Санкт-Петербургский государственный университет

Кафедра, информационно-аналитических систем

Группа 23.Б09-мм

Desbordante: улучшение инфраструктуры

ГАРАЕВ Тагир Ильгизович

Отчёт по учебной практике
в форме «Решение»

Научный руководитель:
ассистент кафедры информационно-аналитических систем, Чернышев Г. А.

Санкт-Петербург
2026

Оглавление

Введение

Развитие высокопроизводительных систем, таких как библиотека для профилирования данных Desbordante [2], требует не только разработки алгоритмов, но и поддержки инфраструктуры. С ростом кодовой базы критически важным становится обновление внешних зависимостей и оптимизация процессов непрерывной интеграции. Игнорирование этих аспектов усложняет поддержку: устаревшие библиотеки блокируют обновление инструментов сборки, а несоответствие стандартам затрудняет статический анализ.

Проект Desbordante развивается на протяжении более семи лет. За это время экосистема C++ и подходы к CI/CD претерпели значительные изменения. Технические решения и зависимости, являвшиеся оптимальными на ранних этапах разработки, со временем естественным образом устарели. В настоящее время выявлен ряд проблем, препятствующих дальнейшему развитию. Во-первых, наличие неподдерживаемых библиотек (`frozen` [3], `atomicbitvector` [7], `better_enums` [1]) создает риски совместимости с новыми компиляторами. Во-вторых, существующая система CI/CD, спроектированная под меньшую нагрузку, выполняла ресурсоемкую сборку бинарных пакетов при каждом изменении, что стало избыточным для текущего темпа разработки.

Работа посвящена модернизации инфраструктуры Desbordante. Для устранения ограничений выполнена замена устаревших зависимостей на собственные реализации и актуальные аналоги с сохранением совместимости Python API. Для оптимизации цикла разработки внедрена стратегия CI, использующая механизм меток для управления ресурсоемкими задачами.

1. Постановка задачи

Целью работы является обновление инфраструктуры проекта Desbordante для устранения технического долга по зависимостям и оптимизации ресурсов непрерывной интеграции.

Для её выполнения были поставлены следующие задачи:

1. заменить неподдерживаемые внешние зависимости (`frozen`, `atomicbitvector`, `better_enums`) на собственные реализации или актуальные аналоги;

2. реализовать механизм трансляции имен перечислений для сохранения обратной совместимости Python API при переходе на нативные `enum class`;

3. оптимизировать сценарии CI/CD, внедрив логику выборочной сборки бинарных пакетов для сокращения времени проверки изменений;

4. провести апробацию решения: подтвердить корректность работы через регрессионное тестирование и оценить снижение нагрузки на сборочную инфраструктуру.

2. Обзор

В разделе описаны технологии и решения, необходимые для понимания работы. Проведен анализ альтернатив для обоснования выбора реализации.

2.1. Реализация ассоциативных массивов времени компиляции

Задача заключалась в замене библиотеки `frozen`, использовавшейся для создания неизменяемых словарей (`constexpr` map). Рассмотрены три подхода:

1. Идеальное хэширование (`frozen`). Библиотека строит идеальную хэш-функцию на этапе компиляции.

- *Преимущество:* поиск за $O(1)$.
- *Недостаток:* сложная внутренняя реализация; библиотека больше не поддерживается разработчиками.

2. Метапрограммирование (`boost::hana::map`). Использование библиотеки Boost.Hana, предназначенной для вычислений на этапе компиляции. Это мощный инструмент для создания гетерогенных контейнеров, где ключами могут выступать типы данных.

- *Преимущество:* дополнительных нет, соответствует требованиям.
- *Недостаток:* библиотека спроектирована для работы с типами, а не значениями; использование её для обычных словарей требует сложного синтаксиса и увеличивает время компиляции.

3. Линейный поиск (собственная реализация). Перебор по сортированному массиву `std::array`.

- *Преимущество:* работа с непрерывными блоками памяти обеспечивает высокую локальность кэша, что компенсирует алгоритмическую сложность на малых объемах данных [5]; отсутствие накладных расходов на вычисление хэша.

- *Недостаток:* асимптотическая сложность поиска $O(N)$.

Сравнительный анализ представлен в таблице 1.

Выбранное решение: линейный список, собственная реализация (`StaticMap`). Для задач сопоставления пар, где их количество невелико, простота реализации и отсутствие внешних зависимостей являются приоритетными факторами.

Таблица 1: Сравнение подходов к реализации Static Map

Решение	**Алгоритм**	**Статус / Зависимость**
`frozen`	Идеальное хэширование	Не поддерживается (Внешняя)
`boost::hana`	Метапрограммирование	Активна (Boost)
`StaticMap`	Линейный поиск	Внутренняя реализация

2.2. Атомарные битовые поля

Для многопоточных алгоритмов необходим битовый массив с поддержкой атомарных операций.

1. Стандартные контейнеры (`boost::dynamic_bitset`). Универсальный класс из библиотеки Boost.

- *Преимущество:* отсутствует.

- *Недостаток:* отсутствует потокобезопасность; требует использования внешних блокировок, что снижает производительность.

2. Библиотека `atomicbitvector`. Header-only библиотека, реализующая вектор над `std::atomic` (lock-free).

- *Преимущество:* высокая производительность за счет неблокирующих алгоритмов.

- *Недостаток:* репозиторий не поддерживается автором; подключение внешней зависимости ради одного файла усложняет сборку.

3. Перенос решения. Перенос кода `atomicbitvector` в утилиты проекта Desbordante.

- *Преимущество:* сохранение производительности lock-free алгоритмов при полном контроле над кодом; удаление неиспользуемого кода (итераторов).

- *Недостаток:* отсутствует.

Сравнение вариантов приведено в таблице 2.

Выбранное решение: перенос кода в проект. Это позволяет устранить риск исчезновения внешней зависимости и упростить конфигурацию сборки, сохранив высокую производительность.

Таблица 2: Варианты реализации атомарных битовых полей

Решение	**Механизм**	**Скорость**	**Статус**
`boost::bitset`	Блокировки (Mutex)	Низкая	Активна (Boost)
`atomicbitvector`	Неблокирующий (Lock-free)	Высокая	Не поддерживается
Перенос	Неблокирующий (Lock-free)	Высокая	Внутренняя

2.3. Статическая рефлексия перечислений

Для конфигурации алгоритмов через Python API необходима конвертация значений перечислений в строки и обратно. В стандарте C++20 встроенная рефлексия отсутствует, поэтому требуются сторонние решения.

1. Эмуляция (`better_enums`). Библиотека заставляет объявлять перечисления через специальные макросы препроцессора.

- *Преимущество:* генерируемые типы не являются стандартными `enum class`, что позволяло использовать стиль `snake_case` в обход проверок статического анализатора.

- *Недостаток:* библиотека не поддерживается; макросы усложняют отладку и поддержку кода.

2. Аннотации (`Boost.Describe`). Использование стандартных перечислений, но с необходимостью их регистрации через макросы Boost.

- *Преимущество:* дополнительных нет, соответствует требованиям.
- *Недостаток:* требует явного дублирования имен значений внутри макроса, что усложняет написание кода.

3. Расширения компилятора (`magic_enum`). Библиотека работает с обычным синтаксисом C++ (`enum class`), автоматически извлекая имена значений из служебной информации компилятора.

- *Преимущество:* нативный синтаксис языка без макросов; возвращает типы под контроль анализаторов кода.
- *Недостаток:* требует реализации механизма адаптации имен для сохранения совместимости с Python API.

Сравнение представлено в таблице 3.

Выбранное решение: `magic_enum`. Несмотря на необходимость реализации адаптера имен, это решение позволяет использовать стандартные средства языка и избавиться от неподдерживаемой библиотеки.

Таблица 3: Сравнение библиотек рефлексии

Библиотека	**Тип данных**	**Объявление**	**Статус**
`better_enums`	Структура (эмуляция)	Макросы	Не поддерживается
`Boost.Describe`	Enum	Макросы	Активна (Boost)
`magic_enum`	Enum class	Нативное	Активна

2.4. Стратегии оптимизации CI/CD

Сборка бинарных пакетов — самая длительная операция в CI. Рассмотрены три подхода к организации процесса.

1. Полная сборка. Безусловный запуск матрицы сборки на каждый Pull Request.

- *Преимущество:* гарантирует качество во всех конфигурациях.

- *Недостаток:* создает очередь задач, блокируя тестирование других изменений.

2. Частичная сборка (Partial Matrix). Сборка только граничных версий Python (старая/новая).

- *Преимущество:* отсутствует.
- *Недостаток:* ненадежность. Во-первых, разные версии Python имеют несовместимый ABI. Во-вторых, последние версии Python могут не иметь поддержки на macOS, а успех на Linux не гарантирует работу на macOS.

3. Сборка по требованию (Explicit Opt-in). Полная матрица запускается при релизах. В Pull Request сборка инициируется вручную через метку `python-packaging-risk`.

- *Преимущество:* баланс скорости и надежности; ресурсы не тратятся на рутинные задачи, но сохраняется возможность полной проверки критических изменений.
- *Недостаток:* отсутствует.

Выбранное решение: внедрена стратегия **сборки по требованию**. Это позволило существенно сократить время ожидания результатов CI для большинства изменений.

3. Решение

В разделе описаны технические решения по замене устаревших зависимостей и обеспечению совместимости между обновленным C++ ядром и Python-интерфейсом.

3.1. Ассоциативные массивы

Библиотеку `frozen`, применявшуюся для `constexpr` словарей, заменил шаблонный класс `util::StaticMap` (Листинг 1).

Решение использует `std::array` и алгоритме линейного поиска. Для задач сопоставления пар, где их количество невелико, это допустимый компромисс: устранение внешней зависимости важнее теоретического снижения асимптотики.

Листинг 1: Реализация поиска в StaticMap

```
template <typename Key, typename Value, std::size_t N>
class StaticMap {
    std::array<std::pair<Key, Value>, N> data_;
public:
    constexpr Value At(Key const& key) const {
        for (auto const& [k, v] : data_) {
            if (k == key) return v;
        }
        throw std::out_of_range("Key not found");
    }
};
```

3.2. Атомарные битовые поля

Логика удаленной библиотеки `atomicbitvector` (lock-free работа с битами) перенесена в кодовую базу проекта Desbordante. Код очищен от неиспользуемых итераторов, а управление памятью переведено на `std::unique_ptr`.

Класс предоставляет потокобезопасный интерфейс, что упрощает использование в многопоточных алгоритмах, таких как FAIDA (Листинг 2).

Листинг 2: Использование AtomicBitVector в алгоритме

```
// Объявление поля в классе
util::AtomicBitVector non_covered_cc_indices_;

// Использование в многопоточном коде
if (set_iter == inverted_index_.end()) {
    if (!non_covered_cc_indices_.Test(combination.GetIndex())) {
        non_covered_cc_indices_.Set(combination.GetIndex());
    }
    return false;
}
```

3.3. Миграция на magic_enum и адаптация API

Библиотека `better_enums` заменена на `magic_enum` для использования стандартных `enum class`.

Это активировало проверки статического анализатора, требующего именования `kCamelCase` (например, `kEuclidean`). Переименование привело код к стандарту, но нарушило совместимость с Python API:

1. **C++:** использует имена вида `kEuclidean`.
2. **Python:** согласно PEP 8 [6], ожидает `snake_case` (например, `euclidean`).

ля решения проблемы реализован механизм двусторонней трансляции имен (Листинг 3). Функции `KCamelToSnake` и `SnakeToKCamel` выполняют преобразование строк для работы с Python API.

Данный код используется в слое привязок для автоматической конвертации пользовательского ввода и генерации сообщений об ошибках.

Листинг 3: Реализация конвертера имен

```cpp
inline std::string KCamelToSnake(std::string_view input) {
    std::string out;
    out.reserve(input.size());
    size_t start = (input.size() > 1 && input[0] == 'k' && std::isupper(input[1])) ? 1 :
    ↪  0;

    for (size_t i = start; i < input.size(); ++i) {
        char c = input[i];
        if (std::isupper(c)) {
            if (i > start) out.push_back('_');
            out.push_back(std::tolower(static_cast<unsigned char>(c)));
        } else {
            out.push_back(c);
        }
    }
    return out;
}

inline std::string SnakeToKCamel(std::string_view input) {
    std::string out;
    out.reserve(input.size() + 1);
    out.push_back('k');

    bool cap_next = true;
    for (char c : input) {
        if (c == '_') {
            cap_next = true;
        } else if (cap_next) {
            out.push_back(std::toupper(static_cast<unsigned char>(c)));
            cap_next = false;
        } else {
            out.push_back(std::tolower(static_cast<unsigned char>(c)));
        }
    }
    return out;
}

template <typename E>
    requires std::is_enum_v<E>
std::string EnumToStr(E value) {
    return KCamelToSnake(magic_enum::enum_name(value));
}

template <typename E>
    requires std::is_enum_v<E>
std::optional<E> EnumFromStr(std::string_view str) {
    return magic_enum::enum_cast<E>(SnakeToKCamel(str));
}
```

3.4. Реализация сборки по требованию

Для экономии ресурсов в конфигурацию GitHub Actions добавлено условие: сборка бинарных пакетов выполняется только если:

1. Событие не является Pull Request'ом.
2. Pull Request имеет метку `python-packaging-risk`.

Реализация показана в Листинге 4. Запуск модульных тестов остается обязательным для всех событий.

Сценарий работы, задокументированный во внутренней вики[1] проекта Desbordante, предполагает следующий порядок действий. По умолчанию на Pull Request сборка пакетов не запускается, выполняются только тесты. Если изменения затрагивают систему сборки или требуют полной проверки, разработчик добавляет метку `python-packaging-risk`, что инициирует полный запуск матрицы. При снятии метки запущенный процесс автоматически отменяется.

Листинг 4: Условие запуска сборки

```
jobs:
  generate-linux-wheels-matrix:
    if: >-
      github.event_name != 'pull_request' ||
      contains(github.event.pull_request.labels.*.name,
               'python-packaging-risk')
    name: Generate Linux wheel matrix
    runs-on: ubuntu-latest
    # ...
```

[1] https://github.com/Desbordante/desbordante-core/wiki/Development%E2%80%90General#ci-%D1%81%D0%B1%D0%BE%D1%80%D0%BA%D0%B0-python-wheels

4. Апробация

4.1. Эффективность использования ресурсов CI

Для оценки стратегии выборочной сборки проанализированы метрики GitHub Actions. Наибольшая нагрузка приходится на агенты macOS, где время ожидания запуска задач достигало 42 минут.

Сравнение сценариев:

1. **Полная сборка:** сборка пакетов (`wheel.yml`) + тесты.
2. **Сборка по требованию:** только тесты (`core-tests.yml`).

Результаты представлены в Таблице 4. Главный критерий — цикл обратной связи (время от коммита до отчета CI) [4].

Таблица 4: Показатели производительности CI на macOS

Метрика	**Полная сборка**	**Сборка по требованию**
Параллельных задач	≈ 14	4
Время в очереди	≈ 42 мин	< 2 мин
Время выполнения	1 ч 14 мин	25 мин
Цикл обратной связи	≈ 1 ч 55 мин	≈ 27 мин

В базовом сценарии время ожидания складывалось из простоя в очереди и длительной сборки. В оптимизированном сценарии количество задач (4) не превышает лимиты платформы, что устраняет очередь и сокращает время получения результата в 4 раза.

4.2. Тестирование зависимостей

Работоспособность замены библиотек подтверждена тестами:

1. **StaticMap.** Модульные тесты Denial Constraints успешно пройдены. Это подтверждает, что замена хэш-таблиц на линейный поиск не изменила наблюдаемое поведение алгоритма.
2. **AtomicBitVector.** Тесты алгоритма FAIDA, запущенные с ThreadSanitizer, завершились без ошибок. Диагностических сообщений о гонках данных зафиксировано не было.

4.3. Проверка совместимости API

Работоспособность механизма трансляции проверена на существующем наборе тестов Python-биндингов.

В Листинге 5 приведен фрагмент конфигурации тестов. Видно, что параметры передаются в Python стиле `snake_case` (например, `"metric": "euclidean"`), несмотря на то, что внутри C++ код теперь использует `kEuclidean`.

Листинг 5: Фрагмент тестов совместимости (bindings-tests.py)

```
    (desb.mfd_verification.algorithms.MetricVerifier, [
        OptionContainer(
            "TestLong.csv",
            {},
            {
                "metric": "euclidean",
                "metric_algorithm": "approx",
                # ...
            },
        ),
        # ...
    ])

    def test_correct_load(self):
        for algo, option_containers in ALGO_CORRECT_OPTIONS_INFO:
            for option in option_containers:
                self._test_correct_option_setting(...)
```

Тесты подтвердили, что слой совместимости корректно преобразует строки, скрывая изменение внутренней реализации.

Заключение

Целью работы являлось обновление инфраструктуры проекта Desbordante для устранения технического долга по зависимостям и оптимизации ресурсов непрерывной интеграции.

В ходе работы были получены следующие результаты:

1. Заменены неподдерживаемые внешние зависимости (`frozen`, `atomicbitvector`, `better_enums`) на собственные реализации или актуальные аналоги.

2. Реализован механизм трансляции имен перечислений для сохранения обратной совместимости Python API при переходе на нативные `enum class`.

3. Оптимизированы сценарии CI/CD, внедрена логика выборочной сборки бинарных пакетов для сокращения времени проверки изменений.

4. Проведена апробация решения: подтверждена корректность работы через регрессионное тестирование и оценено снижение нагрузки на сборочную инфраструктуру.

Изменения, касающиеся оптимизации CI/CD, приняты в основную ветку проекта[2]. Замена зависимостей и адаптация API находится на стадии рецензирования[3].

[2] https://github.com/Desbordante/desbordante-core/pull/619
[3] https://github.com/Desbordante/desbordante-core/pull/659

Список литературы

[1] Bachin Anton. Better Enums: Compile-time enum to string, iteration, and more for C++. — 2024. — URL: https://github.com/aantron/better-enums.

[2] Desbordante: from benchmarking suite to high-performance science-intensive data profiler / George Chernishev, Michael Polyntsov, Anton Chizhov et al. // Proceedings of the 8th International Conference on Data Science and Management of Data (12th ACM IKDD CODS and 30th COMAD). — CODS-COMAD '24. — New York, NY, USA : Association for Computing Machinery, 2025. — P. 234–243. — URL: https://doi.org/10.1145/3703323.3703725.

[3] Guelton Serge. Frozen: A header-only, constexpr, zero-overhead std-like associative container library. — 2024. — URL: https://github.com/serge-sans-paille/frozen.

[4] Humble Jez, Farley David. Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. — Addison-Wesley Professional, 2010.

[5] Meyers Scott. Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library. — Addison-Wesley Professional, 2001.

[6] Van Rossum Guido, Warsaw Barry, Coghlan Nick. PEP 8 – Style Guide for Python Code. — https://peps.python.org/pep-0008/. — 2001. — Accessed: 2024-01-20.

[7] ekg. Atomic bitvector: an atomic bitset/bitvector with size determined at runtime. — URL: https://github.com/ekg/atomicbitvector.