Санкт-Петербургский государственный университет

Смирнов Олег Евгеньевич

Выпускная квалификационная работа

Поддержка языка Python в системе автоматизации разработки Paddle

Уровень образования: бакалавриат

Направление *09.03.04 «Программная инженерия»*

Основная образовательная программа *СВ.5080.2018 «Программная инженерия»*

Научный руководитель:
к. т. н., доцент Брыксин Т. А.

Консультант:
программист ООО "Интеллиджей Лабс" Танков В. Д.

Рецензент:
программист ООО "Интеллиджей Лабс" Сазанович В. В.

Санкт-Петербург
2022

Saint Petersburg State University

Oleg Smirnov

Bachelor's Thesis

Python support in Paddle

Education level: bachelor

Speciality *09.03.04 «Software Engineering»*

Programme *CB.5080.2018 «Software Engineering»*

Scientific supervisor:
Ph.D., associate professor T. A. Bryksin

Advisor:
software engineer, IntelliJ Labs Co. Ltd V. D. Tankov

Reviewer:
software engineer, IntelliJ Labs Co. Ltd U. V. Sazanovich

Saint Petersburg
2022

Оглавление

Введение

Системы сборки программных проектов используются сегодня практически во всех крупных информационных системах. Когда программа становится программным продуктом [4], а кодовая база проекта начинает достигать десятков и сотен тысяч строк, системы автоматизации сборки становятся не просто удобным инструментом, а жизненной необходимостью.

В стандартный набор задач таких систем обычно входит построение ориентированного графа зависимостей между задачами сборки необходимых артефактов в проекте — в частности, запуск скриптов компиляции исходного кода в исполняемые модули. Современные системы автоматизированной сборки, такие как Gradle [15] или Maven [3], часто также реализуют управление зависимостями проекта, в том числе, поиск подходящих для конкретного проекта версий библиотек и сторонних пакетов.

Кроме того, у программистов возникает потребность в автоматизации и других процессов разработки, таких как удаленное исполнение кода, тестирование и линтинг [23] (статический анализ кода на предмет наличия нежелательных конструкций и потенциальных ошибок). Часть этих задач обычно выполняется системами непрерывной интеграции и развертывания (Continuous Integration & Continuous Deployment, CI/CD), но только в контексте запуска конкретных скриптов на удалённых вычислительных узлах. Компиляция необходимых для этого артефактов, разрешение зависимостей и запуск задач на локальной машине пользователя все еще являются привилегией системы сборки.

В этом смысле стоит обратить внимание на организацию программных проектов на языке Python. В то время как Python занимает первую строчку в рейтинге TIOBE[1] по популярности в сообществе разработчиков на май 2022 года, для него не существует единого удобного инструмента автоматизации всех описанных процессов разработки, встраиваемого в интегрированную среду разработки (Integrated Development

[1]Рейтинг TIOBE: https://www.tiobe.com/tiobe-index/ (дата обращения: 19.05.2022)

Environment, IDE). Процесс написания и поддержки кода на Python также включает в себя использование множества специфичных для этого языка программирования паттернов и концепций, таких как виртуальные окружения и взаимодействие с удалёнными PyPi-репозиториями пакетов. Данные особенности являются неотъемлемой частью любой предполагаемой системы автоматизации разработки и сборки проектов на Python.

Более того, в индустрии для работы с проектами на Python все чаще используются так называемые *монорепозитории* [5] — подход, предполагающий работу с множеством подпроектов и их зависимостями в контексте одного репозитория в системе контроля версий. Существующие популярные инструменты автоматизации разработки на Python не поддерживают процессы сборки и публикации таких проектов, и разработчикам приходится делать это вручную.

С описанными проблемами столкнулись разработчики платформы анализа естественного языка Grazie[2] в компании JetBrains[3], использующие монорепозитории с проектами на Python в области машинного обучения. В результате был создан прототип системы автоматизации разработки Paddle [33]. Чтобы упростить процесс интеграции Paddle в IDE на основе IntelliJ Platform [19] (основная кодовая база которой написана на языке Java), для реализации был выбран язык Kotlin. В данной же выпускной квалификационной работе будет описана дальнейшая разработка системы Paddle, в ходе которой было реализовано расширение для поддержки языка Python.

Постановка задачи

Целью данной работы является расширение системы автоматизации разработки проектов Paddle для поддержки языка Python. Для достижения указанной цели в рамках данной выпускной квалификационной работы были поставлены следующие задачи.

[2]Grazie Platform: https://lp.jetbrains.com/grazie-platform-landing/
[3]JetBrains: https://www.jetbrains.com/

1. Провести обзор предметной области в сфере систем сборки проектов и автоматизации разработки на языке Python.

2. Реализовать подсистему управления пакетами и виртуальными окружениями.

3. Реализовать поддержку автоматизированной сборки монорепозиториев с Python-проектами.

4. Интегрировать реализованную функциональность в плагин для IDE PyCharm.

5. Продемонстрировать возможности системы на нескольких существующих репозиториях с Python-проектами.

1. Обзор предметной области

В этой главе будут рассмотрены некоторые известные системы сборки общего назначения, приведён их анализ с точки зрения архитектурной расширяемости и поддержки в IDE. Кроме того, будут выделены их ключевые особенности и недостатки. После этого будет приведено описание некоторых специфичных инструментов разработки на языке Python, использование которых подразумевает любая современная система автоматизации сборки проектов.

1.1. Автоматизация сборки ПО

Изначально основной целью систем сборки являлась компиляция исходного кода проекта в некоторый набор исполняемых модулей или библиотек [11]. Такой инструмент помогал избежать рутинного написания одинаковых скриптов по сборке каждый раз в разных проектах [26]. Идея здесь достаточно простая: разработчики всего лишь поддерживают файл с описанием процесса сборки и местонахождением конкретных файлов и артефактов в проекте, а система сборки уже разрешает необходимые зависимости в порядке создания указанных файлов исходного кода и вызывает компилятор для создания искомых исполняемых модулей (или, например, пакетов с байт-кодом в случае JVM). Для этого вводится концепция *графа зависимостей задач* (task dependency graph), которая используется подавляющим большинством современных систем сборки (Make [12], Bazel [13], Apache Ant [2], Gradle [15] и другими). Задачи компонуются в вершины ориентированного ациклического графа, где рёбра исходят из тех задач, которые должны быть выполнены раньше других для получения необходимых артефактов при сборке.

Кроме того, многие современные системы сборки проектов берут на себя и другие задачи по *автоматизации процессов разработки*. Вот лишь некоторые наиболее популярные из них:

- загрузка библиотек из удалённого репозитория пакетов по сети;
- запуск модульных и интеграционных тестов;

- линтинг [23] (статический анализ кода);
- развертывание проекта на удалённом сервере;
- генерация и копирование конфигурационных файлов;
- генерация и обновление документации к коду.

Может показаться, что какую-то часть этих задач сейчас выполняют CI/CD-сервисы — системы непрерывной интеграции и развертывания программных проектов, такие как Jenkins [18], TeamCity [20], Travis CI [45] и другие. Однако с технической точки зрения CI/CD-системы обычно делегируют задачи запуска тестов, линтинга или сборки артефактов проекта для развертывания другим утилитам, которые чаще всего и являются частью системы сборки. Таким образом, система сборки потенциально может выполнять часть задач системы непрерывной интеграции, но не наоборот. Кроме того, облачные CI/CD-сервисы обычно разворачиваются на выделенных удалённых серверах — они не призваны решать проблемы запуска задач на локальной машине разработчика и практически не интегрируются в IDE.

Упомянутые загрузки в проект библиотек из удалённых репозиториев, как и разрешение подходящих версий зависимостей, могут являться ответственностью “пакетного менеджера” — специальной утилиты для установки, удаления и настройки различных сторонних компонент. Такая концепция используется во многих современных языках программирования: например, `npm` [30] для JavaScript или `pip` [32] для Python. В то же время это может являться и частью системы сборки, как это принято, например, в JVM (в Maven или Gradle). В этом смысле, здесь и далее в работе понятия *система автоматизации сборки* и *система автоматизации процессов разработки* программных проектов употребляются как синонимы, хотя, формально, это и не совсем так.

1.2. Системы сборки общего назначения

Системы автоматизированной сборки проектов могут разрабатываться как программное обеспечение для конкретного языка програм-

мирования или платформы и не иметь широкой популярности за их пределами. Такими системами являются, например, SBT для Scala или Stack для Haskell. Однако большинство систем сборки по своей сути являются системами сборки *общего назначения* и могут использоваться для сборки проектов на практически любом языке программирования. Но и тут проблем возникает достаточно: от специфичных для языка репозиториев пакетов до особенностей системы версионирования библиотек. Чаще всего эти проблемы решаются за счет написания плагинов для систем сборки с поддержкой нужного языка (разумеется, если это предусмотрено архитектурой исходной системы). В ином случае, приходится описывать необходимые манипуляции по сборке проекта на каком-либо скриптовом языке (например, Bash).

Далее кратко рассмотрим некоторые известные системы автоматизации сборки общего назначения с целью выявления их ключевых особенностей, достоинств и недостатков.

Make [12]

Одна из первых систем сборки, архитектура которой была описана еще в 1978 году [11]. Make оперирует понятием *задач* (в оригинале "правил сборки", build rules) и предполагает, что разработчик декларативно описывает получение всех необходимых для сборки проекта артефактов в специальном make-файле (он же Makefile). Внутри таких правил сборки артефактов (чаще всего, исполняемых модулей) Make позволяет вызывать любые shell-скрипты, что делает его достаточно мощным инструментом.

Рассмотрим для примера простейший Makefile:

```
util.o: util.h util.c
    gcc -c util.c
main.o: util.h main.c
    gcc -c main.c
main.exe: util.o main.o
    gcc util.o main.o -o main.exe
```

Этот Makefile описывает 3 задачи: компиляция объектного файла библиотеки утилит `util.o`, компиляция объектного файла `main.o`, а также линковка созданных артефактов с целью создания исполняемого файла `make.exe`. Такое текстовое описание процесса сборки содержит информацию об ориентированном графе зависимостей, который и используется утилитой Make для сборки проекта. Кроме того, Make реализует множество полезных для сборки утилит вроде функций по манипуляции со строками, условных конструкций и шаблонов подстановки символов (wildcards).

Несмотря на внешнюю простоту, за более чем 40 лет истории Make доказал свою универсальность. Он относительно быстрый с точки зрения производительности [43], а также реализует паттерн *инкрементальной сборки* [9], опираясь на *время редактирования файла* в файловой системе. При инкрементальной сборке задача не выполняется заново в случае, если используемые ей данные (input) и её результат (output) не изменялись с момента последнего запуска. Вместе с тем Make может быть недостаточно производительным для сборки действительно огромных проектов, в частности, из-за времени построения собственных структур данных после синтаксического разбора makefile парсером [44].

Из-за своей простоты Make достаточно гибок в смысле использования, и его часто можно встретить как часть полноценной системы сборки в современных проектах в связке с некоторым менеджером задач или системой управления пакетами (для экосистемы Python это может быть, например, `poetry` [35]). Однако сам по себе Make обладает нерасширяемой архитектурой: она не предусматривает написание плагинов для решения повторяющихся и специфичных для языка или платформы задач. Make в принципе лишен таких абстракций как “проект” или “зависимость”, что очень усложняет его использование. Практически любой шаг в сторону от стандартного процесса сборки исходных файлов в исполняемые превращается в написание огромного количества скриптов разработчиком. Make также по умолчанию не предлагает решение ни одной из описанных ранее в обзоре *дополнительных задач*

системы автоматизированной сборки для проектов на конкретном языке (например, Python).

Bazel [13]

Был опубликован в open-source компанией Google лишь в 2015 году (релиз первой стабильной версии состоялся в 2019), однако он уже успел стать сравнительно популярным инструментом в сообществе разработчиков[4]. Bazel кроссплатформенный и имеет расширяемую за счёт плагинов архитектуру. Конфигурация сборки в Bazel включает в себя как декларативное описание вызовов нужных правил сборки целевых артефактов в BUILD-файле проекта, так и процедурное расширение в виде отдельных .bzl-файлов с константами, функциями и макросами, описанными на языке Starlark. Данный скриптовый язык синтаксически является подмножеством языка Python.

Bazel, так же как и Make, использует концепцию графа зависимостей задач сборки. Каждая сборка отдельного артефакта или библиотеки здесь запускается в отдельной изолированной среде (sandbox environment) — тем самым Bazel гарантирует надёжность и воспроизводимость сборок. Он также поддерживает и инкрементальное выполнение задач, но вместо времени редактирования файлов подсчитывает хеши их содержимого. Это позволяет более точно понимать, изменился ли артефакт сборки, чтобы собирать его заново, а также открывает возможность использования удалённых репозиториев артефактов сборки (content addressable remote cache [28]). Последняя особенность является ключевой для Bazel, так как он позиционируется как *облачная система сборки*, то есть позволяет командам разработчиков пользоваться общим хранилищем собранных артефактов. Более того, он позволяет производить сборку проекта и запуск тестов не только на одной локальной машине пользователя, но и на распределённом кластере из нескольких машин[5].

[4]Корпорации, использующие Bazel: https://bazel.build/users.html

[5]Распределённая сборка на кластере в Bazel: https://docs.bazel.build/versions/main/remote-execution.html

Одним из недостатков Bazel по мнению сообщества является зависимость от JVM/JRE[6]: в то время, как разработчиками поддерживаются плагины для многих основных языков программирования, таких как C++ или Python, Bazel всё еще требует наличия виртуальной Java-машины для запуска скриптов сборки. Кроме того, опыт использования Bazel в проектах по машинному обучению команды Grazie в компании JetBrains показывает, что даже при наличии в нём специального плагина для Python конфигурация элементарных для этого языка задач (таких, как создание виртуальной среды разработки или запуск тестов) занимает немалое количество строк инфраструктурного кода на Starlark. Скорее всего это обусловлено универсальностью данной системы автоматизации сборки и использованием процедурного языка для описания расширений.

Gradle [15]

Ещё одной системой автоматизации сборки проектов, заслуживающей упоминания, является Gradle, первый релиз которого состоялся в 2007 году. Gradle также является кроссплатформенной системой сборки общего назначения, хоть и требует предустановленного JRE для запуска.

Gradle (как и все последние рассмотренные системы сборки) использует граф задач для разрешения зависимостей (см. Рис. 1) и обладает расширяемой архитектурой за счёт использования *плагинов*, которые позволяют инкапсулировать практически любые пользовательские расширения логики сборок. Так, например, существует (но уже не поддерживается) плагин PyGradle для автоматизации сборки проектов на Python от компании LinkedIn[7]. Стоит отметить, что Gradle предоставляет возможности расширения своей функциональности прямо внутри исходного проекта: например, новые типы задач можно описывать прямо в директории `buildSrc` собираемого проекта, и они будут доступны к использованию в обычных скриптах сборки.

[6]Обсуждение на Y-combinator: https://news.ycombinator.com/item?id=21385013

[7]Репозиторий PyGradle: https://github.com/linkedin/pygradle

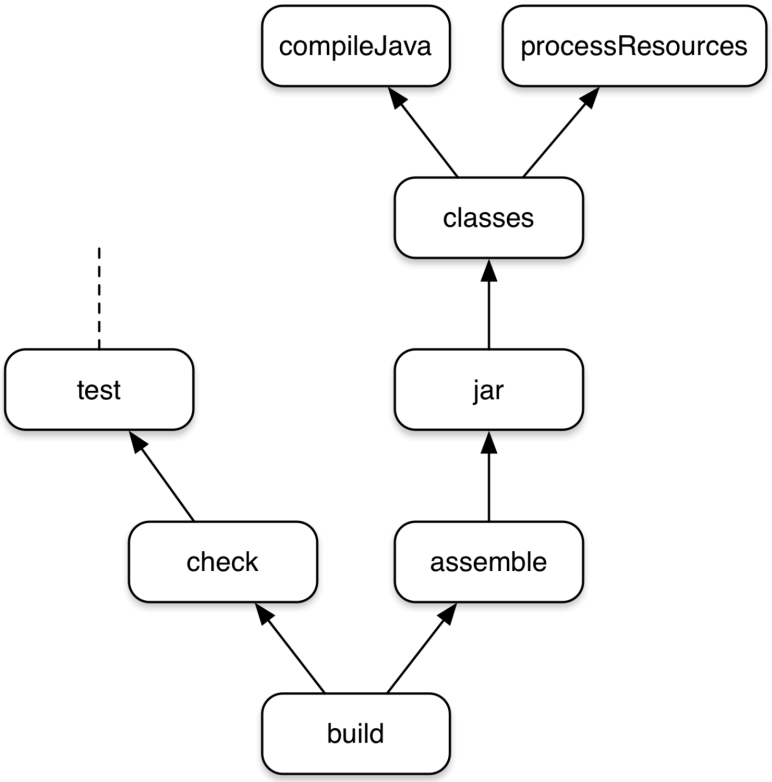

Рис 1: Пример типичного графа задач сборки Gradle для Java-проекта.

Изначально Gradle поддерживал Groovy DSL в качестве скриптового языка для описания сборок, но с относительно недавнего времени в нём появилась и поддержка Kotlin DSL. Важной особенностью Gradle в работе с языком описания сборок является тот факт, что разработчик добавляет лишь описание необходимых ему элементов, в то время как Gradle автоматически генерирует по своей проектной модели остальные нужные для работы артефакты (классы и функции) и продолжает сборку с ними. В частности, с этим связано и сравнительно долгое время сборки проектов на Gradle по сравнению с другими системами автоматизации (для JVM, например, это Maven и Ant).

Pants [34]

Возвращаясь к исходной проблеме поддержки языка Python в современных системах сборки и системах автоматизации разработки, нельзя не упомянуть Pants. Будучи относительно малоизвестной open-source

системой сборки общего назначения, она довольно целенаправленно фокусируется на поддержке языка Python (хоть сейчас и позволяет работать с Java, Scala и Go). Pants расширяема плагинами, поддерживает кэширование артефактов и инкрементальные сборки. За последние годы в ней появилась поддержка множества существующих инструментов для современной разработки на Python — таких, как менеджеры виртуальных окружений, тестовые фреймворки, статические анализаторы и кодогенераторы (подробнее см. раздел 1.3).

Полноценная поддержка монорепозиториев (с корректным управлением версиями зависимостей в разных модулях одного проекта), однако, была анонсирована совсем недавно, в марте 2022 года[8]. Кроме того, при всех достоинствах Pants на данный момент она никак не интегрируется в IDE. Реализация же соответствующего плагина к IDE (в частности, на основе IntelliJ Platform) является очень трудоёмкой задачей, так как ядро системы сборки реализовано на языке Rust. Взаимодействовать с её абстракциями представляется возможным только путем синтаксического разбора журналов исполнения её команд или же интеграцией собственных интерфейсов в исходный код системы.

1.3. Python: инструментарий разработчика

Современная промышленная разработка на Python обычно подразумевает использование множества паттернов и концепций, часть из которых возможно реализовать как задачи системы автоматизации сборки проектов. Стоит отметить, что задачи линковки и разрешения зависимостей локальных модулей в проектах на Python решает сам интерпретатор, когда компилирует исходную программу в байт-код (хранящийся в .pyc-файлах директории `__pycache__` проекта), а позже исполняет её с помощью виртуальной машины Python. Под автоматизацией сборки здесь скорее понимается именно автоматизация *процессов разработки* для всего проекта на Python, таких, как разрешение зависимостей на внутренние модули и на сторонние пакеты, поддержка виртуального

[8]Несколько lock-файлов в монорепозитории Pants: https://blog.pantsbuild.org/multiple-lockfiles-python/

окружения, тестирование, статический анализ кода на предмет уязвимостей, генерация и публикация дистрибутивов с пакетами, и так далее. О некоторых из этих процессов и пойдёт речь в данном разделе.

Системы управления пакетами

Пакетный менеджер (или система управления пакетами) — это специальная утилита, предназначенная для установки, удаления и конфигурации пакетов и сторонних библиотек. Для этого системы управления пакетами взаимодействуют с удалёнными репозиториями, которые могут быть публичными (поддерживаться сообществом) или приватными (например, внутри компании). В ответственность пакетного менеджера также входит и разрешение версий этих самых зависимостей, то есть поиск таких версий пакета в репозитории, которые бы удовлетворяли текущей конфигурации проекта. Необходимо учитывать состояние версий других установленных пакетов (могут возникнуть конфликты), версию платформы, интерпретатора и текущей операционной системы. Кроме того, система управления зависимостями обычно может производить снимки существующего в проекте набора зависимостей и записывать их в специальные .lock-файлы. Это необходимо для воспроизводимости сборки на другой машине, платформе или в CI-сервисе. Если с каждой зависимостью в таком файле указана её уникальная подпись (например, хеш содержимого пакета), .lock-файлы могут использоваться и для верификации сборок, то есть с их помощью проверяется, не заменил ли злоумышленник содержимое никакого из пакетов в удаленном репозитории (supply chain attack [29]).

Стандартом де-факто в Python считается пакетный менеджер pip [32]. Его основная функция — установка пакетов из PyPi-репозиториев, спецификация которых описана в PEP 503 [31]. Для конфигурации зависимостей проекта pip использует файл requirements.txt, уникальный для (под)проекта и содержащий набор искомых зависимостей кода.

Существуют и другие пакетные менеджеры для Python, среди которых особый интерес представляет Poetry [35]. Кроме непосредственного управления зависимостями Poetry предоставляет возможность удоб-

ного управления виртуальными окружениями разработки проектов, а также упрощает публикацию собственных пакетов до написания единственной консольной команды. По сравнению с другим известным менеджером `pipenv` [48] Poetry обладает более интуитивным интерфейсом и способен корректнее разрешать версии зависимостей в определённых случаях[9]. В то же время Poetry является консольной утилитой и не стремится решать остальные задачи системы автоматизированной сборки проектов.

Виртуальные окружения

Виртуальная среда разработки (virtual environment) — специальная абстракция, существующая в экосистеме Python, подразумевающая собой некоторое изолированное окружение, которое создаётся для каждого нового проекта. Технически такое окружение содержит нужную версию интерпретатора Python и множество установленных в нём пакетов, библиотек и скриптов. Нужно это для того, чтобы не возникало конфликтов в случаях, когда на одной локальной рабочей машине в разных проектах используются разные версии одной и той же библиотеки. В таком случае каждый проект будет пользоваться своим собственным виртуальным окружением, что позволит избежать лишних конфликтов.

Начиная с версии 3.3 в Python встроен собственный менеджер виртуальных окружений `venv` [50], однако он имеет лишь некоторое подмножество функциональных особенностей менеджера `virtualenv` [51], который является сейчас наиболее популярным. Кроме них существует и множество других консольных утилит, предоставляющих возможность управления виртуальными окружениями со своими особенностями: уже упомянутые Poetry и pipenv, Conda [8] (параллельно развивающийся продукт со своими репозиториями пакетов, популярный в сообществе машинного обучения), pyenv [49] (позволяющий конфигурировать нужную версию интерпретатора Python, см. Рис. 2) и другие.

[9]Разрешение зависимостей в Poetry: https://github.com/python-poetry/poetry#dependency-resolution

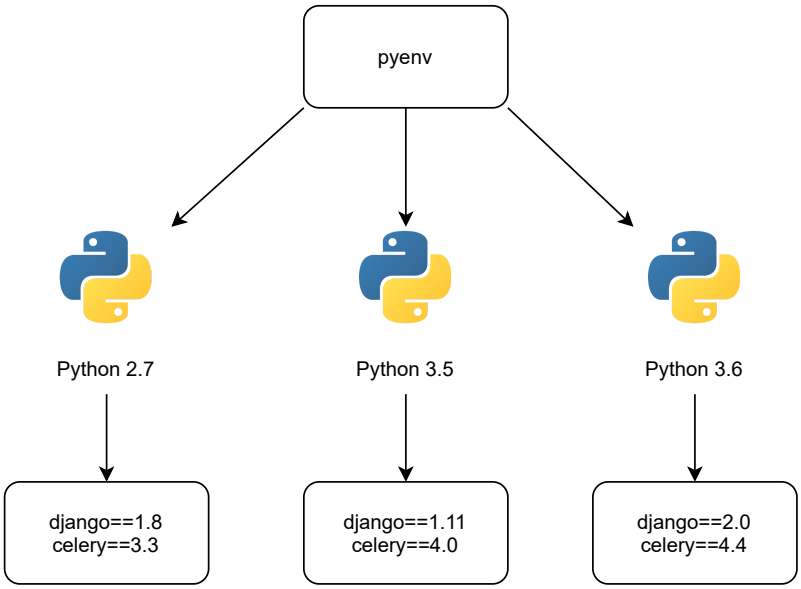

Рис 2: Концепция виртуальных окружений в pyenv: у каждого проекта (среды разработки) могут быть собственные версии интерпретатора Python и зависимости на сторонние пакеты.

Тестирование, статический анализ, исполнение кода

Для того чтобы тесты было удобно писать, запускать и поддерживать, существуют *тестовые фреймворки*: для Python это встроенный Unittest [38], не менее популярный pytest [39] и другие. Система автоматизации сборки проектов с тестами может включать в себя функциональность создания задач по запуску тестов, а также добавление на них зависимостей для других команд вроде сборки и публикации пакета.

Для Python также существует множество различных линтеров, таких как Flake8 (проверяет соответствие стилю PEP8), Pylint [37] (проверки на наличие определенных нежелательных паттернов), Муру [27] (статические проверки соответствия типов) и других. Запуск задач линтинга может являться частью процесса сборки пакета.

Исполнением кода занимается интерпретатор Python. Однако нередко возникает и необходимость запуска интерпретатора на определенной платформе и в специальном окружении. Для таких целей используют Docker [1], позволяющий запускать приложения в специальном “контейнере” изолированно от других процессов на локальной машине. Для удалённого запуска кода проекта можно воспользоваться, например,

протоколом `ssh` [40]. Задачи запуска в определённом окружении сейчас решаются разработчиками на Python каждый раз вручную, им приходится тратить время на написание конфигураций и инфраструктурного кода. Эти задачи также могла бы брать на себя система автоматизации сборки проектов, оставляя за пользователем лишь обязательство декларативно описать необходимую конфигурацию целевой платформы.

1.4. Интегрированные среды разработки

Использование функций системы автоматизированной сборки напрямую из IDE делает процессы разработки намного эффективнее [22]: разработчик не отвлекается на запуск команд из терминала и управляет всеми процессами напрямую из пользовательского интерфейса. Так, например, IntelliJ IDEA [16] (IDE, разрабатываемая JetBrains) имеет встроенную поддержку систем сборки Maven, Gradle и Ant для JVM приложений, а CLion [6] позволяет работать с CMake. Некоторые из рассмотренных ранее в обзоре систем сборки общего назначения имеют плагины для IDE, которые поддерживаются сообществом и в лучшем случае подразумевают лишь подсветку синтаксиса в скриптах сборки и возможность запуска отдельных задач внутри самой IDE. В то же время с точки зрения пользовательского опыта не лишним было иметь и другие средства для удобной работы в данных системах сборки, например, разметку проекта по модулям в соответствии с системой зависимостей между подпроектами или возможность автодополнения кода в редакторе для файлов с конфигурацией сборки.

Касательно Python стоит отметить, что даже с наличием таких инструментов, как PyGradle [**?**], PyCharm [36] (будучи самой популярной IDE для Python на 2021 год [21]) изначально не поддерживает Gradle как систему сборки. Интеграция Gradle в PyCharm выглядит довольно сложной и относительно бесполезной задачей, так как с марта 2020 года плагин PyGradle не поддерживается создателями.

1.5. Выводы

Таким образом, существующие системы автоматизированной сборки общего назначения хоть и способны выполнять задачи, специфичные для проектов на языке Python, их конфигурация и настройка требуют очень много времени у программистов. В то же время рутинные задачи по созданию и управлению виртуальными окружениями, публикации пакетов, запуску тестов и удалённому исполнению кода могли бы выполняться единой системой сборки с декларативным описанием процессов и низким порогом вхождения. Кроме того, такая система сборки позволяла бы поддерживать зависимости по артефактам между задачами, а также зависимости между разными модулями (подпроектами) одного большого проекта в монорепозиториях с кодом.

На данный момент все упомянутые задачи выполняются разными консольными утилитами, а когда проект вырастает до размеров нескольких десятков модулей со своими собственными зависимостями, поддерживать такую инфраструктуру становиться очень сложно. За время поиска существующих продуктов и обзора релевантной литературы также не было найдено ни одной системы автоматизации разработки, выполняющей все описанные задачи и интегрирующейся в IDE (в частности, в PyCharm).

Со всеми этими проблемами столкнулись и разработчики платформы анализа естественного языка Grazie в компании JetBrains, работающие с моделями машинного обучения на Python в монорепозиториях, где каждая модель использует свой специфичный код и наборы зависимостей. Когда число таких моделей (и, соответственно, подпроектов в одном монорепозитории) растёт, использовать единственное виртуальное окружение становится затруднительно, так как миграция на новую версию одной из используемых библиотек для конкретной модели влечет за собой и миграцию для всех остальных моделей. Для работы с такими монорепозиториями изначально использовался Bazel [13], однако даже с плагином для языка Python конфигурация этой системы сборки занимала сотни строк кода.

В результате разработчиками был спроектирован прототип расширяемой системы автоматизации сборки Paddle [33]. С плагином для поддержки языка Python данная система являлась бы компромиссом между громоздкими системами сборки общего назначения и консольными утилитами для автоматизации процессов в проектах на Python. В качестве основного языка реализации был выбран Kotlin, из-за чего Paddle с лёгкостью интегрируется в IDE на основе IntelliJ Platform (в частности, PyCharm), так как позволяет переиспользовать логику отдельных своих компонент. Если будет нужна легковесная консольная утилита и захочется избавиться от зависимостей на JRE, можно будет воспользоваться технологиями Kotlin Multiplatform [24] или GraalVM [14].

2. Архитектура системы Paddle

В основе архитектуры Paddle лежит несколько важных концепций:

- Расширяемость — функциональность исходной системы сборки естественно ограничена, но допускает расширение за счет системы плагинов.
- Декларативность — в отличие от процедурных языков описания сборки, декларативный язык обладает более низким порогом вхождения для новых пользователей [25].
- Интегрируемость с IDE — в случае с IDE на базе IntelliJ Platform интегрируемость достигается уже за счёт выбора языка реализации (Kotlin) и модульной архитектуры.
- Инкрементальность — нет смысла запускать задачу сборки, если её входные данные или результат её исполнения остались неизменными с прошлого запуска.

Эти концепции находят отражение в структурной диаграмме компонент описываемой системы сборки (см. Рис. 3).

Основными исходными компонентами системы являются её ядро (Core), содержащее в себе главную абстракцию над проектом пользователя (Project), системы задач (Tasks) и функциональных расширений (Extensions); репозиторий плагинов (Repository); и, наконец, пользовательские интерфейсы — консольный (модуль CLI) и графический, в виде плагина к IDE PyCharm (модуль Plugin). Далее предлагается более подробное рассмотрение и анализ каждой из этих компонент.

2.1. Проект

Проект является основной абстракцией системы сборки и представляет программный проект пользователя. Любая функциональность, расширяющая возможности системы Paddle, добавляется в проект при помощи плагинов: они подключаются через специальную точку расшире-

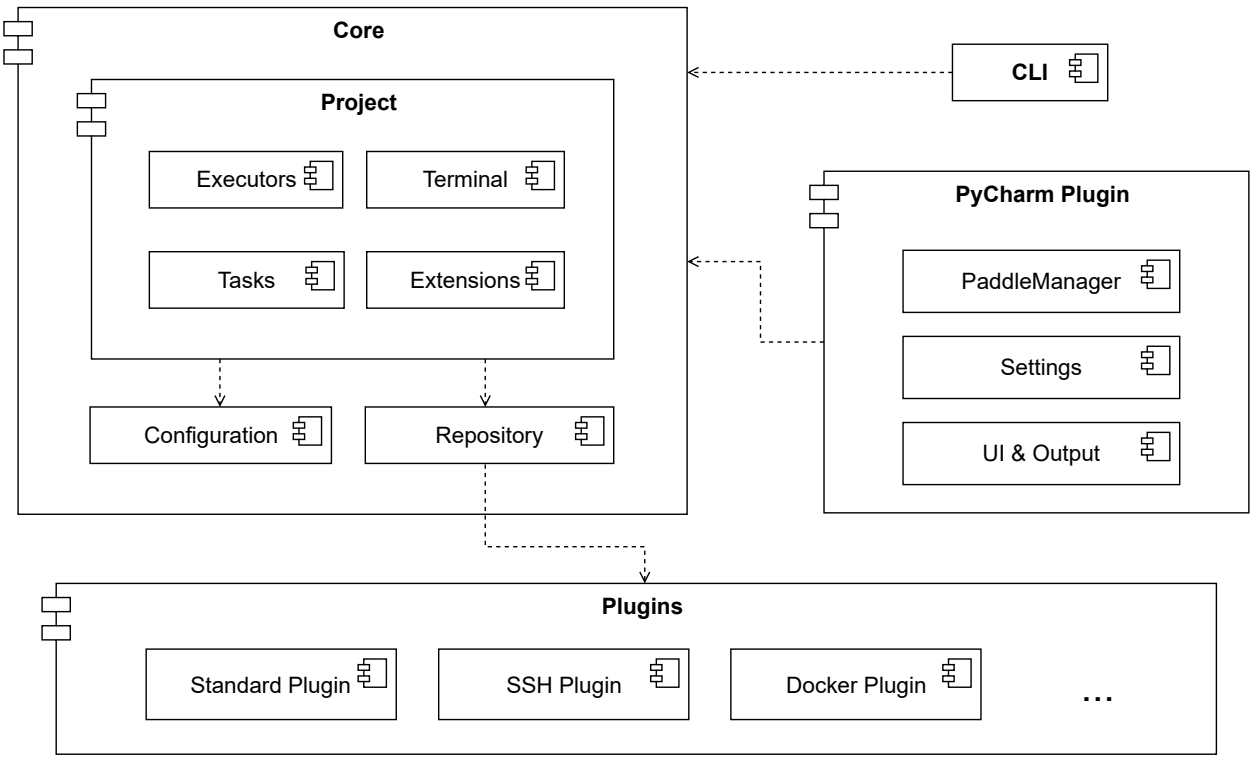

Рис 3: Исходная архитектура системы сборки Paddle.

ния — репозиторий плагинов. Технически это достигается за счёт использования механизма расширений (extensions[10]) в языке Kotlin, которые представляют собой некоторое переосмысление паттерна проектирования “Декоратор” [10].

Проект строится на основе конфигурации (Configuration), которую предоставляет пользователь в специальном YAML-файле (paddle.yaml). Такой файл должен содержаться в единственном экземпляре в корневой директории проекта, с его помощью Paddle способен автоматически распознавать и восстанавливать представление проекта. YAML [46] как язык для файла конфигурации был выбран неслучайно: он достаточно известный в сообществе разработчиков, преподносит себя как “дружественный” формат сериализации данных и активно используется для конфигурации запуска процессов сборки в CI/CD-сервисах.

Пример описываемой конфигурации представлен ниже:

[10]Kotlin Extensions: https://kotlinlang.org/docs/extensions.html

```
descriptor:
    name: example
    version: 0.1.0

roots:
  sources:
    - src/main
  tests:
    - src/test

plugins:
    enabled:
        - python
        - docker
        - ssh
```

Каждый блок данных конфигурации в таком файле (`descriptor`, `roots`, `plugins`) является отдельным расширением системы через определенный плагин, в данном случае стандартный, то есть встроенный в саму систему сборки Paddle. На примере данного файла видно, каким образом указывается название и версия текущего проекта, положение директорий с исходным кодом, тестами и файлами ресурсов, а также список подключаемых плагинов.

2.2. Задачи

Задачи — это единицы исполнения в системе автоматизированной сборки. Каждая задача имеет свой уникальный идентификатор, список задач-зависимостей (тех задач, которые должны быть успешно выполнены до вызова данной), а также соответствующие ей процедуры инициализации и запуска скриптов с искомой функциональностью задачи. С помощью списков зависимостей можно выстраивать иерархии задач произвольной сложности, при этом гарантируя, что, например,

задачи сборки проекта или запуска тестов всегда будут выполнены перед запуском задачи его публикации.

При запуске задачи системой сборки она переходит в состояние `EXECUTING`. В случае её успешного завершения состояние задачи изменяется на `DONE`, в случае происхождения какой-либо исключительной ситуации — на `FAILED`.

Кроме того, Paddle поддерживает концепцию *инкрементальности*: если задача уже выполнялась, и при этом её входные данные (input) и её результат (некоторый артефакт сборки, output) остались неизменными, запускать задачу на исполнение заново не имеет смысла. В таком случае у задачи будет статус `UP-TO-DATE`. Вместо того чтобы сравнивать входные данные и результаты задач на идентичность напрямую, сравниваются их хеши, вычисленные с помощью алгоритма MD5 [42]. В случае отдельных артефактов-файлов хеш считается от набора байт всего файла; в случае директории комбинируются хеши от вложенных директорий и файлов. Позже в рамках данной работы этот шаг был оптимизирован, и для сравнения файловых структур стали использоваться контрольные суммы, вычисленные с помощью сторонней Java-реализации алгоритма Adler32 [7].

2.3. Плагины

Как уже было упомянуто, плагины представляют собой модули для расширения функциональных возможностей системы Paddle. В коде они подключаются при сборке самого Paddle через сущность репозитория плагинов, но также возможно и использование плагинов в виде скомпилированных JAR-файлов, которые система умеет загружать из указанной директории.

Плагины обычно расширяют функциональность с помощью добавления новых задач. Так, например, в стандартный плагин для Paddle входит типичная задача очистки директорий от артефактов сборки `CleanTask`. Однако плагины могут и модифицировать (формально, декорировать, так как большинство сущностей в Paddle неизменяемы)

поведение других компонент системы — проекта, его расширений, модулей ввода-вывода команд, и так далее. Плагины, уже реализованные на данный момент вне рамок данной работы, позволяют запускать задачи на исполнение удалённо через ssh-протокол или производить запуск команд внутри Docker-контейнера. Конфигурация настроек любых новых плагинов также производится через файл конфигурации проекта paddle.yaml, схема которого изменяется в соответствии с используемыми плагинами.

Далее, в главе 3, будет подробно рассмотрен плагин для поддержки языка Python, добавляющий функциональность работы с виртуальными окружениями, внутренними и внешними репозиториями пакетов, версиями интерпретатора и lock-файлами зависимостей.

2.4. Пользовательские интерфейсы

Исходно Paddle содержит в себе два модуля, представляющих собой интерфейсы для использования разработчиком: CLI (Command Line Interface, консольное приложение) и Plugin (плагин для IDE PyCharm).

Консольная утилита предоставляет возможность загрузки и установки системы Paddle из удалённого GitHub-репозитория [33], а также запуска конкретных задач для конкретного проекта в рамках одной рабочей сессии.

Плагин для IDE обладает более сложной архитектурой. Он реализован с использованием IntelliJ Platform SDK [19] — открытого набора инструментов для создания плагинов к любым IDE на базе IntelliJ Platform, поддерживаемого компанией JetBrains. Основной класс в коде плагина, содержащий в себе точку входа, реализует стандартный интерфейс `ExternalSystemManager`, предоставляющий возможность интеграции произвольной системы сборки в IDE. С этим же интерфейсом, к примеру, в проекте IntelliJ IDEA Community Edition[11] реализована поддержка системы сборки Gradle [15]. Исходно, однако, в плагине для Paddle были реализованы лишь самые необходимые части, связанные

[11]Репозиторий проекта IntelliJ IDEA Community Edition:https://github.com/JetBrains/intellij-community

с синтаксическим разбором файла paddle.yaml, с поддержкой списка задач для запуска системой сборки, а также с выводом результатов их исполнения на консоль в IDE. О тех компонентах, которые были добавлены в плагин к IDE PyCharm в рамках данной работы, рассказывается в главе 5.

3. Поддержка языка Python

В данной главе речь пойдет о плагине для системы автоматизации сборки Paddle, расширяющем её функциональные возможности по поддержке проектов на языке Python. Диаграмма компонент Python-плагина представлена на Рис. 4.

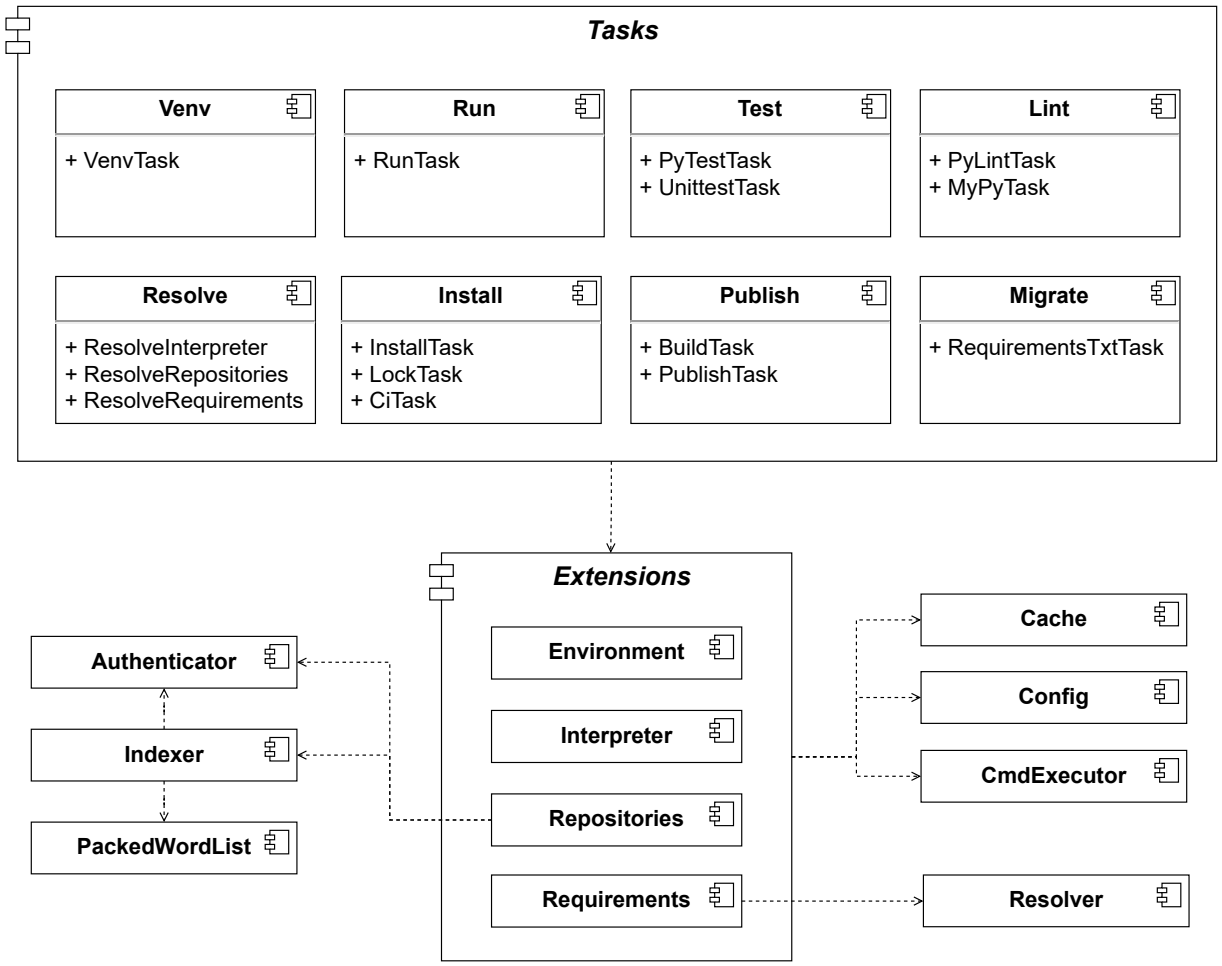

Рис 4: Предлагаемая архитектура плагина для поддержки Python.

Плагин добавляет в систему Paddle несколько новых групп задач:

1. `venv` — задачи по созданию и управлению виртуальными окружениями пакетов;
2. `resolve` — задачи по разрешению версий зависимостей, индексированию PyPi-репозиториев и поиску нужного интерпретатора;
3. `install` — задачи по установке пакетов и созданию lock-файлов;

4. `publish` — задачи по сборке дистрибутивов пакетов и их публикации в PyPi-репозитории пакетов;

5. `run` — задачи по запуску Python-скриптов и модулей;

6. `migrate` — задачи по миграции конфигурации пакетов из стандартных средств языка Python (например, `requirements.txt`) в средства Paddle;

7. `lint` — задачи по запуску линтеров Pylint [37] и Муру [27];

8. `testing` — задачи по запуску тестов с помощью Unittest [38] и pytest [39].

Во время своего исполнения задачи вызывают методы компонент-расширений (модуль Extensions), которые, в свою очередь, реализуют основную логику работы системы с виртуальными окружениями (Environment), интерпретатором Python (Interpreter), репозиториями пакетов (Repositories) и набором зависимостей проекта (Requirements). Кроме того, компонента Config осуществляет синхронизацию классов системы Paddle в оперативной памяти с пользовательскими настройками и кэшем с диска. По умолчанию все данные сохраняются в директории `$PADDLE_HOME` (на данный момент поддерживаются только операционные системы на основе Unix: семейства Linux и MacOS). О большинстве остальных же компонент пойдет речь в следующих разделах этой главы.

3.1. Локальный репозиторий пакетов

Со встроенным Python-плагином система сборки Paddle способна управлять зависимостями проекта из PyPi-репозиториев пакетов. Она добавляет функциональность создания и очистки виртуальных окружений в виде задач для Paddle-проекта, делегируя эти процедуры утилите `virtualenv` [51]. Однако при стандартном подходе к разработке на Python данная утилита каждый раз пытается установить дистрибутивы

пакетов *заново* в каждый новый проект с новым виртуальным окружением. Под установкой здесь понимается распаковка дистрибутивов пакетов с расширением `.whl` или архивов `.tar.gz`, компиляция отдельных частей пакета в байт-код (`.pyc`-файлы), и другое. В то же время на локальной машине пользователя кэшируется только сам дистрибутив во избежание повторных скачиваний из удалённых PyPi-репозиториев. При том, что наборы Python-пакетов современных фреймворков и библиотек могут занимать десятки гигабайт после установки (см. раздел 6.2), такая трата ресурсов диска пользователя расточительна.

Предлагаемое решение этой проблемы — устанавливать пакет определённой версии из определённого PyPi-репозитория только один раз в *локальный репозиторий пакетов* системы Paddle на машине разработчика, а после этого создавать символические (символьные) ссылки[12] на установленный пакет уже внутри виртуальных окружений проектов пользователя. Фактически первый раз пакет устанавливается в некоторое внутреннее виртуальное окружение Paddle с помощью менеджера пакетов pip, затем копируется в локальный репозиторий (откуда уже создаются символьные ссылки), а самое окружение очищается (см. Рис. 5).

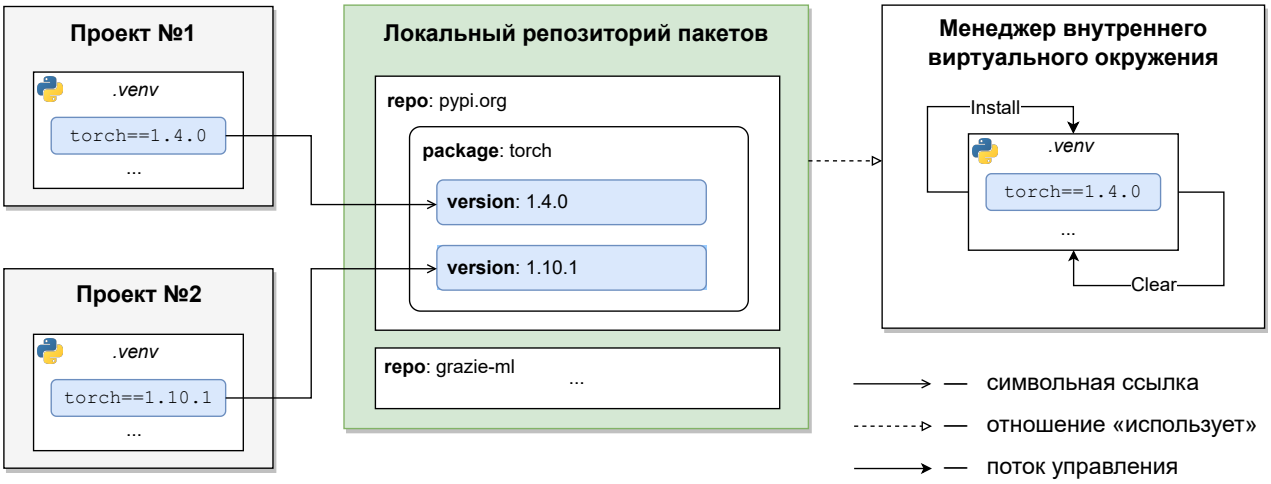

Рис 5: Реализация локального репозитория на примере использования нескольких версий пакета `torch` в разных проектах.

Такой подход, как показывают эксперименты в разделе 6.2, позво-

[12]Символьные ссылки в Linux: https://man7.org/linux/man-pages/man2/symlink.2.html

ляет экономить существенный объём памяти на диске пользователя. Выбор символьных ссылок (symlinks) на установленные пакеты в пользу жёстких (hardlinks) обоснован тем, что хоть они и занимают незначительно больше места в памяти, при удалении оригинального файла (то есть очистке локального репозитория вручную пользователем), они перестают быть корректными, в то время как жёсткие ссылки будут работать до тех пор, пока не удалится каждая ссылка в каждом виртуальном окружении. В то же время, например, пакетный менеджер Conda [8] создаёт жёсткие ссылки, но уже на пакеты из того виртуального окружения, куда они были установлены, тем самым создавая сложную структуру перекрёстных ссылок между виртуальными окружениями. В данной работе было решено отказаться от его использования в силу громоздкости самого менеджера Conda и тех файловых структур, которые он создаёт, в силу использования им внутренних каналов (channels) для скачивания пакетов, а также в силу коммерческого лицензирования данного продукта.

3.2. Разрешение зависимостей

Перед тем как устанавливать набор пакетов из PyPi-репозитория в локальный репозиторий пакетов Paddle, необходимо проверить, что данный набор пакетов *совместим*. Понятие *совместимости* означает, что ограничения по диапазонам версий, наложенные на указанные зависимости (а также зависимости этих зависимостей), не противоречат друг другу. Когда пакеты содержат скомпилированные файлы с бинарным кодом под определённую платформу или архитектуру процессора, это также приходится учитывать. Компонент, ответственный за эти процессы в системе управления пакетами, называется Resolver.

По умолчанию, в Python разрешением зависимостей занимается сам пакетный менеджер `pip`, но обычным пользователям через консольный интерфейс эта функциональность недоступна, потому что является лишь частью процесса установки пакетов. В связи с этим был создан

клон[13] репозитория pip с расширением стандартного набора команд командой `pip resolve`. Этой утилите Paddle (вернее, его Python-плагин) и делегирует разрешение версий зависимостей. В рамках будущей работы предполагается перенести реализацию на Kotlin для унификации кодовой базы проекта и ускорения работы системы. Тем не менее во избежание повторных вызовов сторонней консольной утилиты из основного процесса Paddle, работающего на виртуальной машине Java, каждый её вызов кэшируется. Для каждого разрешённого пакета впоследствии создается абстракция в оперативной памяти (класс PyPackage), с которой Paddle и продолжает работу. Если в момент кэширования такую абстракцию нужно загрузить с диска (или наоборот, по окончании сборки сохранить на диск), Paddle использует механизм сериализации данных в Kotlin[14].

3.3. Поиск интерпретатора Python

В Python-плагин для Paddle также была встроена функциональность поиска и установки релевантной версии интерпретатора Python. При указании версии в YAML-файле конфигурации в формате `python: x.y.z` сначала проверяется её корректность, затем производится поиск на локальной машине пользователя (среди стандартных директорий, а также среди обнаруженных виртуальных окружений менеджера Conda) и сравнение версий (с возможностью округления “минорных” версий и “патчей” — в примере выше, переменные `y` и `z` соответственно). В случае если искомая версия не обнаружена, происходит скачивание её дистрибутива с FTP-сервера `python.org` в директорию `$PADDLE_HOME/interpreters`, распаковка архива, автоматическая настройка переменных среды и установка всех необходимых требований.

[13]Клон репозитория pip (`pip-resolver`): https://github.com/SmirnovOleg/pip/releases/tag/22.1.dev0-beta

[14]Сериализация данных в Kotlin: https://kotlinlang.org/docs/serialization.html

3.4. Поддержка PyPi-репозиториев пакетов

Для поддержки сторонних PyPi-репозиториев пакетов (публичных и приватных), их индексирования и определения приоритета загрузки пакетов (в случае указания нескольких репозиториев сразу) было реализовано расширение системы компонентой Repositories.

Индексирование

При конфигурации внешних PyPi-репозиториев пакетов в файле paddle.yaml производится их индексирование, необходимое, в частности, для ускорения выполнения запросов по имени или версии пакета в системе автодополнения кода в редакторе IDE (см. раздел 5.1). Сохраняется весь список имён существующих пакетов, а при запросе конкретного пакета кэшируется список его версий и доступных дистрибутивов. Так как список имён *всех* пакетов из того же репозитория https://pypi.org/ в виде стандартных Java-строк занимает больше нескольких сотен мегабайт, наивно хранить его в оперативной памяти неразумно. В связи с этим было использовано несколько оптимизаций. Во-первых, имена пакетов могут включать лишь ограниченный набор Unicode-символов (латинский алфавит, цифры, дефисы, символы подчёркивания). Поэтому вместо 16-битных Char-представлений символов в Java используется 8-битный Byte. Во-вторых, применяется структура данных “сжатый список строк”, хранящая строки наборами по одной длине, отсортированными в лексикографическом порядке. Такой подход позволяет искать строки по префиксу бинарным поиском, что особенно полезно при реализации системы автодополнения кода в редакторе файла конфигурации. По сравнению со структурой данных “бор”, применяемой для подобных задач, описываемые массивы байт отлично кэшируются процессором, и потенциально могут занимать даже меньше места в оперативной памяти, так как не требуют хранения 64-битных ссылок на другие части структуры данных, как вершины в боре.

Поддержка приватных репозиториев

Для доступа к приватным PyPi-репозиториям, которые могут использоваться, например, для дистрибуции пакетов в рамках одной компании, пользователю нужна учётная запись. Чаще всего проверка таких учётных данных происходит с помощью механизма Basic Auth [41]. В силу того, что учётные данные пользователя нельзя хранить в открытом виде в конфигурации проекта, Paddle реализует несколько способов их предоставления системе аутентификации:

- `.netrc` — стандартный способ аутентификации в Unix-системах, в котором учётные данные пользователя хранятся в `.netrc`-файле;
- `keyring` — способ, использующий различные сервисы в качестве backend-модуля (например, Keychain на MacOS), популярный среди Python-разработчиков, так как его интерфейсы (API) поддерживаются в Python-пакете keyring;
- `profiles` — способ аутентификации через Paddle-профили пользователя, которые хранятся в файле `$PADDLE_HOME/profiles.yaml`. Дизайн данного метода позаимствован из IAM-профилей разработчика для AWS CLI[15].

У пакетного менеджера pip также присутствует возможность сохранения учетных данных пользователя вместе с URL-адресом PyPi-репозитория через команду `pip config`. Но для предупреждения возможных конфликтов между Paddle и pip на одной локальной машине (так как файл конфигурации у pip только один для одного окружения), а также для явного разделения URL-адресов репозиториев (которые можно хранить явно) и учетных данных (которые не должны попасть в открытый репозиторий с кодом), от данного способа было решено отказаться.

[15]Конфигурация профилей для AWS CLI: https://docs.aws.amazon.com/cli/latest/userguide/cli-configure-profiles.html

К примеру, в файле paddle.yaml итоговая конфигурация для приватного репозитория `grazie-ml` с использованием аутентификации через Paddle-профили будет выглядеть следующим образом:

```
repositories:
  - name: grazie-ml
    url: https://packages.jetbrains.team/grazie-ml/simple
    default: False       # приоритет ниже, чем у pypi.org
    auth:
      type: profile
      username: User.Name
```

Все запросы к серверам PyPi-репозиториев в системе сборки происходят асинхронно с использованием технологии Kotlin Coroutines[16] и http-клиента фреймворка Ktor[17].

3.5. Lock-файлы

Для поддержки воспроизводимых установок пакетов было реализовано расширение системы, позволяющее работать с lock-файлами. Lock-файл — специальный файл в репозитории проекта, загружаемый в том числе и в систему контроля версий, фиксирующий определённый набор конкретных установленных в текущем вирутальном окружении пакетов. Paddle, наподобие систем сборки в других языках, создаёт такой файл `paddle-lock.json` в корневой директории проекта. От файла `requirements.txt`, к примеру, он отличается тем, что содержит точные версии и имена дистрибутивов пакетов (а не ограничения на версии), их зависимостей, а также MD5/SHA256-хеши содержимого пакетов. Последние позволяют производить верификацию пакетов во избежание “атак на цепочку поставок” (supply chain attack [29]), когда злоумышленник подменяет содержимое пакета-зависимости из открытого репозитория с целью добавления туда уязвимостей. Кроме того,

[16]Kotlin Coroutines: https://kotlinlang.org/docs/coroutines-overview.html
[17]Ktor Http Client: https://ktor.io/docs/create-client.html

lock-файлы позволяют ускорять сборки проектов на CI-сервисах, когда необходимо создавать новое виртуальное окружение и устанавливать зависимости туда: Paddle добавляет задачу `ci`, которая верифицирует и устанавливает указанные версии пакетов напрямую из lock-файла, избегая шага с разрешением зависимостей.

4. Поддержка монорепозиториев

Одной из исходных задач данной работы была поддержка монорепозиториев проектов на языке Python. Напомним, что под монорепозиторием здесь понимается особая организация проекта с несколькими модулями (подпроектами), где все они хранятся в одном репозитории системы контроля версий. При этом нужно было поддерживать возможность создания зависимостей между подпроектами, индексировать их структуру и уметь встраиваться в систему импортов языка Python. Обо всех этих тонкостях рассказывается в данной главе.

4.1. Подпроекты

Модель подпроектов была реализована в виде расширения Subprojects для стандартного плагина системы сборки Paddle. Каждый подпроект предполагает наличие своего paddle.yaml-файла с конфигурацией в корневой директории. На их основе строится и индексируется глобальная модель монорепозитория, в которой уже далее создаются отдельные Paddle-проекты. За загрузку проектов и синхронизацию модели с файловой системой отвечает компонента `ProjectProvider`.

4.2. Система импортов

Расширение Subprojects добавляет возможность указания списка зависимостей на подпроекты в специальном подразделе файла paddle.yaml. Чтобы поддержать данные зависимости и в системе импортов самого языка Python, в задачи по запуску Python-скриптов был интегрирован механизм изменения переменной среды `$PYTHONPATH`. Тем самым, при запуске скрипта из какого-либо подпроекта ему становятся доступны импорты из всех пакетов, указанных в Paddle-конфигурации подпроектов.

4.3. Граф зависимостей задач

Когда у проекта появились зависимые подпроекты, стало необходимо поддерживать эти зависимости и в графе зависимостей задач. Исходно в прототипе системы сборки Paddle у каждой задачи был лишь список зависимостей на другие задачи, которые линейно запускались перед её исполнением. Хоть система и поддерживала инкрементальность, что позволяло не запускать задачи по несколько раз с одним и тем же результатом, подсчёт хешей для обновления состояния задачи в `UP-TO-DATE` занимал некоторое время. Данный модуль системы был переписан, и теперь в рамках одной сборки каждая задача запускается лишь один раз, что достигается топологической сортировкой ориентированного графа задач. Для поддержки подпроектов в зависимости каждой задачи просто добавляются задачи с тем же самым идентификатором, но из подпроектов данного проекта. Так, например, полный идентификатор задачи `lock` из подпроекта `:subproject` основного проекта `:project` будет иметь вид `:project:subsproject:lock`, а граф зависимостей между задачами и подпроектами будет выглядеть следующим образом (см. Рис. 6).

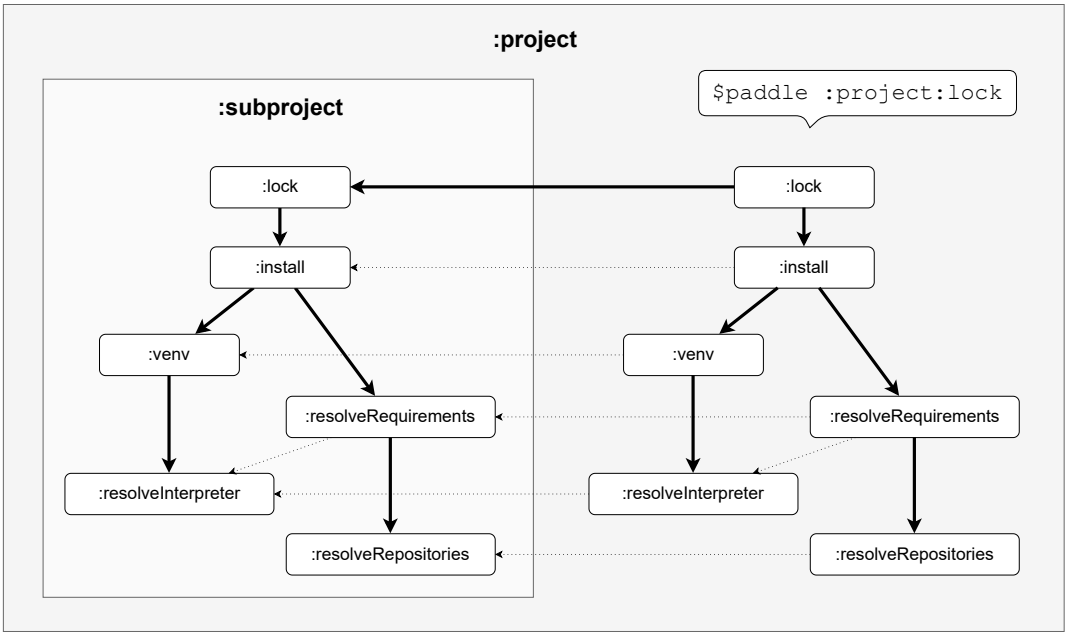

Рис 6: Топологический порядок обхода графа задач, исполняемых при запуске задачи `lock` в проекте `:project` с подпроектом `:subproject`.

4.4. Сборка и публикация пакетов

Для сборки и публикации дистрибутивов пакетов в Python-плагин системы были также добавлены задачи `build` и `publish`. Задача `build` использует стандартный пакет setuptools для генерации .whl- и .tar.gz-дистрибутивов, при этом генерируя на ходу все необходимые конфигурации (вроде файлов pyproject.toml и setup.cfg), извлекая данные о проекте из его модели в оперативной памяти и файла paddle.yaml. Задача `publish` даёт возможность публикации пакетов с помощью разных сервисов: стандартного пакета twine и утилиты JFrog [17], используемой внутри JetBrains для публикации приватных дистрибутивов. Обе задачи корректно работают с подпроектами, собирая и публикуя сначала зависимости основного проекта, и лишь затем приступая к нему.

5. Интеграция в плагин для IDE PyCharm

На Рис. 7 показаны добавленные в ядро системы новые компоненты (Subprojects и ProjectProvider для поддержки монорепозиториев), а также плагин-расширение для поддержки языка Python.

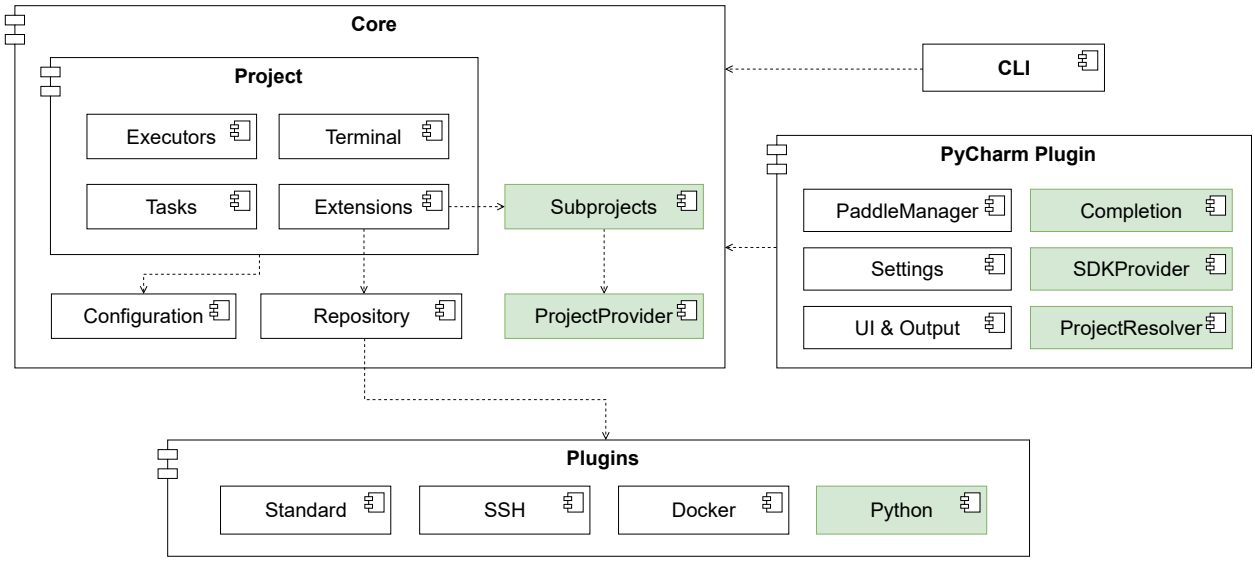

Рис 7: Архитектура системы Paddle после поддержки подпроектов и добавления Python-плагина. Зелёным здесь выделены компоненты, целиком реализованные в рамках данной работы.

Как можно заметить, в плагин для IDE PyCharm также добавилось несколько компонент, связанных с новой функциональностью системы Paddle. В данном разделе предлагается рассмотреть их подробнее.

5.1. Автодополнение фрагментов кода и инспекции

Для улучшения пользовательского опыта взаимодействия с системой внутри IDE была реализована подсистема автодополнения фрагментов кода конфигурации. В частности, это касается имён и версий пакетов в разделе `requirements` файла paddle.yaml. Paddle автоматически загружает необходимые индексы указанных PyPi-репозиториев из секции `repositories`, которые были описаны в разделе 3.4, и использует поиск по префиксу (который уже набрал на клавиатуре пользователь) чтобы найти все подходящие варианты имён и версий пакетов

из репозитория. Технически это реализуется с помощью одной из точек расширения IDE на основе IntelliJ Platform — Completion Contributor.[18]

Также был реализован набор *инспекций* для поддержания целостности и корректности файла конфигурации paddle.yaml. В IntelliJ Platform, инспекции — специальные фоновые процессы, статически анализирующие код пользователя[19]. В случае с Paddle набор реализованных инспекций проверяет, не указал ли пользователь несколько PyPi-репозиториев "по умолчанию" (пакетный менеджер pip может работать только с одним), не оставил ли разработчик по неосторожности свои учётные данные для аутентификации (например, в URL репозитория в формате `https://username:password@path/to/repo`) и так далее.

5.2. Разметка проекта и модулей

Кроме поддержки подпроектов в модели проекта системы сборки Paddle также была необходима и их поддержка в модели проекта в самой IDE. Здесь подразумевается разметка модулей проекта в терминах IDE, в случае с PyCharm — в терминах, общих для всех IDE на основе IntelliJ Platform. Для каждого подпроекта системы Paddle создаётся отдельный модуль в IDE (вместе с соответствующим .iml-файлом в корне подпроекта), добавляются указанные корневые директории (Source Roots) для исходного кода, тестов и ресурсов — так, как они указаны в конфигурационном файле paddle.yaml. Кроме того, для каждого модуля создаётся набор необходимых зависимостей на другие модули, а также генерируется дерево для окна задач системы Paddle.

5.3. Конфигурация SDK

Software Development Kit (SDK) — термин, общий для всех IDE на основе IntelliJ Platform, в данном случае подразумевает директорию с виртуальным окружением проекта. По умолчанию PyCharm предлага-

[18]Completion Contributors: https://plugins.jetbrains.com/docs/intellij/completion-contributor.html

[19]Code Inspections: https://www.jetbrains.com/help/idea/code-inspection.html

ет указать лишь один SDK для использования в проекте. IDE автоматически распознает версию интерпретатора, используемый менеджер зависимостей (pip, Poetry, Conda), а также позволяет разрешать зависимости в коде проекта и осуществлять навигацию по ним[20]. Плагин для системы Paddle позволяет автоматически (после исполнения задачи `venv`) конфигурировать Python SDK для *каждого* модуля в отдельности в соответствии с его конфигурацией в файле paddle.yaml.

Окно активной IDE PyCharm с поддержкой плагина для системы сборки Paddle, а также указанные элементы, поддержка которых была реализована в рамках данной работы, отображены на Рис. 8.

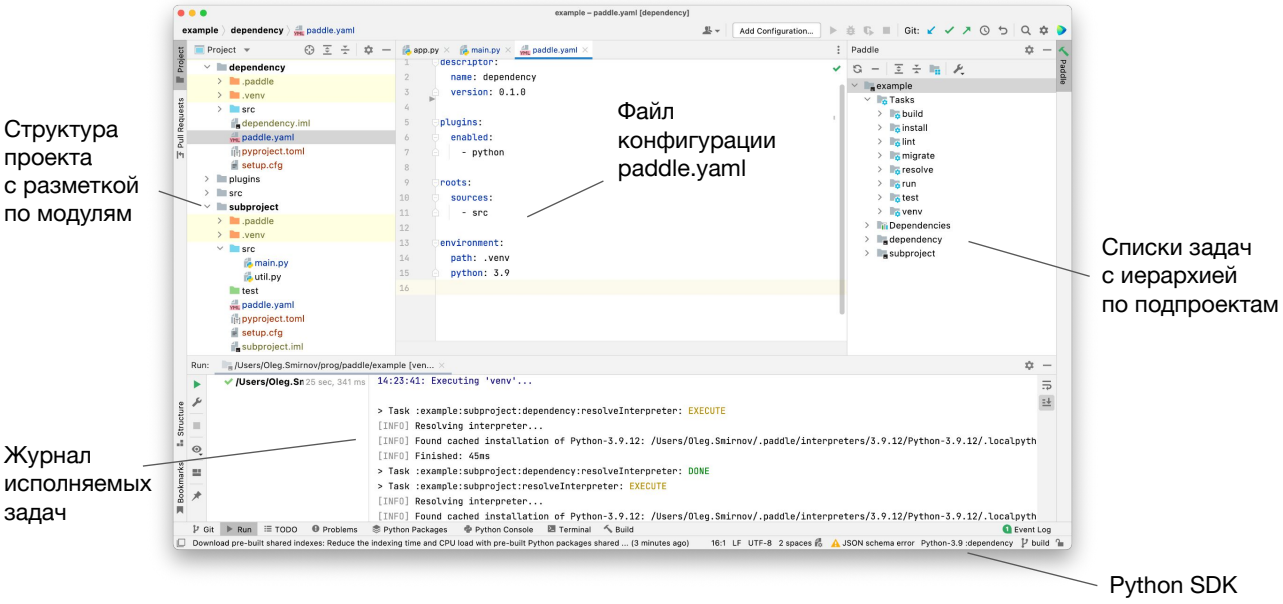

Рис 8: IDE PyCharm при запуске с плагином для системы Paddle.

[20]Примеры использования Import Resolver Extension Point: https://plugins.jetbrains.com/intellij-platform-explorer?extensions=Pythonid.importResolver

6. Апробация

В качестве апробации инструмент автоматизации процессов сборки и разработки Paddle был протестирован в нескольких проектах на языке Python разной сложности. В частности, были рассмотрены следующие внутренние проекты по машинному обучению и анализу естественного языка команды Grazie в компании JetBrains:

- `grazie-crowd` — репозиторий с утилитами для автоматизированного взаимодействия с сервисом краудсорсинга Yandex Toloka [47];
- `grazie-cloud` — монорепозиторий, содержащий различные сервисы платформы анализа естественного языка Grazie;
- `grazie-ml` — монорепозиторий для обучения моделей машинного обучения и их последующей публикации в `grazie-cloud`, на данный момент использующий систему сборки Bazel [13].

Первый из указанных проектов сравнительно простой, не использует систему зависимостей между подпроектами и является монолитным Python-пакетом по своей архитектуре. Два других, напротив, представляют собой монорепозитории с множеством подпроектов, между которыми присутствуют зависимости.

6.1. Репозиторий `grazie-crowd`

Так как данный проект содержал только один Python-пакет, его перенос на систему сборки Paddle вручную не представлял проблем — необходимо было лишь создать и сконфигурировать единственный paddle.yaml-файл. Дальнейшая апробация (в том числе, и с двумя остальными репозиториями) подразумевала запуск различных сценариев использования системы сборки, таких как:

- задачи по преобразованию исходного requirements.txt-файла в формат файла сборки paddle.yaml;

- задачи по разрешению и установке зависимостей, а также генерации lock-файла;
- задачи по сборке и публикации .whl-дистрибутива пакета;
- задачи по запуску тестов (с помощью тестового фреймворка pytest) и линтеров (Pylint и Муру).

6.2. Репозиторий `grazie-cloud`

Данный монорепозиторий содержал внутри себя 15 различных подпроектов, поэтому в систему Paddle были добавлены задачи для автоматической миграции requirements.txt-файлов в файлы системы сборки Paddle (paddle.yaml). В ходе апробации также с помощью локального репозитория пакетов (см. раздел 3.1) удалось значительно сократить объём памяти, занимаемой на диске виртуальными окружениями каждого из 15-ти подпроектов. Исходно код проекта вместе со всеми установленными зависимостями занимал 32,1 Гб, в то время как с использованием системы автоматизации сборки Paddle суммарный объём занимаемой проектом памяти сокращается до 4,1 Гб. Из них сам репозиторий занимает всего 274,6 Мб — это доступный разработчику код проекта и виртуальные окружения с символьными ссылками, остальное же хранится во внутренних директориях локального репозитория пакетов, которыми управляет Paddle.

6.3. Репозиторий `grazie-ml`

В ходе экспериментов над монорепозиторием `grazie-ml` (самым объёмным из рассмотренных, 3,6 Гб исходно), использующем систему сборки Bazel, перенести на систему Paddle целиком к моменту написания данной работы удалось лишь несколько подпроектов. Кроме того, репозиторий использует единственное pipenv-окружение для всех модулей одновременно (что само по себе уже является большой проблемой при миграции отдельных библиотек на новые версии), из-за чего оценить

итоговую разницу в занимаемой виртуальными окружениями памяти не представляется возможным.

Кроме того, были выяснены основные функциональные особенности, которые необходимо поддержать в будущем в системе автоматизации разработки Paddle. В частности, это реализация императивных расширений для системы сборки уже на языке Python (для тонкой конфигурации и настройки системы разработчиком), поддержка кэширования установленных пакетов на уровне организации, а также реализация удалённого и распределённого (в смысле исполнения на кластере вычислительных узлов) запуска задач сборки.

Заключение

Таким образом, в ходе данной выпускной квалификационной работы удалось достигнуть следующих результатов.

1. Проведен обзор систем сборки общего назначения и основных инструментов разработки на языке Python, рассмотрены системы управления пакетами и виртуальными окружениями.

2. Реализована подсистема для поддержки языка Python в системе автоматизации разработки Paddle.

   (a) Реализованы расширения для управления пакетами, для создания и хранения виртуальных окружений, а также для установки произвольной версии интерпретатора.

   (b) Реализована поддержка приватных PyPi-репозиториев и lock-файлов зависимостей проекта.

   (c) Реализовано расширение для работы с монорепозиториями Python-проектов.

3. Выполнена интеграция Python-расширения системы Paddle в соответствующий плагин для IDE PyCharm, реализована поддержка систем синхронизации состояния проекта, конфигурации SDK, автодополнения кода.

4. Проведены эксперименты по использованию системы автоматизации Paddle на трёх репозиториях компании JetBrains в области машинного обучения, рассмотрены основные сценарии применения системы и выделены направления дальнейшей работы (поддержка императивных расширений на Python, кэширование установленных пакетов на уровне организации, распределённое исполнение задач сборки).

Репозиторий проекта находится в открытом доступе на GitHub[21].

[21] Система автоматизации разработки Paddle: https://github.com/TanVD/paddle

Список литературы

[1] Anderson Charles. Docker [software engineering] // Ieee Software. — 2015. — Vol. 32, no. 3. — P. 102–c3.

[2] Apache. ANT. — 2000. — Access mode: https://ant.apache.org/ (online; accessed: 06.05.2022).

[3] Apache. Maven. — 2008. — Access mode: https://maven.apache.org/ (online; accessed: 06.05.2022).

[4] Brooks Jr Frederick P. The mythical man-month (anniversary ed.). — 1995.

[5] Brousse Nicolas. The issue of monorepo and polyrepo in large enterprises // Proceedings of the Conference Companion of the 3rd International Conference on Art, Science, and Engineering of Programming. — 2019. — P. 1–4.

[6] CLion. — Access mode: https://www.jetbrains.com/clion/ (online; accessed: 06.05.2022).

[7] Castagnoli Guy, Brauer Stefan, and Herrmann Martin. Optimization of cyclic redundancy-check codes with 24 and 32 parity bits // IEEE Transactions on Communications. — 1993. — Vol. 41, no. 6. — P. 883–892.

[8] Conda. — Access mode: https://docs.conda.io/en/latest/ (online; accessed: 06.05.2022).

[9] Demers Alan, Reps Thomas, and Teitelbaum Tim. Incremental evaluation for attribute grammars with application to syntax-directed editors // Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. — 1981. — P. 105–116.

[10] Gamma Erich, Helm Richard, Johnson Ralph, and Vlissides John. Design patterns: Abstraction and reuse of object-oriented design // Eu-

ropean Conference on Object-Oriented Programming / Springer. — 1993. — P. 406–431.

[11] Feldman Stuart I. Make—A program for maintaining computer programs // Software: Practice and experience. — 1979. — Vol. 9, no. 4. — P. 255–265.

[12] GNU. Make. — 1976. — Access mode: https://www.gnu.org/software/make/ (online; accessed: 06.05.2022).

[13] Google. Bazel. — 2015. — Access mode: https://bazel.build/ (online; accessed: 06.05.2022).

[14] GraalVM. — Access mode: https://www.graalvm.org/ (online; accessed: 06.05.2022).

[15] Gradle. — 2008. — Access mode: https://gradle.com/ (online; accessed: 06.05.2022).

[16] IntelliJ IDEA. — Access mode: https://www.jetbrains.com/idea/ (online; accessed: 06.05.2022).

[17] JFrog. — Access mode: https://jfrog.com/ (online; accessed: 06.05.2022).

[18] Jenkins. — 2011. — Access mode: https://www.jenkins.io/ (online; accessed: 06.05.2022).

[19] JetBrains. IntelliJ Platform. — 2000. — Access mode: https://plugins.jetbrains.com/docs/intellij/welcome.html (online; accessed: 06.05.2022).

[20] JetBrains. TeamCity. — 2006. — Access mode: https://www.jetbrains.com/teamcity/ (online; accessed: 06.05.2022).

[21] JetBrains. The State of Developer Ecosystem 2021. — 2021. — Access mode: https://www.jetbrains.com/lp/devecosystem-2021/python/ (online; accessed: 06.05.2022).

[22] Johnson Philip M. You can't even ask them to push a button: Toward ubiquitous, developer-centric, empirical software engineering // The NSF Workshop for New Visions for Software Design and Productivity: Research and Applications / Citeseer. — 2001.

[23] Johnson Stephen C. Lint, a C program checker. — Bell Telephone Laboratories Murray Hill, 1977.

[24] Kotlin MultiPlatform. — Access mode: https://kotlinlang.org/docs/multiplatform.html (online; accessed: 06.05.2022).

[25] Lloyd John W. Practical Advtanages of Declarative Programming. // GULP-PRODE (1). — 1994. — P. 18–30.

[26] Mokhov Andrey, Mitchell Neil, and Peyton Jones Simon. Build systems à la carte // Proceedings of the ACM on Programming Languages. — 2018. — Vol. 2, no. ICFP. — P. 1–29.

[27] MyPy. — Access mode: https://mypy.readthedocs.io/en/stable/ (online; accessed: 06.05.2022).

[28] Nath Partho, Urgaonkar Bhuvan, and Sivasubramaniam Anand. Evaluating the usefulness of content addressable storage for high-performance data intensive applications // Proceedings of the 17th international symposium on High performance distributed computing. — 2008. — P. 35–44.

[29] Wolff Evan D, Growley KM, Gruden MG, et al. Navigating the solarwinds supply chain attack // The Procurement Lawyer. — 2021. — Vol. 56, no. 2.

[30] Node Package Manager, NPM. — 2014. — Access mode: https://www.npmjs.com/ (online; accessed: 06.05.2022).

[31] PEP 503. — Access mode: https://www.python.org/dev/peps/pep-0503/ (online; accessed: 06.05.2022).

[32] PIP, Python Package Manager. — 2011. — Access mode: https://pypi.org/project/pip/ (online; accessed: 06.05.2022).

[33] Paddle. — 2021. — Access mode: https://github.com/TanVD/paddle (online; accessed: 06.05.2022).

[34] Pants. — 2014. — Access mode: https://www.pantsbuild.org/ (online; accessed: 06.05.2022).

[35] Poetry. — 2018. — Access mode: https://python-poetry.org/ (online; accessed: 06.05.2022).

[36] PyCharm. — Access mode: https://www.jetbrains.com/pycharm/ (online; accessed: 06.05.2022).

[37] Pylint. — Access mode: https://pylint.pycqa.org/en/latest/ (online; accessed: 06.05.2022).

[38] Pytest. — Access mode: https://docs.python.org/3/library/unittest.html (online; accessed: 06.05.2022).

[39] Pytest. — Access mode: https://docs.pytest.org/en/7.1.x/ (online; accessed: 06.05.2022).

[40] RFC 4253. — Access mode: https://datatracker.ietf.org/doc/html/rfc4253 (online; accessed: 06.05.2022).

[41] RFC 7617. — Access mode: https://datatracker.ietf.org/doc/html/rfc7617 (online; accessed: 06.05.2022).

[42] The MD5 message-digest algorithm : Rep. ; executor: Rivest Ronald : 1992.

[43] Röthlisberger D. Benchmarking the Ninja build system. — 2012. — Access mode: https://david.rothlis.net/ninja-benchmark/ (online; accessed: 06.05.2022).

[44] Szorc G. Makefile Execution Times. — 2012. — Access mode: https://gregoryszorc.com/blog/2012/07/28/makefile-execution-times/ (online; accessed: 06.05.2022).

[45] Travis CI. — 2011. — Access mode: https://www.travis-ci.com/ (online; accessed: 06.05.2022).

[46] YAML. — 2009. — Access mode: https://yaml.org/ (online; accessed: 06.05.2022).

[47] Yandex Toloka. — Access mode: https://toloka.yandex.ru/ (online; accessed: 06.05.2022).

[48] pipenv. — 2018. — Access mode: https://pipenv.pypa.io/en/latest/ (online; accessed: 06.05.2022).

[49] pyenv. — 2017. — Access mode: https://github.com/pyenv/pyenv/ (online; accessed: 06.05.2022).

[50] venv. — Access mode: https://docs.python.org/3/library/venv.html (online; accessed: 06.05.2022).

[51] virtualenv. — 2007. — Access mode: https://virtualenv.pypa.io/en/latest/ (online; accessed: 06.05.2022).