

# Конфигурируемые графы обработки данных в эксперименте Baikal-GVD: архитектура устойчивого научного ПО

А.Г. Соловьев, Т.В. Елзов, Б.А. Шайбонов

**Аннотация** — Статья посвящена развитию системы автоматизации обработки данных нейтринного телескопа Baikal-GVD. Ключевым шагом стало разделение логики отдельных задач и структуры их взаимосвязей: граф зависимостей вынесен из кода задач в конфигурационный файл. При этом вычислительная логика существующих задач не изменялась, старые сценарии обработки сохранены без нарушений, а новые — добавлены через конфигурацию. Обновленная архитектура использует возможности фреймворка Law — расширения Luigi, предоставляющего готовые интерфейсы для работы с удаленным хранилищем EOS и системами пакетной обработки HTCondor и SLURM. Достигнутые в результате модульность, возможность многократного использования задач и защищенность от ошибки воплощают на практике принципы устойчивого научного ПО. Подход может быть применен в других научных экспериментах.

**Ключевые слова** — Baikal-GVD, Luigi, Law, конфигурируемые графы, конфигурационные файлы, EOS, HTCondor, SLURM, устойчивое научное ПО.

## I. ВВЕДЕНИЕ

*А. Нейтринный телескоп Baikal-GVD и его программная инфраструктура*

Baikal-GVD (Gigaton Volume Detector) — строящийся нейтринный телескоп на озере Байкал. Его назначение — регистрация потоков астрофизических нейтрино высоких энергий. Телескоп имеет модульную архитектуру: базовые элементы (оптические модули) объединяются в секции, секции — в гирлянды, гирлянды — в кластеры. Каждый кластер функционирует как автономный детектор.

Благодаря такой модульной архитектуре установка ежегодно расширяется на 1–2 кластера, достигнув к настоящему времени эффективного объема 0.8 км<sup>3</sup>. Рост объемов данных и усложнение алгоритмов обработки требуют постоянной эволюции программной инфраструктуры.

Статья получена 22 июня 2026.

А.Г. Соловьев — старший научный сотрудник Лаборатории информационных технологий имени М.Г. Мещерякова, Объединенный институт ядерных исследований (e-mail: solovjev@jinr.ru); ORCID: <https://orcid.org/0009-0001-9170-362X>

Т.В. Елзов — ведущий инженер Лаборатории ядерных проблем им. В.П. Джелепова, Объединенный институт ядерных исследований (e-mail: telzhov@jinr.ru)

Б.А. Шайбонов — старший научный сотрудник Лаборатории ядерных проблем им. В.П. Джелепова, Объединенный институт ядерных исследований (e-mail: bair@jinr.ru)

Для обработки данных Baikal-GVD используется комплекс программ на C++/ROOT [1] в составе фреймворка BARS (Baikal Analysis and Reconstruction Software) [2]. Программы реализуют все этапы обработки: чтение сырых данных, формирование событий, реконструкцию мюонных треков и ливней заряженных частиц. Весь процесс обработки естественным образом представляется в виде ориентированного графа, где результат одних программ служит входными данными для последующих.

Запуск этих программ и контроль за их выполнением осуществляются **системой автоматизации** на Python. Эта система организует программы в граф обработки и обеспечивает их выполнение в нужном порядке с учетом зависимостей.

*В. Постановка задачи: разработка системы офлайн обработки*

Система автоматизации, называемая **системой быстрой обработки** (Fast Data Processing System) [3], функционирует с 2021 года. Она обеспечивает автоматическую обработку данных по мере их поступления и реализует два конвейера: облегченную обработку каждого файла, ориентированную на минимальную задержку, и полную — для завершенного сеанса набора данных (рана).

Обработка выполняется в облачной инфраструктуре ОИЯИ [4]. Каждая виртуальная машина обрабатывает данные одного кластера, что позволяет распараллелить обработку по числу кластеров.

В 2025 году возникла новая задача — обеспечить возможность повторной обработки большого объема накопленных ранее данных с использованием новых версий BARS. Для решения этой задачи требуется разработать **систему офлайн обработки** (Offline Data Processing System), которая дополнит существующую.

Такая повторная обработка уже не будет автоматической, а будет запускаться по требованию: набор ранов и состав задач, применяемых при обработке, будут указываться пользователем-физиком. Специфика научной группы — разработка и поддержка ведутся силами 1–2 человек — делает критичным требование: система должна позволять пользователю запускать офлайн обработку самостоятельно, не правя код задач и не привлекая разработчика.

## II. ИСХОДНАЯ АРХИТЕКТУРА И ЕЕ ОГРАНИЧЕНИЯ

Система автоматизации обработки данных Baikal-GVD построена на фреймворке Luigi [5]. Выбор обусловлен

двумя причинами: он в точности соответствует задаче (построить граф обработки и запустить его), а его открытый исходный код и легковесность не создают проблем с его освоением и последующим сопровождением построенной на его основе системы.

Luigi оперирует двумя абстракциями: Task (задача) — единица работы, и Target (цель) — маркер ее выполнения. В контексте Baikal-GVD каждая задача служит оберткой для одной программы BARS и определяет три метода: `run()` (запуск программы), `requires()` (задачи-предшественники), `output()` (объекты Target, соответствующие выходным файлам). Если все Target из `output()` уже существуют, Luigi считает задачу выполненной и не запускает ее повторно. Это обеспечивает идемпотентность: после сбоя или прерывания обработка возобновляется с задачи, на которой остановилась, а не с самого начала. На основе `requires()` Luigi автоматически строит граф зависимостей и выполняет задачи в нужном порядке.

В существующей системе быстрой обработки реализованы два таких графа, каждый из которых решает свою задачу [6].

**Пофайловый граф** (реализован в модуле `filetasks.py`) обрабатывает отдельные файлы по мере их поступления (рисунок 1). Его основное назначение — максимально быстро выдать предварительную оценку события. Это критически важно для участия Baikal-GVD в многоканальной астрономии: при регистрации кандидата в нейтринное событие высоких энергий система должна в течение нескольких минут сформировать и разослать алерт (координаты, время, оценку энергии) в глобальную сеть телескопов и обсерваторий для последующих наблюдений одного и того же астро-

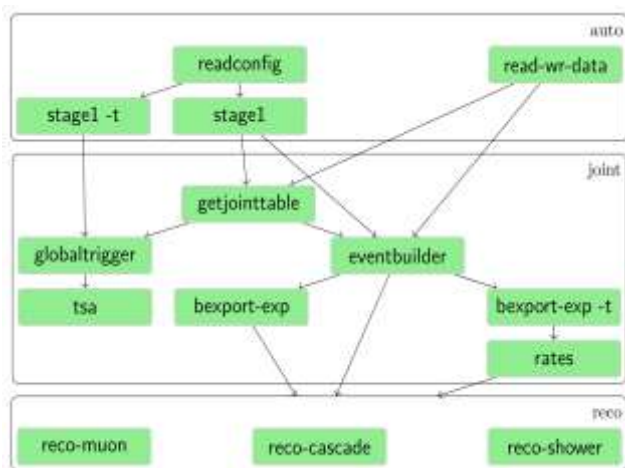


Рисунок 1: Пофайловый граф системы быстрой обработки  
номического объекта различными инструментами.

**Порановый граф** (реализован в модуле `runtasks.py`) обрабатывает целый ран после его завершения (рисунок 2). Этот граф использует более точные калибровки, данные акустической системы позиционирования и выполняет полную реконструкцию событий. Результат — финальные данные, пригодные для

физического анализа. Обработка одного рана занимает несколько часов после его окончания.

Выполнение этих графов осуществляется скриптами `process_file.py` и `process_run.py`. Каждый скрипт предоставляет около трех десятков опций командной строки. Помимо опций, задающих данные для обработки (`--season`, `--cluster`, `--run`), есть опции, включающие или исключающие определенные задачи (`--exclude-wr`, `--exclude-dqm`, `--joint-entry` и т. д.) и управляющие таким образом составом графа обработки. Каждая такая опция потребовала от разработчика добавления условной логики в метод `requires()` соответствующего класса.

Такой способ организации графа означает, что изменить порядок выполнения или состав задач без правки исходного кода невозможно. Если возникает потребность выполнить нестандартную последовательность, для которой не предусмотрены готовые опции командной строки, разработчику приходится добавлять новые опции и пробрасывать их через всю цепочку зависимостей. Эти изменения сопряжены с риском ошибок в зависимостях. Ошибка этого рода редко приводит к явному сбою — чаще она проявляется в потере параллелизма. Если разработчик по ошибке добавляет зависимость между двумя независимыми задачами, Luigi выстраивает их в последовательную цепочку. Обработка идет успешно, но занимает больше времени, при этом факт потери параллелизма остается незамеченным без пристального анализа графа.

Кроме того, чтобы оценить последствия вносимых изменений, необходимо видеть структуру графа целиком. Однако эта структура нигде не представлена в обозримом виде. Штатный веб-интерфейс Luigi для мониторинга недостаточно нагляден и отображает граф только в процессе выполнения, не позволяя анализировать его до запуска. Единственный способ понять устройство графа — изучать код, что при большом объеме файлов становится трудоемкой задачей.

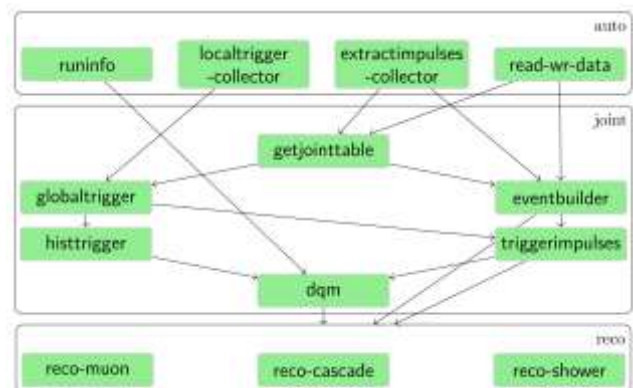


Рисунок 2: Порановый граф системы быстрой обработки.

Эти проблемы — жесткость графа (невозможность изменить его без правки кода) и непрозрачность структуры (невозможность увидеть граф целиком) — не дефекты реализации, а прямое следствие архитектурного решения: граф задается в коде задач. Задача офлайн об-

работки потребовала снять эти ограничения. Требования к новой системе сформулированы ниже.

### III. ТРЕБОВАНИЯ К СИСТЕМЕ ОФЛАЙН ОБРАБОТКИ

В отличие от системы быстрой обработки, система офлайн обработки должна обеспечивать иной режим работы. Прежде всего, большой объем данных (один сезон — более 2000 ранов) требует параллельной их обработки на кластере с сопоставимым количеством вычислительных узлов.

Среднее время обработки одного рана составляет 4–6 часов и практически не зависит от того, выполняется ли она на отдельной виртуальной машине или на узле кластера: длительность определяется вычислительной логикой BARS и объемом входных данных. Поскольку раны независимы, их можно обрабатывать параллельно.

Сценарий обработки каждого рана предполагает:

1. скачивание входных данных из общего хранилища (EOS) на вычислительный узел;
2. локальную обработку на вычислительном узле;
3. загрузку результатов обратно в общее хранилище (EOS, CephFS).

Для реализации такого сценария требуется:

- гибко, без правки кода существующих задач, настраивать:
  - **граф обработки** — какие задачи и в каком порядке выполняются (например, нужна не полная цепочка, показанная на рисунке 2, а ее часть, включающая только программы реконструкции);
  - **расположение данных** — какие файлы должны быть получены с хранилища на вычислительный узел перед обработкой и какие результаты должны быть отправлены обратно;
- запускать **пакетную обработку** на кластере — сотни и тысячи ранов одной командой, с автоматическим распределением по узлам.

Эти три требования — настраиваемый граф, работа с хранилищем и пакетный запуск — определили как архитектурные решения, так и выбор инструментов, рассматриваемые в следующих разделах.

### IV. АРХИТЕКТУРНОЕ РЕШЕНИЕ: КОНФИГУРИРУЕМЫЕ ГРАФЫ ОБРАБОТКИ

#### A. Задачи без зависимостей

Ключевая идея предлагаемой архитектуры — разделение вычислительной логики задач и структуры их взаимосвязей. Задача выполняет только свою непосредственную работу, а граф задается не в коде, а во внешнем конфигурационном файле. Это перекликается с идеей инверсии зависимости [7]: не задача управляет графом через жесткие ссылки на другие задачи, а граф, заданный конфигурацией, определяет, какие задачи и в каком порядке выполняются.

Первым шагом в реализации этой идеи стали задачи, не определяющие собственный метод `requires()`. Задача больше не содержит информацию о том, какие

задачи должны отработать перед ней. **Единственная ответственность** задачи — запуск программ BARS в ее методе `run()`, который остался неизменным.

В результате задачи стали **перепериспользуемыми** — одну и ту же задачу можно без каких-либо изменений ее кода использовать в разных графах обработки, предоставив необходимые входные данные. Модули `filetasks.py` и `runtasks.py` превратились из описаний двух монолитных графов в полноценные библиотеки задач.

#### B. Граф как конфигурация

Структура графа — какие задачи и в каком порядке выполняются — описывается в конфигурационном файле в формате YAML. На листинге 1 приводится фрагмент такого файла. Главное, что дает конфигурационный файл — **граф виден целиком** в одном месте, а не распределен по коду задач.

Другое преимущество — в этом же файле можно адресно задать и **параметры задач**: каждая задача получает только те параметры, которые нужны именно ей, и они указаны в ее секции `params` (например, `res_id` и `geom` на листинге 1). Раньше их приходилось передавать из аргументов командной строки через все промежуточные задачи: даже если параметр нужен только одной задаче, объявить его требовалось в каждой задаче цепочки.

```
tasks:
  JointRunEntry:
    requires:
      - "EventBuilderRun"
      - "TriggeredImpulseRun"
      - "DQMCLiReview"
      - "DQMCLi"

  RecoMuonRun:
    requires:
      - "GeomExportRootRun"
      - "JointRunEntry"
    params:
      res_id: "scl_nu_promptReco"
      geom: "hybrid"

  RecoCascadeRun:
    requires:
      - "GeomExportRootRun"
      - "JointRunEntry"
    params:
      res_id: "prompt"
      geom: "hybrid"
```

Листинг 1: Фрагмент конфигурации графа для офлайн обработки.

Тем самым конфигурационный файл собирает в одном месте все, что определяет конкретный сценарий обработки: состав задач, их связи и параметры. Пользователь может прочитать конфигурационный файл, понять структуру обработки и изменить ее в соответствии со своими задачами. Разработчик системы автоматизации при этом не нужен.

Следует, однако, отметить, что конфигурационный файл определяет порядок запуска, но не снимает проблему согласования входных данных между задачами. Знание о входных файлах зашито в код программ BARS и раньше было жестко зафиксировано в коде запускаю-

щих их задач. Теперь эта проблема в явном виде стоит перед тем, кто пишет конфигурационный файл.

Конфигурационный файл, сохраненный в системе контроля версий, служит **документом**, фиксирующим, какая именно цепочка обработки применялась к данному набору данных — это важно для воспроизводимости научного результата.

### С.Связывание задач

Работоспособность идеи обеспечивают два класса. Бэк-энд `PipelineLoader` отвечает за всю работу с конфигурационным файлом. Используя его фронтенд `RequiresMixin` — примесь, добавляющая задачам метод `requires()`. Теперь этот метод не переопределяется в коде каждой задачи, а реализуется программно на основе конфигурации.

Примесь `RequiresMixin` включена в базовый класс всех наших задач, поэтому ее поведение автоматически получает каждая задача. Метод `requires()`, добавляемый примесью, обращается к `PipelineLoader` и возвращает список зависимостей из секции `requires` конфигурации для данной задачи.

При этом задачи, у которых метод `requires()` определен в коде, продолжают использовать его. Задача начинает получать зависимости из конфигурации только после удаления из нее `requires()`. Это позволяет переводить задачи на новую архитектуру по одной, не нарушая работу системы.

Кроме того, примесь отвечает за параметры. При создании любой задачи — и той, что запускает пользователь, и тех, что создаются как ее зависимости — `PipelineLoader` находит задачу в конфигурации по имени класса и присваивает ее атрибутам значения из секции `params`. Если один и тот же параметр передан и через командную строку, и через конфигурацию, изменяется значение из конфигурации — это гарантирует, что сценарий обработки, зафиксированный в файле, не может быть случайно изменен аргументами запуска. Каждая задача-зависимость при создании получает те же параметры, что и текущая, если в конфигурации не указано иное.

`PipelineLoader` при первом обращении читает конфигурационный файл и проверяет его корректность: существование задач, указанных в зависимостях, наличие соответствующих классов в зарегистрированных модулях, отсутствие циклов в графе. Ошибки выявляются до запуска обработки. Результат кэшируется, так что при последующих обращениях файл не перечитывается.

Помимо простого задания зависимости строкой с именем задачи `PipelineLoader` поддерживает расширенную спецификацию словарем, который дополнительно к имени задачи может включать:

- `params` — параметры, передаваемые этой задаче. Если значение параметра содержит выражение в фигурных скобках, оно вычисляется в контексте остальных параметров задачи. На листинге 2 задача `Stage1` получает параметр

`file`, вычисленный как `{file - 1}`, — на единицу меньше, чем у текущей задачи;

- `when` — условие, при котором зависимость включается в граф (например, `"file > 1"` или `not exclude_dqm`) и вычисляется в контексте параметров задачи. На листинге 2 зависимость `Stage1` включается в граф только при `file > 1`.
- `for_each` — итератор, размножающий зависимость: для каждого значения, возвращаемого итератором, создается отдельный экземпляр задачи. Итераторы регистрируются отдельно и могут добавляться без изменения кода `PipelineLoader`. На листинге 3 для каждого номера секции, полученного от итератора `masters()`, создается экземпляр `ExtractedImpulsesCollector`.

Следует различать два уровня возможностей конфигурации. Первый — статический граф: список задач с параметрами и зависимостями, достаточный для офлайн-обработки и не требующий от пользователя ничего, кроме понимания последовательности обработки. Второй — программируемый генератор графа: условная логика (`when`), итераторы (`for_each`) и выражения в параметрах.

Расширенные возможности покрывают сценарии, задаваемые опциями командной строки скриптов `process_file.py` и `process_run.py`, и реализованы для последующего постепенного вывода этих старых точек входа из эксплуатации. Дальнейшего усложнения конфигурации не ожидается: обработка данных `Baikal-GVD` сформировалась, и интенсивного роста числа задач не предполагается.

```
GetJointTableFile:
  requires:
    - "Stage1"
    - task: "Stage1"
      when: "file > 1"
  params:
    file: "{file - 1}"
```

Листинг 2: Условная зависимость с выражением в параметре.

```
EventBuilderRun:
  class: "EventBuilderRun"
  requires:
    - "RunInfo"
    - "ReadConfig"
    - "GetJointTableRun"
  task: "ExtractedImpulsesCollector"
  for_each: "sdc in masters(season, cluster,
                           include_sync_master=True,
                           include_white_rabbit=False)"
```

Листинг 3: Пример итератора.

Готовые инструменты (`Airflow` [8], `Snakemake` [9]) обеспечивают сопоставимую гибкость, но их интеграция потребовала бы написания и сопровождения предметно-специфичного кода адаптации. Кроме того, переход на сторонний оркестратор означал бы либо остановку действующей быстрой обработки на время миграции,

либо поддержание двух параллельных систем с дублированием задач.

Напротив, система уже построена на Luigi, а Law предоставляет готовые бэкэнды для HTCondor и SLURM и абстракции удаленного хранилища как основные возможности, сопровождаемые сообществом физики высоких энергий. Переход на Airflow или Snakemake означал бы отказ от этой основы.

Другим преимуществом собственной реализации является возможность отладки конфигурации стандартными средствами Python: вся логика работы с конфигурацией сосредоточена в PipelineLoader, что позволяет использовать точки останова и логирование для анализа структуры графа.

PipelineLoader поддерживает композицию конфигурационных файлов через стандартную директиву YAML !inc: общие блоки задач выносятся в отдельный файл и подключаются из нескольких сценариев, что исключает дублирование.

#### V. ИНСТРУМЕНТАЛЬНЫЕ РЕШЕНИЯ: ХРАНИЛИЩЕ И ПАКЕТНАЯ ОБРАБОТКА

##### A. Фреймворк Law и его возможности

Law (Luigi Analysis Workflow) [10] — это надстройка над Luigi, разработанная для нужд научных вычислений. Выбор Law обусловлен тем, что он совместим с Luigi и расширяет его, а заложенные в нем возможности — работа с удаленными хранилищами, поддержка систем пакетной обработки и механизм параллелизма на уровне данных — прямо соответствуют требованиям, сформулированным в разделе III.

##### B. Работа с удаленным хранилищем

Данные Baikal-GVD хранятся в распределенной файловой системе EOS [11]. Law предоставляет для работы с такими данными базовый класс RemoteFileTarget, который берет на себя высокоуровневую логику, обеспечивая при этом кэширование, повторные попытки при сбоях, атомарное копирование. Для его работы требуется низкоуровневый бэкэнд — реализация интерфейса взаимодействия с конкретным хранилищем.

В составе Law есть готовый бэкэнд для EOS через библиотеку gfal2, однако ее установка в нашем программном окружении оказалась затруднительной. Поэтому мы реализовали собственный низкоуровневый бэкэнд — XRootDFileInterface, работающий через протокол XRootD.

Помимо обязательных файловых операций (проверка существования, копирование, создание директорий) в этом бэкэнде реализована функция преобразования локальных путей в пути на EOS и обратно. Иерархия каталогов, в которой хранятся данные, строго определена и едина для BARS и системы автоматизации. Она задана файлом directory.json из состава BARS. В системе автоматизации за формирование путей по этой иерархии отвечает класс PathConfig, который по заданным в конфигурации корневому пути и компонентам (сезон, кластер, тип данных — экспериментальные или смоделированные — номер рана и идентификатор версии

BARS) собирает путь для конкретного рана из его атрибутов.

Преобразование путей выполняется с помощью двух экземпляров класса PathConfig. Один, настроенный на directory.json, извлекает из локального пути к данным рана компоненты: сезон, кластер, тип данных, номер рана и идентификатор версии. Другой, параметризованный словарем eos\_config, собирает из тех же компонентов полный путь в EOS, следуя шаблону eosDataTemplate от корня eosDataRoot.

Словарь eos\_config можно определить в том же конфигурационном файле, что и структура графа. На листинге 4 показан фрагмент конфигурации, в котором в секции params задачи JointRunEntry появился этот словарь. Там же указан идентификатор версии BARS (identifier), которой были обработаны эти данные.

```
JointRunEntry:
  requires: [...]
  params:
    eos_config:
      eosDataRoot: "/eos/baikalgvd/processed"
      eosDataTemplate:
        - "season"
        - "cluster"
        - "datasource"
        - "datatype"
        - "identifier"
        - "run"
      identifier: "v2.3-79-gfefb58"

EOSUploader:
  requires: ["RecoMuonRun", "RecoCascadeRun"]
  params:
    eos_config:
      eosDataRoot: "/eos/baikalgvd/processed"
      eosDataTemplate:
        - "season"
        - "cluster"
        - "datasource"
        - "datatype"
        - "identifier"
        - "run"
      identifier: "v2.6"
```

Листинг 4: Конфигурация EOS для загрузки (JointRunEntry) и выгрузки (EOSUploader) данных.

На основе этой конфигурации JointRunEntry при старте определяет, откуда скачать файлы, и для каждого вызывает функцию копирования на локальную машину copy\_to\_local(). Если запрошенный файл в EOS отсутствует, задача JointRunEntry завершается с ошибкой. В лог выводится полный путь к отсутствующему файлу, включая сконструированный из конфигурации путь в EOS.

Отправку результатов в EOS выполняет задача EOSUploader, конфигурация которой также показана в листинге 4. Для каждого выходного файла задач, указанных в ее секции requires, она создает RemoteFileTarget со своим eos\_config и вызывает copy\_from\_local() — обратную операцию, копирующую локальный файл в EOS.

При необходимости результаты могут быть направлены в другое хранилище без изменения кода задач. Например, для записи в CephFS, подмонтированную к

узлам кластера, достаточно реализовать класс `FileCopier`, копирующий файлы в указанную директорию, и заменить точку входа с `EOSUploader` на `FileCopier` в конфигурационном файле. Благодаря композиции конфигураций новая конфигурация ссылается на основную, переопределяя только изменившуюся часть.

Таким образом, загрузка данных из EOS и отправка результатов обратно оформлены как задачи графа. Пути к данным задаются в конфигурации наравне с зависимостями и параметрами задач.

### С.Пакетная обработка

Law предоставляет встроенный тип задачи — `Workflow` — для параллельной обработки независимых порций данных.

Для сценария офлайн обработки этот функционал реализован в классе `OfflineProcessingWorkflow`, показанном в листинге 5. Этот класс — точка входа для офлайн обработки: пользователь запускает его одной командой, указав список ранов и конфигурационный файл.

```
class OfflineProcessingWorkflow(
    law.LocalWorkflow,
    SlurmWorkflow,
    HTCondorWorkflow,
    law.Task
):
    runs_list_file = luigi.Parameter(
        default=None, significant=False)
    processing_config = luigi.DictParameter(
        default={}, significant=False)

    pipeline_config = luigi.Parameter(
        default="offline_v2.6.yaml")

    def create_branch_map(self):
        """Каждый ран — отдельная ветвь"""
        return {
            i: {
                'season': s,
                'cluster': c,
                'runnum': r
            }
            for i, (s, clusters) in enumerate(
                self.processing_config.items())
            for c, runs in clusters.items()
            for r in runs
        }

    def run(self):
        """Запуск графа обработки
        для текущего рана"""
        params = self.branch_data
        loader = PipelineLoader.for_format(
            self.pipeline_config)
        task = loader.build_task(
            loader['entry_point'],
            season=params['season'],
            cluster=params['cluster'],
            runnum=params['runnum'],
            pipeline_config=self.pipeline_config,
        )
        yield task
```

Листинг 5: Точка входа офлайн обработки — класс `OfflineProcessingWorkflow`.

Метод `create_branch_map()` возвращает словарь, где ключ — номер ветви, а значение — данные,

необходимые этой ветви (`branch_data`). Каждой ветви соответствует один ран. Метод `run()` запускает для этого рана граф обработки, заданный конфигурационным файлом. Путь к нему передан через параметр `pipeline_config`.

Где именно будут выполнены получившиеся ветви — на локальной машине или на узлах кластера — определяют бэкэнды Law, перечисленные в родительских классах: `LocalWorkflow`, `SlurmWorkflow` и `HTCondorWorkflow`. Конкретный бэкэнд выбирается один раз при запуске опцией `--workflow`. В текущей эксплуатации Baikal-GVD используется HTCondor. В прототипе рассматривался бэкэнд SLURM. Код вычислительной логики при этом не меняется. Настройка бэкэнда сводится к определению нескольких методов в классе, наследующем базовый бэкэнд Law. Для HTCondor в методе `htcondor_job_config` задаются образ контейнера и требования к узлу — число ядер, объем памяти.

Сбой отдельной ветви не затрагивает остальные: они продолжают выполняться. По умолчанию Law выполняет пять повторных запусков неудачных задач, это число может быть изменено в конфигурационном файле `law.cfg`. В бэкэнде HTCondor можно задать предельное время выполнения, по истечении которого планировщик будет принудительно завершать зависшие ветви.

Таким образом, `OfflineProcessingWorkflow` размножает граф обработки на множество ранов, а бэкэнды Law обеспечивают выполнение задач на доступных вычислительных ресурсах.

## VI. ИНФРАСТРУКТУРНОЕ РЕШЕНИЕ: ВЫХОД НА КЛАСТЕР

Выполнение задач на сотнях и тысячах узлов кластера требует единого программного окружения, доступного на каждом узле без предварительной настройки. Эта задача решается интеграцией системы автоматизации со стандартными для физики высоких энергий технологиями: распределенной файловой системой CVMFS, контейнеризацией Singularity и системой пакетной обработки HTCondor (рисунок 3). Такая связка хорошо зарекомендовала себя в экспериментах на LHC и других крупных научных проектах; ее применение в Baikal-GVD позволило не разрабатывать специализированное решение.

Программное окружение собирается в три слоя (операционная система и системные библиотеки, BARS, система автоматизации на Luigi и Law), конвертируется в песочницу Singularity и публикуется в репозитории CVMFS. Песочница, в отличие от монолитного SIF-файла, представляет собой директорию с файловой системой образа, что удобно для распространения через CVMFS: файлы загружаются по мере обращения и кэшируются локально на каждом узле. После публикации образ становится доступен всем узлам кластера.

При запуске офлайн обработки Law формирует задание для HTCondor, в котором среди прочих параметров указан путь к образу в CVMFS (директива `+SingularityImage`). Планировщик HTCondor

направляет задание на свободный узел, где запускается контейнер Singularity с этим образом. Внутри контейнера Law выполняет граф обработки: загружает входные данные из EOS на локальный диск узла, запускает задачи, отправляет результаты обратно. Ни настройка окружения на узле, ни запуск контейнера не требуют вмешательства пользователя — все выполняется автоматически.

Обработка запустится на кластере, число доступных узлов которого сопоставимо с числом ранов в сезоне. Однако развертывание на полном сезоне не проводилось: система находится в стадии внедрения, поэтому количественная оценка масштабируемости — предмет дальнейшей работы.

#### В. Характеристики полученного решения

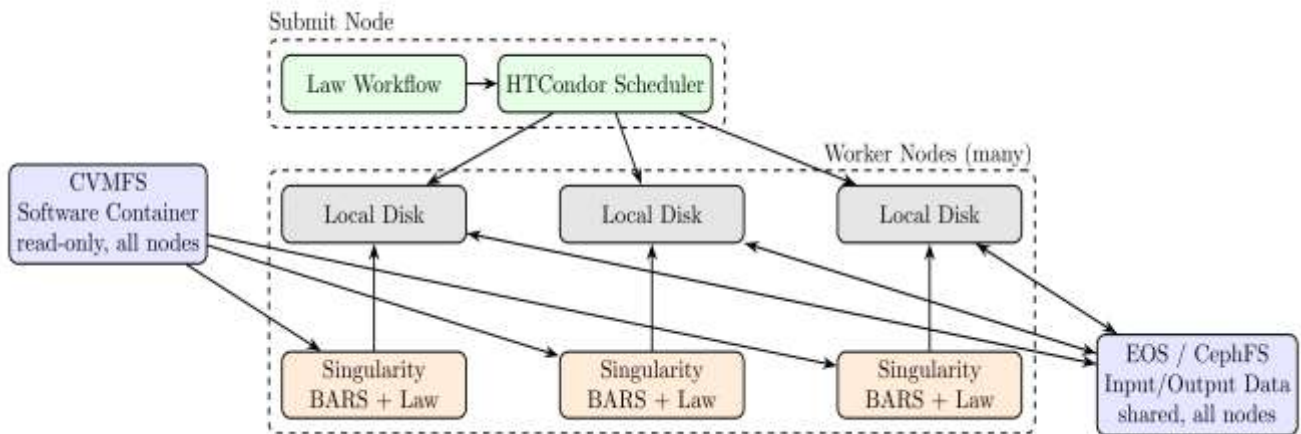


Рисунок 3: Инфраструктура для выполнения офлайн обработки на кластере.

## VII. ЗАКЛЮЧЕНИЕ

### А. Сценарий использования

Для запуска офлайн обработки пользователю предоставлены три компонента, подготовленные разработчиком: программа `OfflineProcessingWorkflow`, конфигурационный файл с описанием графа и параметрами хранилища, и образ контейнера с BARS и системой автоматизации, развернутый на всех узлах кластера через CVMFS.

Действия пользователя сводятся к формированию списка ранов и выполнению одной команды<sup>1</sup> на узле доступа к кластеру:

```
law run OfflineProcessingWorkflow \
  --workflow htcondor \
  --pipeline-config offline_v2.6.yaml \
  --runs-list-file runs.txt
```

`OfflineProcessingWorkflow` читает список ранов, вызов `create_branch_map()` разбивает их на независимые ветви, и HTCondor распределяет ветви по узлам кластера. На каждом узле внутри контейнера Singularity выполняется граф обработки. Ход выполнения можно отслеживать через стандартные средства Law и HTCondor.

Таким образом, для офлайн обработки данных с новой версией BARS пользователю достаточно выбрать конфигурационный файл, сформировать список ранов и выполнить одну команду. Ни правки кода задач, ни вмешательства разработчика не требуется.

Для систематизации изменений, произошедших в системе, мы используем категории качества, определенные в ГОСТ Р ИСО/МЭК 25010-2015 [12]. Это дает нам терминологическую основу, позволяющую структурировать качественное сравнение с исходной архитектурой.

#### • Сопровождаемость.

- **Модульность.** В исходной архитектуре зависимости между задачами определялись непосредственно в их коде: класс, реализующий вычислительную логику, одновременно задавал свое место в графе обработки. После рефакторинга задачи стали независимы: их код не содержит ссылок на другие задачи, а связи вынесены в конфигурационный файл. Граф и задачи можно менять независимо.
- **Возможность многократного использования.** В исходной архитектуре задача была жестко привязана к конкретным предшественникам через метод `requires()`. Использовать ее в другом графе — с иным набором зависимостей — без правки кода было невозможно. В новой архитектуре одни и те же классы задач применяются и в быстрой, и в офлайн обработке — различие задается исключительно конфигурационными файлами.
- **Модифицируемость.** Ранее изменение состава графа требовало правки кода на Python в нескольких классах и добавления опций командной строки — действий, которые мог выполнить только разработчик.

<sup>1</sup>Перед первым запуском необходимо однократно выполнить две вспомогательные команды: `source setup.sh` и `law index`.

Теперь достаточно изменить конфигурационный файл; это может сделать физик, не владеющий внутренним устройством системы автоматизации.

- *Анализируемость.* В исходной архитектуре структура графа была распределена по методам `requires()` нескольких классов; единственным способом понять ее было изучение кода. В новой архитектуре граф описан в одном конфигурационном файле, связи между задачами видны явно.
- **Удобство использования.**
  - *Изучаемость.* Пользователю больше не требуется извлекать структуру графа из кода задач — граф компактно описан в конфигурационном файле. Это снижает порог входа: участник коллаборации, понимающий алгоритм обработки, может прочитать и изменить сценарий для своих нужд.
  - *Защищенность от ошибки.* В исходной архитектуре ошибка в зависимостях могла оставаться незамеченной длительное время. В новой архитектуре граф весь на виду, а валидация выявляет проблему на старте. В контексте пакетной обработки автоматический повтор упавших ветвей снижает риск потери результатов, а независимость ветвей исключает влияние сбоя на остальные раны.
- **Функциональная пригодность.**
  - *Функциональная полнота.* К действующей системе быстрой обработки, работающей в автоматическом режиме по мере поступления данных, добавлена система офлайн обработки, позволяющая выполнять переобработку архивных данных с новыми версиями BARS по требованию.

Таким образом, внедрение описанных в настоящей статье решений позволило качественно улучшить сопровождаемость, удобство использования и функциональную полноту программного комплекса без остановки его работы.

Чтобы количественно подтвердить улучшение *модифицируемости* и *изучаемости*, оценим, во что обошлось бы добавление двух задач — `EOSUploader` и `FileCopier` — и двух сценариев с их участием в старой архитектуре. В новой архитектуре эти задачи заняли ~40 строк Python (только вычислительная логика), а их подключение к обработке — 18 строк декларативного описания в конфигурационном файле (весь этот файл занимает ~50 строк). В старой архитектуре помимо добавления самих этих задач (те же ~40 строк кода) потребовались бы дополнительные правки: объявления зависимостей и параметров, опции командной строки для проброса параметров и выбора точки входа, а также доработка механизма построения путей — суммарно ~100 строк кода в пяти файлах.

### С.Извлеченные уроки

Работа над системой автоматизации Baikal-GVD — частный случай более общей проблемы: как развивать научное ПО в условиях, когда разработка ведется силами 1–2 человек, а код должен работать годами и адаптироваться к меняющимся научным требованиям. В ходе подобных работ происходит осознание общих принципов, ведущих к созданию **устойчивого научного ПО**. Результатом проделанной работы стало практическое подтверждение и уточнение принципов, сформулированных в работе [13] и согласующихся с практиками, рекомендованными сообществом Software Carpentry [14].

- **Использовать работающий код для поэтапного развития системы.** Задачи, лежащие в основе быстрой обработки, не были переписаны полностью — зависимости вынесены в конфигурацию, вычислительная логика сохранена без изменений. Это позволило расширить *функциональную полноту* системы, добавив офлайн обработку без миграции на другой инструмент и без нарушения работы действующей системы. Такой способ развития — новая функциональность надстраивается рядом со старой, используя те же строительные блоки, — соответствует паттерну «удушающий фикус» (Strangler Fig) [15], примененному не к системе в целом, а к способу конфигурирования графа.
- **Следовать конвейерной архитектуре.** Конвейерная архитектура — сквозной принцип программного обеспечения Baikal-GVD: она реализована и на уровне системы автоматизации (графы Luigi/Law), и внутри самих программ BARS, каждая из которых реализует конвейер обработки событий. Данная работа следовала этому принципу. Офлайн обработка не стала отдельным инструментом — она построена на тех же задачах и в той же среде, что и быстрая обработка, что повысило *изучаемость* системы в целом.
- **Выделять абстракции, соответствующие предметной области.** В ходе работы были осознаны и разделены две ключевые абстракции: задача (запуск одной программы BARS) и граф (структура обработки). Вынос графа в конфигурацию обеспечил *возможность многократного использования* задач. В результате повысилась *модифицируемость* системы: граф и задачи можно менять независимо друг от друга.
- **Выбирать внешние библиотеки, доступные для освоения своими силами.** Критериями служат легковесность, качество документации и прозрачность внутреннего устройства. Фреймворки Luigi и Law удовлетворяют этим критериям. Выбор таких инструментов обеспечил *сопровождаемость* системы автоматизации силами группы.
- **Снижать зависимость пользователя от разработчика.** Этот принцип подчеркивался на

протяжении всей работы. Следование ему повышает *удобство использования* и снижает нагрузку на разработчика: пользователь, которому нужно выполнить офлайн обработку, запускает ее одной командой — разработчик не требуется ни на этапе подготовки, ни на этапе выполнения. Этот принцип прямо соответствует рекомендации Software Carpentry об автоматизации повторяющихся операций [14].

- **Сопровождать разработку научными публикациями.** Описание архитектурных решений в научных статьях служит одновременно документацией, средством верификации и гарантией долгосрочной доступности, что напрямую влияет на *удобство использования* и *сопровождает* программное обеспечение. Данная статья выполняет именно эту функцию по отношению к описанной архитектуре.

Эти принципы проверены в условиях, типичных для научного эксперимента: отсутствие штата разработчиков, программное окружение должно развиваться без остановки действующей обработки данных. Сходство выводов, полученных независимо в разных научных проектах — обработка данных нейтринного телескопа и обработка данных малоуглового рассеяния нейтронов [13], — говорит в пользу их более широкой применимости.

#### БЛАГОДАРНОСТИ

Авторы выражают признательность коллаборации Baikal-GVD за предоставление материалов и условий для работы.

#### БИБЛИОГРАФИЯ

- [1] R. Brun, F. Rademakers, «ROOT — An object oriented data analysis framework», Nuclear Instruments and Methods in Physics Research A. 1997. Vol. 389, Issues 1–2, pp. 81–86. DOI: 10.1016/S0168-9002(97)00048-X
- [2] A.D. Avrorin et al. (Baikal-GVD Collaboration), «Automatic data processing for Baikal-GVD neutrino observatory», Proceedings of Science. 2021. Vol. ICRC2021, pp. 1040. DOI: 10.22323/1.395.1040.
- [3] V.A. Allakhverdyan et al. (Baikal-GVD Collaboration), «Design and implementation of the fast data processing system for the Baikal-GVD neutrino telescope», Journal of Instrumentation, 2026. Vol. 21. DOI: 10.1088/1748-0221/21/01/P01025
- [4] A.V. Baranov, N.A. Balashov, N.A. Kutovskiy et al. «JINR cloud infrastructure evolution» Phys. Part. Nuclei Lett. Vol. 2016. 13, № 5(203): 672.
- [5] Spotify, «Luigi python package», Documentation: <https://luigi.readthedocs.io/>. GitHub: <https://github.com/spotify/luigi>
- [6] A.G. Solovjev and et al (Baikal-GVD Collaboration), «Review of software for automatic processing and analysis of data from the Baikal-GVD neutrino observatory», In Parallel Computational Technologies, volume 2241 of Communications in Computer and Information Science, pages 80–91, Cham, 2024. Springer. Presented at 18th International Conference on Parallel Computational Technologies (PCT 2024), Chelyabinsk, Russia. DOI: 10.1007/978-3-031-73372-7\_6
- [7] R.C. Martin, «Agile Software Development: Principles, Patterns, and Practices», 2002. Prentice Hall.
- [8] Apache Airflow. Documentation: <https://airflow.apache.org/docs/>. GitHub: <https://github.com/apache/airflow>
- [9] F. Mölder et al. "Sustainable data analysis with Snakemake", F1000Research, 2021. Vol. 10, p. 33. DOI: 10.12688/f1000research.29032.2
- [10] M. Rieger, «Law: Luigi Analysis Workflow», Documentation: <https://law.readthedocs.io/>. GitHub: <https://github.com/riga/law>
- [11] A.J. Peters, E.A. Sindrilaru, G. Adde «EOS as the present and future solution for data storage at CERN» J. Phys.: Conf. Ser. 2015. Vol. 664. P. 042042.
- [12] ГОСТ Р ИСО/МЭК 25010-2015. Информационные технологии. Системная и программная инженерия. Требования и оценка качества систем и программного обеспечения (SQuaRE). Модели качества систем и программных продуктов. — М.: Стандартинформ, 2015.
- [13] А.Г. Соловьев и др. «Принципы устойчивого научного ПО: опыт разработки программы обработки данных малоуглового рассеяния нейтронов», Компьютерные исследования и моделирование, 2026. Т. 18, № 2, сс. 335–358.
- [14] G. Wilson et al. «Best Practices for Scientific Computing» PLOS Biology. 2014. Vol. 12, No. 1. P. E1001745.
- [15] M. Fowler. "StranglerFigApplication", 2004. <https://martinfowler.com/bliki/StranglerFigApplication.html>

# Configurable data processing pipelines in the Baikal-GVD experiment: architecture of sustainable scientific software

A.G. Solovjev, T.V. Elzhov, B.A. Shaybonov

**Abstract** — The article is devoted to the development of an automated data processing system for the Baikal-GVD neutrino telescope. The key step was the separation of the logic of individual tasks from the structure of their interconnections: the dependency graph was moved out of the task code and into a configuration file. At the same time, the computational logic of the existing tasks remained unchanged, previous processing pipelines were preserved without disruption, and new ones were added through configuration. The updated architecture leverages the capabilities of the Law framework — an extension of Luigi that provides ready-to-use interfaces for working with the EOS remote storage and the HTCondor and SLURM batch processing systems. The resulting modularity, reusability of tasks, and robustness against errors embody the principles of sustainable scientific software in practice. This approach can be applied in other scientific experiments.

**Keywords** — Baikal-GVD, Luigi, Law, configurable pipelines, configuration files, EOS, HTCondor, SLURM, sustainable scientific software.

## REFERENCES

- [1] R. Brun, F. Rademakers, «ROOT — An object oriented data analysis framework», Nuclear Instruments and Methods in Physics Research A. 1997. Vol. 389, Issues 1–2, pp. 81–86. DOI: 10.1016/S0168-9002(97)00048-X
- [2] A.D. Avrorin et al. (Baikal-GVD Collaboration), «Automatic data processing for Baikal-GVD neutrino observatory», Proceedings of Science. 2021. Vol. ICRC2021, pp. 1040. DOI: 10.22323/1.395.1040.
- [3] V.A. Allakhverdyan et al. (Baikal-GVD Collaboration), «Design and implementation of the fast data processing system for the Baikal-GVD neutrino telescope», Journal of Instrumentation, 2026. Vol. 21. DOI: 10.1088/1748-0221/21/01/P01025
- [4] A.V. Baranov, N.A. Balashov, N.A. Kutovskiy et al. «JINR cloud infrastructure evolution» Phys. Part. Nuclei Lett. Vol. 2016. 13, № 5(203): 672.
- [5] Spotify, «Luigi python package», Documentation: <https://luigi.readthedocs.io/>. GitHub: <https://github.com/spotify/luigi>
- [6] A.G. Solovjev and et al (Baikal-GVD Collaboration), «Review of software for automatic processing and analysis of data from the Baikal-GVD neutrino observatory», In Parallel Computational Technologies, volume 2241 of Communications in Computer and Information Science, pages 80–91, Cham, 2024. Springer. Presented at 18th International Conference on Parallel Computational Technologies (PCT 2024), Chelyabinsk, Russia. DOI: 10.1007/978-3-031-73372-7\_6
- [7] R.C. Martin, «Agile Software Development: Principles, Patterns, and Practices», 2002. Prentice Hall.
- [8] Apache Airflow. Documentation: <https://airflow.apache.org/docs/>. GitHub: <https://github.com/apache/airflow>
- [9] F. Mölder et al. "Sustainable data analysis with Snakemake", F1000Research, 2021. Vol. 10, p. 33. DOI: 10.12688/f1000research.29032.2
- [10] M. Rieger, «Law: Luigi Analysis Workflow», Documentation: <https://law.readthedocs.io/>. GitHub: <https://github.com/riga/law>
- [11] A.J. Peters, E.A. Sindrilaru, G. Adde «EOS as the present and future solution for data storage at CERN» J. Phys.: Conf. Ser. 2015. Vol. 664. P. 042042.
- [12] GOST R ISO/IEC 25010-2015. Information technology. Systems and software engineering. Systems and software quality requirements and evaluation (SQuaRE). System and software quality models. Moscow: Standartinform, 2015. (in Russian)
- [13] Solovjev A.G. et al. Principles of sustainable scientific software: lessons from developing a data processing program for small-angle neutron scattering // Computer Research and Modeling, 2026, vol. 18, no. 2, pp. 335-358. (in Russian)
- [14] G. Wilson et al. «Best Practices for Scientific Computing» PLOS Biology. 2014. Vol. 12, No. 1. P. e1001745.
- [15] M. Fowler. "StranglerFigApplication", 2004. <https://martinfowler.com/bliki/StranglerFigApplication.html>