Санкт-Петербургский государственный университет

Кафедра информационно-аналитических систем

Группа 24.Б41-мм

Модернизация CMake в проекте Desbordante

ГАВРИЛОВ Артём Евгеньевич

Отчёт по учебной практике
в форме «Производственное задание»

Научный руководитель:
ассистент кафедры ИАС, к. ф.-м. н., Чернышёв Г. А.

Санкт-Петербург
2026

Оглавление

Введение

Эффективная система сборки — фундамент для разработки сложного программного обеспечения. Она должна обеспечивать не только компиляцию кода, но и управление зависимостями, конфигурацию под различные платформы и интеграцию с инструментами тестирования. Проект Desbordante [1] — высокопроизводительный профилировщик данных — использует систему сборки CMake[1], наиболее распространенный индустриальный стандарт для проектов на языке C++ на сегодняшний день [4].

Подходы к написанию сценариев сборки CMake со временем значительно эволюционировали. Существующая система сборки проекта, основанная на устаревших парадигмах и зависимая от внешних bash-скриптов, перестала отвечать требованиям расширяемости, затрудняла добавление новых зависимостей и интеграцию в сторонние системы. Современный подход, известный как *Modern CMake*, предлагает переход от манипулирования глобальными свойствами к целеориентированной (target-based) модели, что в итоге позволяет минимизировать побочные эффекты и конфликты конфигураций в крупных проектах.

В рамках данной работы предстояло провести модернизацию сборочной системы Desbordante, направленную на повышение её чистоты, модульности, удобства использования и соответствие лучшим современным практикам разработки программного обеспечения.

[1]Официальный сайт CMake, URL: https://cmake.org (дата обращения: 2 января 2026 г.).

1. Постановка задачи

Цель работы — модернизация системы сборки проекта Desbordante на основе CMake. Для её выполнения были поставлены следующие задачи:

1. Провести обзор существующей конфигурации системы сборки, выявить ключевые проблемы.
2. Реорганизовать цели проекта: уточнить их области видимости, зависимости и свойства, локализовать глобальные настройки.
3. Устранить зависимость от вспомогательных bash-скриптов, интегрировав их функциональность непосредственно в CMake.
4. Выполнить замеры времени выполнения модернизированной системы.

2. Обзор

2.1. Анализ существующей инфраструктуры сборки

На момент начала работы процесс сборки и тестирования проекта базировался на разнородном наборе инструментов, включающем конфигурационные файлы CMake и вспомогательные bash-скрипты. Сборка осуществлялась посредством скрипта `build.sh`, отвечающего за передачу параметров конфигурации в CMake. Подготовка тестового окружения требовала предварительного запуска скрипта `pull_datasets.sh` для загрузки наборов данных, после чего запуск тестов производился утилитой `ctest`.

В ходе анализа текущей инфраструктуры были выявлены следующие недостатки:

1. **Платформозависимость инструментария.** Использование bash-скриптов в качестве единственной точки входа создает жесткую зависимость от Unix-подобного окружения. Это делает возможный будущий переход к нативной разработке в среде Windows затруднительнее, а также усложняет интеграцию проекта с IDE, ожидающих стандартное окружение CMake.

2. **Отсутствие изоляции конфигураций.** Артефакты сборки вне зависимости от выбранного типа (`Debug`, `Release`) сохраняются в единую директорию `build`. Это приводит к необходимости полной очистки кэша и перекомпиляции всего проекта при смене режима сборки, что существенно увеличивает время цикла разработки.

3. **Низкая степень автоматизации работы с данными.** Процесс получения тестовых данных не интегрирован в граф зависимостей системы сборки. Требуется ручное выполнение скрипта загрузки. В случае обновления удаленного репозитория с датасетами локальные данные теряют актуальность.

4. **Избыточность операций при подготовке данных.** Алгоритм работы `pull_datasets.sh` включает полное клонирование репозитория с историей, распаковку множества архивов и повторное перепаковывание файлов в единый архив. Кроме того, при каждом запуске тестов происходит избыточное копирование данных из исходной директории в директорию сборки.

2.2. Анализ сценариев сборки CMake

Описание сборки проекта выполняется с помощью команд CMake, считываемых из файлов `CMakeLists.txt` или файлов, имеющих расширение `.cmake`из. Эволюция данного инструмента, начиная с версии 3.0, ознаменовала смену парадигмы от строко-ориентированной обработки переменных и директорий к целе-ориентированной модели, известной как *Modern CMake* (с выходом версии 4.0 заявляют о *Post-Modern CMake* [2]). Данная концепция рассматривает проект как граф зависимостей между «целями» (targets) — логическими компонентами системы (библиотеками или исполняемыми файлами), инкапсулирующими свои свойства: пути к заголовочным файлам, флаги компиляции и линковки, определения препроцессора, исходные файлы и другие настройки. Важная особенность — управление областями видимости, позволяющее разграничить свойства, необходимые только для сборки самого компонента, и свойства, транзитивно передаваемые зависимым модулям.

Анализ файлов `CMakeLists.txt` проекта Desbordante показал, что конфигурация была построена на устаревших практиках, не соответствующих современным стандартам:

1. **Загрязнение глобальной области видимости.** Широкое использование команды `include_directories()` в корневом файле проекта приводит к тому, что пути поиска заголовков применяются ко всем целям, что нарушает принцип инкапсуляции и провоцирует конфликты имен при добавлении сторонних библиотек.

2. **Использование изменяемого глобального состояния.** Вместо настройки конкретных целей логика сборки опирается на глобальные переменные. Это делает поведение сборки зависимым от порядка подключения подкаталогов `add_subdirectory` и затрудняет понимание, какие настройки применяются к конкретному модулю.

3. **Использование шаблонов поиска файлов.** Для формирования списка исходных кодов применялась команда `file(GLOB)`, из-за которой список файлов фиксируется на этапе генерации файлов проекта, и при добавлении новых исходных файлов система сборки не инициирует автоматическую перегенерацию, а значит, требует от разработчика ручного перезапуска процесса, поэтому использование данной команды не рекомендуется разработчиками CMake[2].

2.3. Выводы

Проведенный обзор показывает, что существующая система сборки, хотя и выполняет базовые функции компиляции, обладает значительным техническим долгом, создающим существенные издержки при сопровождении проекта. Для устранения выявленных проблем необходима модернизация системы сборки с переходом на подход Modern CMake, интеграцией вспомогательных процессов в фазы генерации проектных файлов и сборки.

[2]Замечание об использовании команды file(GLOB) в документации CMake, URL: https://cmake.org/cmake/help/latest/command/file.html#filesystem:~:text=Note%20We,rebuild, (дата обращения: 9 января 2026 г.).

3. Описание решения

3.1. Переход на целеориентированный подход

Ключевым этапом рефакторинга стал отказ от использования глобальных директив, таких как `include_directories()`. Данное изменение необходимо для устранения неявных зависимостей между компонентами системы и обеспечения соответствия Google C++ Style Guide [3]. Первым шагом стала каноникализация путей во включаемых файлах. Согласно принятому стилю, все директивы `#include` должны указываться относительно корневого каталога исходного кода (`src/`). Для автоматизации этого процесса был разработан скрипт[3], выполняющий поиск и замену путей в исходном коде проекта[4,5,6]. Унификация путей позволила сократить количество поисковых директорий с 11 до одной, что значительно упростило структуру сборки.

Для реализации модульной архитектуры алгоритмы, вспомогательные библиотеки, тесты и Python-привязки были выделены в изолированные цели[7]. Для стандартизации их создания были разработаны функции-обёртки:

- `desbordante_add_lib()` — для создания библиотек компонентов;
- `desbordante_add_test()` — для регистрации модульных тестов;
- `desbordante_add_bind()` — для создания Python-привязок.

Стоит отметить, что для функции `desbordante_add_lib()` вариант с полной инкапсуляцией всей логики настройки цели внутри одной вызова был отклонен. Такой подход снижает прозрачность сборочных

[3]desbordante-core PR#633, URL: https://github.com/Desbordante/desbordante-core/pull/633, (дата обращения: 2 января 2026 г.).

[4]desbordante-core PR#629, URL: https://github.com/Desbordante/desbordante-core/pull/629, (дата обращения: 2 января 2026 г.).

[5]desbordante-core PR#631, URL: https://github.com/Desbordante/desbordante-core/pull/631, (дата обращения: 2 января 2026 г.).

[6]desbordante-core PR#632, URL: https://github.com/Desbordante/desbordante-core/pull/632, (дата обращения: 2 января 2026 г.).

[7]desbordante-core PR#652, URL: https://github.com/Desbordante/desbordante-core/pull/652, (дата обращения: 2 января 2026 г.).

сценариев и затрудняет понимание логики сторонними разработчиками. Вместо этого используется комбинированный подход: функция создает базовую цель, а спецификация свойств, например, с помощью `target_link_libraries()`) остается явной. Списки исходных файлов и линкуемых библиотек передаются через параметры `SRCS` и `LIBS` для функций `desbordante_add_test()` и `desbordante_add_bind()`. Такая реализация позволяет централизованно распространять общие требования на все цели проекта: создание псевдонимов с определённым префиксом, установку необходимых флагов компиляции и определений препроцессора.

Дополнительно была проведена работа по изоляции пространства имен опций сборки. Для упрощения интеграции Desbordante в качестве подмодуля во внешние проекты (например, при сборке pip-пакета, ко всем опциям CMake был добавлен префикс проекта[8] (например, `BUILD_NATIVE` → `DESBORDANTE_BUILD_NATIVE`). Также была унифицирована терминология опций: использование глагола `BUILD` вместо `COMPILE` (например, `DESBORDANTE_BUILD_TESTS`). В процессе рефакторинга была устранена избыточная команда отключения санитайзера[9].

Для упрощения интеграции проекта в другие системы также необходимо ограничить CMake опции проекта путём добавления префикса[10], например, `BUILD_NATIVE` → `DESBORDANTE_BUILD_NATIVE`, это позволяет не пересекаться с опциями внешнего проекта. Также для единообразия и сокращения длины слово `COMPILE` в опциях CMake была изменено на `BUILD`, например, `COMPILE_TESTS` → `DESBORDANTE_BUILD_TESTS`. В ходе изменений было замечено, что при сборке pip-пакета используется избыточная команда отключения санитайзера — строчка с командой была удалена[11].

[8]desbordante-core PR#654, URL: https://github.com/Desbordante/desbordante-core/pull/654, (дата обращения: 2 января 2026 г.).

[9]desbordante-core PR#653, URL: https://github.com/Desbordante/desbordante-core/pull/653, (дата обращения: 2 января 2026 г.).

[10]desbordante-core PR#654, URL: https://github.com/Desbordante/desbordante-core/pull/654, (дата обращения: 2 января 2026 г.).

[11]desbordante-core PR#653, URL: https://github.com/Desbordante/desbordante-core/pull/653, (дата обращения: 2 января 2026 г.).

3.2. Автоматизация скачивания тестовых данных

Для исключения ручного управления загрузкой датасетов (ранее выполнявшегося скриптом `pull_datasets.sh`) был внедрен механизм синхронизации на уровне CMake. Рассматривался вариант использования стандартного модуля `ExternalData`[12], однако он требует хранения данных под именами, соответствующими их хэш-суммам, что усложняет навигацию по репозиторию данных.

В качестве альтернативы было реализовано решение на основе команды `file(DOWNLOAD)`[13]. Разработанный механизм работает следующим образом:

1. В репозитории `desbordante-data`[14] хранятся файлы-маркеры, содержащие актуальные контрольные суммы архивов с данными. Обновление маркеров автоматизировано посредством pre-commit хуков[15].

2. На этапе конфигурации CMake скачивает легковесные файлы-маркеры и, сравнивая контрольные суммы локальных копий датасетов с находящимися удалённо, проверяет наличие и целостность архивов, хранящихся на устройстве разработчика.

3. При отсутствии датасетов или несовпадении контрольной суммы производится автоматическая загрузка только необходимых архивов.

[12] Документация ExternalData, URL: https://cmake.org/cmake/help/latest/module/ExternalData.html, (дата обращения: 2 января 2026 г.).

[13] Документация file(DOWNLOAD), URL: https://cmake.org/cmake/help/latest/command/file.html#download, (дата обращения: 2 января 2026 г.).

[14] https://github.com/Desbordante/desbordante-data

[15] desbordante-data PR#4, URL: https://github.com/Desbordante/desbordante-data/pull/4, (дата обращения: 2 января 2026 г.).

3.3. Стандартизация конфигураций сборки (CMake Presets)

Для унификации процесса сборки и обеспечения воспроизводимости среды разработки был внедрен механизм пресетов[16] (CMake Presets[17]). Были зафиксированы типовые конфигурации:

- `Debug` — отладочная сборка;
- `Release` — оптимизированная сборка;
- `DebugSan` — отладочная сборка с включенными санитайзерами.

Логика определения флагов компилятора была вынесена в отдельный файл, что позволило отделить описание целей от глобальных настроек.

В ходе внедрения пресетов была выявлена и устранена проблема сборки с AddressSanitizer[18]. Из-за некорректного использования функций CMake флаги санитайзера перезаписывались. Причиной проблемы была ошибка в стандартной библиотеке `libc++` (до версии 19 включительно)[19], приводившая к конфликтам флагов. Для решения проблемы версия `libc++` в CI-окружении была обновлена до 20-й.

Скрипт `build.sh` — привычная точка входа для разработчиков, поэтому он был адаптирован для использования пресетов, что позволило устранить дублирование логики конфигурации и сохранить обратную совместимость.

3.4. Сопутствующие улучшения кодовой базы

В рамках работы над системой сборки был выполнен ряд исправлений, направленных на повышение стабильности и гигиены кода:

[16]desbordante-core PR#657, URL: https://github.com/Desbordante/desbordante-core/pull/657, (дата обращения: 2 января 2026 г.).

[17]Документация cmake-presets, URL: https://cmake.org/cmake/help/latest/manual/cmake-presets.7.html, (дата обращения: 9 января 2026 г.).

[18]Документация AddressSanitizer, URL: https://clang.llvm.org/docs/AddressSanitizer.html, (дата обращения: 2 января 2026 г.).

[19]libc++ issue#59432, URL: https://github.com/llvm/llvm-project/issues/59432, (дата обращения: 2 января 2026 г.).

- Исправлено поведение при сборке библиотеки `GTest` с использованием `libc++`. Из-за ошибки[20] для сборки проекта требовался флаг `-Wno-error=character-conversion`. Ранее этот флаг применялся глобально, теперь[21] с помощью `target_compile_options()` область его видимости ограничена исключительно целью библиотеки для тестирования.

- Конфигурация внешних зависимостей была вынесена в отдельный CMake-модуль, что улучшило структурированность сценариев сборки CMake[22].

- Усовершенствован механизм проверки совместимости компилятора[23]. В ходе последующего код-ревью была выявлена и исправлена логическая ошибка в проверке версий[24].

- Удален пустой файл, выявленный в ходе работы над проектом[25].

[20] GoogleTest issue#4762, URL: https://github.com/google/googletest/issues/4762, (дата обращения: 2 января 2026 г.).

[21] desbordante-core PR#656, URL: https://github.com/Desbordante/desbordante-core/pull/656, (дата обращения: 2 января 2026 г.).

[22] desbordante-core PR#658, URL: https://github.com/Desbordante/desbordante-core/pull/658, (дата обращения: 2 января 2026 г.).

[23] desbordante-core PR#598, URL: https://github.com/Desbordante/desbordante-core/pull/598, (дата обращения: 2 января 2026 г.).

[24] desbordante-core PR#627, URL: https://github.com/Desbordante/desbordante-core/pull/627, (дата обращения: 2 января 2026 г.).

[25] desbordante-core PR#626, URL: https://github.com/Desbordante/desbordante-core/pull/626, (дата обращения: 2 января 2026 г.).

4. Эксперимент

4.1. Условия эксперимента

Проведено сравнительное тестирование двух конфигураций проекта: изначальной — исходная архитектура сборки, изменённой — с применением модификаций, сделанных в ходе учебной практики. В эксперименте рассматриваются три сценария: генерация — конфигурация системы сборки, полная сборка — компиляция и линковка с нуля, инкрементальная сборка — пересборка после изменения даты файла `src/core/algorithms/euler.cpp`, файл выбран, т.к. он отражает типичный сценарий изменения реализации отдельного алгоритма.

Аппаратная платформа: ЦП — Intel Core i7-11800H, ОЗУ — 16 Гб. Программное окружение: ОС — Arch Linux x86_64, ядро — Linux 6.18.3-arch1-1, Ninja — 1.13.2, CMake — 4.2.1, компилятор Clang — 21.1.6. Данные измерений были получены с помощью утилиты `perf-stat`[26] версии 6.18-1, сравнивался показатель события `duration_time`. При замерах для генерации программа запускалась на одном ядре, а для сборок на всех.

Перед измерениями были отключены Intel Turbo Boost, гиперпоточность, рандомизация размещения адресного пространства, были ограничены частота процессора до базовой, параметр `vm.swapiness` до десяти. При замерах на одном ядре все прерывания и системные процессы, которые можно было перенести, были перемещены на остальные ядра. Перед каждым замером сбрасывался кэш файловой системы. Подробнее ознакомиться с окружением можно в специально созданном репозитории `bench_env`[27].

При генерации использовались флаги: `-DCMAKE_BUILD_TYPE=Debug -DCOMPILE_BENCHMARKS=ON -DPYTHON=COMPILE -G Ninja` и `--preset Debug -DDESBORDANTE_BUILD_BENCHMARKS=ON -DDESBORDANTE_BINDINGS =BUILD -DDESBORDANTE_FETCH_DATASETS=OFF` для изначальной и изме-

[26]Вики утилиты perf, URL: https://perfwiki.github.io/main, (дата обращения: 8 января 2026 г.).

[27]Репозиторий bench_env, URL: https://github.com/metrapoliten/bench_env, (дата обращения: 10 января 2026 г.).

нённой конфигураций соответственно.

4.2. Результаты

Результаты замеров приведены в нижеприведённой таблице. Внедрение изменений привело к незначительной регрессии времени генерации (9%) и полной сборки (1%). Увеличение времени обусловлено усложнением графа зависимостей CMake и ростом числа целей. Однако в сценарии инкрементальной сборки, доминирующем в процессе разработки, достигнуто ускорение около 23%. Это объясняется гранулярностью линковки: в изменённой конфигурации модификация одной единицы трансляции требует перелинковки только маловесных библиотек или исполняемых файлов, тогда как в изначальной это вызывало перелинковку монолитных исполняемых файлов и крупной библиотеки с дорогостоящей операцией ввода-вывода.

Для сценария генерации было произведено по 500 измерений, для полной сборки — по 20, для инкрементальной — по сто на каждую конфигурацию. В качестве случайной погрешности была выбрана средняя абсолютная ошибка (средняя абсолютная погрешность, MAE). Эмпирические распределения времени выполнения сценариев, полученные в ходе замеров, в ряде случаев не удовлетворяли критериям нормальности, например, при сценарии генерации в изменённой конфигурации. В качестве целевого значения для расчёта средней абсолютной ошибки принималось среднее арифметическое.

Таблица 1: Сравнение реального времени выполнения различных сценариев, в с.

Сценарий	**Изначальная**	**Изменённая**
Генерация	3.131 ± 0.011	3.440 ± 0.309
Полная сборка	279.2 ± 0.5	282.6 ± 0.5
Инкрементальная	27.96 ± 1.03	21.68 ± 0.14

Заключение

В рамках учебной практики была проведена комплексная модернизация системы сборки проекта Desbordante. По итогам работы были достигнуты следующие результаты:

1. Выполнен аудит инфраструктуры сборки. Были выявлены недостатки существующей системы, связанные с несоответствием современным стандартам CMake.

2. Осуществлен полный переход к целеориентированной модели. Реализована строгая инкапсуляция зависимостей.

3. Устранена зависимость от внешнего скрипта загрузки датасетов. Внедрен механизм CMake Presets для унификации конфигураций сборки.

4. Выполнены замеры реального времени выполнения.

Результаты работы в виде пулл-реквестов доступны на GitHub, из них девять[28,29,30,31,32,33,34,35,36] принято в основную ветку репозитория, три[37,38,39] находятся на стадии рассмотрения и доработки, и ещё три

[28]desbordante-core PR#598, URL: https://github.com/Desbordante/desbordante-core/pull/598, (дата обращения: 10 января 2026 г.).
[29]desbordante-core PR#599, URL: https://github.com/Desbordante/desbordante-core/pull/599, (дата обращения: 10 января 2026 г.).
[30]desbordante-core PR#626, URL: https://github.com/Desbordante/desbordante-core/pull/626, (дата обращения: 10 января 2026 г.).
[31]desbordante-core PR#629, URL: https://github.com/Desbordante/desbordante-core/pull/629, (дата обращения: 10 января 2026 г.).
[32]desbordante-core PR#631, URL: https://github.com/Desbordante/desbordante-core/pull/631, (дата обращения: 10 января 2026 г.).
[33]desbordante-core PR#632, URL: https://github.com/Desbordante/desbordante-core/pull/632, (дата обращения: 10 января 2026 г.).
[34]desbordante-core PR#653, URL: https://github.com/Desbordante/desbordante-core/pull/653, (дата обращения: 10 января 2026 г.).
[35]desbordante-core PR#654, URL: https://github.com/Desbordante/desbordante-core/pull/654, (дата обращения: 10 января 2026 г.).
[36]desbordante-core PR#660, URL: https://github.com/Desbordante/desbordante-core/pull/660, (дата обращения: 10 января 2026 г.).
[37]desbordante-ccore PR#633, URL: https://github.com/Desbordante/desbordante-core/pull/633, (дата обращения: 10 января 2026 г.).
[38]desbordante-core PR#655, URL: https://github.com/Desbordante/desbordante-core/pull/655, (дата обращения: 10 января 2026 г.).
[39]desbordante-data PR#4, URL: https://github.com/Desbordante/desbordante-data/pull/4, (дата

ожидают рассмотрения[40,41,42].

обращения: 10 января 2026 г.).

[40]desbordante-core PR#656, URL: https://github.com/Desbordante/desbordante-core/pull/653, (дата обращения: 10 января 2026 г.).

[41]desbordante-core PR#657, URL: https://github.com/Desbordante/desbordante-core/pull/654, (дата обращения: 10 января 2026 г.).

[42]desbordante-core PR#658, URL: https://github.com/Desbordante/desbordante-core/pull/660, (дата обращения: 10 января 2026 г.).

Список литературы

[1] 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.

[2] Gamberini Vito. Post-Modern Cmake - From 3.0 to 4.0. — 2025. — URL: https://www.youtube.com/watch?v=K5Kg8TOTKjU (дата обращения: 2 января 2026 г.).

[3] Google C++ Style Guide: Names and Order of Includes, Google. — URL: https://google.github.io/styleguide/cppguide.html#Names_and_Order_of_Includes (дата обращения: 2 января 2026 г.).

[4] Results summary: 2025 Annual C++ Developer Survey "Lite", Standard C++ Foundation. — URL: https://isocpp.org/files/papers/CppDevSurvey-2025-summary.pdf (дата обращения: 2 января 2026 г.).