**Simplr Insites LLC Technical Specifications**

Andrew Evans

Draft

# Purpose

This document outlines the backend and associated software for Simplr Insites, LLC. All components of this system are open sourced under Apache v2.0. This system does not pertain to the actual brokers and GUI tool, and other components built on the system which will contain proprietary and paid options. Deliverable pipelines fall under the ownership of our clients.

Actual requirements docs should be created for Sprints. This document can be presented to a client to aid them in the event of contract termination.

# Editing the Document

Receive approval before editing the document. Phases can be added but do not go back to a phase that is complete and add more details that were not a part of this phase. Instead, create a new phase add an ETA, change the total development time, and add the appropriate information such as objectives, structures, and protocols.

Link to relevant documentation as well. Make each section thorough.

# Overview

The Simplr Insites, LLC backend combines the speed of modern tools and programming languages in a simple and easy to manipulate platform. This platform allows the company to interchange technologies as needed while standardizing messaging and easily expanding microservices and pipelines.

If you have questions, please contact Andrew Evans at aevans48@simplrinistes.com

# Goals

The goals of the backend are to:

* Implement open source technologies that interact seamlessly with SI infrastructure
* Creation of a Pub/Sub system for a fast and scalable producer-consumer job queue
* A system capable of self-management
* Be able to incorporate any paid and proprietary products and allow them to easily interchange with open source systems
* Meet the needs of our clients for large scale, intuitive, and simple backends
* A faster and less resource intensive system than Celery using modern tools for brokers, backends, scheduling, and management (Celery is somewhat outdated)

# Why a new System and Why Queues and Topics?

Celery is getting outdated and still does not support faster systems such as Kafka. This system will be able incorporate modern technologies while also using faster languages such as Rust. Due to the use of Python, Celery lacks the ability to appropriately thread tasks as well, eating entire OS threads for a single task and more for messaging. In short, the only existing system that is production ready is expensive, slow, and old.

Queues and topics are resilient. Despite having some overhead, they create a failsafe system capable of persisting messages. Modern system scale well and are appropriate for running batches of tasks, managing systems such as Spark, and performing multiple tasks at once.

# Concept

Our backend tools combine the power of a broker such as RabbitMQ and Kakfa with a fast and scalable result queue or similar Kafka topic. Built on the ideas of Celery, standard messaging protocol and an RPC-like format allow you to easily scale and build services. While initially planned as a port, this tool is not a direct copy of Celery due to the needs of the company and our clients.

Brokers or the emulation of brokers on services such as Kafka support streaming and job queueing. A standard protocol pushes message in a format that our workers can comprehend. RPC-like calls support running tasks

The UI being designed and developed at Simplr Insites, LLC will use the backend as will any custom broker. Other specifications will lay out this incorporation in depth. This system does not include these components.

# Achieving Simplicity

The major goal of this system is to achieve simplicity through abstraction. The core tools allow developers to write small classes and pipelines within a standardized and manageable framework. Flexibility achieved through actual development as opposed to the use of tools such as Pentaho allows these frameworks to support a wider range of services.

A user should be able to setup a client or worker using an application. For instance:

DataCannon::new(name, ...).run();

Creating tasks should be equally simple:

struct Task{

...args...

}

Impl MyTask for Task{

...impl...

}

All tasks must be present on the worker and accessible through a registry. Registration might look like:

fn register\_task(f: &dyn Fn(..args..) -> Box<AsyncResult>) {}

The main loop in clients and workers should block until the futures receiving and sending messages close. Tokio or an asynchronous library will handle message passing to allow for thousands of tasks to run at once. Ideally you should limit the number of futures to your core count while being mindful of the number of channels or other structures you create.

# Library Structure

There will be three main Rust libraries. These are:

* datacannon-rs-core: Core utilities used by both a worker and client
* datacannon-rs-worker: Worker utilities
* datacannon-rs-client: Client utilities
* datacannon-rs-scheduling: Scheduling utilities
* datacannont-rs-healthchecker: Shared health checking utilities

# A Note on Blame

If you change a package, change the author so we can come to you when something goes wrong. Thanks.

# Development Methodology

Simplr Insites uses Agile/Scrum and Kanban within the behaviorally driven design paradigm. Phases are major elements with incremental improvement over other sections. However, we will try to accomplish the major goals in each phase before proceeding to the next. We are trying to avoid Waterfall development.

Each phase implicitly includes fixing issues and making small, non-breaking changes over existing code. Tests should account for this. That allows the system to be robust.

# Plan

Development proceeds in multiple phases with client data being incorporated as soon as possible. These are:

|  |  |  |  |
| --- | --- | --- | --- |
| **Phase** | **Goal** | **ETA** | **Status** |
| **1** | Research | 1 week | Complete |
| **2** | Establish brokers and backends | 2 weeks | In Progress |
| **3** | Client and Worker | 2 weeks | In Progress |
| **4** | Cluster Communications | 1 week | TODO |
| **5** | Testing and deployment of previous phases using client data (Tim’s 13-15 sources and scrapers) | 2 weeks | TODO |
| **6** | Monitoring and health checking | 3 weeks | TODO |
| **7** | Scaling on Kubernetes and testing on a 1 U | 4 Week | TODO |
| **8** | Scheduling | 3 weeks | TODO |
| **9** | Integrate with Python | 2-3 weeks | TODO |
| **10** | Upgrade system | 3 weeks | TODO |
| **11** | Security and Authentication | 2 weeks | TODO |
| **12** | Evaluation and improvement | Unknown | TODO |
| Total |  | 19- 20 weeks + Unknown | In Progress |

The total estimated time to evaluation and deployment is 24-26 weeks or roughly 6-7 months. The client should be receiving data from the system much sooner and this project will be built alongside runnable sources. User data will run on the system and produce actual paid results within 7-8 weeks or 2 months.

To ensure that the project works smoothly, specifications will try to be as feature complete as possible. Each phase will lay any groundwork for known future phases.

# Phase 1

**Goal : Understand the System**

Phase 1 is research. The questions to be answered are:

* Which backend produces the best result?
* How do we incorporate Rust into the project?
* Which backends are available and useful?
* How can we stream in real time?
* What is the timeline for development?
* Is Kafka faster than RabbitMQ?

# Phase 2

**Goal: Setup a standardized and interchangeable way to send messages and receive results**

Phase 2 involves writing individual brokers and backends capable of:

* Permitting streaming
* Storing messages
* Routing to relevant queues or topics
* Supporting scale
* Auditing as required
* Integrating in a way that supports new technologies

## Messaging

A standard format is used to package messages. Messages are defined in [Celery.](https://docs.celeryproject.org/en/latest/internals/protocol.html) This project uses message format v2.0 which includes:

* Headers including relevant task information, ids, retries, time limits
* Properties such as a correlation id, content type, content encoding and reply topic or queue
* A body including args, kwargs, and streaming information

## Authorization

Messages should allow for authorization. Authorization uses Branca tokens, a standardized way to perform OAuth with better encryption, and a modular builder. Every message needs:

* Space for a token

Futures will authenticate the token against an existing cache. Acquisition will be handled in the application at startup using provided credentials.

## Task Message

Tasks are the main unit of work in the system. Messages embody tasks to be executed. Workers run tasks using the arguments and mapped arguments passed by the client. A basic task in message form includes:

* A unique id for identifying the actual unit of work
* A reply queue or topic for sending results if required or a redis url
* Arguments
* Mapped arguments
* A task name for calling from the registry

## Configuration

Configuration allows users to create and manipulate defaults for brokers, backends, workers, monitoring, networking, and streaming. Configuration variables include:

|  |  |  |  |
| --- | --- | --- | --- |
| **Variable** | **Description** | **Type** | **Default** |
| connection\_inf | Broker connection Information | Object |  |
| result\_backend | Backend Connection Information | Object |  |
| routers | A set of routers | Object |  |
| cache\_backend | Backend for caching tasks | Optional Object |  |
| send\_events | Whether to send events | Bool |  |
| default\_exchange | Default exchange name | String | cannon |
| default\_exchange\_type | Type of default exchange | Enum | Direct |
| default\_queu | Default queue for sending tasks | String | cannon |
| event\_queue | Default event queue | String | cannon |
| event\_exchange | Default event exchange name | String | cannonevent |
| event\_exchange\_type | Type of exchange | Enum | Topic |
| event\_routing\_key | Routing key for events | String | cannonevent |
| result\_exchange | Exchange for results where appropriate | String | cannonresult |
| accept\_content | Type of content to send and receive (currently only json) | String | application/json |
| worker\_prefetch\_multiplier | Number of messages to prefetch for a worker | I8 | 4 |
| default\_delivery\_mode | Delivery persistence type | Object | Persistent |
| default\_routing\_key | Default task routing key | String | cannon |
| broker\_connection\_timeout | Timeout for the broker connection | I64 | 5000ms |
| num\_broker\_threads | Number of threads for the broker runtime | usize | cpus/2 |
| num\_broker\_connections | Number of connections for the broker (number of futures) | usize | Cpus |
| num\_backend\_connections | Number of backend connections | Usize | cpus/2 |
| num\_backend\_threads | Number of connections for the broker (number of futures) | Usize | cpus/2 |
| ha\_policy | Optional high availability policy | Option Object | None |
| create\_missing\_queues | Whether to create the missing quues | Bool | True |
| broker\_transport\_options | Additional options for message sent to the broker | Option Map | None |
| task\_queue\_max\_priority | Max priority for a task queue | u8 | 1 |
| task\_default\_priority | Default priority for tasks | I8 | 0 |
| maximum\_allowed\_failures | Allowed failures in sending messages before killing the program | u8 | 2 |
| maximum\_allowed\_failuers\_per\_n\_calls | Number of calls to measure failures over | U8 | 10 |
| broker\_connection\_max\_retries | Maximum number of retries in a broker | U8 | 1000 |
| task\_retries | Number of times to repeat a task on failure | u8 | 3 |
| default\_lang | Default\_language for tasks | String | rs |
| app\_lang | Language for the application | String | rs |

## System Application

The application starts on a runtime. The broker and backend run on a tokio runtime. Sending and receiving messages occurs asynchronously. The runtime is managed by a client or worker application with the underlying system being almost entirely asynchronous. Most tasks are small and most time is spent in I/O making asycnio a good option.

The total number of threads managed by the runtime is the sum total of the backend and broker threads. Connections are placed on the runtime.

## Underlying Queues

The underlying queue structures should maintain functionality as well. This includes *send* functionality. Queues need to be as immutable as possible.

The queue base trait must be implemented. Individual, connection specific traits must be implemented as well. The base trait contains metadata functions such as *get\_name*. Connection specific functions must include *setup*, send, *create*, *drop,* and *teardown*.

Queue options return a result wrapping a success status and, on error, a QueueError.

## Routers

The router structure stores a set of queues connected to a routing key. Implementation takes a key and returns a subset of queues to send to. In Kafka the result is used to send a message to a specific topic. In RabbitMQ, the structure serves to maintain a list of routes between tasks and queues.

The structure is:

*/// impl contains getters and setters*

*pub struct Router{*

*key: String,*

*queues: Vec<GenericQueue>,*

*exchange: String,*

*}*

The individual router is maintained by a map in *routers.rs*. Every router maintains its routing key. Match functions performed on the key may be regex or direct. Which function is used is defined in the configuration or overridden in a relevant function. These routes should be difficult to change and are set in stone by the program to ensure consistency.

Queues must be a member of the GenericQueue enum. Rust is not object oriented.

## Brokers

Actual brokers pass messages to consumers and store messages for consumption from producers. This application serves as middleware between a broker and application. Messages are sent to the broker and retried on failure. This application facilitates routing while managing multiple broker connections as well.

Available broker types are:

* RabbitMQ
* Kafka

Brokers, as well as backends, spawn Futures that handle messaging on a Tokio loop for asynchronous processing. Futures may fail up to a maximum number of times per a given cycle before the application terminates with error.

### Restarting failed Futures

A future can break down. In order to appropriately handle failures, Rust requires monitoring for a *SendError* or *TrySendError*. Futures are considered terminated on failure and restarted. A maintained failure count is incremented and reset after *maximum\_allowed\_failures\_per\_n\_calls* which is described below.

A maximum number of failures is allowed through the *maximum\_allowed\_failures* variable which is the numerator in a ratio containing a denominator of *maximum\_allowed\_failures\_per\_n\_calls*. When the ratio is exceeded, the entire program fails regardless of whether the ratios exceeded in the broker or backend. The ratio is checked after n calls since the last reset are reached.

#### Failing with a Error

When the number of failures exceeds the allotted amount, the application terminates with an error. The *MaximumFailureErro*r is returned to the starting application. Each application should terminate appropriately.

### General Broker Traits

All broker and backends need to be interchangeable. Therefore, brokers must:

* Maintain a list of routers that:
  + Contain a routing key
  + Contain a list of queues or topics to route to
* Maintain queues or topics in routers that:
  + Maintain a list of broker related queues
  + Set a default exchange if required
  + Set a default routing key
  + Set whether to create queues
  + Maintain receivers and senders for any futures

### Tasks

Tasks are passed using a *TaskConfig*. The config includes additional options, some of which overwrite config options.

These are:

* task\_name
* args
* kwargs
* reply\_to
* correlation\_id
* result\_expires
* priority
* time\_limit
* soft\_time\_limit
* eta
* retries
* lang
* shadow

### Message Retries and Acknowledgment

Because of the way that Rust works (e.g. futures do not yet allow you to poll for completion status externally), it is necessary to send acknowledgement of task receipt so that a task can be removed. This happens on receipt to avoid further crashing futures. This is also necessary to avoid losing tasks.

Acknowledgment occurs over the communications channel. Brokers send a message, check this channel, and then wait for more work from the application. A message id is stored by the broker along with a receipt date checked on each loop iteration of the *run* loop. After a specified *retry\_send\_after* period, the message is considered to have failed and an attempt to send will be made for *broker\_retry* or *backend\_retry* attempts.

NOTE: There is a hint that querying for status will be allowed in the future. This could eliminate the need for acknowledgment to a degree. A failed message should still be reattempted.

### Chains and Chords in the Broker

WARNING: Experimental

First, a note on *root\_id* and *parent\_id* in task and the headers. These represent pieces of the chain. The *root\_id* is the starting message and the *parent\_id* is the message id where the most recent step started.

Chains and chords can run within the futures which should not block the main thread. Running too many at once may be a problem if tens of thousands start at once. This is under research.

A chain is the area of research as each element in the chain executes in new futures within the future running the previous step. Errors should bubble up to the future represented by the root id. Failures are handled as normal.

Chords merely execute multiple functions at once. This blocks the thread on which the future operates but using a broker allows tasks to run for a period of time.

### Kafka Broker

A Kafka broker combines topics in a manner that allows for streaming and exactly-once or as close to exactly-once processing as possible. The Broker:

* Maintains a list of consumers
* Acquires the list of consumer topics from the broker or Zookeeper with filterable names
* Emulates exchanges
* Filter the list of consumers and support routing from the tool (e.g. routes to topic lists)
* Provide options for broadcasting to all members of a routing key through regular expression matching or direct matching
* Incorporate celery-like retry options using the config
* Creation and destruction of topics at startup and teardown
* Manage message sending to a queue like structure where:
  + A consumer has a topic supporting a routing key (e.g. my\_route.my\_group as with Celery)
  + The broker manages the routing strategy (round robin at first but we can include better strategies down the road and add them to the configuration)
* Maintains a separate routing structure for supporting different routes
* Any remaining features in Celery
* Creation of an event topic if not present and handling of event requests

### RabbitMQ Broker

A RabbitMQ Broker supports:

* Dynamic Routing with direct matching and fanout
* RabbitMQ-like exchanges and routing
* Broadcasting capability
* Message sending to queues
* Any support for rolling upgrades that does not break interchangeability
* Features supported by Celery
* Queue management such as queue creation when *create\_queues* is set
* Creation of an event queue and response to event messages

### Sending Messages to the Broker

Functions that send messages to brokers use a specific set of arguments. These are encapsulated by *SendArgs*. *SendArgs* contains:

* message: An encapsulated Message
* exchange: Optional name of the exchange
* routing\_key: Optional routing key for the message
* shadow: Optional name for logging
* options: Additional options to be included with the configs transport options

## Backend

Backends facilitate the return of results in the system. These systems attach task ids from the message to keys in the return. Backends may pass results in order to a subscribed consumer or may pass any result to a consumer without regard to the key as can occur when using RabbitMQ or a Kafka topic. Available backends are Redis, RabbitMQ, and Kafka

### Result Format

Result formats allow for identification of a task, error messages, and other features. On the client side, a user can wait for a result returned as a Future. Workers package results in accordance with the Celery message protocol.

### Redis

Redis queues provide a key value store as well as subscriptions to a specific key. This implementation utilizes these properties to:

* Store task ids as keys and results as values
* Retrieve information from Redis using a unique task id
* Clearing of keys and values on termination

### RabbitMQ

A RabbitMQ backend comes with significant overhead but is useful for testing new mechanisms and avoiding deployment of additional systems. RabbitMQ also supports MQTT. The SI RabbitMQ backend includes:

* Storage of results within a unique queue or passage of results to a queue posted by reply\_to
* Retrieval of tasks by consuming from the one-shot queue or reply\_to queue
* Generation of a one shot queue if and only if the queue must be created
* Dropping a result queue after consumption or on termination
* Responding to event messages and creation of an event queue if needed
  1. *Kafka*

This implementation of Kafka streaming uses a separate topic if the backend is also kafka to send results to a consumer. Order is not guaranteed and delivery is the best effort at exactly once. The implementation includes:

* The creation of a Kafka topic for results
* Potential to register the client to a consumer group for processing
* Implementation of nearly exactly-once processing
* Dropping a result topic on termination
* Responding to event messages and creation of an event queue as needed

# Phase 3

**Goal: Process, send, and receive messages in a standardized way across any system**

Developers deal directly with clients and workers in the system. A client serves as the producer of tasks and consumer of results while workers consume tasks and produce results.

This phase needs to include the groundwork for a notification system. Anything major should be considered here to avoid complexity later.

## Worker

The worker consumes messages passed to the broker by the consumer. These messages are dealt with using common code. Results are pushed to a backend if specified. Workers should acknowledge the receipt of messages when RabbitMQ is the broker.

### Task Format

Developers create tasks. Tasks have names and other associated information following the Celery messaging protocol guidelines. The follow code should instantiate a task in the Rust based system:

fn my\_task(args: Args, kwargs: Kwargs, options: some(Options)) -> Box<AsyncResult>

The task name is completely up to the developer. Args, kwargs, and options follow Celery. The remaining arguments are contained by the message which is generated by the worker from broker data.

### Task Registry

The task registry is a mapping of task name to their respective functions. This looks as follows in the worker:

fn add\_task(f: Fn(Args, Kwarsg, Some(Options)) -> Box<AsyncResult>)

Tasks are stored in a mapping with a name potentially generated programatically.

### Worker Pool, Broker, and Backend

Worker tasks run in a future where they receive information from a broker. Messages contain rpc-like calls. If a backend is provided, then results are sent back over a provided tool.

Each worker future receives a channel if required for brokers and backends. They may also poll for information depending on the backend.

### Running in Python

It should be possible to run the rust code from Python. A worker should be created to interact with the system in this way which will require the crate pyo3. This will allow developers to use the underlying system in Python, speeding up NLP and other tasks on the system.

To speed up releases to the client, it is also possible to run Python from Rust. Pyo3 also works for this task.

### Stream Support

Stream support occurs using chains and chords. For a chain, the chain information is returned to the backend. For chords, functions are executed sequentially before the worker waits for more work. Results are passed as an argument to a chord.

Chains require the provision of a backend while chords do not.

## Client

The client sends tasks to the system. Tasks are converted to messages passed using the appropriate protocol. Clients maintain a pool of connections to a broker and, if required, a backend.

### Sending Tasks

Task results and messages follow the Celery message format. This protocol contains arguments and messaging information.

### Sending Multiple Tasks at Once

One major improvement this system has over Celery is the lack of a GIL. Clients can run and monitor multiple tasks at once. A client broker pool will allow tasks to run at a specified level of asynchronousity.

A specified number of futures start that send information to RabbitMQ or Kafka. When given a backend, each will also establish a connection to the backend and wait for an appropriate result.

### Retrieving a result

Results are retrieved in the futures when a pool is provided. Each future pushes information to the relevant broker before waiting on a backend. Results from this future are passed back to the actual client over a mpsc channel.

### Streaming with Chains and Chords

Chains and chords are provided in the message body. The client future pops the current part of a chain and sends the task with any result back to the broker. Chords are executed at once in the worker. They should be allowed to be part of a chord.

Chains require a backend but chords do not.

### Running Groups

Groups execute functions in parallel. The application sends each function to a channel in round robin fashion. Results of a group come back in a random order through the return channel. They can be part of a chain or chord as well. Groups should be part of a chain.

# Phase 4

**Goal: Create an inter-cluster communications framework**

Inter-cluster communications are the key to a healthy system. Celery accomplishes this task on AMQP brokers through an events queue. The same will occur in this system.

The communications subsystem lays the groundwork for healthchecking and Zookeeper maintenance, event requests, upgrades, and targeted communications to specific nodes.

Common use cases for notifications:

* Upgrades
* Health Checking
* Notification of a new control node (these must be known)

## Worker and Client Notifications (Phase 3)

Handling notifications in the worker and client lays the groundwork for more in depth communications. Clients and workers will respond to notifications over a separate queue or topic.

Notifications are not always specific. Health check requests ask for a ping from all parts of the system. Therefore, messages broadcast to every node. However, a filter will allow the system to target specific nodes.

## Message format

A general message format for the subsystem includes:

* Sender
* Any authorization token with an appropriate scope
* A Message body with:
  + Filter for nodes (each node has a name)
  + Task to accomplish
  + An embodied task message

Messages should embody the task message as the functions will be registered for flexibility as they are in the worker.

## Event Queue

A topic or event queue named *datacannon\_events\_topics* forms the backbone of messaging. Consumers and workers subscribe to the queue. The health check system then submits requests to the queue.

A separate future on a client and worker responds solely to these events. Statistics and other data should be readily available for acquisition.

## Response Queue

The response queue, *datacannon\_event\_requests*, handles responses passed to the health checker. Should this node go down, a backup node should be able to request a response over the events topic and start handling requests with an alert being handled by ElasticAPM as described in health checking and monitoring.

Response are handled as described in the health checking section.

# Phase 5

**Goal: Test and start to use the system**

Phase 5 starts the application of the system within client pipelines. Every component is unit tested under the behaviorally driven design framework. Integration testing occurs on the dev box. This is the first time a system is tested for deployment.

Build the required sources and thoroughly load test the system. Push data to a production system and ensure that the components scale well and efficiently. Create a test plan before testing in this phase.

# Phase 6

TODO

# Phase 7

TODO

# Phase 8

TODO

# Phase 9

TODO

# Phase 10

TODO

# Phase 11

TODO

# Phase 12

TODO

# Total System

Additional phases may be added to this document as scope increases. The goal of the system is to provide a working backend platform capable of scaling from a mini-pc to tens of thousands of nodes in the cloud.