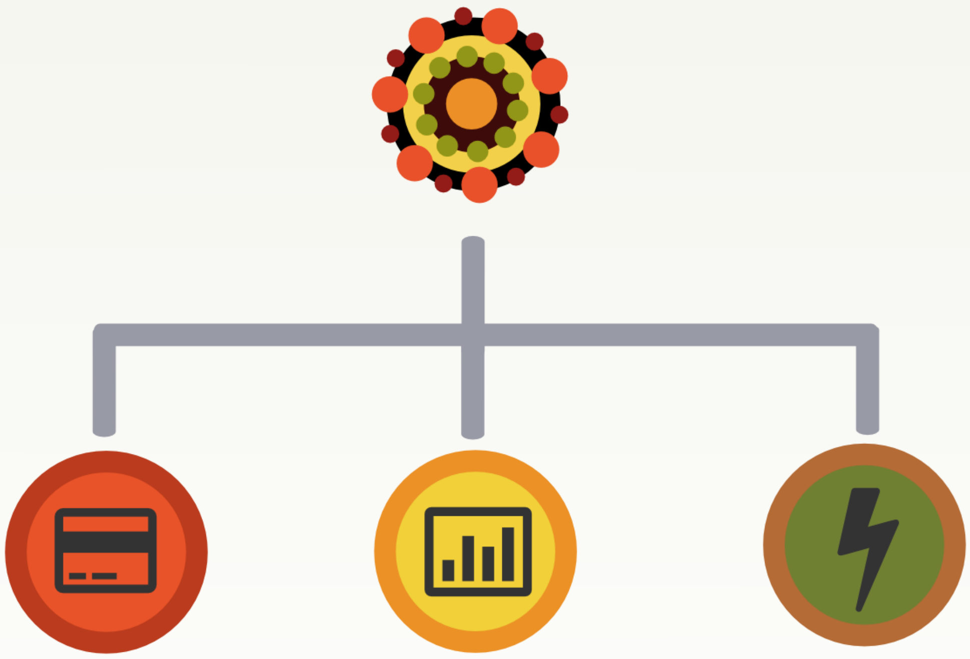
OPENRISK WHITE PAPER

Introducing the Open Risk API

Authors: Øyvind Foshaug, Philippos Papadopoulos

May 31, 2015



www.openrisk.eu

The open future of risk management

SUMMARY

We develop a proposal for an open source application programming interface (API) that allows for the distributed development, deployment and use of financial risk models. The proposal aims to explore the following key question: how to integrate in a robust and trustworthy manner diverse risk modeling and risk data resources, contributed by multiple authors, using different technologies, and which very likely will evolve over time.

The proposal builds on two key modern technological frameworks, Semantic Data and RESTful API's, which in turn are both examples of rapidly adopted and evolving Web technologies. We review the motivation for such an infrastructure, the concepts and tools that can enable such a design and various related initiatives. We describe in detail the current version of the API specification.

For definiteness we illustrate the concept with an open source implementation that takes a use case from the analysis of credit risk in loan portfolios. The implementation consists of demo model and data servers and clients implemented using Python and MongoDB.

The white paper has three main sections:

- A *Review* section discussing the motivation, concepts and precedents
- The *API Specification* section offering precise definitions of the current version of the API
- An *Implementation* section describing the concrete proof-of-principle example

Further Resources

The Open Risk Forum is a meeting place for risk managers and the primary venue for discussing open source risk model development. The Open Risk Academy offers a range of online courses around risk and portfolio management, which utilize the latest in interactive eLearning tools. Please inquire at **info@openrisk.eu** about course schedules.

OpenRisk

OpenRisk is an independent provider of training and risk analysis tools to the broader financial services community. Our mission is captured by the motto: The open future of risk management. Learn more about our mission at: `www.openrisk.eu`

Copyright 2015, OpenRisk. All rights reserved.

Motivation

The challenges of modern financial risk modeling

Financial risk modeling, the quantification of the variety of risks facing financial services companies (banks, insurance, asset managers, brokers etc) is a very broad knowledge domain. Applications range from the provision of low level business insights (e.g., individual client assessment or transaction valuations) to enterprise wide risk frameworks and the fulfillment of formal regulatory requirements (e.g., calculations of required risk capital).

The global financial crisis revealed huge cracks in the risk management capability of the financial sector. We do not propose to undertake here a review of all the specific failures. Even a cursory enumeration, though, suggests weaknesses were revealed across all three classic risk management areas, namely credit, market and operational risk. While the most egregious challenges are, as always, linked to human behavior, which in turn are affected by incentives and governance, the risk infrastructure component was also found to be lacking. The opening paragraph of the ”Principles for effective risk data aggregation and risk reporting” [1] makes for somber reading in this respect:

> *”One of the most significant lessons learned from the global financial crisis that began in 2007 was that banks information technology (IT) and data architectures were inadequate to support the broad management of financial risks. Many banks lacked the ability to aggregate risk exposures and identify concentrations quickly and accurately at the bank group level, across business lines and between legal entities. Some banks were unable to manage their risks properly because of weak risk data aggregation capabilities and risk reporting practices. This had severe consequences to the banks themselves and to the stability of the financial system as a whole”*

The BIS paper goes on to identify the following principles as requirements for ensuring strong risk data aggregation capabilities, which in turn will enhance risk management and decision making processes:

1. Governance: risk data aggregation capabilities and risk reporting practices should be subject to strong governance arrangements
2. Data architecture and IT infrastructure: Fully support risk data aggregation capabilities and risk reporting practices not only in normal times but also during times of stress or crisis
3. Accuracy and Integrity: Generate accurate and reliable risk data to meet normal and stress/crisis reporting accuracy requirements. Data should be aggregated on a largely automated basis so as to minimize the probability of errors
4. Completeness and Granularity: Capture and aggregate all material risk data across the firm. Data should be available by business line, legal entity, asset type, industry, region and other groupings, as relevant for the risk in question, that permit identifying and reporting risk exposures, concentrations and emerging risks
5. Timeliness: Generate aggregate and up-to-date risk data in a timely manner
6. Adaptability: Generate aggregate risk data to meet a broad range of on-demand, ad hoc risk management reporting requests, including requests during stress/crisis situations, requests due to changing internal needs and requests to meet supervisory queries.

7. Accuracy: Risk management reports should accurately and precisely convey aggregated risk data and reflect risk in an exact manner. Reports should be reconciled and validated

8. Comprehensiveness: Risk management reports should cover all material risk areas within the organisation. The depth and scope of these reports should be consistent with the size and complexity of the firms operations and risk profile, as well as the requirements of the recipients.

9. Clarity and usefulness: Risk management reports should communicate information in a clear and concise manner. Reports should be easy to understand yet comprehensive enough to facilitate informed decision-making. Reports should include meaningful information tailored to the needs of the recipients.

10. Frequency: The board and senior management (or other recipients as appropriate) should set the frequency of risk management report production and distribution. Frequency requirements should reflect the needs of the recipients, the nature of the risk reported, and the speed, at which the risk can change, as well as the importance of reports in contributing to sound risk management and effective and efficient decision-making across the bank. The frequency of reports should be increased during times of stress/crisis.

The identified weaknesses and concomitant requirements enumerated above must be seen against a backdrop of massive ongoing IT and Risk Management expenditure by financial institutions that reaches multiple hundreds of billions annually. Yet clearly existing approaches to developing risk management infrastructure have failed to deliver the soundness that is expected from a global industry that is deeply involved in almost every aspect of economic activity. Without diminishing the real challenge involved, the failure is at least in part due to reduced organizational efficiency in adopting and fully utilizing rapidly evolving information technologies.

One of the most profound messages from the technology sector is that even the best founded companies cannot sustain large scale *proprietary* software systems. The challenge of maintaining complex software is so big that only access to the widest talent pool, ability to look under the hood, peer review and other enabling ecosystem attributes of *open source* development can deliver viable solutions. The best example of this dynamic at work is probably the linux operating system, which has become a de-facto platform across the widest range of computing platforms even there is no single corporation claiming the IP behind the system.

Open Source Risk Models

The situation described in the previous section has created a visible (and understandable) change of attitude amongst regulatory bodies to rely less on *internal* risk frameworks developed by financial institutions and more on standardized and simplified measures. This direction can lead to various distortions, not least a vicious cycle of underinvestment in internal risk frameworks.

Reversing that trend will not be an easy undertaking, yet there are very promising ideas that have not been explored. A recent paper[2] presented a new policy proposal for building a *public* framework for gathering, measuring and disclosing financial risk information. A fairly exhaustive list of issues that bedevil the currently predominant risk frameworks is identified, along with the potentially positive impact of a *centralized* open source approach:

1. Black-box/Opacity: Under a public open-source model black boxes turn grey, public model methodology is completely transparent to users.
2. Uncertainty of Robustness/Opacity: All stakeholders work together to ensure the robustness of the new model while significantly expanding knowledge inputs.
3. Uncertainty of Reliability/Monitoring: More heavily scrutinized by a larger number of persons who will have access to methodology and testing.
4. Risk Talent Supply: Centralized design of the construct defines the scope and breadth of the project. One model is developed and maintained, not thousands.
5. Supply of Qualified Regulatory Talent: Under the new public open-source model, the centralized design of the construct defines the scope and breadth of the project. One model is supervised, not thousands.
6. Fragmented and Incomplete Data: Comprehensive OFR data calibrates the model. This is mandated by statute.
7. Complex Risk System Maintenance and Documentation: A centralized design of the construct allows depth of study and analysis. One model is developed and maintained, not thousands.
8. Incomplete Risk Coverage: Under the new public open-source model, having only one model allows a build with a comprehensive list of attributes.
9. Individually Immaterial/Material in Aggregate: Under the new public open- source model, risks can be aggregated in different ways correctly because a complete or more comprehensive dataset is available.
10. High Cost/Access: Under the new public open-source model, efforts are collective, inefficiencies are eliminated and costs are shared.
11. Internal Model Inconsistency: Under the new public open-source model, regulatory capital is applied consistently so that everyone knows they are paying a cost proportional to risk taken.
12. Compliance and Regulatory Cost/Arbitrage: Under the new public open-source model, the cost of compliance is proportional to risk taken.
13. Flexibility/Upgradeability/Scalablility: Under the new public open-source model, it will be easier to effect changes on one model compared to thousands.

A ”Bottom-Up” Open Source Architecture

We propose in this white paper a slightly different open source architecture which can be described as a ”bottom-up” and loosely coupled open source framework compared against the more ”top-down” design sketched in [2].

While all significant open source projects acquire sooner or later some degree of centralized organizational steering, the presence of such a framework is not required for the project to take off and excessive governance may even be counter-motivating for adoption. The key initial challenge is to enable, inspire and incentivise a broad based set of contributors and users, principally by offering rapid productivity.

Towards that goal we propose the development of a well defined *risk models API* (an application programming interface), that is a set of semantic conventions and communication patterns that will govern the interaction of the various resources (models, data) that constitute the open source risk modeling framework.

Required Elements

While an ”API” is currently the preferred design framework to support large scale inter-connected applications, to be effective versus the unique challenges of financial risk management we need to augment this more IT oriented specification with additional domain specific requirements. Hence while the starting point is to provide production level capabilities for all required risk management tasks, in addition we need:

1. To support enhanced / automated calculation audits (workflow provenance, reporting of accuracy etc)
2. Enable efficient implementation validations (comparisons of production tools against specifications)
3. Facilitate conceptual model validations (assessment of model risk via alternative models)
4. Allow for an evolving landscape of inter-operable resources (continuous upgrades)
5. Cost efficiency (while supporting the above)

The possibility of *continuous upgrades* is probably the iconic requirement *not met* but current risk technology frameworks. E.g., the industry is notoriously hampered when requested to implement updated regulatory requirements. Yet in many other areas of technology there is now an expectation of an almost automated upgrade process that works flawlessly.

Meeting the above requirements is a highly non-trivial task and prompts for re-thinking many technical areas that are not strictly in scope of this paper. In this paper we focus on specifying an API for coordinating loosely coupled risk methodology resources (models, data) across an organization. We believe such an API is an essential first step for creating an attractive open source ecosystem. But before we can outline the API it is necessary to sketch the technology context in which it is to be understood.

Core Technologies

A fundamental challenge with risk management oriented IT infrastructure is that it is very seldom, if ever, a single and well defined computational system. In fact the extreme opposite is usually the case: a sprawling array of resources, spanning technologies originating in different decades, ranging in size from massive data-centers to ubiquitous excel models, and being either standalone risk solutions or add-ons to other business and/or finance systems. Therefore the design of a risk API leads naturally to the discussion how to best organize distributed computational systems. Fortunately this is an area very well trodden.

The evolution of client/server interfaces

The *client - server* model of computing is one of the oldest paradigms for designing distributed computational infrastructure. It separates tasks or workloads between specialized providers of a resource or service, called servers, and, (typically more numerous and less potent) service consumers, called clients. For comparison, in the competing *peer-to-peer* design, all computing resources are - in principle - equally capable. How much computation or data storage is required by the client environment is denoted as the "thickness" (or thinness) of the client.

Over the course of the decades advances in the availability and cost of processing power and the speed and capacity of networks have influenced which of the two paradigms attracts more attention. For example initially the dominant paradigm was a *mainframe server*, with the clients being the attached terminals.

This model has ceded space in the 80's and 90's to powerful networked workstations, typically running Unix and later more cost efficient Microsoft (Windows) desktops.

The advent of the web (and its current cloud incarnation) is signaling a swing back to client-server designs. In a fully web/cloud centric architecture the client needs, in principle, only to run a *web browser* as all the significant computation and storage is done on servers.

The popularity of *portable devices* marks another potential turning point in this debacle as portable devices are standalone and don't necessarily need a network to operate. Yet the limitations imposed by portability and the massive computational and storage demands of large scaled networked applications means that the network based client-server paradigm will likely continue being the core architecture for the foreseeable future, with massive servers offering the computing capability and desktop browsers along with portables interacting with the servers in various ways.

The general client-server concept can be implemented in a variety of ways, but as indicated above, it is essential to contemplate a proliferation of *different* servers with diverse capabilities, which exchange data both with other servers and with clients. A most typical example in financial sector context is the result of a merger, which invariably leads to multiple incompatible systems.

What is the cause of this incompatibility? The exchange of data requires a communication (messaging) protocol between the different communicating components. This is a key design element that admits various options. A *protocol* denotes the precise specification of *what* information is exchanged and *how* the message is structured. Over the course of the decades there have been numerous protocols defined and many are concurrently in use today. We do not aim to cover them in detail but only insofar as they are relevant for specifying our API.

Communication protocols

The fundamental low level protocol on which the Internet is build is commonly known as TCP/IP, because its most important protocols, the Transmission Control Protocol (TCP) and the Internet Protocol (IP), were the first networking protocols defined in this standard. TCP/IP enables *wire connectivity* by specifying how data can be segmented into packets, addressed to the different element of the network, transmitted through wires, *routed* through and finally received at the destination.

At the most basic level we suggest the adoption of "Internet based designs" and protocols for the integration of risk resources (models/data) in a loosely coupled network. These designs have been proving remarkably robust to scaling (managing complexity) and receive continuous development attention due to widespread use. This choice, being low level, still leaves considerable freedom (additional ”middleware” layers) that need to be specified before we reach the level of risk systems.

Remote Procedure Calls (RPC)

A remote procedure call (RPC) is a client-server communication pattern that allows a computer program running on the client to cause a program to execute in another resource. The main abstraction involved is to make the invocation of the remote program identical to executing locally on the client.

An example of RPC style communications is the Common Object Request Broker Architecture (CORBA). As the name implies CORBA provides access to remote objects, thereby enabling a distributed object oriented architecture. A more recent and open source RPC framework is ZeroC ICE.

RPC type protocols may have an advantage in specialized applications (e.g. very high rates of data transmission) but for more than a decade the emphasis for general purpose applications has shifted to http.

Hypertext Transfer Protocol (HTTP)

HTTP[3] is a document-based client-server protocol and is the basis for communication in the World Wide Web.

The HTTP protocol is organized around the concept of client initiated ”sessions”. A sequence of request-response transactions over the network. An important part of the HTTP design is that it is always the client that initiates a request, by establishing a connection to a server. The server sends back a status response and a message response. The body of the message is the requested resource (data) which is then further processed on the client, typically for display in a browser. Client requests must utilize one of the available HTTP Methods (GET,HEAD,PUT,DELETE,POST) which determines the nature of the principal data exchange (from client to server or vice versa).

While HTTP/1.1 is the currently widely used version of the protocol (standardized in 1997), there is now a new version (HTTP/2), the first update of the protocol. HTTP/2 promises to improve latency but also, importantly, to allow *persistent connections* between client and server. The server can now autonomously send data to the client (also called server push technology).

HTTP is still a relatively low level protocol. Besides a set of required metadata and the above mentioned methods, it places few restrictions on how the information is structured. To facilitate development of *web services* (distributed applications based on the http protocol) it is advantageous to structure the information exchanged in more standardized formats. Again there are many possible options.

Simple Object Access Protocol (SOAP)

The Simple Object Access protocol (SOAP)[4], is a specification for exchanging structured information in the implementation of web services. It uses XML as its message format. SOAP essentially uses an envelope to wrap underlying data and provide context for their semantics and use. This is done by means of WSDL (Web Services Description Language) and an XML Schema.

While SOAP and WSDL offer a complete solution for designing web services, and despite the ”S” in SOAP, the protocol has faced resistance in developer communities due to perceived complexity and verbosity. One of the contributing factors is its reliance on XML which has been replaced by JSON as the preferred format for exchanging data between web browsers and servers.

JavaScript Object Notation (JSON)

JSON (JavaScript Object Notation)[5] is a lightweight data-interchange format. It is human readable / editable but at the same time easy to parse and generate programmatically. The JSON schema is built on only two structures: i) A collection of name/value pairs which can be realized at runtime as an common object, structure or dictionary. ii) An ordered list of values, realized as an array, vector, list, or sequence. As the name suggests, JSON is native to the javascript language (which in turn is the basis for script execution in all modern browsers and explains its popularity as a message exchange format between clients and servers). One of the weaknesses of JSON is that it does not support binary data formats. A relatively new specification (BSON) aims to remedy this.

RESTful API's

REST (Representational State Transfer) is a web services architecture that has grown in popularity as a viable solution for large scale network applications[6]. Confusingly, it is not a defined protocol, but rather a set of principles (or constraints), which leads to endless discussions as to whether a given implementation is RESTful or not. It should be clear though that implementing client-server communications according to REST is a distinct alternative to the above mentioned defined SOAP protocol.

REST it provides a set of architectural constraints for distributed web services that results in benefits in terms of loose coupling, maintainability, evolvability, and scalability. As discussed in the previous section we consider these attributes essential (even if not sufficient) requirements for supporting the next generation of risk management architecture.

The emphasis above is in the ”constraints” keyword. The aim of REST is to reduce the complexity of developing large scale applications by *simplifying* the permissible interaction patterns into the smallest possible standardized vocabulary.

The main elements of a REST approach are listed below.

- Separation of concerns (between clients and servers): REST is a client-server architecture as previously discussed.
- Uniform interface: Resources are uniquely identified by uniform resource identifiers (URIs) with no assumptions about the internal structure of resources. Data and computational services are available as addressable resources via URL (uniform resource locators). When such a URL is dereferenced (accessed with a client), the server responds with a resource *representation*. This allows the further discovery and manipulation of resources. A representation utilizes a serialization format (which transforms arbitrary resources into a serial file such as JSON or XML) that has a known (to the client) media type.
- Hypermedia: Server responses should enable HATEOAS (hypermedia as the engine of application state). That means that the format should include *links* to further resources that are part of the API. Combined with the previous point of URI opaqueness, this constraint means the mechanism for discovering resources is simple and uniform across the API, essentially *self-documenting*.

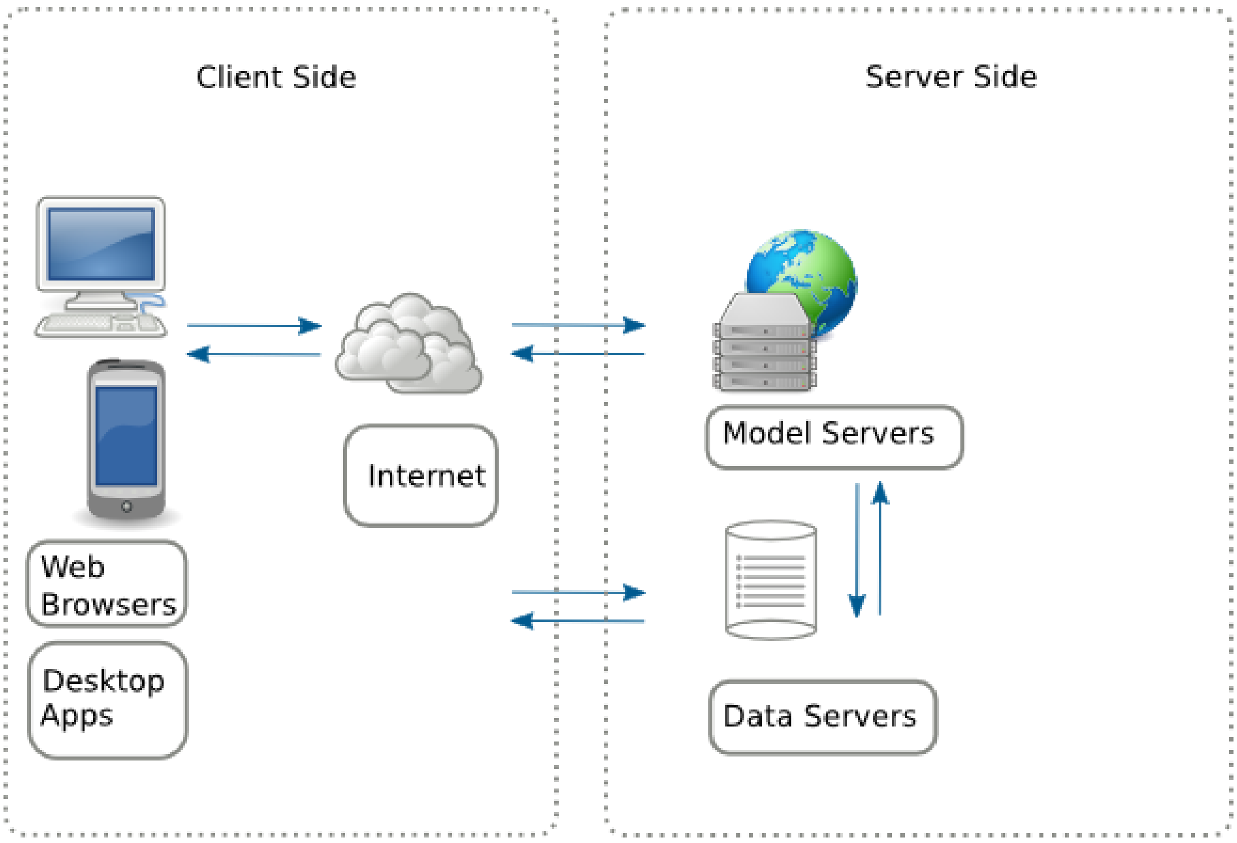


Figure 1: The components of an internet based risk architecture. Clients are completely separated and generally thin (they don't perform significant model calculations, nor do they store data). Via code-on-demand (e.g. javascript) clients can be empowered for example to visualize outcomes in user customizable ways. Model and data servers are distinct and both can be accessed directly from clients. A complete calculation will typically involve accessing a collection of model and data servers. For illustration purposes it is worth mentioning that a spreadsheet calculator does **not** fit this pattern unless the spreadsheet is a registered and connected model/data server

- Statelessness: Requests from clients to servers contain all of the information necessary to understand the request, and do not rely on any internally stored context on the server[1]. A test of statelessness is whether the interaction can be replicated as-is (with the same client request data) after a reboot of the server.
- Cache-ability: Responses can be cached and stored with the client and then (if required) served locally instead for sending the same request to the server.
- Layered system: There can be multiple data / model servers. For any given request, a given component does not interact with the others beyond its immediate layer. Hence a client can issue a request to a server which may trigger a cascade of requests to other servers, but these requests are not visible to the client.[2]
- Code on demand: Clients may receive from the server code snippets (e.g. javascript code that facilitates rendering server data graphically on the client)

REST and functional versus object oriented programming

Ultimately in internal digital representation both what we would normally call ”data” (positions, market data etc) and what we would recognize as ”models” (functions, calculators) are all bits in memory (or permanent storage) and the difference stems from how these bits are used by the processing unit.

High level programming languages further exaggerate these differences to facilitate problem solving by human operators. For example technical computing languages such as R typically use a functional style of programming, where methods take a role and notation similar to functions in mathematics. On the other hand object oriented languages such as C++ or Java are built around the concept of data objects that represent a ”state” and methods that are applied to objects and can modify their state. In simple terms, in functional programming equations are written as y = f(x) rather than the object oriented y = x.f(). While superficially similar, in the second notation a method f() is invoked on an object x and may modify the state of this particular object. The method f() does not exist independently of the concrete object x. In an object oriented approach models live through data objects. To implement a generic interface that is available before any concrete object classes have been define one must use the more elaborate concept of *abstract classes*.

When trying to map the computational process of a risk model calculation into a web service context this distinction becomes quite apparent. REST does not map very naturally to the functional paradigm of technical computation software. Languages where functions are first class citizens suggest more RPC flavored interfaces, which according to Fielding are by definition not restful.

Evolvable Interfaces

REST API’s are currently enormously popular as they have facilitated the development of large scale web applications (including social media platforms). Current implementations ignore an important subset of RESTs constraints, namely demanding self-descriptive messages and requiring the use of hypermedia. In

[1]in mathematical language we would say that the system is Markovian. Past server states (history) are not relevant for its response to the current request

[2]This does not prohibit to support also direct requests to lower level services if part of the API

these partially RESTful approaches the nature of web services included in the API is fixed in time and cannot evolve. This results in more tightly coupled systems and thus less robust to changing requirements. This can be a severe limitation when designing an API that cannot anticipate in detail the future type of messages to be exchanged between clients and servers.

One of the technical options to support an evolvable interface is to incorporate links between services using technologies of the so-called Semantic Web.

The Semantic Web

Semantic Web denotes a number of interlinked technologies including RDF, OWL, SPARQL (definitions below) which have been proposed quite some time ago as an *extension* of the web that will enable more efficient identification and interaction with the exponentially growing number of resources. While early adoption was slow, possibly hindered by overly academic and verbose implementations, practical imperatives have increasingly been working in favor of operationalizing semantic web concepts (e.g. the concepts are already in use by search engines)

The Resource Description Framework (RDF)[7] is a specification for a ”metadata” model. It is similar to specifying entity-relationships in an abstract database model. Its core syntax is a graph, where nodes are subjects or objects and are connected by predicates. The powerful link with web services comes with assigning to all of these three elements their own URI. A database of such triples is called a triple-store.

The Web Ontology Language (OWL) is a systematic framework for describing objects and their relationships. It allows defining classes and individuals (instances) belonging to classes along with object properties. A OWL ontology can be implemented via an RDF graph.

SPARQL (SPARQL Protocol and RDF Query Language) is the query language for extracting information from RDF triple-stores. It is the analogous to SQL for querying a relational database.

Currently various projects create large collections of RDF data by transforming structured data sources into RDF using specialized mappings, and exposing the generated RDF dataset as RDF triple stores with SPARQL endpoints. As mentioned, and not unlike the reasons behind the unpopularity of the CORBA or SOAP frameworks, complexity and verbosity of semantic web technologies have been a hindrance in their adoption. Increasingly there is a push to simplify the framework so as to make it more widely adopted.

Description of a Project (DOAP)

DOAP[8] is an example of a successful documentation project based directly on semantic data technologies. It creates an RDF/XML vocabulary to describe (document) software projects, and in particular open source projects. There are currently generators, validators, viewers, and converters to enable more projects to be able to be included in the semantic web. Freshmeat’s 43 000 projects are now available published with DOAP. It is currently used in the Mozilla Foundation’s project page and in several other software repositories, notably the Python Package Index. The following is an example of an RDF/XML file with high level information about a project

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:doap="http://usefulinc.com/ns/doap#">
 <doap:Project>
  <doap:name>Example project</doap:name>
```

```
   <doap:homepage rdf:resource="http://example.com" />
   <doap:programming-language>javascript</doap:programming-language>
   <doap:license rdf:resource="http://example.com/doap/licenses/gpl"/>
  </doap:Project>
 </rdf:RDF>
```

Linked Data

The rise of both REST and Semantic Web technologies has raised the question of how the two standards interact and in particular how the hypermedia component of REST might be served using semantic data principles[9, 10].

JSON-LD (JSON based Linked Data)[11], is a recent effort to standardize a media type targeted to machine-to-machine communication but with inherent hypermedia and semantic web support. JSON-LD is expressed in traditional JSON format (as opposed to XML that is the basis for much of the Semantic Web), hence developers can leverage the massive investment in tools and libraries for processing JSON messages.

JSON-LD can be used to build fully RESTful services as it supports hyperlinks. At the same time, as it inherits the RDF structure, it integrates the exposed models and data into the Semantic Web[12, 13]. The required additional design costs are significantly outweighed by the achievable benefits in terms of loose coupling, evolvability, scalability, self-descriptiveness, and maintainability.

Risk Management Specific Designs

At this point it might be worth to summarize the technology review thus far. We propose that an internet technology based client-server model with a RESTful communication pattern is the backbone for risk management infrastructure, enabling the key scalability, evolvability and cost efficiency requirements as with many other contemporary large scale applications.

The capture, for audit purposes, of additional information about entities, activities, and people involved in producing a piece of data, which can be used to form assessments about its quality, reliability or trustworthiness is usually denoted as ”provenance”.

This is a further, higher level, layer that is not covered by the architectures discussed so far but which is necessary for our purposes. The utilization of Semantic Web technologies provides the means for documenting resources and calculations so as to satisfy the stringent requirements of a critical risk management framework. For example semantic mediawikis can provide a model documentation repository to support both implementation and theoretical model validations.

In the context of desktop applications there are many examples of provenance frameworks. The corresponding functionality in a web context is less advanced. There is a W3C Working group that has produced the PROV Family of Documents. Those define a model, corresponding serializations and other supporting definitions to enable the inter-operable interchange of provenance information in heterogeneous environments such as the Web.

A workflow is the composition (chaining) of several steps to generate output by processing successive inputs. In the context of the Web, the single processing steps are the invocations of individual web services. From the workflow perspective, a web service is seen as an atomic activity whose internals are hidden from the workflow system.

The composition, storage and retrieval of workflow information and the optimal arrangement of model documentation and model validation resources is not defined further in this paper but we leave it for future work.

The Open Risk Models API

The Open Risk Models API serves to integrate arbitrary collections of data and modeling resources that are used in the context of assessing and managing financial risk, typically (but not exclusively) by financial services providers. The API envisages a wide variety of conforming resources with different access privileges[2].

API Design

In this section we lay out the overall envisaged architecture of components. The API is the protocol that integrates and allows the interoperability of these components.

Objectives and Constraints

In this section we lay out the API objectives and imposed constraints revisiting the "Required Elements" of the previous section. The API aims to support the following,

- Support quality outcomes (Required Elements 2, 3, 4): Facilitate transparency and quality control of risk management results via good documentation, multiple validation layers. Enable standardization and automated testing.
- Support production level use (Required Element 1): Subject to conforming to the API, any standard data and/or modeling infrastructure should be integrable into the risk framework. Private resources (models, data) should be able to co-exist with public (open) components, subject to adhering to the API. There should be a well defined mechanism to integrate legacy resources, subject to developing an interface (API toolkit)
- Enhance usability (Required Elements 5, 6): Facilitate easy human interaction with risk resources via automated discovery, publication of interfaces, and streamlined resource linkages.
- Cost reduction (Required Elements 5, 6): Enable cost efficient deployment and evolution of risk management resources (models / data sources) that will not break interoperability in future configurations. To the degree possible the API should be implemented using generic standards that are adopted more widely.

The first objective, as stated in the introduction is a particularly important consideration for risk models. The other three objectives are in principle equally important in other domains, although the number and complexity of modeling resources may vary in each instance

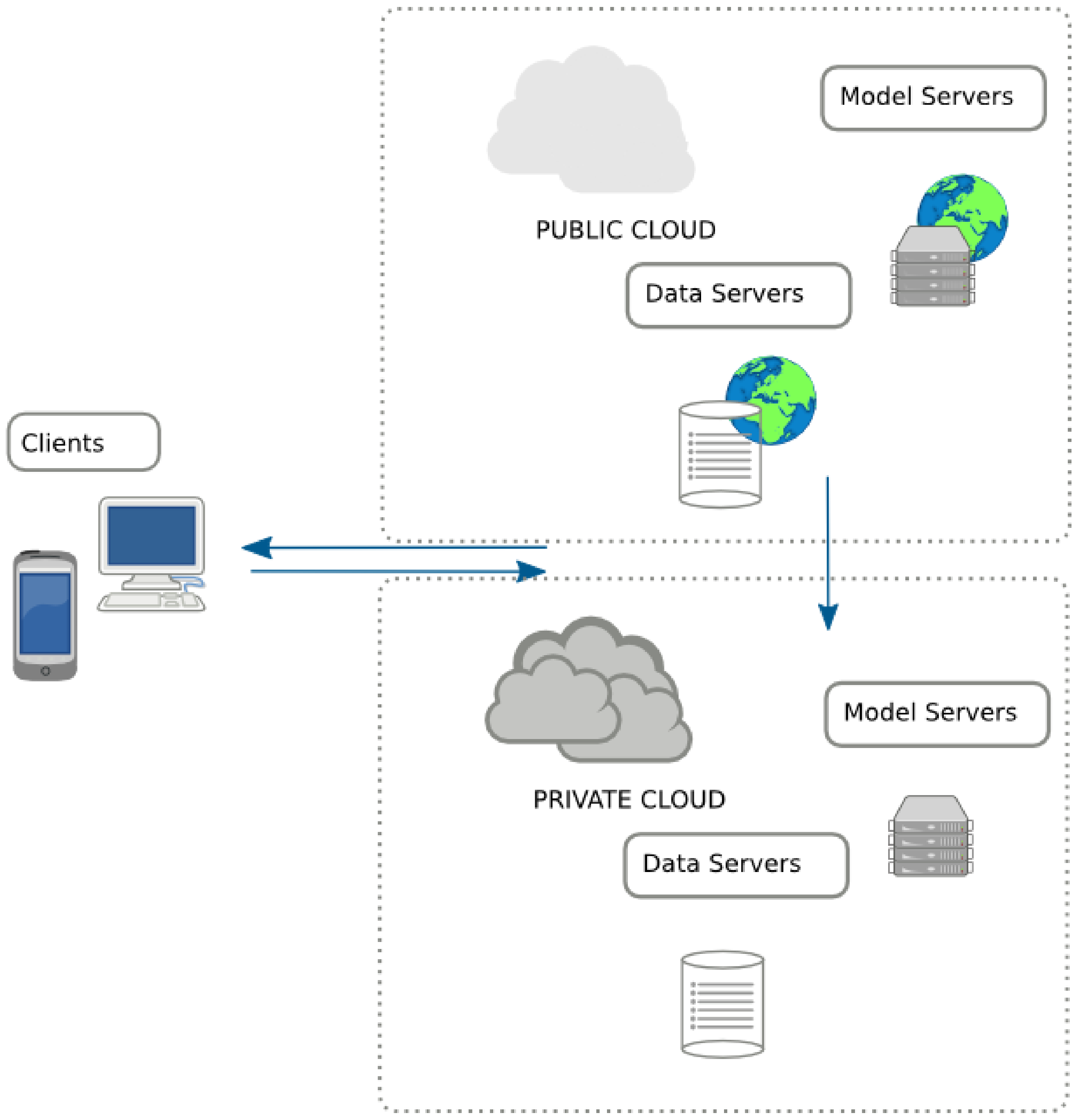


Figure 2: The separation of public and private domains. Public and private components communicate via the same API, similarly model versus data resources (servers). The primary differentiation as far as the API is concerned is authentication and authorization for access and use of resources

Underlying Protocols and Message Formats

The API is to be deployed over standard HTTP protocol. The use of HTTP ensures that the API is leveraging one of the globally most widely used communications technologies. At the same time this implies that there is less flexibility in optimize the API to very specific use cases, e.g., to achieve best possible performance. As bidirectional communication protocols evolve (HTTP/2, WebSockets) and get standardized we would expect the API to evolve to encompass these.

The exchange of information between clients and data / model servers is done using JSON-LD or JSON formats (BSON for binary data). JSON-LD has recently been adopted as a W3C recommendation [14, 15]. Where necessary there are available tools for converting other formats (such as XML or CSV) into the above.

Resources

A resource is either

- a registered risk model (calculator) or
- a registered risk data repository

”Registered” means that the resource has been validated as conforming with the API and is listed in a resource registry.

Each unique resource has an URI (Uniform Resource Identifier).

Registered resources are linked via metadata to other resources that provide e.g., human readable documentation, source code etc. These further resources do not necessarily conform to the API for further access. E.g., source code may be only available via an ftp server.

Model Resources

A model resource exposes a collection of models (one of more concrete functions). A model server:

- Serves as an API documentation endpoint for the implemented model
- Performs calculations
- Maintains a log of input and output communications (but no actual results)
- May contact other services providers for data and/or further models

A collection of models is described by a model template (model metadata)

A model is a concrete function (in the mathematical sense), thus a mapping from a defined set of data inputs into a defined set of data outputs.

Model functions can be multi-valued.

Model inputs and outputs can be both categorical and numerical in nature.

All model resources are documented according to established formats. The documentation constitutes model metadata.

Data Resources

Data resources are persistent stores (databases) of client, product, market history or performance data and similar. A data server:

- Serves as an API documentation endpoint for the data provided (All data resources must allow the exploration of their context via a REST API starting from the starting URI)
- Offers create, read, update and delete (CRUD) operations on datasets
- Maintains a log of input and output communications
- May contact other services providers for data (layering)

There is no restriction on the nature of the data store but it must support communications via the HTTP protocol either directly or via an intermediate server.

The *end transmission* format for data must be JSON.

Resource Registries

A resource registry is an auxiliary component of the system that permits the easy exploration of a large set of resources. It forms the entry point to the API for users that have limited knowledge of the available resources, but also a documentation point for audit purposes.

The registry is optional in the sense that the information it contains is derived from the modeling or data resources (e.g. a crawler may automatically collect such metadata for all live servers)

Workflows

A workflow is a specification of a sequence of operations that uses a combination of resources to achieve a certain objective. That objective can be a normal production calculation, a validation sequence etc. In the current version of the API a workflow is private information stored with the user client. We envision a future extension that will more formally specify workflow[16] resources along with their tracking and storing for audit purposes.

Example context: Calculate concentration risk for portfolio XYZ

1. User wants to discover resources available (models, data) and their interoperability as the precursor activity to constructing and committing a workflow. Using a client (e.g web browser) the user interacts with an information aggregation endpoint. Search model registry: find concentration index calculators (DOAM) Returns: endpoints that match description
2. User wants to construct and store a workflow. By interrogating information aggregation endpoints and individual resource endpoints the user compiles sufficient information
   (a) Search data registry: find credit portfolios Special Case: there is only one portfolio - known URL
   (b) Query Selected Model server Returns: - required data interface (semantic description of risk data) - registered data points - similar models - disclose dependencies (submodels)

   (c) Construct Calculation Request: - model endpoint / inputs file - inputs file: data endpoints / options

   (d) Create New Calculation Request (PUT) to model server (callbacks, asynchronous API, long polling)

3. User wants to test / validate a workflow and receive a report

4. User wants to automate a workflow

5. User wants to introduce / remove / update a resource (change its interface)

Description of a Model

We propose to standardize the metadata capturing model documentation requirements using a variation of the DOAP project. Entirely analogous to that specification, we create an RDF/XML vocabulary to describe (document) model services. An example of a suitable vocabulary definition is included in the appendix. Using that vocabulary we can describe a model using a simple XML or JSON-LD file.

```
<?xml version="1.0" encoding="UTF-8"?>
<Model  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
        xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
        xmlns:foaf="http://xmlns.com/foaf/0.1/"
        xmlns="https://openriskplatform.org/ns/doam#">

        <name>HHI Calculator</name>
        <homepage rdf:resource="https://github.com/open-risk/concentration_library" />
        <created>2015-01-01</created>

        <shortdesc xml:lang="en">
        Calculation of the HHI Index
        </shortdesc>

        <description xml:lang="en">
        Calculates the HHI Index from given portfolio data.
        Part of a library for calculating concentration metrics
        </description>

        <service-endpoint rdf:resource="http://localhost/5001" />
        <data-endpoint rdf:resource="http://localhost/5002" />

        <mailing-list rdf:resource="https://www.openrisk.eu" />

        <maintainer>
                <foaf:Person>
                        <foaf:name>OpenRisk</foaf:name>
                <foaf:homepage rdf:resource="https://www.openrisk.eu" />
            </foaf:Person>
        </maintainer>

        <release>
            <Version>
                <name>unstable</name>
                <created>2015-01-01</created>
                <revision>0.0.1</revision>
            </Version>
        </release>

        <!-- Model category:  Credit Risk :: Concentration -->
        <category rdf:resource="http://software.freshmeat.net/browse/1020/" />

        <license rdf:resource="http://spdx.org/licenses/MIT" />
        <bug-database rdf:resource="http://bugzilla.gnome.org/" />
        <screenshots rdf:resource="https://www.openrisk.eu" />
</Model>
```

The API specification

API Methods

1. Discovery

The target use is to discover service endpoint(s) (urls of servers where model / data resources are available). The appropriate methods are indicated in Table (1)

HTTP Method	Resource	Response Content	Representation	Status Code
GET	/registry/models	model catalog	JSON-LD	200 OK
GET	/registry/models/{model_id}	info about specific model	JSON-LD	200 OK
GET	/registry/data	data catalog	JSON-LD	200 OK
GET	/registry/data/{data_id}	info about specific data	JSON-LD	200 OK

Table 1: API Methods for resource discovery. The information returned is in JSON-LD format and provides the available metadata / links to further resources. Curly brackets indicate a parametric specification. Named resources can also be accessed by name (instead of an ID). Only the expected return codes are indicated. When resources are unavailable servers will respond with 4XX or 5XX error codes as appropriate.

An example of JSON-LD response content to the first method (GET /registry/models/) is included below. It is a JSON-LD document that contains model metadata. The most important data point per model is its URI, the location where the model can be accessed for further information. The snippet illustrates also the use of external namespaces for importing common vocabularies instead of replicating them (in the example re-using the "friend-of-friend" vocabulary to indicate a certain data point is a name or a nickname for a model)

Exhibit 1: An example of data returned by a GET request to a model registry server

```
[
  {
    "@id": "http://127.0.0.1:5012/models/hhi",
    "@type": ["http://openriskplatform.org/ns/doam#model"],
    "http://openriskplatform.org/ns/doam#name": [ { "@value": "HHI" } ],
    "http://xmlns.com/foaf/0.1/name": [ { "@value": "Hirschman" } ],
    "http://xmlns.com/foaf/0.1/nick": [ { "@language": "en", "@value": "hhi" } ]
  },
  {
    "@id": "http://127.0.0.1:5012/models/shannon",
    "@type": [ "http://openriskplatform.org/ns/doam#model" ],
    "http://openriskplatform.org/ns/doam#name": [ { "@value": "Shannon" } ],
    "http://xmlns.com/foaf/0.1/name": [ { "@value": "Shannon" } ],
    "http://xmlns.com/foaf/0.1/nick": [ { "@language": "en", "@value": "sha" } ]
  }
]
```

Once we have a list of URI for resources we can inquire them for further information. This is done via a second GET query utilizing the URI of the desired model/data. For example:

GET /modelserver/models/shannon will return the JSON-LD response indicated in Exhibity 2. From this response we are informed via further metadata pertaining to the model (in a fully deployed production environment this metadata would conform to the DOAM vocabulary we introduced in the previous section) and would contain pointers to every piece of information relevant for the model.

In this instance, the primary new information we will use is pointers as to where we can find valid inputs and where we can store outputs from a calculation. These two pieces of information are linked to formal resources with an URI. So we are informed that the model "shannon" has input that is the data source "obligors" and we are given its URI. We are also given the information that the model has an output relationship with the "results" URI.

Exhibit 2: An example of data returned by a GET request for specific model data

```
[
  {
    "@id": "http://127.0.0.1:5012/models/shannon",
    "@type": ["http://openriskplatform.org/ns/doam#model"],
    "http://openriskplatform.org/ns/doam#hasInput":
            [{ "@id": "http://127.0.0.1:5011/obligors/" }],
    "http://openriskplatform.org/ns/doam#hasOutput":
            [{ "@id": "http://127.0.0.1:5010/results/" }],
    "http://openriskplatform.org/ns/doam#name":
            [{ "@value": "shannon" }],
    "http://xmlns.com/foaf/0.1/mbox":
            [ {"@id": "mailto:models_r_us@example.org" }],
    "http://xmlns.com/foaf/0.1/name":
            [ {"@value": "shannon" }]
  }
]
```

The relationship of these two newly identified resources with the model is not implicit but is contained in the DOAM vocabulary. E.g., the portfolio resource is an *input relation*. Clients can automatically use this information to ensure this resource is used properly.

Obviously the DOAM vocabulary can be extended to any desired degree of detail. For example by introducing more specific relationships such as hasPortfolioInput, hasControlInput, hasMarketDataInput etc. we can fine tune the relationship of a model with other resources.

In an entirely similar manner metadata information can be obtained about data servers. For example risk data that are not primitive but are obtained through the application of lower level models can be documented via linkages to these models and the underlying further data sources.

Proceeding with our example, from the previous query we obtained the URI /dataserver/obligors/ as the location of suitable data. Issuing GET /dataserver/obligors/1 we obtain the response indicated in Exhibit 3. The response is in JSON format with embedded links.

Exhibit 3: An example of data returned by a GET request for specific obligor data

```
{
 "_id": "55585d88a738bd73a9622cd4",
 "id": 1,
 "PD": 0.9371103083249182,
 "EAD": 0.46735685504972935,
 "LGD": 0.0925403158180415 6,
 "_etag": "9e1936c823b9982cd146a27bcae10d7fee8bd6c8",
 "_links": {"self": {"href": "obligors/555853a9622cd4", "title": "obligor"},
             "collection": {"href": "obligors", "title": "obligors"},
             "parent": {"href": "/", "title": "home"}},
 "_created": "Thu, 01 Jan 1970 00:00:00 GMT",
 "_updated": "Thu, 01 Jan 1970 00:00:00 GMT"
}
```

It is a design choice to be refined in further revisions of the API to specify the boundary of the semantic layer (exchanges of information that are using JSON-LD) with the potentially simpler underlying data layer (which we have specified as based on the JSON format) In the above example we have an intermediate situation where the message exchange includes some links, yet the semantics of the different fine grained resources (PD, EAD, LGD) are left unspecified. This means that the user needs to *know* what to do with these data resources.

Promoting the data interchange from JSON to JSON-LD and adding the corresponding semantics to the DOAM namespace would characterize e.g., that a PD is the resource that is *probability*, hence has a certain range, can be transformed by standard methods (e.g. a hazard rate can be computed). At the same time the API would have to become more complex and granular. For example we should be able to access the granular resource /dataserver/obligors/1/PD as a permanent resource.

2. CRUD Operations

Once portfolio resources are discovered fetching data directly is done via a straightforward CRUD API which *minimally* includes the following operations.

HTTP Method	Resource	Action	Format	Return Code
POST	/datapoint/data/{X}	create data X	JSON	201 Created
GET	/datapoint/data/{X}	retrieve data X	JSON	200 OK
PUT	/datapoint/data/{X}	update data X	JSON	200 OK
DELETE	/datapoint/data/{X}	delete data X		200 OK

Table 2: Basic API Methods for Create Read Update Delete (CRUD) operations

An important reason to leave the data side of the API only specified at high level at present is the fact that CRUD oriented REST API's to interact with data stores see rapid development in standards see e.g., OData[17]. We provide some indicative examples of filtered data extraction in the implementation demo.

3. Calculations

Calculations are performed using combinations of model service endpoints and data endpoints. To initiate a calculation, a client must compile a valid workflow document. All the information required for constructing such a document must be provided by the API.

Once the client has constructed a calculation workflow, the key interaction is a POST to the target model server. Upon receipt of the workflow payload, the model server proceeds with fulfilling the request. Doing so may well require accessing other registered resources (models and/or data points) The client need not know of these further interactions (layered approach).

The precise interaction path in the steps following the initial POST request depends on whether the calculation is expected to be a time intensive operation or not. The timescale for what is ”time intensive” is the possible inactivity timeout of any intermediate servers between the client and the calculation server. In the former case (long calculation), an *asynchronous* interaction pattern is required, where the model server does not immediately return the calculation results but instead returns a URI where those will appear in due course. The client can regularly *poll* that URI to obtain progress updates on the calculation and eventually the results.

HTTP Method	Resource	Response Content	Format	Return Code
POST	/server/{model_id}/	output URI (new)	JSON	201 Created
GET	/server/{model_id}/inputs	list of input sets	JSON	200 OK
GET	/server/{model_id}/inputs/{input_id}	output URI (cached)	JSON	200 OK
GET	/server/{model_id}/outputs	list of output sets	JSON	200 OK
GET	/server/{model_id}/outputs/{output_id}	calculation results	JSON	200 OK

Table 3: API Methods for asynchronous calculations

For simplicity in the implementation example we illustrate a *synchronous approach* where the calculation results are expected to be available ”immediately”. For future compatibility we nevertheless assume that the outcome of the initial post is NOT the result set, but a URI where the results can be found

HTTP Method	Resource	Response Content	Response Format	Return Code
POST	/server/{model_id}/	output URI (new)	JSON	201 Created
GET	/server/{model_id}/{output_id}	calculation results	JSON	200 OK

Table 4: Simplified API for synchronous calculations

Further Work

This is the initial (alpha) release of the Open Risk Models API and we envisage work to continue in refining and extending in several directions. We already indicated the need to integrate into the API various provenance and validation functionalities that will enable enhanced risk data / models quality control. Here we indicate some further areas for future work.

Integrating RPC services

We discussed already the tension between REST and RPC style invocations of modeling resources and the better overall suitability of the former for a scalable and evolvable risk framework. Yet the fact remains that significant amount of existing implementations of risk models, whether proprietary frameworks or open source, e.g., R packages are served via RPC. A very relevant example here is the interfacing to an R Server. In this direction there is recent in the for of the OpenCPU API[18].

A POST request in the OpenCPU API always invokes a remote procedure call (RPC). Requests targeting a function object result in a function call where the HTTP parameters from the post body are mapped to function arguments. E.g., the request: POST /ocpu/library/stats/R/rnorm will invoke the rnorm function with client provided arguments.

While not fully RESTful, a *hybrid* extension of our API that will enable the integration of popular R services is a future extension.

Interfacing with Statistical and Market Data Servers

A large component of data used for risk management is of statistical nature. In this space an emerging standard is SDMX[19]. SDMX stands for Statistical Data and Metadata Exchange, the electronic exchange of statistical information. The BIS (Bank for International Settlements), ECB (European Central Bank), EUROSTAT (Statistical Office of the European Union), IMF (International Monetary Fund), OECD (Organization for Economic Co-operation and Development), UN (United Nations), and the World Bank have joined together to focus on business practices in the field of statistical information that would allow more efficient processes for exchange and sharing of data and metadata within the current scope of their collective activities. The SDMX goal is to explore e-standards that could allow efficiency gains and avoid duplication of effort in the field of statistical information. There are already initial examples of REST based interfaces to SDMX data providers (ECB SDW) including JSON based data exchange. Existing SDMX services can be partially integrated into the Open Risk Models API via an intermediate server.

Another relevant data exchange standardization initiative is OpenMAMA (Open Middleware Agnostic Messaging API)[20], a lightweight vendor-neutral integration layer for systems built on top of a variety of message orientated middlewares. The objective of OpenMAMA is to enable users to develop high-performance, event driven applications against a single standard API. OpenMAMA is addressing primarily the needs of real time trading in capital markets and as it is based on special purpose messaging protocols and not HTTP the possibility of useful *direct* interface is not apparent.

Further Semantic Tools

We have restricted the semantic layer of our API to the minimal required. Further structure can be introduced e.g., to enable programmatic validation of the interoperability of different resources.

The current stack of semantic tools includes SKOS (Simple Knowledge Organization System) whic can be used to describe concept hierarchies and vocabularies. While SKOS, RDF Schema and OWL can be used to define classes, properties and relationships between these describing general computational behavior of *objects* is not within the scope of these standards. Object oriented languages defining object behavior by associating *methods* with class members. The SPARQL Inferencing Notation (SPIN) combines concepts from object oriented languages, query languages, and rule-based systems to describe object behavior on the web of data. Using SPIN it may be possible to perform constraint checking of objects in the absence of tight coupling. This utilizes the class definition of the RDF description. SPIN can recursively resolve resources for calculation workflows.

Auxiliary Services

In practical and production oriented implementations there are several other required services that need to be codified in the API, including authentication and authorization, proxy servers, load balancing etc.

Open Source Implementation Demo

A software implementation and related documentation of the metrics developed in this paper are available and released by OpenRisk under an Open Source license.

The released implementation is at this stage preliminary (alpha - not recommended for production) and consists of a lightweight (console based) user interface and back-end database and web servers that implement the workflow execution engines. The user interface is a command line application. Not all elements of the API are implemented at this stage.

Use Case: Concentration Measurement applied to Loan Portfolios

The demo builds around a simplified example of calculating a concentration metric (such as HHI) from credit portfolio data. The concentration metric calculation serves as a simple ”model” example. The model server offers access to a collection of different metrics.

Requirements

To try the API implementation you will need

- The source code from our Github repository (`https://github.com/open-risk/Open_Risk_API`)
- A functioning MongoDB installation for providing a data server
- A functioning python installation on any platform (available for all major platforms, see `www.python.org`)
- A few additional python modules: {numpy, flask, eve}. Check the respective websites for installation instructions (`www.numpy.org`, `www.scipy.org`)

Documentation

Documentation of the client and server implementations is provided in the source code repository and within the code itself.

For simplicity, in this version we have the functionality of the registry bundled with the model / data servers.

The implementation consists of three directories: Description of servers etc.

Data Server

The data server is implemented with MongoDB as a backend and the python Eve framework (built on top of flask) providing REST based web access to the portfolio data.

Model Server

The model server is implemented using Python flask. Documentation for the available risk functions is provided in the Open Risk Manual:

- The Concentration Ratio
- The HHI index
- The Gini index
- The Shannon index
- The Hannah-Kay index

The Open Risk Manual is a Semantic Mediawiki server that enables the enhanced annotation of human readable documentation so that it becomes also programmatically accessible via an API. In the current version of the API this functionality is not integrated but this concept will be pursued in future revisions.

Client

An example web client is provided in the form of a python script. The script illustrates some key interactions with the framework: discovery of model lists, fetching model specifics, compiling a calculation ticket, requesting a calculation and inspecting the results.

Appendix

Illustrative ”description of a model” vocabulary

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:owl="http://www.w3.org/2002/07/owl#"
  xmlns:vs="http://www.w3.org/2003/06/sw-vocab-status/ns#"
  xmlns:foaf="http://xmlns.com/foaf/0.1/"
  xmlns:dc="http://purl.org/dc/elements/1.1/"
  xmlns:doam="http://openriskplatform.org/ns/doam#"
>

<owl:Ontology rdf:about="https://openriskplatform.org/ns/doam#">
        <owl:imports rdf:resource="http://xmlns.com/foaf/0.1/index.rdf" />
        <dc:title>Description of a Model (DOAM) vocabulary</dc:title>
        <dc:description>The Description of a Model (DOAM) vocabulary, described
        using W3C RDF Schema and the Web Ontology Language. Based on the Description of a
        Model (DOAP) vocubulary developed by Edd Dumbill</dc:description>
        <dc:creator>OpenRisk</dc:creator>
        <dc:format>application/rdf+xml</dc:format>
        <dc:rights>Copyright  2015 OpenRisk</dc:rights>
        <foaf:maker>
                <foaf:Person>
                        <foaf:name>Philippos Papadopoulos</foaf:name>
                        <foaf:mbox rdf:resource="mailto:info@openrisk.eu" />
                </foaf:Person>
        </foaf:maker>
</owl:Ontology>

<rdfs:Class rdf:about="https://openriskplatform.org/ns/doam#Model">
        <rdfs:isDefinedBy rdf:resource="https://openriskplatform.org/ns/doam#" />
        <rdfs:label xml:lang="en">Model</rdfs:label>
        <rdfs:comment xml:lang="en">A distinct registered model</rdfs:comment>
        <rdfs:subClassOf rdf:resource="http://xmlns.com/wordnet/1.6/Model" />
        <rdfs:subClassOf rdf:resource="http://xmlns.com/foaf/0.1/Model" />
</rdfs:Class>

<rdfs:Class rdf:about="https://openriskplatform.org/ns/doam#Version">
        <rdfs:isDefinedBy rdf:resource="https://openriskplatform.org/ns/doam#" />
        <rdfs:label xml:lang="en">Model Version</rdfs:label>
        <rdfs:comment xml:lang="en">Version information of a released model.</rdfs:comment>
</rdfs:Class>

<rdfs:Class rdf:about="https://openriskplatform.org/ns/doam#Specification">
        <rdfs:isDefinedBy rdf:resource="https://openriskplatform.org/ns/doam#" />
        <rdfs:label xml:lang="en">Specification</rdfs:label>
        <rdfs:comment xml:lang="en">A specification of the model</rdfs:comment>
        <rdfs:subClassOf rdf:resource="http://www.w3.org/2000/01/rdf-schema#Resource"/>
</rdfs:Class>

<rdfs:Class rdf:about="https://openriskplatform.org/ns/doam#Repository">
        <rdfs:isDefinedBy rdf:resource="https://openriskplatform.org/ns/doam#" />
        <rdfs:label xml:lang="en">Repository</rdfs:label>
        <rdfs:comment xml:lang="en">Source code repository for the model.</rdfs:comment>
</rdfs:Class>

<rdfs:Class rdf:about="https://openriskplatform.org/ns/doam#GitBranch">
        <rdfs:isDefinedBy rdf:resource="https://openriskplatform.org/ns/doam#" />
        <rdfs:label xml:lang="en">Git Branch</rdfs:label>
        <rdfs:comment xml:lang="en">Git source code branch.</rdfs:comment>
        <rdfs:subClassOf rdf:resource="https://openriskplatform.org/ns/doam#Repository" />
</rdfs:Class>

<rdf:Property rdf:about="https://openriskplatform.org/ns/doam#service-endpoint">
        <rdfs:isDefinedBy rdf:resource="https://openriskplatform.org/ns/doam#" />
        <rdfs:label xml:lang="en">Service Endpoint</rdfs:label>
        <rdfs:comment xml:lang="en">The URI of a web service endpoint</rdfs:comment>
        <rdfs:domain rdf:resource="https://openriskplatform.org/ns/doam#Model" />
        <rdfs:range rdf:resource="http://www.w3.org/2000/01/rdf-schema#Resource" />
</rdf:Property>

<rdf:Property rdf:about="https://openriskplatform.org/ns/doam#data-endpoint">
        <rdfs:isDefinedBy rdf:resource="https://openriskplatform.org/ns/doam#" />
        <rdfs:label xml:lang="en">Data Endpoint</rdfs:label>
        <rdfs:comment xml:lang="en">The URI of a data endpoint</rdfs:comment>
        <rdfs:domain rdf:resource="https://openriskplatform.org/ns/doam#Model" />
        <rdfs:range rdf:resource="http://www.w3.org/2000/01/rdf-schema#Resource" />
</rdf:Property>

<rdf:Property rdf:about="https://openriskplatform.org/ns/doam#release">
        <rdfs:isDefinedBy rdf:resource="https://openriskplatform.org/ns/doam#" />
        <rdfs:label xml:lang="en">Release</rdfs:label>
        <rdfs:comment xml:lang="en">A model release.</rdfs:comment>
        <rdfs:domain rdf:resource="https://openriskplatform.org/ns/doam#Model" />
        <rdfs:range rdf:resource="https://openriskplatform.org/ns/doam#Version" />
</rdf:Property>

<rdf:Property rdf:about="https://openriskplatform.org/ns/doam#category">
        <rdfs:isDefinedBy rdf:resource="https://openriskplatform.org/ns/doam#" />
        <rdfs:label xml:lang="en">Category</rdfs:label>
        <rdfs:comment xml:lang="en">Model Category.</rdfs:comment>
        <rdfs:domain rdf:resource="https://openriskplatform.org/ns/doam#Model" />
</rdf:Property>

<rdf:Property rdf:about="https://openriskplatform.org/ns/doam#license">
        <rdfs:isDefinedBy rdf:resource="https://openriskplatform.org/ns/doam#" />
        <rdfs:label xml:lang="en">License</rdfs:label>
        <rdfs:comment xml:lang="en">The URI of an RDF description of the license</rdfs:comment>
</rdf:Property>

<rdf:Property rdf:about="https://openriskplatform.org/ns/doam#wiki">
        <rdfs:isDefinedBy rdf:resource="https://openriskplatform.org/ns/doam#" />
        <rdfs:label xml:lang="en">wiki</rdfs:label>
        <rdfs:comment xml:lang="en">URL of Wiki for collaborative discussion</rdfs:comment>
        <rdfs:domain rdf:resource="https://openriskplatform.org/ns/doam#Model" />
</rdf:Property>

<rdf:Property rdf:about="https://openriskplatform.org/ns/doam#bug-database">
        <rdfs:isDefinedBy rdf:resource="https://openriskplatform.org/ns/doam#" />
        <rdfs:label xml:lang="en">Bug Database</rdfs:label>
        <rdfs:comment xml:lang="en">Bug tracker for a project.</rdfs:comment>
        <rdfs:domain rdf:resource="https://openriskplatform.org/ns/doam#Model" />
</rdf:Property>

<rdf:Property rdf:about="https://openriskplatform.org/ns/doam#maintainer">
        <rdfs:isDefinedBy rdf:resource="https://openriskplatform.org/ns/doam#" />
        <rdfs:label xml:lang="en">Maintainer</rdfs:label>
        <rdfs:comment xml:lang="en">Maintainer of a project</rdfs:comment>
        <rdfs:domain rdf:resource="https://openriskplatform.org/ns/doam#Model" />
        <rdfs:range rdf:resource="http://xmlns.com/foaf/0.1/Person" />
</rdf:Property>

<rdf:Property rdf:about="https://openriskplatform.org/ns/doam#developer">
        <rdfs:isDefinedBy rdf:resource="https://openriskplatform.org/ns/doam#" />
        <rdfs:label xml:lang="en">developer</rdfs:label>
        <rdfs:comment xml:lang="en">Developer of code</rdfs:comment>
        <rdfs:domain rdf:resource="https://openriskplatform.org/ns/doam#Model" />
        <rdfs:range rdf:resource="http://xmlns.com/foaf/0.1/Person" />
</rdf:Property>

<rdf:Property rdf:about="https://openriskplatform.org/ns/doam#validator">
        <rdfs:isDefinedBy rdf:resource="https://openriskplatform.org/ns/doam#" />
        <rdfs:label xml:lang="en">validator</rdfs:label>
        <rdfs:comment xml:lang="en">Validator of code</rdfs:comment>
        <rdfs:domain rdf:resource="https://openriskplatform.org/ns/doam#Model" />
        <rdfs:range rdf:resource="http://xmlns.com/foaf/0.1/Person" />
</rdf:Property>

<rdf:Property rdf:about="https://openriskplatform.org/ns/doam#validation-report">
        <rdfs:isDefinedBy rdf:resource="https://openriskplatform.org/ns/doam#" />
        <rdfs:label xml:lang="en">Validation report</rdfs:label>
        <rdfs:comment xml:lang="en">URI of a validation report</rdfs:comment>
        <rdfs:domain rdf:resource="https://openriskplatform.org/ns/doam#Model" />
        <rdfs:range rdf:resource="http://www.w3.org/2000/01/rdf-schema#Resource" />
</rdf:Property>

<rdf:Property rdf:about="https://openriskplatform.org/ns/doam#documenter">
        <rdfs:isDefinedBy rdf:resource="https://openriskplatform.org/ns/doam#" />
        <rdfs:label xml:lang="en">documenter</rdfs:label>
        <rdfs:comment xml:lang="en">Contributor of documentation</rdfs:comment>
        <rdfs:domain rdf:resource="https://openriskplatform.org/ns/doam#Model" />
        <rdfs:range rdf:resource="http://xmlns.com/foaf/0.1/Person" />
</rdf:Property>

<rdf:Property rdf:about="https://openriskplatform.org/ns/doam#tester">
        <rdfs:isDefinedBy rdf:resource="https://openriskplatform.org/ns/doam#" />
        <rdfs:label xml:lang="en">tester</rdfs:label>
        <rdfs:comment xml:lang="en">A tester or other quality control</rdfs:comment>
        <rdfs:domain rdf:resource="https://openriskplatform.org/ns/doam#Model" />
        <rdfs:range rdf:resource="http://xmlns.com/foaf/0.1/Person" />
</rdf:Property>
```

```
<rdf:Property rdf:about="https://openriskplatform.org/ns/doam#helper">
        <rdfs:isDefinedBy rdf:resource="https://openriskplatform.org/ns/doam#" />
        <rdfs:label xml:lang="en">helper</rdfs:label>
        <rdfs:comment xml:lang="en">Model contributor.</rdfs:comment>
        <rdfs:domain rdf:resource="https://openriskplatform.org/ns/doam#Model" />
        <rdfs:range rdf:resource="http://xmlns.com/foaf/0.1/Person" />
</rdf:Property>


</rdf:RDF>
```

Bibliography

[1] Basel Committee on Banking Supervision. Principles for effective risk data aggregation and risk reporting, 2013.

[2] Jong Ho Hwang. Open-source financial risk model. January 2014.

[3] HTTP online documentation. Link.

[4] SOAP online documentation. Link.

[5] JSON online documentation. Link.

[6] R. Fielding. Architectural styles and the design of network-based software architectures. Link.

[7] RDF online documentation. Link.

[8] DOAP online documentation and repository. Link.

[9] Erik Wilde and Michael Hausenblas. Aligning SPARQL with REST and resource orientation. WEWST 2009, 2009.

[10] Rosa Alarcon and Erik Wilde. Linking data from RESTful services. LDOW 2010, 2010.

[11] JSON-LD online documentation. Link.

[12] Markus Lanthaler and Christian Gütl. On using JSON-LD to create evolvable RESTful services. In *Proceedings of the Third International Workshop on RESTful Design*, WS-REST '12, pages 25–32, New York, NY, USA, 2012. ACM.

[13] Markus Lanthaler. Creating 3rd generation web APIs with hydra. WWW 2013 Companion. ACM, 2013.

[14] M. Sporny et al. JSON-LD 1.0: A JSON-based serialization for linked data.

[15] D. Longley et al. JSON-LD 1.0 processing algorithms and API.

[16] K. Baierer K. Eckert, D. Ritze and C. Bizer. Restful open workflows for data provenance and reuse. WWW14 Companion. ACM, 2014.

[17] OData online documentation. Link.

[18] Jeroen Ooms. The OpenCPU system: Towards a universal interface for scientific computing through separation of concerns.

[19] SDMX online documentation. Link.

[20] OpenMAMA online documentation. Link.