# Resource Informatics or Production Core

## Specific Aims

**Aim 1.** Develop, use, and extend data exchange standards such as file formats, APIs, and controlled vocabularies.

**Aim 2.** Assure the quality and understandability of genomics data sets as they are imported.

**Aim 3.** Maintain software quality through a combination of automated and manual testing and best practices.

**Aim 4.** Provide a fast, stable computer hardware environment for Genome Browser developers and users.

## Research Strategy

### *Aim 1. Develop, use, and extend data exchange standards such as file formats, APIs, and controlled vocabularies*

#### Significance

Most research labs use data and programs from a wide variety of sources. As more individuals and groups develop biomedical databases and software, it becomes increasingly important for them to establish and use common standards rather than each representing their own data in an independent fashion. Standard file formats, APIs, and controlled vocabularies allow users to combine data from different sources with less effort, and are particularly helpful for groups such as our own that are focused on data integration.

Standards can reduce our workload significantly, if intelligently designed and widely adopted. The ideal standard is simple to understand, easy to implement, flexible enough to have wide applicability, and stable across years, if not decades. The worst standards are complex, narrow, and unstable, and compete against other standards in the same domain. Most standards fall somewhere in the middle of the spectrum.

At UCSC we are involved in both defining standards and supporting standards developed elsewhere. We are cognizant of the difficulties caused by the existence of multiple competing standards, which in extreme cases can be worse than having no standards at all, and therefore selectively develop new file formats only when no existing format fits the need. For instance, we abandoned the internally developed short read alignment and variant call formats that we used during the early days of next-generation sequencing when the binary alignment/map format (BAM) and variant call format (VCF) were released and adopted as standards, rather than investing in further development of our internal formats.

During the past sixteen years we have developed and promoted a number of formats to fill unmet needs in advance of other formats, and most of these have become widely adopted. For example, we developed MAF, a format for multiple genome alignments, to address limitations with previous multiple alignment formats that were originally designed for proteins and could not accommodate forward and reverse strands, inversions, and long genomic sequences. We developed BED, a simple line-oriented format that can efficiently accommodate a wide range of genomic annotations. To enable the viewing of large, remotely hosted files at a variety of scales, we developed the BigBed format that handles a wide range of annotations like BED, and the BigWig format that efficiently handles numerical data associated with genome coordinates. In conjunction with the ENCODE Consortium, we defined specific variants of BigBed – NarrowPeak and BroadPeak – for ChIP-seq and DNAse-seq data.

Another widely adopted set of standards we have developed at UCSC is that defining the track hub and assembly hub systems, which are useful to scientific consortia and labs who need to view many related genomics files in an organized manner. Track hubs, which are essentially directories of genomics files accessible via an http, https, or ftp site, are ideal for viewing data from many individuals, tissues, and even assays, including data stored in Simple Storage Service (S3) systems in the cloud through providers such as Amazon. A supplemental text file, trackDb.txt, allows the hub provider to configure labeling and coloring information on the files, and to arrange the files into a hierarchy of tracks. Our track hub system, which has connected to nearly 45,000 hubs, is currently used by many large consortia such as Roadmap Epigenomics and ENCODE, as well as many smaller groups who want to display their data on the UCSC browser. The track hub display standards have been adopted by several other genome browsers, including the Dalliance (1), Ensembl (2), and WashU Roadmap EpiGenome (3) browsers.

We have recently expanded the track hub system to support assembly hubs, which contain files with genome sequence information in addition to the annotation tracks. Through the use of assembly hubs, users can display and annotate genome sequences for assemblies not hosted in the UCSC Genome Browser database.

In addition to well-documented, standard file formats, we also release data through SQL database tables, both through an online public MySQL database and via tab-separated table dumps. Our database schema, which can be viewed interactively through our Table Browser tool, includes names and descriptions of each field in each table. The SQL language provides a powerful, well understood mechanism for querying our databases, and many third-party bioinformaticians make good use of it.

Most of our code is developed using C, a simple, stable language that has regained its status as the most popular programming language, overtaking Java. We place the most generally useful parts of our code into libraries with clearly documented interfaces. These libraries include code that reads and writes files in standard genomics formats, rapidly finds overlaps and correlations between two different data sets, loads annotations within a specific region of the genome, aligns two sequences, and performs many other common and useful bioinformatics programming tasks. External users have written wrapper application program interfaces (APIs) for some of these functions in many other languages, including C++, Python, and Perl. Many groups also interface with our file formats and databases through command-line interfaces on UNIX and Linux systems.

While much of our work focuses on the efficient display and analysis of large data sets, we also invest considerable effort in maintaining high-quality metadata, i.e., the sets of data that describe the genomics data. It can be a challenge to find which of thousands of genomics files are relevant to a particular biomedical question; therefore, ideally these metadata are broken into logical parts that can be stored in databases, allowing them to be easily searched by computers as well as browsed by human eyes.

One key to ensuring high-quality metadata is the use of controlled vocabularies, which prevents different scientists from using varying terms to describe the same thing. An extreme example of this is the “sex” field in GenBank, which currently has 233 values. Beyond male and female, there are approximately a dozen terms reflecting real biological differences in form (including hermaphrodites and various fungal mating types), but the vast majority of these 233 values are simply different ways of expressing that a sample is a mixture of two sexes. Currently we work with controlled vocabularies from the Gene Ontology (4,5) and the Sequence Ontology (6) groups within our metadata systems.

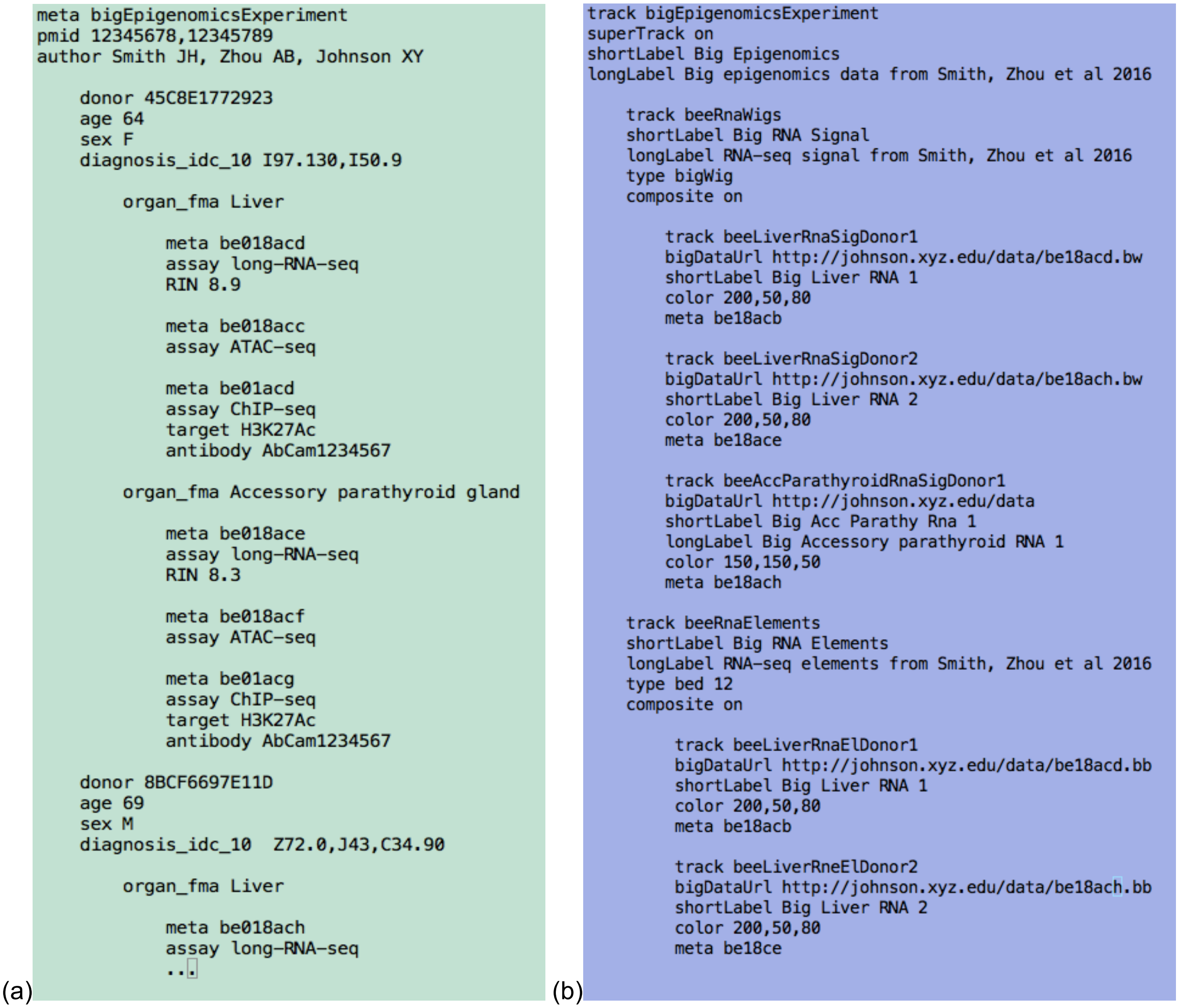
## Innovation

In addition to maintaining our existing data interfaces, we intend to expand the data exchange formats and APIs for our software and databases in three primary ways: extending the track hub system, adding support for more controlled vocabularies, and developing a JSON-based web services API to our databases.

We plan to extend track hubs to accommodate all data types natively displayed on the UCSC Genome Browser, including formats for comparative genomics and the new RNA expression-level displays developed for GTEx that are currently unsupported on hubs. After this extension is complete, we intend to build a track hub containing all of the Genome Browser data. We can reasonably anticipate that new formats for new data types will emerge over the next five years; we will add hub support for those as they are developed.

We also plan to enrich the metadata that can be stored on a hub using a flexible, hierarchical system called TagStorm (Fig. 1) that we developed for ENCODE and have also utilized in a data center we run for the California Institute of Regenerative Medicine (CIRM). This system will allow multiple tracks to refer to the same biosamples and explicitly indicate which biosamples are from the same donor.

The basic idea behind TagStorm is relatively simple. It is a line-oriented format in which each line starts with a field name, followed by a field value. Records are separated by blank lines. The “meta” field, which is indexed, is used to retrieve a particular record. Although tag/value pair formats like TagStorm are logically equivalent to tables, with a tag corresponding to a particular column, they are typically much smaller and easier to read than tables or XML format in instances where there are a large number of tags, of which only a few are used for any particular record. This is commonly the case in biomedical metadata.



(a)

```
meta bigEpigenomicsExperiment
pmid 12345678,12345789
author Smith JH, Zhou AB, Johnson XY

    donor 45C8E1772923
    age 64
    sex F
    diagnosis_idc_10 I97.130,I50.9

        organ_fma Liver

            meta be018acd
            assay long-RNA-seq
            RIN 8.9

            meta be018acc
            assay ATAC-seq

            meta be01acd
            assay ChIP-seq
            target H3K27Ac
            antibody AbCam1234567

        organ_fma Accessory parathyroid gland

            meta be018ace
            assay long-RNA-seq
            RIN 8.3

            meta be018acf
            assay ATAC-seq

            meta be01acg
            assay ChIP-seq
            target H3K27Ac
            antibody AbCam1234567

    donor 8BCF6697E11D
    age 69
    sex M
    diagnosis_idc_10  Z72.0,J43,C34.90

        organ_fma Liver

            meta be018ach
            assay long-RNA-seq
            ..
```

(b)

```
track bigEpigenomicsExperiment
superTrack on
shortLabel Big Epigenomics
longLabel Big epigenomics data from Smith, Zhou et al 2016

    track beeRnaWigs
    shortLabel Big RNA Signal
    longLabel RNA-seq signal from Smith, Zhou et al 2016
    type bigWig
    composite on

        track beeLiverRnaSigDonor1
        bigDataUrl http://johnson.xyz.edu/data/be18acd.bw
        shortLabel Big Liver RNA 1
        color 200,50,80
        meta be18acb

        track beeLiverRnaSigDonor2
        bigDataUrl http://johnson.xyz.edu/data/be18ach.bw
        shortLabel Big Liver RNA 2
        color 200,50,80
        meta be18ace

        track beeAccParathyroidRnaSigDonor1
        bigDataUrl http://johnson.xyz.edu/data
        shortLabel Big Acc Parathy Rna 1
        longLabel Big Accessory parathyroid RNA 1
        color 150,150,50
        meta be18ach

    track beeRnaElements
    shortLabel Big RNA Elements
    longLabel RNA-seq elements from Smith, Zhou et al 2016
    type bed 12
    composite on

         track beeLiverRnaElDonor1
         bigDataUrl http://johnson.xyz.edu/data/be18acd.bb
         shortLabel Big Liver RNA 1
         color 200,50,80
         meta be18acb

         track beeLiverRneElDonor2
         bigDataUrl http://johnson.xyz.edu/data/be18ach.bb
         shortLabel Big Liver RNA 2
         color 200,50,80
         meta be18ce
```

**Figure 1.** *Panel (a)* – The start of a TagStorm file describing metadata for a hypothetical experiment involving RNA-seq and ATAC-seq of samples from multiple organs from multiple donors. Child stanzas, which are located beneath parent stanzas and are more deeply indented, inherit tags from parents. Thus the bottom stanza, identified by the meta tag *be018ach*, is associated with tracks that are tagged as liver from a 69-year-old male, and is part of the larger data set submitted by JH Smith and colleagues and associated with two papers in PubMed.

*Panel (b) (blue)* – A fragment of a track hub trackDb.txt file that references the metadata in *Panel (a)*. The trackDb file contains information for labeling, formatting, and displaying the track in the Genome Browser, while the TagStorm metadata file contains information about the biomedical experiment. Multiple tracks, in this case one containing RNA-seq signals in BigWig format and another showing discrete exon-intron structures in BigBed format, can reference the same metadata, in this instance designated by the meta tag *be18acb*.

In addition to the simple tag/value pair structure, TagStorm is also hierarchical. Child records, which are located beneath parent records, are visually indicated by a deeper indentation. A child record consists of those fields it directly defines, as well as any fields defined in a parent or other ancestral record. When both a child and a parent define the same field, the child's value takes precedence within the child record. TagStorm files support as many levels of hierarchy as are needed, which in the case of a single-cell RNA sequencing experiment could easily include four levels: single cell, tissue, donor, and overall experiment. A more conventional experiment might have just three levels, omitting the single cell level. In either case, the TagStorm file format and the code to interpret it are the same. In contrast, most other existing metadata systems define a fixed number of levels, and each level has fields that are specific to that level. For example, GEO (7) has two levels: series and sample. This structure presents two problems. First, much of the design of the experiment is not apparent because the lower, sample level has to accommodate all of the information. Second, the appropriate level for a tag can be ambiguous at times. For instance, in most cases "lab" is a field that is relevant at the series level, but if the data set is from a large consortium, it may be needed at the sample level instead.

In our use of TagStorm for two large projects through our roles as the CIRM Data Coordination and Management Center and as the ENCODE Data Coordination Center, the uniformity, flexibility, and readability of the format has proven itself. As an added advantage, TagStorm is very similar to the hierarchical, stanza-based trackDb format with which track hub users are already familiar. Users will be able to add a "meta" tag to a trackDb stanza to associate it with metadata in the corresponding TagStorm file.

We plan to associate certain TagStorm tags with particular standard controlled vocabularies, tracking the Open Biomedical Ontologies (OBO) (8) project as well as continuing to work with the Sequence Ontology and Gene Ontology projects. For medically related metadata, we plan to capture diagnostic codes, which have become almost universally adopted due to their usage for insurance billing purposes. As we gather more phenotype information, we plan to adopt the Human Phenotype Ontology (HPO) (9), a format widely used for encoding patient phenotypic descriptions.

For programming standards, we plan to add a JSON-based, RESTful web services API to our system. Such APIs have become a common, well understood part of the website programming ecosystem. JSON is the native representation for data in the JavaScript language that is built into all modern web browsers. A web services API is a convention where URLs are constructed to represent a query, and a JSON object is returned in response. If the API is RESTful, the same query always yields the same response, so that the query and result can be cached for better performance by the same systems that cache web pages. JSON web services APIs are relatively easy for programmers to use and test within their own web browsers, and well-tested, supported libraries exist for the APIs in Python, Perl, and R. This wide range of language support makes these APIs useful in genomics analysis pipelines. The native Javascript support is useful for people developing alternative interactive displays of genomics data. We plan to use the API ourselves in the development of a genome browser for mobile applications (Resource Project component, Aim 1).

There are many details to a particular web services API that must be well defined to make it useful, e.g., which fields are returned in a JSON object and the particular format of a query URL. We plan to work with the Global Alliance for Genomics and Health project (GA4GH) (10,11) to define an API that is as broadly compatible with other genomics databases as possible. We intend to support the GA4GH APIs in three major ways. We will create a program that can serve data from a track hub via the GA4GH APIs, as well as a program that can serve data from the Genome Browser internal databases via the API. We will also extend the Genome Browser so that it can act as a client for visualizing data from GA4GH API servers.

## Approach

Our overall approach to maintaining and extending data exchange standards is to track the development of emerging standards and important new tools, which often create *de facto* standards, and integrate these into our resource when appropriate. We plan to continue the development of the track hub and assembly hub standards that we have developed, particularly with regards to improving the metadata features, including controlled vocabulary support. We will work with GA4GH in the definition of web services APIs, and will implement both clients and servers for these APIs.

Ideally, a data exchange standard should include readable but detailed documentation, numerous examples of items conforming to the standard, and a test set that can validate compliance to the standard. It is also beneficial to have code libraries that work with the standard and can be used by other programs. We intend to conform to these ideals for our internally developed standards. For externally developed standards, we will offer feedback and (if necessary) assistance with documentation and implementation to help drive the standards towards these ideals. Specifically, we will develop C language libraries that implement the standard if none exist, generally a necessary step for integrating a standard into our system.

In conjunction with extending track hubs to include metadata, we will develop documentation that describes the valid tags and the controlled vocabulary, if any, associated with a tag. We plan to develop both web-based and command-line validation tools to check metadata files and report errors, and will incorporate these into our existing overall track hub validation systems. We will improve the documentation of our C libraries and utilities that already interact with the TagStorm metadata format, and will develop Python and Javascript library interfaces as well. We will also develop tools that can convert between TagStorm and a recursive JSON format; although the latter format is more difficult for humans to work with because of the required quotation marks, commas, and brackets, it interfaces better with computers. Finally, we plan to develop and publish tools for indexing and searching TagStorm files, which we will incorporate into the Genome Browser hub and track search mechanisms.

In our development of web services APIs, we plan to closely collaborate with Benedict Paten, Director of the Big Data to Knowledge (BD2K) Center for Big Data in Translational Genomics (12) within the UCSC Genomics Institute, which implements GA4GH standards and with whom we have an established working relationship. As a first concrete step, we will work together to build a GA4GH-conforming server for track hubs. Because this work can be done somewhat independently of the large UCSC Genome Browser code base, we can more easily utilize the engineering efforts of Benedict's group as no familiarity with the Genome Browser or its code will be required. Once we can serve track hubs via GA4GH APIs, our own group will independently extend the Genome Browser to be a GA4GH client, able to display data served by this protocol. We anticipate that this process will generate much useful feedback for the GA4GH group and potentially suggest possible extensions to their specifications. We will then test the Genome Browser against other GA4GH servers external to UCSC and, if necessary, modify the Genome Browser to accommodate data from these sources. Finally, once we have completed all the work needed to display the entire Genome Browser data set using a track hub, we will put this hub on a GA4GH server.

## *Aim 2. Assure the quality and understandability of genomics data sets as they are imported*

### Significance

Scientific progress critically depends on the quality and understandability of published data. Though the scientific method is sufficiently robust that most errors are eventually corrected, erroneous data can cost researchers a large amount of time and effort. Undocumented and cryptically documented data also cause many problems, increasing the time and frustration involved in learning a data set, and at times resulting in people simply repeating experiments they don't understand. Therefore, we invest considerable effort in assuring that the data we display is of high quality, and that our documentation is written clearly and directed at an appropriate level for our user base.

The focus of the research community on new findings can unfortunately result in noise and artifacts being mistaken for novelty at times. Experienced scientists have learned to be skeptical of investing in a result "too good to be true". Conversely, using too much caution in interpreting unexpected results can lead to being scooped in publication by a less cautious scientist. Our goal is to achieve a good sense of balance in evaluating the data imported into the Genome Browser.

There have been a few instances in which we have we have published interesting, but unexpected, data on the browser after our own careful analysis, with the data later incorporated into a major scientific publication. One such example involves the discovery of ultraconserved regions in human, mouse, and rat. While aligning the first mouse genome assembly to the human genome, we identified many large regions that were identical between the assemblies, a highly unexpected finding given the long and separate evolution of the two species from their common ancestor. Although many of these regions were attributed to contamination and were removed, we agreed to retain a small fraction of shared sequence that the Broad Institute determined to be connected strongly to sequence unique to the mouse assembly. Subsequently, when many of these conserved regions were later found to be conserved at 100% base identity in the rat, we had the confidence to publish on the ultraconserved regions (13). In this instance, if we had been slightly less skeptical in our quality checking, the mouse assembly would have been released with hundreds of stretches of human contaminants, leading to false conclusions and loss of confidence in the sequence efforts. If we had been too skeptical, a major scientific discovery would have been delayed. In another such example, our decision to include long noncoding RNAs (lncRNAs) with strong evidence in our gene set, in response to comments expressed by Mary-Claire King on our advisory board, helped to support an explosion of research on that gene type.

These two instances notwithstanding, an unfortunately large percentage of novel results can be attributed to contamination, technical artifacts, or simple errors from dropped tubes, typographical errors, and data processing problems. By investing significantly in people and computer programs aimed at detecting such errors in the data imported into the UCSC Genome Browser, we are able to save much wasted time and effort by the scientific community in general and our users in particular.

## Innovation

During the sixteen years of development of this resource, we have established a robust set of procedures and programs that we will apply to the integration of new data sets introduced by this proposal. We have identified a few areas in which we can reduce costs through procedural changes and by adopting or developing new quality assurance tools. However, most of the innovative work in this aim will be focused on encouraging high quality standards for data distributed through track hubs by supplementing our existing format checks with timely, useful automated feedback to the data providers based on data comparisons.

As our data integration and quality assurance processes have matured, we have placed a greater emphasis on staffing these teams with individuals trained in biology rather than computer science. Much of what can be profitably automated by a computer program has already been accomplished, and it is therefore more cost effective to hire biologists skilled at identifying laboratory-generated artifacts via spot checks than it is to hire computer scientists at an equivalent level to write more tests. In the case of repeated errors, the biologists refer the issue to one of the computer scientists for addition to the automated tests.

To expand the automated checking of track hubs, we plan to ask hub providers to complete an online form that will help us better understand their expectations of the data. For example, we might ask whether a data set should overlap or avoid certain regions such as coding exons, promoters, areas of open chromatin, common SNPs, highly conserved regions, 5’ UTRs, 3’ UTRs, mitochondria, and the sex chromosomes. Based on this information, we will be able to compute the enrichment of the data in these specific regions, rather than on the genome as a whole, and can then flag any tracks where the enrichment is not as expected. We will also ask track hub providers to identify at least one native track in the Genome Browser that they expect to correlate or anti-correlate with the tracks in their data set. Finally, we will generate ten random regions within their data set and ask the providers to visually spot check them. Once this new validation process is in place, we plan to require that all track hubs pass these checks to be listed on our site.

## Approach

Over the years we have fine-tuned our “wrangling” process for importing data sets into the browser. Key components of the process include establishing a good initial relationship with the data provider, running automated tools to check for internal consistency and compare to similar existing data sets, and spot-checking the data. In parallel, we work with the contributor to write descriptive pages for the data set aimed at the comprehension level of a first-year graduate student in a biomedical field. After we have completed the tests and documentation, we share our findings with the data provider. If we are both in agreement that the data is of high quality and is represented accurately on our site, we release it to the public.

Forming good relationships with data providers is essential. Because they have the scientific background to understand their data better than we do, they can best determine if we have introduced any obvious errors while reformatting the data and editing the documentation. When present, many errors are discovered simply by having the data providers view the data in our browser, which may have different strengths and weaknesses from the visualization tools the providers use and may present a broader view of the data.

We apply an array of standard tests when checking the quality and consistency of a data set. We overlap the data with an established set of regions (described in the Innovation section). If the data set is an updated, improved version of data already present in the browser, we have tools to compare the two data sets and highlight differences, which we then investigate and (if necessary) discuss with the data provider. We routinely spot check ten randomly selected sites, as well as five sites selected based on biological interest and intuition, and alert the data provider of problems. When hidden problems are uncovered in the original data set, we report back to the data providers for long-term correction, while fixing or filtering out the errors ourselves in the short term.

Other than the data from major consortia such as ENCODE, which we may host pre-publication, most of the data we display has an associated scientific publication that serves an additional validation resource. We often compare data sets with publication figures, and incorporate information from the paper into our documentation.

## *Aim 3. Maintain software quality through a combination of automated and manual testing and best practices*

### Significance

Much like bad data, bad software can significantly impede scientific progress. In an era where most genomics data flows through elaborate data analysis pipelines, bad software can actually create bad data. It can also present data in a misleading fashion, causing data points to deceptively disappear or appear associated with the wrong experiment. Even if software produces and displays a correct result, slow performance and a poor user interfaces can make it cumbersome or impractical to use.

Similar to the quality practices associated with our data integration process (Aim 2), we have developed a solid set of procedures and programs to ensure the high quality of our software, which we describe in the Approach section below. As a result, we are able to offer a highly stable, reliable set of tools to the research community.

### Innovation

To implement the software enhancements proposed in the Resource Project component, we anticipate that we will expand the amount of software development done in coding languages other than C, including languages such as JavaScript that are interpreted rather than compiled. When possible we plan to leverage existing, well-tested library code developed by external groups, in part because we have collectively less experience in these languages, and also because testing is generally more difficult in interpreted languages, which in many cases (such as JavaScript) must be tested on a wide range of web browsers. For interpreted code that originates in our group, we plan to use static program analyzers to duplicate as much as possible the automatic checking that is done during compilation in languages such as C. Though JavaScript is a challenging programming environment, we feel that the enhanced web page interactivity that it enables is worth the effort.

### Approach

We use a wide range of techniques and processes to ensure the high quality of our software. These include the regular execution of several levels of code-checking, the adherence to software development and quality assurance best practices, and collecting input from our active community of Genome Browser users. The end result is software that functions accurately and efficiently and provides a reasonable user interface.

Whenever possible we follow an incremental approach to development, in which software is developed in small pieces that incorporate testability as part of the design. This ensures that we do not invest too much bandwidth on a project that turns out to be intractable, and prevents simple improvements from being delayed while complex improvements are implemented. We maintain a three-week software release cycle. Changes that require more time to implement are developed on a separate branch in our source code control system (git) and then merged into the main branch when ready, accompanied by additional testing. The incremental development practice is applicable to any language. Though it sometimes requires significant thought in the early, architectural phases of design, it is hard to overstate the rewards of a design that can be tested in small pieces as it is built, and gradually extended to accommodate new features.

All but the smallest of our programs are built from smaller pieces known as subroutines, procedures, or (in the C language) as functions. A C function can have side effects, i.e., when a function is run a second time with the same inputs, it can produce a different output from the first run. To compound matters, *other* functions can have different outputs before and after a C function is run. This creates three major classes of problems. First and foremost, it makes it more difficult to understand the code, since merely reading that a function is called does not indicate what the function is doing. Second, it means that the function cannot be used in multiple threads in parallel. Since most computers now support many simultaneous threads, this severely limits performance. Third, side effects prohibit the use of code in recursive programming techniques, where a function calls itself. Recursive techniques simplify many problems; for instance, they are key to the flexible notion of hierarchy in the TagStorm metadata format.

Functional programming is a technique in which subroutines are not allowed to have side effects, behaving more like the mathematical notion of a function. While C language functions are not required to be functional in this sense, several features in the C language make it relatively easy to adopt a style that ensures that functions are indeed functional. Chief among these is ensuring that all global variables are read-only, which in effect restricts them to being constants. This can severely limit a program. In particular it is hard for a program to allow different options, such as which data tracks to display, when taking a pure functional approach. Our pragmatic compromise is to allow programs to write to global variables during the initialization phase, after which all variables are considered read-only except those local to a function. In practice we limit our use of global variables and use naming conventions that easily identify them.

In object-oriented programming, objects created in the code attempt to parallel those with which the program is working. For example, within the UCSC Genome Browser, major objects include tracks, items within a track, and DNA sequences. An object consists of the data that describes it and the functions that manipulate it. Objects can be polymorphic, a characteristic that can be observed in the Genome Browser tracks. In some cases it makes sense to arrange classes of objects into hierarchies with the more abstract general-purpose classes (such as track itself) on top, the specialized classes (such as those representing discrete regions of the genome) in the middle, and classes representing specific items (such as SNPs) at the bottom level. The specialized classes can typically reuse much of the code and data structures of the more general classes. Most programs have classes for internal, computer-oriented items as well; for example, we have rich classes for working with line-oriented text files and HTML pages.

The C language does not directly support classes in the manner that many modern programming languages do. However, as with functional programming, it is possible to adopt conventions that enable object-oriented programming in C. In our convention, we create a C structure (a construct that bundles together multiple variables) that represents the data of the class and carries the class name. Functions that operate on the class (called messages in object-oriented terminology) are named using the class name as a prefix, and take the associated class data structure as their first parameter. In a purely functional style, objects can't be modified after they are created. Although we observe this restriction most of the time, occasionally it is simpler and more efficient to modify a data field within an object rather than create and track a separate slightly different object. We can still allow parallel computation in this case by requiring that a function modify only its own object and nothing else. Parallelism can then be safely executed, if each thread works on a separate object.

A program of more than moderate size is built of modules. In C, a module is typically defined by a pair of files: a *.h* file that describes the interface to the module and a *.c* file that implements the module. C places no constraints upon how to modularize the code, but it is good programming practice to place related code in the same module and keep a module as self-contained as possible. By convention we typically put a single object class into a single module. Ideally a module is small enough that it can be read and understood by a programmer in a single work session, but large enough that the programmer is not required to constantly flip between modules to understand the code.

Our engineers follow an established set of coding conventions that guide them towards an incremental, functional, object-oriented, modular style and that help produce code that is readable and unambiguous. We configure our compiler to catch a wide range of errors automatically, including such things as variables that are used before they are initialized, unused variables and functions, function inputs of the wrong type or number, mix-ups of object types in general, empty statements, and differences between *printf* formatting codes and the variables being serialized. We also run automated code analysis tools to identify errors, particularly for non-compiled languages, and when possible we use code generators to implement routine code, for instance code that reads an object from a database table into an object in a computer language.

Our entire project team participates in weekly paired reviews of all code committed to the source code control system during the previous week. This practice nets many benefits. The paired reviews ensure that new code adheres to our coding conventions, keeping our style clean and consistent, and catches a particular set of bugs that are apparent to a human proofreader, such as mixing up X and Y coordinates, but that are difficult for a compiler or other code validator to identify as errors. Reviewers check that appropriate unit tests have been created to provide built-in self-checks for functions, objects, and modules. Finally, the code reviews spread the knowledge of each programmer's work throughout the organization, which encourages reuse of the code by others and makes it easier for one programmer to take over work initiated by another.

We maintain a suite of automated programs that extensively test our code by mimicking the actions of a software user. For example, the test for the main Genome Browser program turns on each track individually, initially displays it at the base level and then zooms out repeatedly until the full chromosome is displayed. Software crashes, error messages, and the number of seconds required for each display are logged. Our quality assurance (QA) group runs these tests and reviews the output prior to each software release.

We have found that it is extremely effective to maintain a QA group separate from our development team and staffed by individuals who are motivated to find bugs—in contrast to software developers who tend to be less enthusiastic about finding problems in their own code. Having a separate QA group allows us to target our hiring at skills specific to software testing and to somewhat reduce our personnel costs, since testers tend to impact the budget less than programmers. In addition to running and reviewing our automated test programs prior to a software release, the QA team manually checks the most important functions of the Genome Browser, tests new features in great detail, and performs random tests that vary from one release to the next. Recurring problems that are uncovered manually may be added to the automated tests if appropriate. The QA group also organizes, reproduces, and logs those bugs reported by our very active user community that have slipped past our extensive testing processes.

Overall we have developed a solid, time-tested process for producing high-quality software that has served us well during the previous grant cycle and reinforces the care, experience, and effort of our project team staff. This emphasis on software quality will become even more important as genomics becomes a discipline used in the clinic as well as in research.

### *Aim 4. Provide a fast, stable computer hardware environment for Genome Browser developers and users*

#### Significance

Our project depends heavily on having a stable hardware environment. Hardware malfunctions can easily make our site unusable or prevent most of our staff from working. Fast hardware, particularly if tuned to the task, can make our site more responsive and consequently improve the productivity of our users and developers. By relying on relatively standard hardware and operating systems, we make it easier for advanced users to mirror and utilize our tools.

#### Innovation

Overall, our existing approach to hardware works well, but we propose two major changes: moving our largest compute jobs to a cloud environment and reconfiguring our web servers for higher reliability and performance.

We plan to migrate most of our work that requires large amounts of computer time in bursts of activity, such as our multiple alignment pipeline, from dedicated local computer clusters to shared cloud-computing environments. Because of the greater number of machines that can be recruited temporarily in the cloud, this should significantly reduce the time needed to complete complex computations. This approach should also be more cost-effective than maintaining a dedicated cluster, because a cluster with sufficient capacity to finish our largest jobs in a reasonable time will be partially idle between such jobs.

We plan to reconfigure our web server system to user fewer, larger, and more self-contained machines. The current system consists of nine machines: a large data file server that is a single point of failure, a server and a backup server for the read/write databases that contain such things as user settings, and six machines that run web servers and the bulk of our software. We plan to switch to a system that requires only two larger servers, either of which is capable of running the entire website and associated databases, thus eliminating our single point of failure and allowing greater parallelization. This conversion will be possible as the use of track hubs grows, reducing our need to locally host many of the large data files, and thus enabling us to duplicate the remaining set of large data. Because modern computers have more many more CPUs, reducing the number of web server machines will not reduce the CPU parallelism. Overall our website is impacted more by data bottlenecks than CPU bottlenecks; therefore, the data parallelism is the more important of the two.

## Approach

Overall, we aim for 99.95% percent hardware uptime for the genome.ucsc.edu public website, and 99.5% uptime for our development website. Achievement of these goals requires reliable power, reliable hardware, and redundant systems. Because most of our code is written in a fast compiled language using good algorithms, the system I/O typically limits our speed rather than the CPU system. Fast I/O devices are expensive, but we deploy them selectively to speed up critical areas.

Reliable power can be somewhat challenging at UCSC; the local commercial power infrastructure has frequent drops in service. Our production server is on UPS backup in one of the few campus buildings backed up by generator power. Unfortunately, the building does not have space for our development machines, which are therefore located in the San Diego Supercomputer Center (SDSC). The reliability of the SDSC power supply is more robust than UCSC's and is backed up by generators, but in practice we average about twice as many power failures to our SDSC-located machines than we experience with our UCSC-based production server.

Modern computer hardware tends to be quite reliable overall. Most problems manifest in new systems that have not yet been tested by a large user community or in old systems with obsolete components. Our computational requirements, while large compared to the needs of an individual or a moderate-sized lab, are relatively modest compared to many organizations. As a result, we can avoid the use of leading edge hardware and instead purchase systems that have been extensively deployed elsewhere, allowing our systems administrators to weigh feedback from other users when making purchasing and configuration decisions. We aim to replace most hardware every two years to remove worn-out components and to take advantage of the higher speeds of newer systems.

To support the redundant, parallel production site configuration we proposed in the Innovation section, we plan to have two servers each hosting a complete set of Genome Browser data on local RAID5 disk, which will provide both redundancy and parallelism since the pool of users can be split between the two machines. Because we can more easily tolerate system downtime on the development servers, the data storage is mirrored on a cheaper, slower device rather than on a device of the same speed. If necessary, we may be able to rely on the Genomics Institute and previous generation hardware for temporary backup computing, in the event that our development server fails and we need to replace it (this has not yet been proven).

Because most of the data displayed on our production website is precomputed, the bulk of a user's lag time can be attributed to delays while I/O operations complete. We design our data-loading software to minimize disk seeks, which are typically the slowest I/O operation on hard disks. However, we do store a relatively small amount of data specific to each user, such as information about which tracks are visible and other browser configuration information. This user data is of variable size and changes in unpredictable ways; therefore, we are not able to apply the same performance improvements that we use on the pre-computed genomics data. We have found that we can improve performance considerably by putting this relatively small amount of data on a solid-state device (SSD). The production system performs well with the bulk of the data residing on relatively inexpensive RAID5 disk.

In contrast to our production website, our development website is responsible in many cases for performing intensive computations on large amounts of data. It is prohibitively expensive to store all of the development data, which is generally much larger than the production data, on an SSD, but relying on RAID5 disk storage is too slow. Instead, we use a general parallel file system (GPFS) that spreads the data across a large number of disks and a moderate number of nodes, and also internally mirrors the data. This system is able to serve the high I/O needs of our development environment at a reasonable price.

### *Milestones*

| Aim | Description of task | Grant Year | | | | |
|---|---|---|---|---|---|---|
| | | 1<br>Q1-4 | 2<br>Q1-4 | 3<br>Q1-4 | 4<br>Q1-4 | 5<br>Q1-4 |
| 1 | Add hierarchical metadata to hubs and improve searching. | ■ | ▨ | | | |
| 1 | Port all native Genome Browser data to track hubs. | ▨ | ░ | ░ | | |
| 1 | Extend the Genome Browser to serve as a GA4GH client. | ░ | ▨ | ▨ | | |
| 1 | Add support for additional data types to track hubs. | | ▨ | ▨ | ▨ | ▨ |
| 1 | Create JSON-based API for our web services. | | ■ | ▨ | | |
| 2 | Further automate the QA and release of public track hubs. | | ▨ | ▨ | ░ | |
| 3 | Run automated code analysis tools on non-compiled code. | ░ | ░ | ░ | ░ | ░ |
| 3 | Run automated tools on the Genome Browser components. | ░ | ░ | ░ | ░ | ░ |
| 3 | Manually check the software and new features before each release. | ░ | ░ | ░ | ░ | ░ |
| 4 | Move large compute jobs from local cluster to the cloud. | ▨ | ▨ | ░ | ░ | ░ |
| 4 | Reconfigure web servers for higher reliability and performance. | ░ | ░ | ░ | ░ | ░ |
| 4 | Upgrade the web servers. | ░ | | ░ | | ░ |
| 4 | Upgrade the development server. | | ░ | | ░ | |

**Table 1**. Approximate timing and level of effort of specific tasks within this Resource Informatics component. Darker colors indicate greater effort over the given year/quarters.