Opening Reproducible Research System Architecture

1. Introduction and Goals

Preamble

The packaging of research workflows is based on the concept of the **Executable Research Compendium** (ERC, see specification and article). The reproducibility service is defined by a web **API** specification and demonstrated in a **reference implementation**. Both are published under permissive open licenses, as is this document.

The normative specification is given in the Markdown formatted files in the project repository, which form the basis for readable PDF and HTML versions of the architecture. A HTML and PDF version of this document are available at https://o2r.info/architecture/ and https://o2r.info/architecture/o2r-architecture.pdf respectively.

1.1 Requirements Overview

This architecture describes the relationship of a **reproducibility service** with other services from the context of scientific collaboration, publishing, and preservation. Together these services can be combined into a new system for transparent and reproducible scholarly publications.

The reproducibility service must provide a reliable way to create and inspect packages of computational research to support reproducible publications. *Creation* comprises uploading of a researcher's workspace with code, data, and documentation for building a reproducible runtime environment. This runtime environment forms the basis for *inspection*, i.e. discovering, examining details, and manipulating workflows on an online platform.

1.2 Quality Goals

Transparency

The system must be transparent to allow a scrutiny demanded by a rigorous scientific process. All software components must be Free and Open Source Software (FOSS). All text and specification must be available under a permissive public copyright license.

Separation of concern

The system must integrate with existing services and focus on the core functionality: creating interactive reproducible runtime environments for scientific workflows. It must not replicate existing functionality such as storage or persistent identification.

Flexibility & modularity

In regard to the research project setting, the system components must be well separated, so functions can be developed independently, e.g. using different programming languages. This allows different developers to contribute efficiently. It must be possible to provide various computational configurations required by specific ERC which are outside of the included runtime.

1.3 Stakeholders

Role/Name	Goal/point of contact	Required interaction
Author (scientist)	publish ERC as part of a scientific publication process	-
Reviewer (scientist)	examine ERC during a review process	-
Co-author (scientist)	contribute to ERC during research (e.g. cloud based)	-
Reader (scientist)	view and interact with ERC on a journal website	-
Publisher	increase quality of publications in journals with ERC	-
Curator/preservationist	ensure research is complete and archivable using ERC	-
Operator	provide infrastructure to researchers at my university to collaborate and conduct high-quality research using ERC	-
Developer	use and extend the tools around ERC	-

Some of the stakeholders are accompanied by user scenarios in prose.

2. Architecture constraints

This section shows constraints on this project given by involved parties or conscious decisions made to ensure the longevity and transparency of the architecture and its implementations. If applicable, a motivation for constraints is given. (based on biking2)

2.1 Technical constraints

	Constraint	Background and/or motivation
TECH.1	Only open licenses	All third party software or used data must be available under a suitable code license, i.e. either OSI-approved or ODC license.
TECH.2	OS independent development and deployment	Server applications must run in well defined Docker containers to allow installation on any host system and to not limit developers to a specific language or environment.
TECH.3	Do not store secure information	The team members experience and available resources do not allow for handling information with security concerns, so no critical data, such as user passwords but also data with privacy concerns, must be stored in the system.
TECH.4	Configurations for ERC runtimes	ERCs include the runtime environment in form of a binary archive. The architecture must support executing this runtime environment and must be able to provide different configurations outside it, for example computer architectures or operating system kernels. The minimum requirements for the containerisation solution regarding architecture and kernel apply.

2.2 Organizational constraints

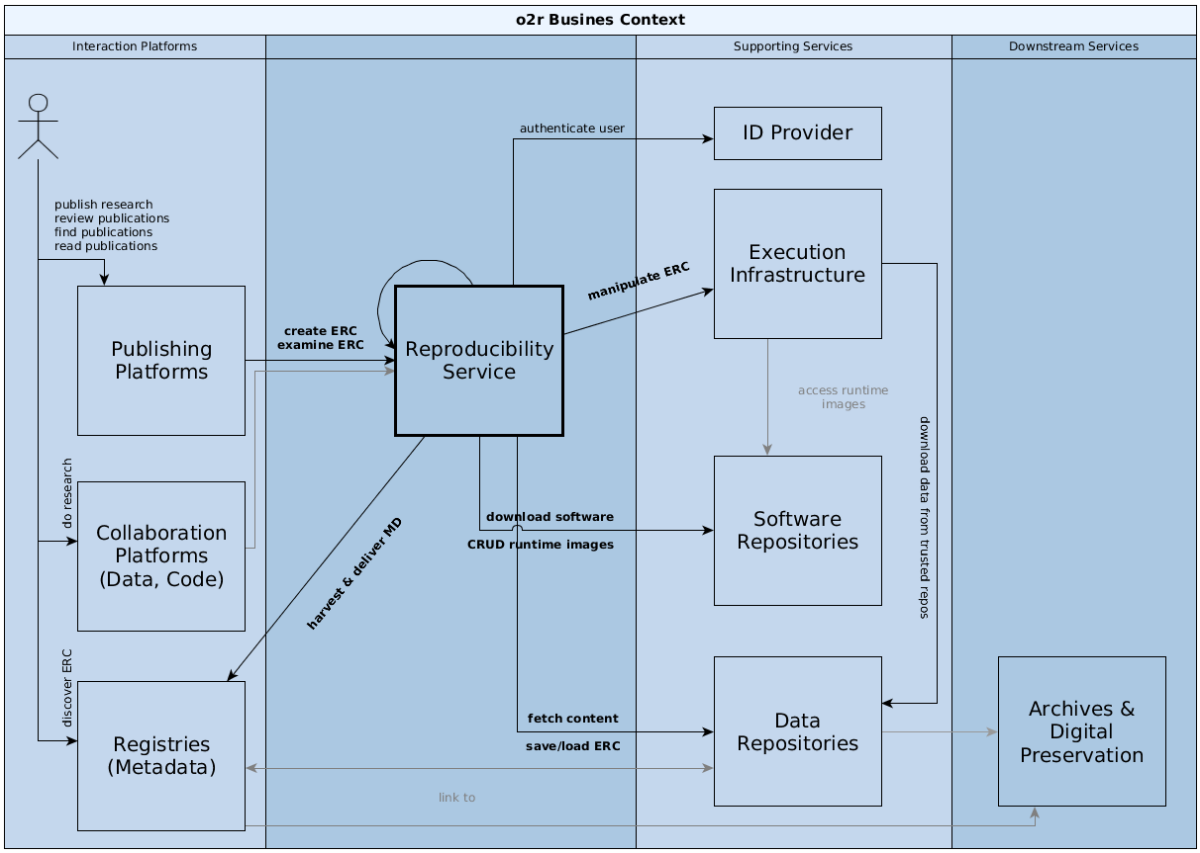
	Constraint	Background and/or motivation
ORG.1	Team and schedule	https://o2r.info/about
ORG.2	Do not interfere with existing well-established peer-review process	This software is *not* going to change how scientific publishing works, nor should it. While intentioned to support public peer-reviews, open science etc., the software should be agnostic of these aspects.
ORG.3	Only open licenses	All created software must be available under an OSI-approved license, documentation and specification under a CC license.
ORG.4	Version control/management	Code must be versioned using `git` and published on GitHub.
ORG.5	Acknowledge transfer from group domain to persistent domain	The ERC bundles artifacts coming from a private or group domain for a transfer to a public and persistent domain (cf. Curation Domain Model (in German)), which imposes requirements on the incorporated metadata.

2.3 Conventions

	Constraint	Background and/or motivation
CONV.1	Provide formal architecture documentation	Based on arc42 (template version 7.0).
CONV.2	Follow coding conventions	Typical project layout and coding conventions of the respective used language should be followed as far as possible. However, we explicitly accept the research project context and do *not* provide full tests suites or documentation beyond what is needed by project team members.
CONV.3	Documentation language is British English	International research project must be understandable by anyone interested; consistency increases readability.
CONV.4	Use subjectivisation for server component names	Server-side components are named using personalized verbs or (ideally) professions: *muncher*, *loader*, *transporter*. All git repositories for software use an `o2r-` prefix, in case of server-side components e.g. `o2r-shipper`.
CONV.5	Configuration using environment variables	Server-side components must be configurable using all caps environment variables prefixed with the component name, e.g. `SHIPPER_THE_SETTING`, for required settings. Other settings should be put in a settings file suitable for the used language, e.g. `config.js` or `config.yml`.

3. System scope and context

3.1 Business context


o2r Busines Context
Interaction Platforms
Supporting Services
Downstream Services
publish research
review publications
find publications
read publications
Publishing Platforms
do research
Collaboration Platforms (Data, Code)
discover ERC
Registries (Metadata)
create ERC
examine ERC
Reproducibility Service
harvest & deliver MD
authenticate user
ID Provider
manipulate ERC
Execution Infrastructure
access runtime images
download software
CRUD runtime images
Software Repositories
download data from trusted repos
fetch content
save/load ERC
Data Repositories
link to
Archives & Digital Preservation

Communication partner	Exchanged data	Technology/protocol
Reproducibility service, e.g. o2r reference implementation	publication platforms utilize creation and examination services for ERC; reproducibility service uses different *supporting services* to retrieve software artifacts, store runtime environment images, execute workflows, and save complete ERC	`HTTP` APIs
Publishing platform, e.g. online journal website or review system	users access ERC status and metadata via search results and paper landing pages; review process integrates ERC details and supports manipulation;	system's API using `HTTP` with `JSON` payload
Collaboration platform	provide means to collaboratively work on data, code, or text; such platforms support both public and private (shared) digital workspaces	`HTTP`
ID provider	retrieve unique user IDs, user metadata, and authentication tokens; user must log in with the provider	`HTTP`
Execution infrastructure	ERC can be executed using a shared/distributed infrastructure	`HTTP`
Data repository	the reproducibility service fetches (a) content for ERC creation, or (b) complete ERC, from different sources; it stores created ERC persistently at suitable repositories, which in turn may connect to long-term archives and preservation systems	`HTTP`, `FTP`, `WebDAV`, `git`
Registry (metadata)	the reproducibility service can deliver metadata on published ERC to registries/catalogues/search portals directly and mediately via data repositories; the service can also retrieve/harvest contextual metadata during ERC creation to reduce required user inputs; users discover ERC via registries	(proprietary) `HTTP` APIs, persistent identifiers (`DOI`), `OAI-PMH`
Software repository	software repository provide software artifacts during ERC creation and store executable runtime environments	`HTTP` APIs
Archives and digital preservation systems	saving ERCs in preservation systems includes extended data and metadata management (cf. private/group domain vs. persistent domain in the Curation Domain Model (in German)), because a different kind of access and re-use is of concern for these systems; these concerns are relevant in so far as the intermediary *data repositories* must be supported, but further aspects, e.g. long-term access rights, are only mediately relevant for the reproducibility service	metadata in `JSON` and `XML` provided as part of `HTTP` requests or as files within payloads

3.2 Technical context

All components use `HTTP(S)` over cable networks connections for communication (metadata documents, ERC, Linux containers, etc.).

4. Solution strategy

This section provides a short overview of architecture decisions and for some the reasoning behind them.

Web API

The developed solution is set in an existing system of services, and first and foremost must integrate well with these systems, focussing on the specific missing features of building and running ERCs. These features are provided via a *well-defined RESTful API*.

Microservices

To allow a dynamic development and support the large variety of skills, all server-side features are

developed in independent *microservices*. These microservices handle only specific functional parts of the API and allow independent development and deployment cycles. Core components are developed using server-side JavaScript based on Node.js with Express while other components are implemented in Python.

We accept this diversification *increases complexity* of both development and testing environments and the deployment of said services.

Required documentation is minimal. The typical structure should follow common practices of the respective language and tools.

Storage and intra-service communication

In accordance with the system scope, there is no reliable storage solution implemented. The microservices simply share a common pointer to a local file system path. Storage of ERC is only implemented to make the solution independent during development and for the needs of core functionality (temporal storage), but it is not a feature the solution will eventually provide.

The unifying component of the architecture is the *database*. It is known to all microservices.

Some microservices communicate via an eventing mechanism for real-time updates, such as the search database and the component providing live updates to the user via WebSockets. The eventing is based on the operation log of the database (which is normally used to synchronise database nodes). This is a clear *misuse of an internal feature*, but a lot simpler than maintaining a full-blown eventing solution.

Demonstration, user data & authentication

To be able to demonstrate the system, a *browser-based client application* is developed. It uses the RESTful API to control the system. *OAuth 2.0* is used for authentication and minimal information, which is already public, is stored for each user. This information is shared between all services which require authentication via the database.

The client application manages the control flow of all user interactions.

Tools

If standalone tools are developed, they provide a command-line interface (CLI). The CLI allows integration into microservices when needed and to package tools including their dependencies as containers and distributing them using a container registry. These *2nd level containers* are started by the microservices and can run either next to the microservices or in an independent container cluster, providing scalability. It must only be ensured they are correctly configured in each microservice. The only required documentation is the installation into a container and usage of the CLI.

5. Building block view

5.1 Refinement Level 1

5.1.1 Blackbox Publication Platforms

Publications platforms are the online interaction points of users with scientific works. Users create publications, e.g. submitting to a scientific journal, publishing on a pre-print server, publishing on a self-hosted website, or collaborating in online repositories. Users examine publications, e.g. browsing, searching, reading, downloading, or reviewing.

5.1.2 Blackbox ID Provider

Identification information of distributed systems is crucial, and for security reasons as well as for limiting manual reproduction of metadata, a central service can provide all of

- unique *identification of users* and *metadata on users*,
- *authentication* of users, and
- metadata on a user's *works*, e.g. publications or ERC.

Persistent identifiers for artifacts in the reproducibility service itself are *not required*, as these are provided by data storage and registries. However, services such as ePIC could allow to retrieve persistent IDs.

5.1.3 Blackbox Execution Infrastructure

The execution infrastructure provides CPU time and temporary result storage space for execution of ERC, both "as is" and with manipulation, i.e. changed parameters. It also provides different architectures and operating system kernel configurations which are outside of the scope of ERC's runtime environments based on containers.

5.1.4 Blackbox Data Repositories

Data repositories are all services storing data but not software. More specifically, they may store software "as data", but not with software-specific features such as code versioning or installation binaries for different computer architectures. Data repositories may be self-hosted or public/free, domain-specific or generic. They typically provide persistent identifiers or handles, e.g. a DOI or URN. They are used both for loading created ERC and for storing the ERC created by the reproducibility service.

5.1.5 Blackbox Registries

Registries are metadata indexes or catalogues.

They are recipients of metadata exports by the reproducibility service to share information about ERC, e.g. add a new ERC to an author's profile. This requires the reproducibility services to translate the internal metadata model into the recipients data model and encoding.

They are sources of metadata during ERC creation when the information in the fetched content is used to query registries for additional information which can be offered to the user.

5.1.6 Blackbox Software Repositories

Software repositories are a source and a sink for software at different abstraction levels. They are a source for software dependencies, such as system packages for installing a library. They are a sink for executable images, which comprise a number of software artifacts and their dependencies, for a specific ERC instance.

5.2 Refinement Level 2

5.2.1 Whitebox Publication Platforms

Publication platforms can be roughly divided into two groups. They can be either specific journals hosted independently, such as JStatSoft or JOSS, or a larger platform provided by a publisher to multiple journals, such as ScienceDirect, MDPI, SpringerLink, or PLOS. To some extend, pre-print servers, for example OSF or arXiv.org, can also fall into the latter category.

Integration with the reproducibility service can happen via plug-ins to generic software, e.g. OJS, or by bespoke extensions. Integrations are based on the service's public API.

5.2.2 Whitebox ID Provider

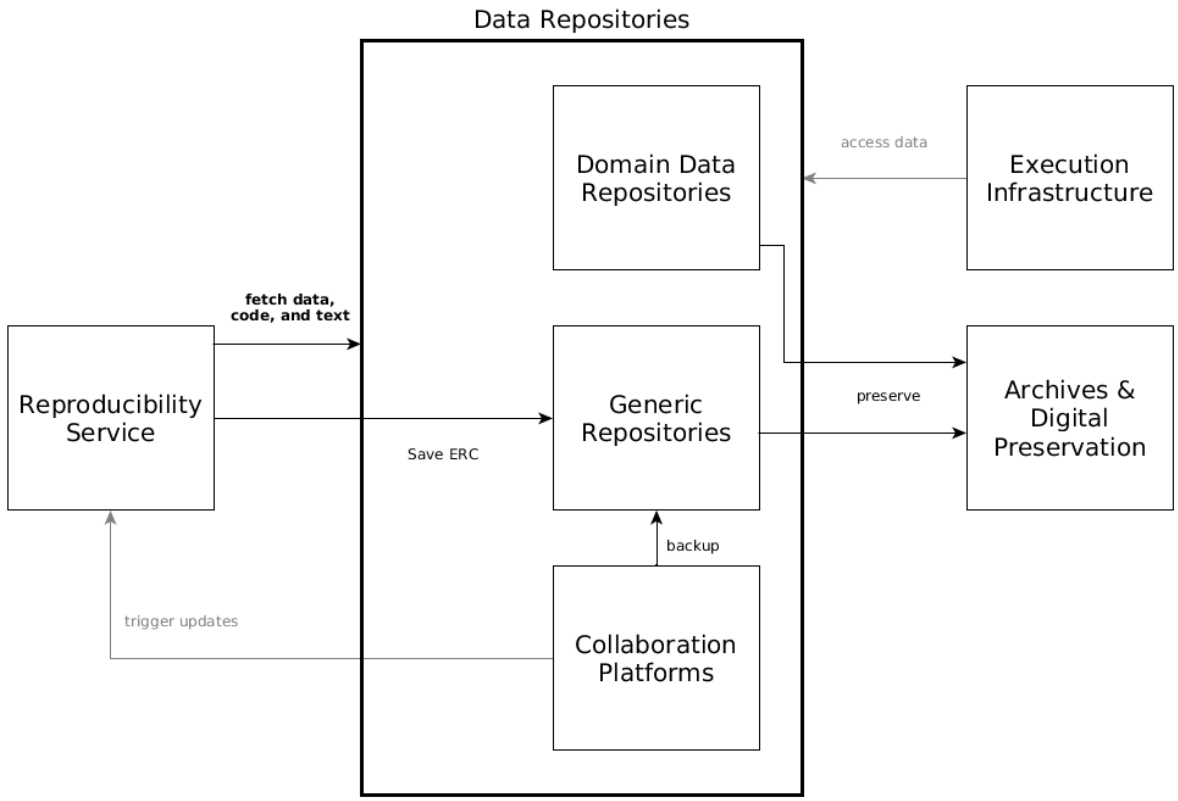
The reproducibility service uses ORCID to authenticate users and retrieve user metadata. The reproducibility service does not use the ORCID authorisation to edit ORCID user data or retrieve non-public data from ORCID, thus this process is pseudo-authentication using OAuth. Internally, the user's public `ORCID` is the main identifier. User have different levels, which allow different actions, such as "registered user" or "administrator". These levels are stored in the reproducibility service.

5.2.3 Whitebox Execution Infrastructure

Such an infrastructure could be either self-hosted, e.g.Docker Swarm-based, use a cloud service provider, such as Amazon EC2, Docker Cloud, or even use continuous integration services such as Travis CI or Gitlab CI. Or it could use a combination of these.

Not all of these options provide the flexibility to provide configurations outside of containers, for example specific operating system kernels. An implementing system must manage these independently, for example by mapping ERC requirements like an operating system, to a part of the execution infrastructure that supports it.

5.2.4 Whitebox Data Repositories



The reproducibility service *does not persistently store anything*. It only keeps copies of files during creation and inspection. So where are ERCs saved and where is their data coming from?

Collaboration platforms, e.g. ownCloud/Sciebo, GitHub, ShareLatex, OSF, allow users to create, store, and share their research (code, text, data, et cetera). Besides being an interaction platform for users, they can

also be seen simply as a data repository. The reproducibility service fetches contents for building an ERC from them based on public links, e.g. a public GitHub repository or shared Sciebo folder. It is possible to link ERC creation to an project/repository under development on a collaboration platform as to trigger an ERC (re-)creation or execution when changes are made.

Protocols: `WebDAV`, `ownCloud`, `HTTP` (including webhooks), `git`

Domain data repositories, e.g. PANGAEA or GFZ Data Services, can be accessed by the reproducibility service during creation and execution of ERC to download data. Allowing access to data repositories reduces data duplication but requires control over/trust in the respective repository.

Protocol: `HTTP` APIs

Generic **Repositories**, e.g. Zenodo, Mendeley Data, Figshare, OSF, provide (a) access to complete ERC stored in repositories for inspection and execution by the reproducibility service, and (b) storage of created ERC. repositories.

Protocols: (authenticated) `HTTP` APIs

Archives and digital preservation solutions can provide long-term preservation of ERC. The data repository and/or one of the involved platform providers are responsible for preservation. A data repository might save the hosted content to an archive, be regularly harvested by an archive, or be part of a distributed dark archive, e.g. CLOCKSS. A platform provider might supply a digital preservation service, e.g. an installation of Archivematica.

Protocol: `HTTP` carrying bitstreams and metadata

> *Data Curation Continuum*
>
> The Data Curation Continuum (cf. diagram by Andre Treloar), describes how data moves from the private domain of a researcher to the public domain of data repositories over the course of conducting research. It describes the properties of data and important aspects of the transitions. In a publishing process based on the reproducibility service, the full migration process is run through.

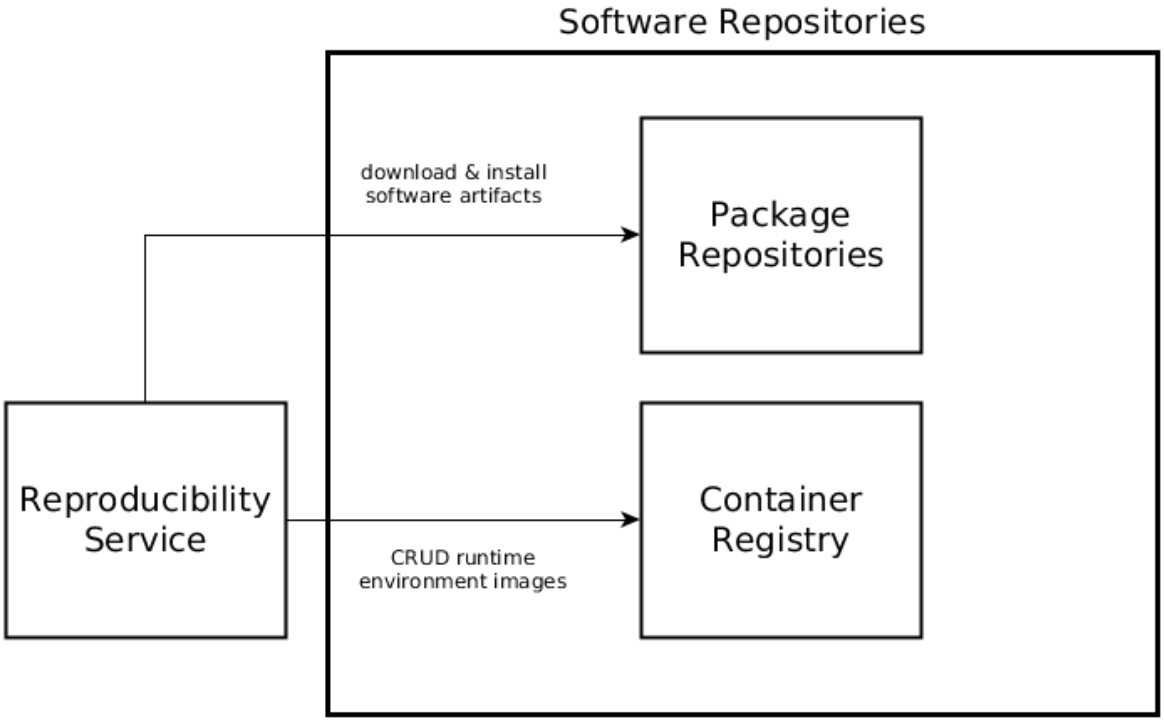
5.2.5 Whitebox Registries

Research data registries and websites, for example (CRIS, DataCite, Google Scholar, Scopus, Altmetric, to name just a few, collect metadata on publications and provide services with this data. Services comprise discovery but also derivation of citation data and creating networks of researchers and publications.

The listed examples include open platforms, commercial solutions, and institution-specific platforms. Some of the registries offer a public, well-defined API to retrieve structured metadata and to create new records.

Protocol: `HTTP` APIs

5.2.6 Whitebox Software Repositories



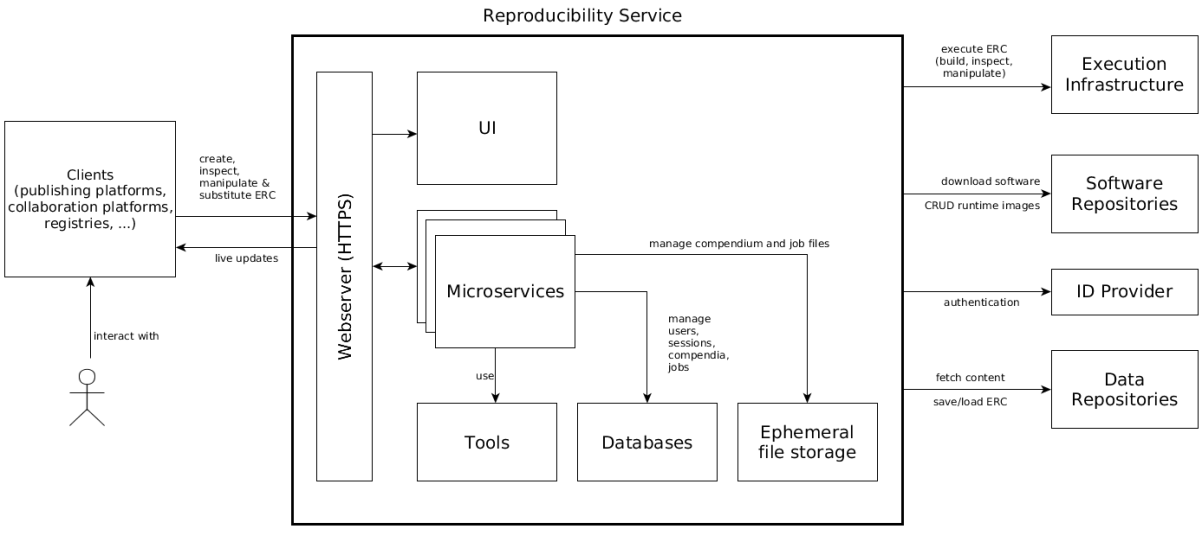
5.2.6.1 Blackbox Package repositories

Package repositories are used during ERC creation to download and install software artifacts for specific operating systems, e.g. Debian APT or Ubuntu Launchpad, for specific programming languages or environments, e.g. CRAN, or from source, e.g. GitHub.

5.2.6.2 Blackbox Container registries

Container registries such as Docker Hub, Quay, self-hosted Docker Registry 2.0 or Amazon ERC, store executable images of runtime environments. They can be used to distribute the runtime environments across the execution infrastructure and provide an intermediate ephemeral storage for the reproducibility service.

5.2.7 Whitebox Reproducibility Service



5.2.7.1 Blackbox Webserver

A webserver handles all incoming calls to the API (`/api/v1/`) via `HTTPS` (`HTTP` is redirected) and distributes them to the respective microservice. A working nginx configuration is available in the test setup.

5.2.7.2 Blackbox UI

The UI is a web application based on Angular JS, see o2r-platform. It connects to all microservices via their API and is served using the same webserver as the API.

5.2.7.3 Blackbox Microservices

The reproducibility service uses a microservice architecture to separate functionality defined by the **web API specification** into manageable units.

This allows scalability (selected microservices can be deployed as much as needed) and technology independence for each use case and developer. The microservices all access one main database and a shared file storage.

5.2.7.4 Blackbox Tools

Some functionality is developed as standalone tools and used as such in the microservices instead of re-implementing features. These tools are integrated via their command line interface (CLI) and executed as *2nd level containers* by microservices.

5.2.7.5 Blackbox Databases

The *main document database* is the unifying element of the microservice architecture. All information shared between microservices or transactions between microservices are made via the database, including session state handling for authentication.

A *search database* is used for full-text search and advanced queries.

The database's operation log, normally used for synchronization between database nodes, is also used for

- event-driven communication between microservices, and
- synchronization between main document database and search index.

> **Note**
>
> This eventing "hack" is expected to be replaced by a proper eventing layer for productive deployments.

5.2.7.6 Blackbox Ephemeral file storage

After loading from external sources and during creation of ERC, the files are stored in a file storage shared between the microservices. The file structure is known to each microservice and read/write operations happen as needed.

5.3 Refinement Level 3

5.3.1 Whitebox microservices

Each microservice is encapsulated as a Docker container running at its own port on an internal network and

only serving its respective API path. Internal communication between the webserver and the microservices is unencrypted, i.e. `HTTP`.

Testing: the reference implementation provides instructions on running a local instance ofr the microservices and the demonstration UI.

Development: the o2r-platform GitHub project contains docker-compose configurations to run all microservices, see repository file `docker-compose.yml` and the project's `README.md` for instructions.

The following table describes the microservices, their endpoints, and their features.

Project	API path	Language	Description
muncher	`/api/v1/compendium` and `/api/v1/job`	JavaScript (Node.js)	core component for CRUD of compendia and jobs (ERC execution)
loader	`/api/v1/compendium` (`HTTP POST` only)	JavaScript (Node.js)	load workspaces from repositories and collaboration platforms
finder	`/api/v1/search`	JavaScript (Node.js)	discovery and search, synchronizes the database with a search database (Elasticsearch) and exposes read-only search endpoints
transporter	`~ /data/` and `~* \.(zip|tar|tar.gz)`	JavaScript (Node.js)	downloads of compendia in zip or (gzipped) tar formats
informer	`~* \.io`	JavaScript (Node.js)	socket.io-based WebSockets for live updates to the UI based on database event log, e.g. job progress
inspecter	`/api/v1/inspection`	R (plumber)	allow inspection of non-text-based file formats, e.g. `.Rdata`
substituter	`/api/v1/substitution`	JavaScript (Node.js)	create new ERCs based on existing ones by substituting files
manipulater	`under development`	--	provide back-end containers for interactive ERCs

ERC exporting

Project	API path	Language	Description
shipper	`/api/v1/shipment`	Python	ship ERCs, including packaging, and their metadata to third party repositories and archives

Authentication

Project	API path	Language	Description
bouncer	`/api/v1/auth`, `/api/v1/user/`	JavaScript (Node.js)	authentication service and user management (whoami, level changing)

Supporting services

Existing software projects can be re-used for common functionality, such as gathering statistics. These supporting services run alongside the microservices in their own containers accessible via the main webservice.

Project	Description
Piwik	collect user statistics

5.3.2 Whitebox database

Two databases are used.

MongoDB document database with enabled replica-set oplog for eventing.

Collections:

- `users`
- `sessions`
- `compendia`
- `jobs`
- `shipments`

The MongoDB API is used by connecting microservices via suitable client packages, which are available for all required languages.

Elasticsearch search index, kept in sync with the main document database by the microservice `finder`. The ids are mapped to support update and delete operations.

The two main resources of the API are kept in separate indices due to their different structure/mappings:

- `compendia` with type `compendia`
- `jobs` with type `jobs`

The search index is accessed by clients through the search endpoint provided by `finder`.

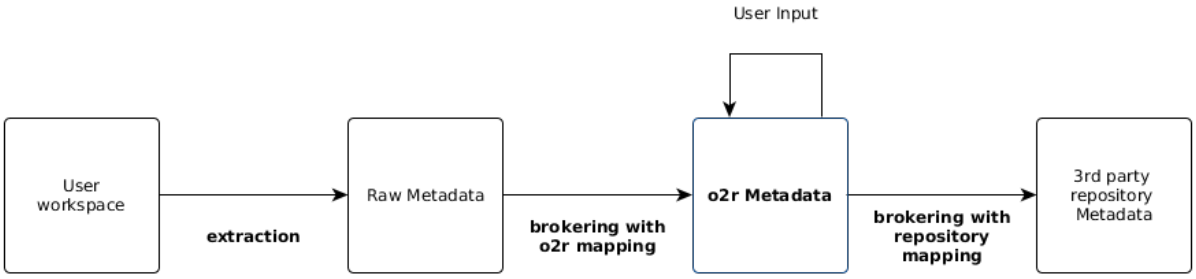
5.3.3 Whitebox tools

project	language	description
meta	Python	scripts for extraction, translation and validation of metadata; for details see metadata documentation
containerit	R	generation of Dockerfiles based on R sessions and scripts

Each tool's code repository includes one or more `Dockerfiles`, which are automatically build and published on Docker Hub. The microservices use the tool's Docker images to execute the tools instead of installing all their dependencies into the microservices. The advantages are a controlled environment for the tool usage, independent development cycles and updating of the tools, smaller independent images for the microservices, and scalability.

Meta

Meta provides a CLI for each step of the metadata processing required in the reproducibility service as shown by the following diagram. After each step the created metadata is saved as a file per model to a directory in the compendium. A detailed view of the meta tool usage in the creation process is provided in the runtime view ERC Creation.



Containerit

The containerit tool extracts required dependencies from ERC main documents and uses the information and external configuration to create a Dockerfile, which executes the full computational workflow when the container is started. Its main strategy is to analyse the session at the end of executing the full workflow.

5.3.4 Whitebox ephemeral file storage

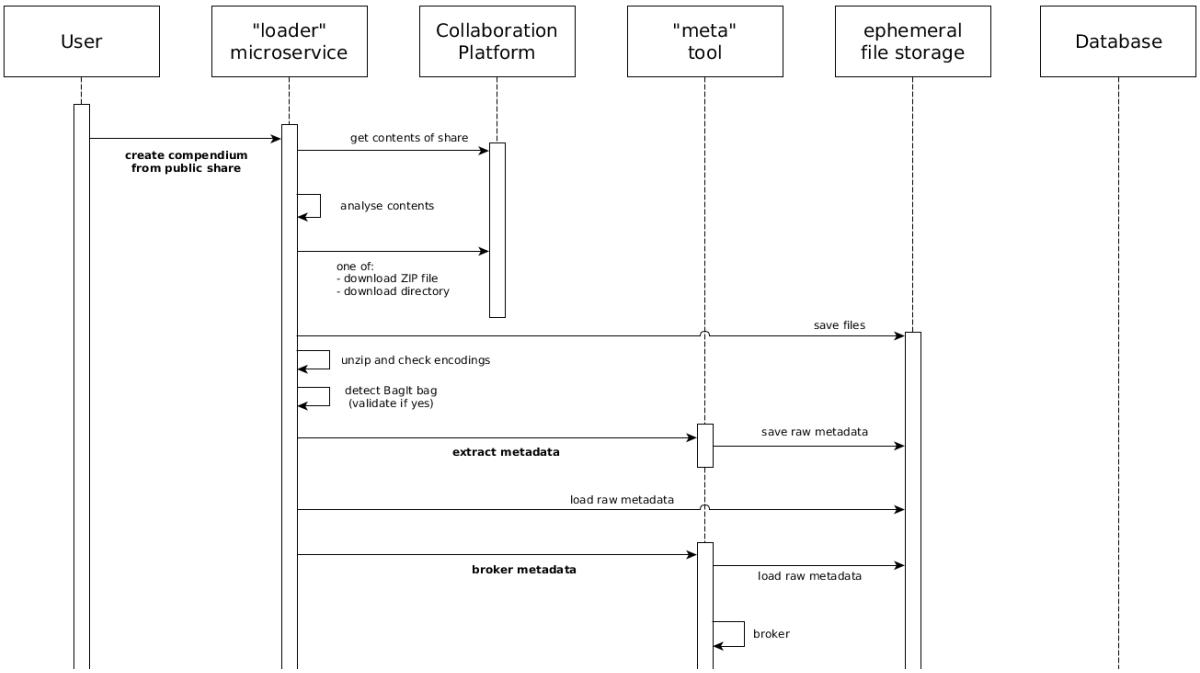
A host directory is mounted into every container to the location `/tmp/o2r`.

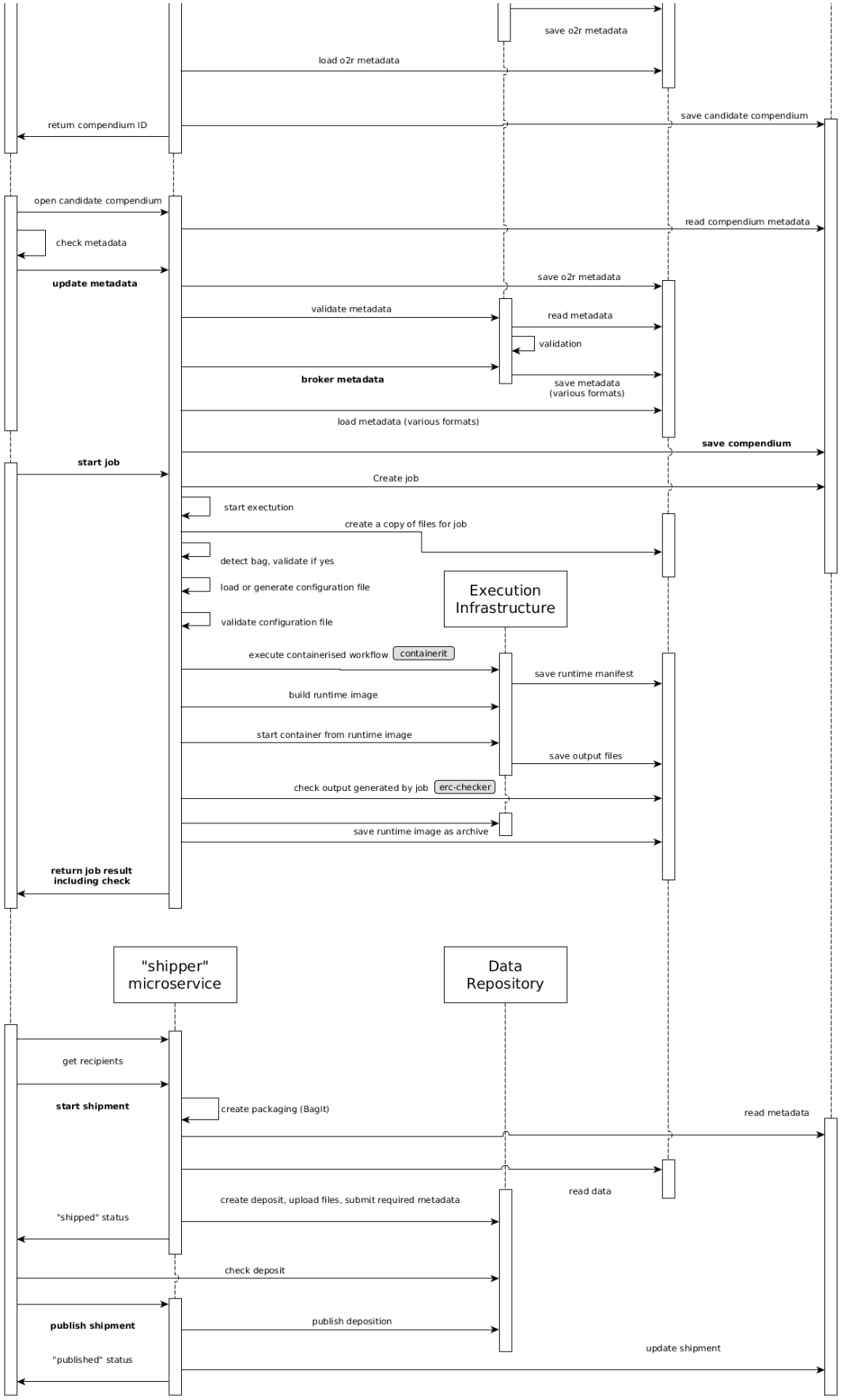
6. Runtime view

The runtime view describes the interaction between the static building blocks. It cannot cover all potential cases and focusses on the following main scenarios.

Scenario	Purpose and overview
ERC Creation	The most important workflow for an author is creating an ERC from his workspace of data, code and documentation. The author can provide these resources as a direct upload, but a more comfortable process is loading the files from a collaboration platform. Three microservices are core to this scenario: `loader`, `muncher`, and `shipper`.
ERC Inspection	The most important workflow for a reviewer or reader is executing the analysis encapsulated in an ERC. The execution comprises creation of configuration files (if missing) from metadata, compiling the a display file using the actual analysis, and saving the used runtime environment. The core microservice for this scenario is `muncher`.

6.1 ERC Creation





First, the user initiates a *creation* of a new ERC based on a workspace containing at least a viewable file (e.g. an HTML document or a plot) based on the code and instructions provided in a either a script or literate programming document), and any other data. The `loader` runs a series of steps: fetching the files, checking the incoming workspace structure, extracting raw metadata from the workspace, brokering raw metadata to o2r metadata, and saving the compendium to the database. The compendium is now a non-public

candidate, meaning only the uploading user or admin users can see and edit it. All metadata processing is based on the tool `meta`.

Then the user opens the candidate compendium, reviews and completes the metadata, and saves it. Saving triggers a metadata validation in `muncher`. If the validation succeeds, the metadata is brokered to several output formats as files within the compendium using `meta`, and then re-loaded to the database for better searchability.

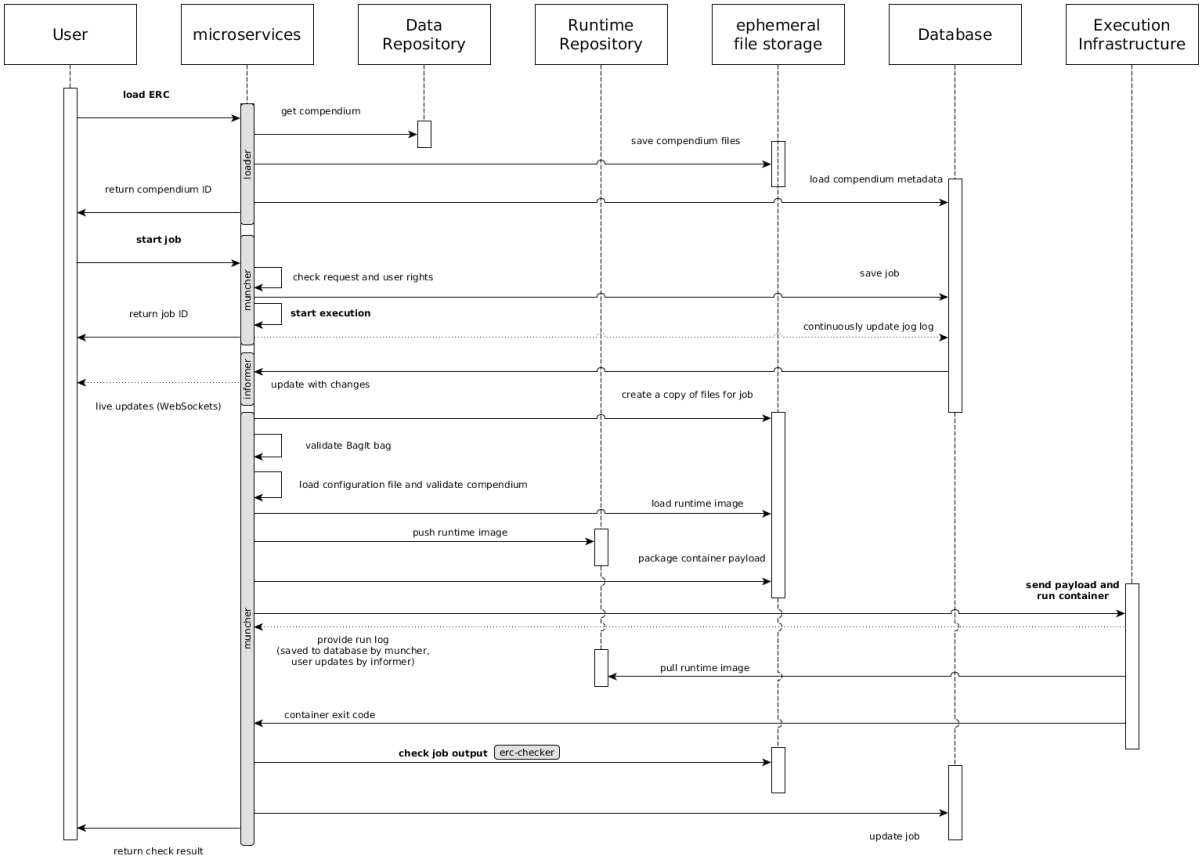
Next, the user must start a *job* to add the ERC configuration and runtime environment to the workspace, which are core elements of an ERC. The ERC configuration is a file generated from the user-provided metadata (see ERC specification). The runtime environment consists of two parts: (a) the runtime manifest, which is created by executing the workflow once in a container based on the tool `containerit`; and (b) the runtime image, which is built from the runtime manifest. A user may provide the ERC configuration file and the runtime manifest with the workspace for fine-grained control; the generation steps are skipped then.

Finally the user starts a shipment of the compendium to a data repository. The `shipper` manages this two step process. The separate "create" and "publish" steps allow checking the shipped files and avoid unintentional shipments, because a published shipment creates an non-erasable public resource.

In the code

The `loader` has two core controllers for direct *upload* and *load* from a collaboration platform. Their core chain of functions are realised as JavaScript Promises, see the code for loader and uploader respectively. The respective steps are shared between these two cases where possible, i.e. starting with the step `stripSingleBasedir`.

6.2 ERC Inspection



The user initiates an *inspection* of an existing ERC by providing a reference such as DOI or URL. `loader` retrieves the compendium files, saves them locally and loads the contained metadata. Then the user can start a new *job* for the compendium. `muncher` checks the request, creates a new job in the database and returns the job ID. The user's client can use the ID to connect to the live logs provided by `informer`. All following steps by muncher regularly update the database, whose change events `informer` uses to continuously update client via WebSockets.

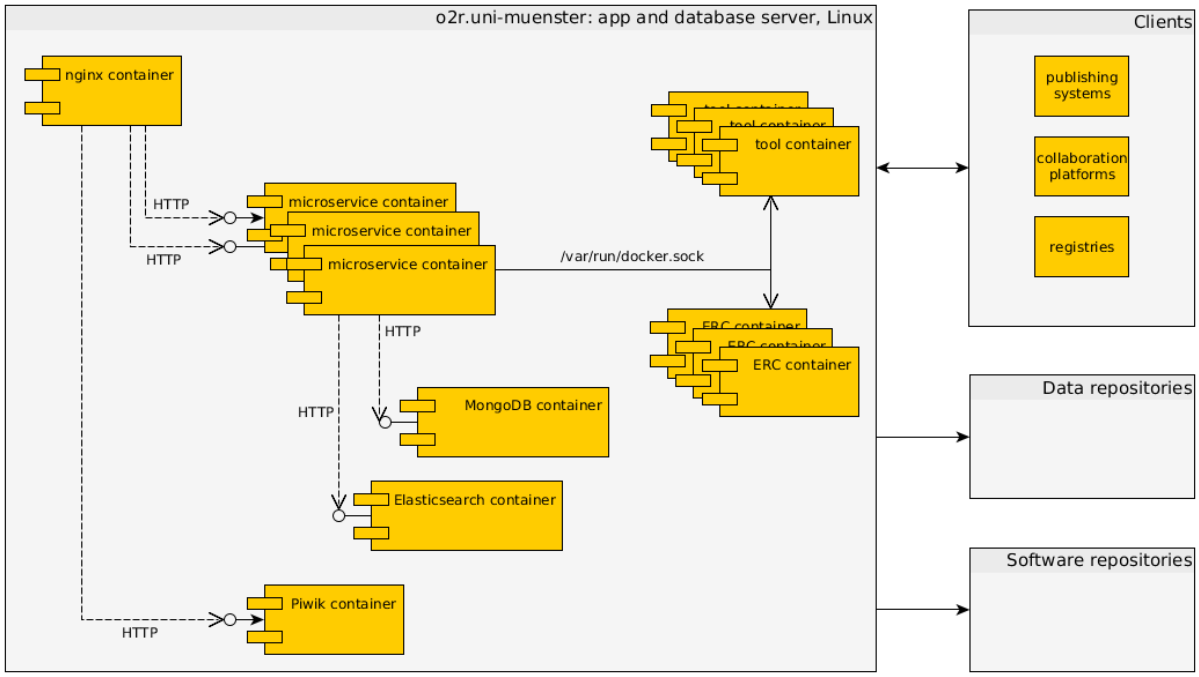
The job starts with creating a copy of the compendium's files for the job. A copy-on-write filesystem is advantageous for this step. Then the archived runtime image is loaded from the file in the compendium into a runtime repository. This repository may be remote (either public or private, e.g. based on Docker Registry, ECR or GitLab) or simply the local image storage. Then all files except the runtime image archive are packed so they can be send to a container runtime. The container runtime can be local (e.g. the Docker daemon), or a container orchestration such as Kubernetes. It provides log updates as a stream to `muncher`, which updates the database, whose changes trigger updates of the user interface via `informer`. When the container is finished, `muncher` compares the created outputs with the ones provided in the compendium and provides the result to the user.

In the code

The `muncher` has two core resources: a *compendium* represents an ERC, a *job* represents a *"run"* of an ERC, i.e. the building, running, and saving of the runtime environment including execution of the contained workflow. The core function for this is the `Executor`, which chains a number of steps using JavaScript Promises, see the code. The check uses the tool `erc-checker`.

7. Deployment View

7.1 Test server https://o2r.uni-muenster.de



Motivation

The o2r infrastructure is driven by the research community's need for user friendly and transparent but also scalable and reliable solutions to increase computational reproducibility in the scientific publication process. To retrieve feedback from the community (public demo) and to increase software quality

(controlled non-development environment), the current development state is regularly published on a test server.

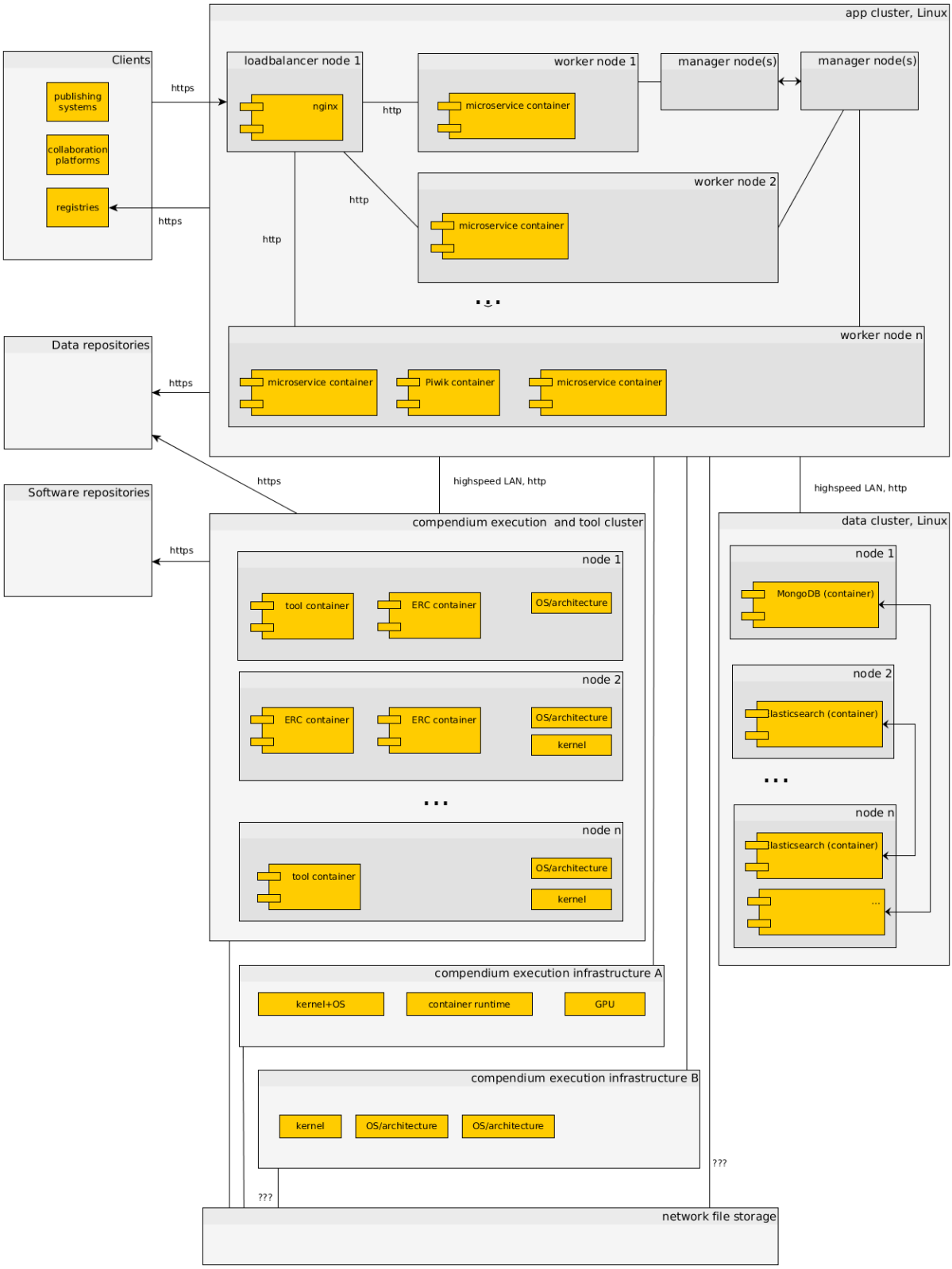
Quality and/or Performance Features

The server is managed completely with Ansible to ensure a well-document setup. The base operating system is CentOS Linux 7. The machine has 4 cores, 8 GB RAM, and a local storage ~100 GB, and runs on a VM host. The one machine in this deployment runs the full o2r reproducibility service, i.e. all microservices and a webserver to serve the user interfaces. It also runs the databases and ancillary services, such as a web traffic statistics service. When executing a compendium, the compendium workspace is packaged in a tarball and send to the Docker daemon. This allows easy switching to remote machines, but also has a performance disadvantage.

Mapping of Building Blocks to Infrastructure

All building blocks run in their own Docker container using an image provided via and build on Docker Hub using a `Dockerfile` included in each microservice's code repository. The server is managed by the o2r team; external building blocks are managed by the respective organisation/provider.

7.2 Production (sketch)



Note

This deployment view is a sketch for a potential productive deployment and intends to point out features of the chosen architecture and expected challenges or solutions. *It is not implemented at the moment!*

Motivation

A productive system must be reliable and scalable providing a single reproducibility service API endpoint. It must also adopt the distribution and deployments of the reproducibility service's microservices. Being based on containers it naturally uses one of the powerful orchestration engines,

such as Docker Swarm or Kubernetes. It can also include multiple execution infrastructures to support multiple container software versions, different architectures, kernels, GPUs, or even specialised hardware. Operators of a reproducibility service can separate themselves from other operators by offering specific hardware or versions.

Quality and/or Performance Features

The services are redundantly provided via separated clusters of nodes for (a) running the reproducibility service's microservices and ancillary services, (b) running the document and search databases, (c) running ERC executions. Separating the clusters allows common security protocols, e.g. the tool and execution cluster should not be able to contact arbitrary websites. The software in the data cluster can run in containers or bare metal. The clusters for app and compendia have access to a common shared file storage, a potential bottleneck. Performance of microservices can be easily scaled by adding nodes to the respective clusters. The diversity of supported ERCs can be increased by providing different architectures and kernels, and hardware. Some requirements could be met on demand using virtualisation, such as a specific operating system version.

Mapping of Building Blocks to Infrastructure

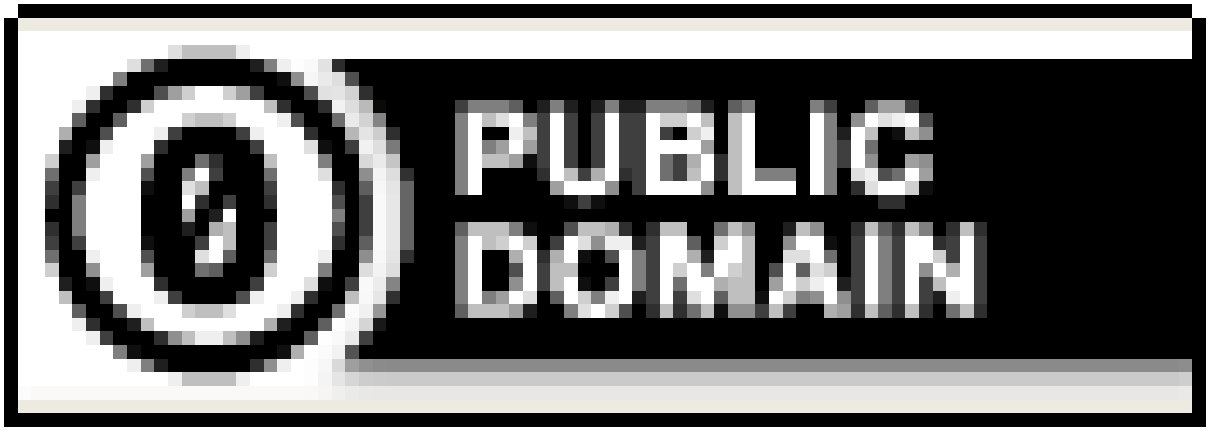
The o2r reproducibility service and execution infrastructures are managed by the o2r team similar to the test server. The other big building blocks, like publishing platforms or data repositories, are managed by the respective organisations.

Credits

This specification and guides are developed by the members of the project Opening Reproducible Research (Offene Reproduzierbare Forschung) funded by the German Research Foundation (Deutsche Forschungsgemeinschaft (DFG) - Projektnummer 274927273) under grant numbers PE 1632/10-1, KR 3930/3-1, and TR 864/6-1).

Opening Reproducible Research (o2r, https://o2r.info/about) is a DFG-funded research project by Institute for Geoinformatics (ifgi) and University and Regional Library (ULB), University of Münster, Germany. Building on recent advances in mainstream IT, o2r envisions a new architecture for storing, executing and interacting with the original analysis environment alongside the corresponding research data and manuscript. This architecture evolves around so called Executable Research Compendia (ERC) as the container for both research, review, and archival.

License



About arc42

arc42, the Template for documentation of software and system architecture.

By Dr. Gernot Starke, Dr. Peter Hruschka and contributors.

Template Revision: 7.0 EN (based on asciidoc), January 2017

Build 00756bc @ 2018-12-12T12:29:15Z

Next
