurrency and Computation: Practice and Experience*, 2014.

[30] M. Lew, "Content-based multimedia information retrieval: State of the art and challenges," *ACM Transactions on Multimedia Computing, Communications, and Applications*, 2006.

[31] D. Garette and E. Klein, "An extensible toolkit for computational semantics," *International Conference on Computational Semantincs*, 2009.

[32] J. Lehmann *et al.*, "Dbpedia - a large scale, multilingual knowledge base extracted from wikipedia," *Semantic Web*, 2012.

[33] J. Myers *et al.*, "Towards sustainable curation and preservation: The sead project's data services approach," *Interoperable Infrastructures for Interdisciplinary Big Data Sciences Workshop, IEEE eScience*, 2015.

[34] M. Dietze, D. LeBauer, and R. Kooper, "On improving the communication between models and data," *Plant, Cell & Environment*, 2012.

[35] J. Han *et al.*, "A neotropical miocene pollen database employing image-based search and semantic modeling," *Apllications in Plant Sciences*, 2014.

[36] E. Spalding and N. Miller, "Image analysis is driving a renaissance in growth measurement," *Current Opinion in Plant Biology*, 2013.

[37] S. P. Satheesan, S. Poole, R. Kooper, and K. McHenry, "Groupscope: A microscope for large dynamic groups research," *IEEE eScience*, 2013.

[38] K. McHenry *et al.*, "A mosaic of software," *The IEEE International Conference on eScience*, 2011.

[39] L. Marini *et al.*, "Versus: A framework for general content-based comparisons," *IEEE eScience*, 2012.

[40] T. Yeh, T. Chang, and R. Miller, "Sikuli: Using gui screenshots for search and automation," *UIST*, 2009.

[41] T. McPhillips *et al.*, "Yesworkflow: A user-oriented, language-independent tool for recovering workflow information from scripts," *International Journal of Digital Curation*, 2015.

[42] M. Beckerle and S. Hanson, "Data format description language (dfdl) v1.0 specification," *Open Grid Forum*, 2014.

[43] A. Luckow *et al.*, "P*: A model of pilot abstractions," *IEEE eScience*, 2012.

[44] L. M. Vaquero, L. Rodero-Merino, and R. Buyya, "Dynamically scaling applications in the cloud," *SIGCOMM Comput. Commun. Rev.*, vol. 41, no. 1, pp. 45–52, Jan. 2011.

[45] M. Satyanarayanan *et al.*, "Olive: Sustaining executable content over decades," *XSEDE*, 2014.

[46] G. Galante and L. C. E. d. Bona, "A survey on cloud computing elasticity," in *Proceedings of the 2012 IEEE/ACM Fifth International Conference on Utility and Cloud Computing*, ser. UCC '12, 2012, pp. 263–270.

[47] C. Bunch *et al.*, "A pluggable autoscaling service for open cloud paas systems," in *Proceedings of the 2012 IEEE/ACM Fifth International Conference on Utility and Cloud Computing*, ser. UCC '12, 2012, pp. 191–194.

[48] J. Yang *et al.*, "Workload predicting-based automatic scaling in service clouds," in *Proceedings of the 2013 IEEE Sixth International Conference on Cloud Computing*, ser. CLOUD '13, 2013, pp. 810–815.

[49] A. Kaebler and G. Bradski, *Learning OpenCV, Computer Vision in C++ with the OpenCV Library*. O'Reilly.

[50] R. Smith, "An overview of the tesseract ocr engine," *International Conference on Document Analysis and Recognition*, 2007.

[51] R. Kooper *et al.*, "Cyberintegrator: A highly interactive scientific process management environment to support earth observations," *Geoinformatics Conference*, 2007.