# RESTful Annotation and Efficient Collaboration

**Jonathan D. Wright**
Linguistic Data Consortium, University of Pennsylvania
3600 Market Street Suite 810, Philadelphia, PA, 19104, USA
jdwright@ldc.upenn.edu

**Abstract**

As linguistic collection and annotation scale up and collaboration across sites increases, novel technologies are necessary to support projects. Recent events at LDC, namely the move to a web-based infrastructure, the formation of the Software Group, and our involvement in the NSF LAPPS Grid project, have converged on concerns of efficient collaboration. The underlying design of the Web, typically referred to as RESTful principles, is crucial for collaborative annotation, providing data and processing services, and participating in the Linked Data movement. This paper outlines recommendations that will facilitate such collaboration.

**Keywords:** annotation, restful, collaboration, NLP, HLT, pipelines, cost reduction

## 1. Introduction

### 1.1. Motivation

With Big Data comes Big Collaboration, and new challenges. An organization may receive Gigaword[1] from LDC[2] in the mail, but then what? Does that organization build its own NER tagger and annotation tools? Do they download existing toolkits to run locally? They may, but the internet allows the possibility of distribution of effort, of machine and human annotation occurring remotely, of the data itself delivered as needed. The challenge is then one of communication, and communication that takes place fast enough to be useful.

At LDC, beyond collaboration with other organizations, efficiency is of paramount importance. At a large corporation, the most sensible course of action may be to throw money at the problem (any problem). LDC is relatively resource limited: even for a well funded project, a small staff must meet short deadlines. Furthermore, computational resources typically aren't the bottleneck anymore, and processing and storage are rarely obstacles. Rather, we must address human efficiency.

In short, collaboration and efficiency converge on similar problems and solutions. Collaboration isn't realistic without a certain degree of efficiency, and efficiency demands within an organization lead to "collaborative" behavior. Annotators work remotely, therefore are identical to collaborators from a technical point of view. The internet services of collaborators are used to avoid reimplementing capabilities locally. When capabilities are implemented locally for some reason, they are designed as internal services that multiple projects share. Internal networks and external networks converge in form. Or put in different terms, *we have a financial incentive to make internal and external operations as similar as possible*.

### 1.2. Dimensions of Efficiency

Programmers are familiar with time and space efficiencies, which generally are opposing. For example, the use of an index decreases query time but increases space consumption. Project managers have to consider further tradeoffs, like programmer effort. The implementation of an index might be costly, and therefore not worth the speed increase. Collaboration brings further considerations. A URL that provides a large corpus in a single response is efficient for the service provider in terms of programmer time, but not for the collaborative effort because the server and client software might not perform well. Here we consider all these dimensions in scenarios typical of the HLT field.

### 1.3. Procedure

The World Wide Web, or more specifically, the network of computers that communicate via HTTP[3], is arguably the most successful distributed architecture in the world. The Web exemplifies at least 3 properties: universal identification, a constrained set of operations, and hypermedia(Richardson and Amundsen, 2013). Representational State Transfer (REST) due to Roy Fielding[4] refers to the notion of transferring resource *representations*, and RESTful has become a cover term for the proceeding properties. Bending annotation and HLT processing to the same principles is the obvious path to efficient collaboration. We'll examine these properties by developing a toy problem as we go, which is trivial in content but realistic in form: annotating Part of Speech (POS) tags. LDC specific implementations will be drawn from but drastically simplified. In general our goal is to promote procedures, not implementations, but we must have concrete reference points. Furthermore, the implementations discussed are ever evolving. The demo accompanying this paper will illustrate whatever implementation is current using LDC's annotation server[5] (Wright et al., 2012).

## 2. Universal Identification

The internet relies on Universal Resource Identifiers (URIs) and Locations (URLs), the latter a subset of the former that is actually resolvable by some protocol. We'll use both

[1] (Parker et al., 2011)
[2] http://www.ldc.upenn.edu/
[3] http://www.w3.org/Protocols/
[4] http://roy.gbiv.com/
[5] https://webann.ldc.upenn.edu/

terms in what follows, but note that URLs are preferable, because resolvable identifiers facilitate machine negotiation of linked data. Resource Description Framework (RDF)[6] often uses URIs that are not URLs, leveraging the uniqueness property but neglecting resolvability. URLs will be used throughout, but shortened to exclude the domain, in other words http://www.ldc.upenn.edu/documents/1 will be shortened to /documents/1. Because our example is illustrative but not functional, the reader shouldn't expect resolvable URLs, but understand that they normally do represent URLs in a working API. Furthermore, internal to LDC, there are multiple schemes of unique identification, which coexist and serve complementary purposes. Here we focus on URLs which provide uniqueness and machine interpretability.

## 3. Protocols

We'll rely heavily on HTTP, which is a protocol with a very limited set of operations, GET and POST the most well-known. We'll also make reference to Create Read Update Delete (CRUD), which is a model of database operation, similar to the POST GET PUT DELETE verbs of HTTP. In both cases the motivation is partly to constrain available operations, while putting the burden of complexity on the data model. This provides advantages such as stability over time and simplicity in reasoning about operations. Remote Procedure Calls (RPC) is a programming paradigm somewhat in conflict with this approach, since it involves custom definition of operations. APIs can be designed in this fashion, for example, the following HTTP request could initiate tokenization.

```
REQUEST
GET /tokenize
Accept: text/plain
RESPONSE
success
```

This uses HTTP in a non-RESTful manner. SOAP[7] is a standardized means of implementing RPC within HTTP (or other protocols). SOAP and REST are not inherently in conflict as is sometimes assumed, but operate at different layers of abstraction. We'll return to this point in a later section.

## 4. Media Types

HTTP relies on a standardized set of media types, as well as a means to notate custom types. Here we begin our annotation example by annotating a word with a POS tag. Here's our document:

```
<DOC id="example">This is text.</DOC>
```

If "example" is unique within the space of identifiers at LDC, a conventional URI could be created like so:

```
http://www.ldc.upenn.edu/example
```

Another conventional pattern would look like this

```
http://www.ldc.upenn.edu/documents/1
```

which refers to the first in a collection of documents (see section on collections). Universal identifiers are somewhat a matter of convenience, and multiple schemes can coexist. The toy document above illustrates a salient problem at LDC, the ambiguity of format, or media type in the parlance of the web. Is the document XML or text, or both? In practice, the history of LDC newswire and its use across projects has led to difficulty in answering this question. It is not always possible to start from scratch with the "right" design, and data must be used as is. The present paper isn't the venue for debating text formats, but we can recommend one way of addressing the existing problem. The name REST is based on the idea that resources themselves aren't transferred, only their representations, of which multiple can exist. Therefore the following exchanges involve different representations of the same content.

```
REQUEST
GET /documents/1
Accept: text/plain
RESPONSE
<DOC id="example">This is text.</DOC>

REQUEST
GET /documents/1
Accept: text/xml
RESPONSE
<?xml version="1.0" encoding="UTF-8" ?>
<DOC id="example">This is text.</DOC>
```

This allows a formal inclusion of both formats at once. Specifications of the text that is annotated will differ, but can be explicitly related to each other.

For data other than the original source material, our media type is JSON[8] (one variant or another). This is currently the serialization format of choice for web APIs. If we serve our toy document as plain text, we can create our first JSON annotation as follows:

```
REQUEST
GET /tokens/1
Accept: application/json
RESPONSE
{
  "offset": 26,
  "length": 4,
  "text": "text",
  "POS": "noun"
}
```

## 5. Annotation Reification

A naive but sound model of annotation would be to have a relational database table for every object or concept of interest. For example, the four fields above could be the columns of a token table. This would lead to CRUDy annotation which can be mapped to the operations of a database and the basic verbs of HTTP. Retrieving the above annotation would execute a SQL SELECT, the Read of CRUD.

[6] http://www.w3.org/RDF/

[7] http://www.w3.org/TR/soap/

[8] http://www.json.org/

This model is simple, robust, and adequate for the generic web application, but for annotation (and likely many pursuits) it has notable shortcomings, explored below. We argue that rather than CRUDy annotation, we use CaReful annotation, limiting ourselves to Create and Read at the highest interface level. The user experience of Update and Delete will be expressed at a lower level. At the lowest level, the database interface, the operations would still be CRUD, but annotation involves software that wraps the database and imposes it's own interface.

First, the Update operation obliterations the original annotation. Not only is this inadequate, but so is a typical log file. Also consider dual annotation. While a token can only have one POS in one syntactic analysis, annotation exists outside that analysis, and one token can have multiple POS tags. We argue that *annotation reification* is necessary, which means the recognition of Annotation as a first class object that abstracts over all domain specific objects like POS tag.

| id | uri | user | label |
|---|---|---|---|
| 1 | /tokens/1 | john | noun |
| 2 | /tokens/1 | jane | verb |
| 3 | /tokens/1 | jane | noun |
| 4 | /tokens/1 | john | verb |

The above example is ambiguous regarding the task: is this collaborative correction, or dual annotation? At LDC we model assignments with *kits*, or put another way, a *kit* is a model that represents the unit of assignment. If we wanted to represent our current example as dual annotation, we could do the following.

| id | uri | user | label | kit |
|---|---|---|---|---|
| 1 | /tokens/1 | john | noun | /kits/1 |
| 2 | /tokens/1 | jane | verb | /kits/2 |
| 3 | /tokens/1 | jane | noun | /kits/2 |
| 4 | /tokens/1 | john | verb | /kits/1 |

Here kits are connected to single users, but that isn't a requirement either. Dual annotation and collaborative correction could coexist. At LDC we have several abstractions of this sort, the details of which are not only outside the scope of this paper, but not necessarily amenable to other organizations. Here we argue that *some* such abstraction is necessary, and the kit will serve present purposes.

The reification of annotation is in some sense passing the buck, since the complexity of an annotation task has to be represented somewhere. However, it is an important step in a RESTful design. At the level of communication, there are no annotation operations, only HTTP requests on annotation resources. We'll return to the hidden complexity below.

# 6. Hypermedia

Hypermedia is the use of URLs that a machine can resolve to receive further media. Returning to our JSON representations, let's examine annotations.

## 6.1. Linked Data

Hypermedia is basically synonymous with Linked Data, RDF, etc. Consider our JSON annotation.

```
{
  "id": 1,
  "uri": "/tokens/1",
  "user": "john",
  "label": "noun",
  "kit": "/kits/1"
}
```

This is still just JSON, but we'd actually like to use JSON-LD[9], which we can do with a few minor changes (some necessary, some convenient).

```
{
  "@context": "/context",
  "@id": "/annotations/1",
  "uri": "/tokens/1",
  "user": "john",
  "label": "noun",
  "kit": "/kits/1"
}
```

The conversion to JSON-LD has two primary benefits. First, it establishes a protocol for machine interpretation of the document via the "context", essentially a schema for the document. Second, it provides the basis for hypermedia, since the context indicates which strings are actually links that can be followed to access another resource. A possible context for our example follows.

```
{
"@context":
{
"uri":
{
"@id": "http://vocab.lappsgrid.org/Token",
"@type": "@id"
}
"user":  "http://xmlns.com/foaf/0.1/name",
"label": "http://schema.org/Text",
"kit":
{
"@id": "http://www.ldc.upenn.edu/kit",
"@type": "@id"
}
}
```

The context combines both standard and custom types, ranging from the well-known FOAF[10] ontology to custom LDC types (also see the WS-EV in the section on LAPPS Grid). The @type: @id key/value pair also indicates the hypermedia links, so a machine can expect to follow /kits/1 (perhaps at http://www.ldc.upenn.edu/kits/1) and receive a representation of a kit. We won't flesh out the context for other types in this paper, in order to save space, but will continue to include the URI in documents.

[9] http://json-ld.org/

[10] http://www.foaf-project.org/

At the highest level, annotation then becomes the use of GET and POST to exchange JSON-LD documents with this context. Since POST requests typically use URL-encoded strings, and JSON is a serialization to a string, it makes sense to embed the document in a single parameter, which would look like the following, if we assume the only key/value pair is label: ”noun”.

```
POST /annotations
Accept: application/ld+json
json=%7Blabel%3A%22noun%22%7D
```

The response to this request might be identical to the response from the following.

```
GET /annotations/1
Accept: application/ld+json
```

For readability, we won't continue to URL encode request bodies here.

### 6.2. Collections

While creating annotations one at a time is reasonable, reading them one at a time is not. This raises the implementation of *collections*, which will be of particular interest when we return to source data representations. Here we will simply illustrate the concept by reading all the annotations for a specific kit.

```
REQUEST
GET /annotations?kit_id=1
Accept: application/json
RESPONSE
{
  "items":
  [
    {
      "@context": "/context",
      "@id": "/annotations/1",
      "uri": "/tokens/1",
      "user": "john",
      "label": "noun",
      "kit": "/kits/1"
    },
    {
      "@context": "/context",
      "@id": "/annotations/4",
      "uri": "/tokens/1",
      "user": "john",
      "label": "verb",
      "kit": "/kits/1"
    }
  ]
}
```

This mimics in part a personal standard called Collection+JSON[11].

[11] http://amundsen.com/media-types/collection/

## 7. The Importance of Standards

The use of standards and widely used software is an important part of our infrastructure, because they reduce costs, but they are a double edged sword. Any particular implementation represents an investment, and the risk of the investment is that it will have to be re-implemented in the face of changing circumstances. Return on investment is limited by the time it takes for the community to replace current practices, since change and reimplementation are inevitable. This issue has to be addressed with almost every concrete application or use case.

HTTP is an example of a standard not likely to disappear tomorrow. Web APIs however have not reached maturity and practices are likely to change. Standards must be used with reservation, to strike the right balance between interoperability and return on investment. XML and SOAP have waned in popularity in favor of JSON based APIs, but this trend is not mature yet, and there is a plethora of specific formats. Personal standards like Collection+JSON and Hydra[12] address important concerns (collection representations and hypermedia controls respectively), but are new and may not receive wide acceptance. JSON-LD seems to have momentum, builds on RDF, and with its context mechanism provides some relief to changing semantics (assuming clients use it).

The LDC is therefore cautiously rolling out such ”standards” in its APIs, without presuming a single standard of *representation*. The best we can do now is adopt the standard of *protocol*, meaning the use of HTTP and hypermedia, which allows clients to reason in the face of change. This has been the way the Web has evolved, as the response to any particular URL changes over time.

## 8. LAPPS Grid

The LDC is participating in the NSF Language Application Services (LAPPS) Grid project (Ide et al., 2014) which is addressing most of the same concerns as this paper through a different perspective. Not only is the project aimed at creating collaboration, it is a use case for LDC's collaborative model, since the Grid can be seen as a single entity with which LDC's software must collaborate. LAPPS aims to achieve efficient collaboration in the following ways.

The current paper promotes no shared software, beyond the use of HTTP servers. We promote design criteria or procedures. This approach is robust to change, but puts the burden on the individual to implement communication software. The LAPPS Grid uses a software package that individuals can adopt to alleviate that burden. The average user or organization may wrap local software or data as a service using LAPPS software, and therefore immediately become a collaborator with anyone that's also on the Grid. LDC is wrapping existing services in the Data Node API created by LAPPS developers so that the same resource accessible via an HTTP request can be accessed via the Grid.

The LAPPS project adopted JSON-LD (among other standards) to integrate with the Linked Data movement, and LDC follows their lead in APIs accessible outside the Grid.

[12] http://www.markus-lanthaler.com/hydra/

LAPPS has developed a Web Service Exchange Vocabulary (WS-EV)[13] to serve as an interlingua among services which don't share input and output formats. The URL for the token type in the previous examples comes from this vocabulary.

## 9. Complex Resource Transfer

Transfer time across the web is a significant concern for any architecture. Not only do large representations take time due to the number of bytes they contain, but if they are produced dynamically, the runtime of their creation may be significant. On top of this, a server must serve multiple users concurrently. Therefore an architecture which simply returns the obvious representation for a resource may be a terribly inefficient one. If the client is an annotation tool, then it doesn't need much data at one time anyway, due to human limits. The problem exists for both source material and annotations. The POS tagging of a corpus would be much larger than the corpus itself due to the overhead of the object representations. Finally, because objects contain other objects, it's unclear how much embedding is appropriate. Imagine the serialization for an SMS conversation. It might contain a list of message objects, which in turn contain token objects. Message objects could contain speaker objects, or speaker objects could be containers for all the messages for that speaker.

As a rule of thumb, we use *two-level embedding*. The collection pattern described earlier is essentially a resource with a list of resources of specific type, leaving open the question of further embedding. We can modify this concept to treat a resource as a potential collection of arbitrary types. As an example, let's define a kit representation to contain all its annotations, plus some other attributes.

```
{
  "@context: "/context",
  "@id": "/kits/1",
  "user": "john",
  "annotations":
  [
    "/annotations/1",
    "/annotations/4"
  ]
}
```

In this representation, values may be URIs or lists of URIs, but no further embedding occurs, leaving it to the client to following the URIs for more data. Smart clients will do so selectively to increase efficiency. Standard link relations like "next" and "prev" can also be used to break up large collections into parts, as is commonly done with HTML and search results.

## 10. Authentication and Authorization

Effective collaboration isnt possible without thorough means of authentication and authorization. Concurrency is key to collaboration at scale, and servers receiving thousands of concurrent requests must identify users (or agents) and grant access to exactly those resources and services permitted. In addition, annotations must be associated with annotators for practical purposes.

While this problem is never simple, the current architecture defines the problem in a simple way, so that reasoning becomes straightforward. Resource representations are exchanged with GET and POST primarily, so we express constraints in those terms. Here we walk through an example as it would be handled by webann. Lets assume Jane attempts to annotate John's kit.

```
POST /annotations
Accept: application/json
Cookie: jane's-secret-key
{
  "@context": "/context",
  "kit": "/kits/1",
  "uri": "/tokens/1",
  "label": "adj"
}
```

Note the use of the Cookie header to pass a secret key. This key combines the notions of username, password, and session, and identifies John as making the request. Note the document lacks the "user" field, because that's filled in based on the secret key, ensuring the annotation matches the current user. Data integrity could be enforced in different ways at this point. Since kits represent who they are assigned to, a user mismatch could be detected and an error thrown. Another approach is to allow this annotation to be created as normal, because whether this is malicious or collaborative can't necessarily be determined from the information at hand. This would require the request for a kit representation to enforce data integrity by excluding inappropriate annotations. The best approach depends on over all system design, but the general model is simple and flexible.

## 11. Structural Annotations

While sequential annotations are efficient in some respects, they are inefficient in others. There is usually some sense of "current" annotations, which is some combination of the annotations at the tail end of the sequence. But the original represent of a token model/table with a field/column is much more efficient for storing the current state, and could coexist with the sequential annotations. However, "current" itself is not well defined. In the above example, John and Jane both have a current version of their POS label. We again need some sort of reification of an annotation, which we'll call a *node*. Consider the following table, based on the same example.

| id | uri | user | label | kit |
|---|---|---|---|---|
| 1 | /tokens/1 | jane | noun | /kits/2 |
| 2 | /tokens/1 | john | verb | /kits/1 |

These are just rows 3 and 4 from the annotations table, but now they represent nodes, structural rather than sequential annotations. From the users' point of view, they are the current annotations. The choice of the word "node" here

[13] http://vocab.lappsgrid.org/

reflects the generalization over any sort of graph of data. Here the concrete model *node* is used as an abstraction over whatever data structure is most useful, and we associate it with our kit abstraction.

annotations

| id | user | kit | node | operation | value |
|---|---|---|---|---|---|
| 1 | john | /kits/1 | /nodes/1 | update | noun |
| 2 | jane | /kits/2 | /nodes/2 | update | verb |
| 3 | jane | /kits/2 | /nodes/2 | update | noun |
| 4 | john | /kits/1 | /nodes/1 | update | verb |

nodes

| id | kit | source | target |
|---|---|---|---|
| 1 | /kits/1 | /tokens/1 | verb |
| 2 | /kits/2 | /tokens/1 | noun |

Here "update" does refer to the Update of CRUD, and we might ask why not allow the API to operate on this table directly. We don't want to abandon the concept of CaReful annotation however. The annotations (sequential representation) represent complete, immutable information, while the nodes (structural representation) represent mutable caches, which in fact could be completely erased and reconstructed from the annotations. The operation column in the current example is similar to RPC described above. The inventory of operations is intended to be constrained to CRUD when possible, but sometimes it is expedient to introduce a custom operation. Since the annotations are the primary representations which are CaReful, allowing the nodes to be CRUDy is not risky. Data integrity problems, once discovered, can be remedied via the annotations table.

## 12. RESTful Annotation Tools

Thus far we have focused on APIs and machine interpretable semantics. However, a key component of LDC's current infrastructure is a web-based annotation tool framework, since even LDC employees may not work in the office. These tools use the API described above. At the highest level, all the annotation tools are a *single* tool, and the notion of tool really becomes a human based categorization. The initial request made by an annotator, via their web browser, includes a kit parameter, therefore accesses a particular set of nodes, and the tool is just a representation of the structural annotations (nodes). The browser then continues to access the API in the manner described above in response to user input. This approach drastically reduces (without eliminating) the amount of custom code that a tool requires. This reduces programmer effort, and therefore cost.

A great deal of power has been delegated from developer to manager, which not only makes local management more efficient, but remote management easier. Workflow selection, work assignment, progress tracking, etc., are all possible through the web browser. With a WYSIWYG editor, a variety of annotation tools can be created from scratch in the browser, and many properties can be modified at runtime since they are parameters in a database. One application of standards here is the use of CSS and HTML tags for customization where possible, which are widely understood by non-developers, further minimizing the effort required by developers. The connection of this editor to the API describe above lies in the structural annotations (nodes). While the user/manager continues to operate within the metaphor of "tool", they are actually defining the structural relationships over which the sequential annotations operate. We hope to see this "meta-tool" used outside the LDC as another example of what's possible when web-services are fully leveraged.

## 13. HLT Pipelines

A number of collaborations are underway that combine annotation tools with HLT technologies like Speech Activity Detection. These pipelines may be deployed through the Grid or solely through WebAnn. Currently the pipelines are not exposed over the Web, but nevertheless follow REST principles like universal identification and CRUD-like operations. Forcing what clearly is an open ended set of operations into a resource oriented framework fosters efficiency and replicability, not to mention eventually exposure over the Web. Here standards are less clear, but there are broad communities working on the same domain, like CLARIN[14] and META-NET[15]. For now, simply following RESTful principles may be sufficient to participate in such communities.

Pipelines present particular problems when distributed. Imagine a request to tokenize Gigaword, and how long the client would have to wait for a response. Web communications must also deal with dropped connections, browser refresh, web server overload, etc. Constructing a pipeline across sites exacerbates these problems. There's also the problem of new input being added after the initial output is delivered.

When can apply RESTful principles to these problems as well. In the following we'll gloss over many details of actual representations to focus on the problem. We'll assume a pipeline is something a client creates as follows.

```
REQUEST
POST /pipelines
Accept: application/json
{
  "@context": "/context",
  "process": "/pos_tagger"
  "input": "/documents/1"
}
RESPONSE
{
  "@context": "/context",
  "@id": "/pipelines/1",
  "process": "/pos_tagger"
  "input": "/documents/1"
  "output": "/kits/3"
}
```

Note how the request and response are basically the same document, but with additional fields that reflect the creation

[14]http://www.clarin.eu/

[15]http://www.meta-net.eu/

of a new pipeline resource, as well as a new output resource. Consider this request to occur at time $t_0$, while two requests for the output occur at $t_1$ and $t_2$ as follows.

```
REQUEST
GET /kits/3
Accept: application/json
RESPONSE
{
  "@context": "/context",
  "@id": "/kits/3"
}

REQUEST
GET /nodes/3
Accept: application/json
RESPONSE
{
  "@context": "/context",
  "@type": "/...",
  "@id": "/kits/3",
  "nodes": [
  {
    "@id": "/nodes/1",
    "source": "/tokens/1",
    "target": "noun"
   }
   ]
}
```

At $t_1$, the pipeline hasn't actually executed yet, but a resource for its output does exist. The *representation* for that resource is valid but impoverished, until time $t_2$. This is just par for the course in RESTful communication, but is not the norm for NLP pipelines. It is a sort of asynchronous processing, especially if you consider that the input and output may not just have empty and full states, but incremental states as well. But URIs and hypermedia make the model straightforward.

Architecture designers may need to change their thinking from the assumption that resources are complete when they can access them. This has been important in a completely internal sense at LDC, where a data collection occurs over time. Waiting for collection to finish before processing any data is impractical. Processing the entire collection multiple times isn't much better. The optimization is to incrementally process the input, which leads to something like a RESTful design.

## 14. Licensing

For raw data, licensing can be treated as a subproblem of authorization when the resource is requested. Pipelines add considerable complexity to the problem, as there are multiple components with different licenses that are distributed across sites. A well defined pipeline may fail at arbitrary points due to licensing. The asynchronous model described above decouples execution from access, since the resources are identified before meaningful execution takes place. Licensing again reduces to authorization on individual resources. If licensing cannot be determined due to a gap in the database, it doesn't block pipeline creation or execution necessarily, only the retrieval of output.

## 15. Conclusion

Here we have focused not on standards that must be adopted, but on a set of procedures that organizations can follow to collaborate efficiently and reduce cost. In many ways we are arguing for the path of least resistance. Open standards should be adopted and developed, but in proportion to their probable return on investment. For example, JSON APIs are a safe bet, but the various extensions to the format should be used cautiously. At the same time, available technologies should be used rather than reinvented, for example, relying on existing media types whenever possible. Distributed architectures that depend on URIs and hypermedia bring great flexibility in a changing world. URIs that are actually URLs should be used when reasonable so that true hypermedia exists. While complex programming ultimately involves complex operations, hiding these operations at the lowest levels simplifies collaboration. APIs should limit the inventory of operations whenever possible, shifting the burden of complexity to representations and hypermedia. Finally, cost reduction can be achieved by assuming internal architecture follows the same patterns as external architecture.

## 16. Acknowledgments

This work was supported by National Science Foundation grants NSF-ACI 1147944 and NSF-ACI 1147912.

This material is based upon research supported by the Defense Advanced Research Projects Agency (DARPA) Contract No. HR0011-11-C-0145 and Air Force Research Laboratory agreement number FA8750-13-2-0045. The U.S. Government is authoized to reporoduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Air Force Research Laboratory and Defense Advanced Research Projects Agency or the U.S. Government.

## 17. References

Ide, N., Pustejovsky, J., Cieri, C., Nyberg, E., DiPersio, D., Shi, C., Suderman, K., Verhagen, M., Wang, D., and Wright, J. (2014). The Langauge Application Grid. In *Proceedings of the Tenth International Language Resources and Evaluation (LREC14)*, Reykjavik, Iceland. European Language Resources Association (ELRA).

Parker, R., Graff, D., Kong, J., Chen, K., and Maeda, K. (2011). English gigaword fifth edition.

Richardson, L. and Amundsen, M. (2013). *RESTful Web APIs*. O'Reilly, Sebastopol, CA.

Wright, J., Griffitt, K., Ellis, J., Strassel, S., and Callahan, B. (2012). Annotation Trees: LDC's Customizable, Extensible, Scalable, Annotation Infrastructure. In *Proceedings of the Eighth International Language Resources and Evaluation (LREC12)*, Istanbul, Turkey. European Language Resources Association (ELRA).