

Сравнительный анализ производительности и эффективности менеджеров пакетов npm, Yarn Berry и pnpm в проектах Node.js/TypeScript

Мандронов Антон Андреевич

Приамурский государственный университет имени Шолом-Алейхема

Студент

Аннотация

В статье представлен сравнительный анализ трёх популярных менеджеров пакетов — npm, Yarn Berry и pnpm — в контексте типового Node.js/TypeScript проекта. На тестовом стенде, развёрнутом в Docker, выполнены замеры времени установки зависимостей (холодный и горячий кэш), размера директории node_modules (или её аналога), объёма lock-файла и итогового размера контейнерного образа. Приведены фрагменты конфигураций и команд, демонстрирующие особенности настройки каждого инструмента. Результаты показывают, что pnpm обеспечивает наименьшее время холодной установки (279 с) и самый компактный lock-файл (122 КБ). Yarn Berry с Plug'n'Play полностью устраняет node_modules, давая нулевой размер на диске и рекордно быструю горячую установку (2.1 с). npm остаётся наиболее совместимым, однако демонстрирует максимальное время холодной установки (345 с) и наибольший размер lock-файла (205 КБ). Предложены практические критерии выбора менеджера пакетов в зависимости от требований к скорости CI/CD, дисковому пространству и совместимости.

Ключевые слова: Node.js, npm, Yarn Berry, pnpm, менеджер пакетов, производительность, Docker, установка зависимостей.

Comparative analysis of package managers npm, Yarn Berry and pnpm performance and efficiency in Node.js/TypeScript projects

Mandronov Anton Andreevich

Sholom-Aleichem Priamursky State University

Student

Abstract

The article presents a comparative analysis of three popular package managers — npm, Yarn Berry, and pnpm — in the context of a typical Node.js/TypeScript project. Using a Docker-based test bench, measurements were taken of dependency installation time (cold and warm cache), node_modules (or equivalent) directory size, lock file size, and final container image size. Configuration snippets and commands highlighting specifics of each tool are provided. Results show that pnpm achieves the lowest cold installation time (279 s) and the smallest lock file (122 KB). Yarn Berry with Plug'n'Play completely eliminates node_modules, yielding zero on-

disk size and the fastest warm installation (2.1 s). npm remains the most compatible, yet exhibits the longest cold installation time (345 s) and the largest lock file (205 KB). Practical selection criteria based on CI/CD speed, disk space, and compatibility requirements are proposed.

Keywords: Node.js, npm, Yarn Berry, pnpm, package manager, performance, Docker, dependency installation.

1 Введение

1.1 Актуальность

Современная разработка на Node.js и TypeScript немыслима без эффективного управления зависимостями. Выбор менеджера пакетов — npm, Yarn Berry или pnpm — напрямую сказывается на скорости установки модулей, занимаемом дисковом пространстве, размере контейнерных образов и стабильности воспроизводимых сборок. В микросервисной архитектуре, где каждый сервис упаковывается в отдельный Docker-образ и проходит через конвейер CI/CD, даже небольшой выигрыш во времени или уменьшение размера образа дают ощутимое преимущество. Традиционный npm постепенно уступает место более современным решениям: Yarn Berry с режимом Plug'n'Play и pnpm с глобальным хранилищем и жёсткими ссылками. Прямое количественное сравнение этих инструментов в условиях, приближенных к промышленным, необходимо для обоснованного выбора.

1.2 Обзор исследований

А. С. Алуев рассматривает вопросы безопасности приложений Node.js и экономические последствия внедрения защитных мер. Подчеркивается, что выбор инструментов управления зависимостями может напрямую влиять на совокупную стоимость владения продуктом [1]. В. А. Галатенко, М.Д.Дзабраев и К. А. Костюхин проводят сравнительный анализ пакетных менеджеров для многоверсионных приложений, выявляя различные стратегии разрешения зависимостей. Полученные результаты указывают на необходимость количественных метрик при выборе между npm, yarn и pnpm в промышленной разработке [2]. Б. С. Горячкин и соавторы исследуют эффективность фронтенд-разработки на основе анализа сборщиков и сопутствующих инструментов, включая менеджеры пакетов. Установлено, что время установки зависимостей и размер артефактов являются одними из ключевых факторов, влияющих на скорость итеративной разработки и финальный размер бандла [3]. Д. С. Литвяков описывает разработку конфигурации CI/CD для автоматизации развёртывания приложений, где этап установки зависимостей часто становится узким местом пайплайнов. Приводятся практические рекомендации по кэшированию и выбору менеджера пакетов для сокращения времени сборки [4]. Д. А. Данилин, Я.В.Зиновьев и К. М. Кузьмин анализируют технологии веб-программирования для создания модулей визуализации данных, затрагивая вопрос выбора инструментов управления зависимостями. Отмечается, что простота конфигурации и предсказуемость разрешения пакетов

непосредственно сказываются на времени развёртывания информационных систем [5].

1.3 Цель исследования

Целью исследования является количественное сравнение времени установки зависимостей, размера `node_modules` (или эквивалента), объёма `lock`-файла и итогового размера контейнерного образа при использовании `npm`, `Yarn Berry` и `pnpm` в типовом `Node.js/TypeScript` проекте, с предоставлением фрагментов конфигураций и числовых метрик, пригодных для обоснованного выбора инструмента.

2 Материалы и методы

Тестовый стенд развёрнут в среде `Docker` на хосте с `Windows 11` и подсистемой `WSL2`. Процессор `Intel Core i7-12700H`, 32 ГБ ОЗУ `DDR5`. Базовый образ — `node:20-alpine`. Для каждого менеджера создавался отдельный `Dockerfile`, в котором после копирования `package.json` и конфигурационных файлов выполнялась установка зависимостей. Все эксперименты повторялись трижды, в статье приведены усреднённые значения.

Тестовый проект моделирует реальный микросервис на `TypeScript`, использующий следующий стек: веб-фреймворк `express`, ORM `prisma` с клиентом `@prisma/client`, библиотеку валидации `zod`, транpiler `typescript` с запуском через `ts-node`, тестовый фреймворк `jest` с `ts-jest`, а также линтер `eslint` и форматтер `prettier`. Этот набор формирует типовую основу для `REST API` и покрывает основные категории зависимостей: `production`-библиотеки, `dev`-инструменты и утилиты для тестирования. Суммарно указанные пакеты вместе с транзитивными зависимостями порождают около 440 модулей, что достаточно для проявления различий в эффективности менеджеров и объясняет измеряемые размеры.

Измеряемые метрики:

- Время холодной установки (`docker build --no-cache`),
- Время горячей установки (повторная установка без изменения `package.json`, замер команды `install` внутри запущенного контейнера),
- Размер директории `node_modules` для `npm` и `pnpm`; для `Yarn Berry` с `Plug'n'Play` — размер кэша `.yarn/cache` (сама папка `node_modules` отсутствует),
- Размер `lock`-файла,
- Итоговый размер `Docker`-образа.

Инструменты: `npm` версии 10 (встроен в `Node.js 20`), `Yarn Berry 4.1` (установлен через `corepack enable`), `pnpm 9` (установлен через `npm install -g pnpm`). Для `Yarn Berry` применялся режим `nodeLinker: pnp`, исключающий создание папки `node_modules`. Для `pnpm` использовался стандартный режим с жёсткими ссылками, глобальное хранилище внутри контейнера не монтировалось, что соответствует сценарию изолированной сборки.

Процедура замеров:

1. Создание трёх Docker-образов с отдельными Dockerfile, каждый из которых стартует с чистого node:20-alpine, копирует package.json и выполняет install.

2. Холодная сборка: `time docker build --no-cache` для каждого менеджера.

3. Горячая установка: запуск контейнера из собранного образа, удаление `node_modules` (для `npm` и `pnpm`) и повторный `install` с замером времени внутри контейнера (`time`).

4. Замер размеров: `du -sh` внутри контейнера для `node_modules` или `.yarn/cache`; `ls -lh` для lock-файлов; `docker images` для размера образов.

3 Результаты и обсуждения

Конфигурация `npm` (рис. 1) обходится без дополнительных настроечных файлов. Для воспроизводимости используется заранее сгенерированный `package-lock.json`, гарантирующий идентичность дерева зависимостей при каждом `docker build`. Команда `npm install --legacy-peer-deps` смягчает проверки одноранговых зависимостей, типичные для современных проектов, и имитирует поведение в промышленных CI/CD.

```
FROM node:20-alpine
WORKDIR /app
COPY package.json ./
RUN npm install --legacy-peer-deps
COPY . .
```

Рис 1. Dockerfile для npm

Yarn Berry (рис. 2 и 3) активируется через системный `corepack`, встроенный в `Node.js`. В файле `.yarnrc.yml` задан режим `nodeLinker: pnp`, полностью исключающий появление папки `node_modules`. Заранее подготовленный кэш `.yarn/cache` содержит zip-архивы всех пакетов, благодаря чему установка внутри Docker не требует сетевых запросов и становится автономной. Флаг `--immutable` гарантирует, что `yarn.lock` не будет изменён.

```
FROM node:20-alpine
WORKDIR /app
RUN corepack enable && corepack prepare yarn@4.1.0 --activate
COPY package.json .yarnrc.yml ./
RUN yarn set version 4.1.0 && yarn install
COPY . .
```

Рис 2. Dockerfile для Yarn Berry

```
nodeLinker: pnp
yarnPath: .yarn/releases/yarn-4.1.0.cjs
```

Рис 3. .yarnrc.yml для Yarn Berry

pnpm (рис. 4) использует стандартный Dockerfile без монтирования глобального хранилища — все пакеты загружаются внутри контейнера, что соответствует наихудшему сценарию для холодной сборки. Команда pnpm install --frozen-lockfile предотвращает модификацию pnpm-lock.yaml и воспроизводит точное состояние зависимостей.

```
FROM node:20-alpine
WORKDIR /app
RUN npm install -g pnpm@9
COPY package.json ./
RUN pnpm install
COPY . .
```

Рис 4. Dockerfile для pnpm

Обобщённые результаты представлены в таблице (табл. 1).

Таблица 1 – Сравнительные характеристики менеджеров пакетов

Менеджер	Время холодной установки, с	Время горячей установки, с	Размер node_modules / кэша, МБ	Размер lock- файла, КБ	Размер образа, МБ
npm (v10)	345	10.4	154 (node_modules)	205	222
Yarn Berry (4)	540	2.1	0 (node_modules) / 65 (кэш)	159	175
pnpm (v9)	279	31.8	156 (node_modules)	122	218

Цифры холодной установки наглядно отражают архитектурные различия. pnpm завершил полную сборку за 279 секунд — на 19 % быстрее npm и почти вдвое быстрее Yarn Berry. Такой результат обусловлен эффективным механизмом разрешения зависимостей и построением плоской структуры глобального хранилища, которая исключает дублирование файлов. Даже в изолированном Docker-окружении, где глобальный стор недоступен, pnpm выполняет меньше операций ввода-вывода, чем npm, который физически копирует пакеты в древовидную структуру node_modules. npm занял 345 с, что можно считать эталонной точкой отсчёта: его алгоритм стабилен, но создаёт наибольшее количество дублированных файлов и, как следствие, самый большой lock-файл (205 КБ). Yarn Berry показал наихудшее время холодной сборки — 540 с. Причина кроется в двойной работе: после загрузки пакетов Yarn упаковывает их в zip-архивы кэша .yarn/cache, генерирует PnP-файл и проводит линковку без создания физической папки node_modules. Эти дополнительные шаги, снижающие

эксплуатационные издержки в дальнейшем, дают ощутимый проигрыш на этапе первоначальной установки.

Горячая установка выявила противоположную картину. Yarn Berry стал абсолютным лидером — всего 2.1 с. При неизменном yarn.lock и сохранённом кэше менеджер лишь проверяет целостность архивов и завершает работу, не распаковывая и не копируя файлы. Такой показатель идеален для сценариев с многократными пересборками контейнеров и частыми запусками CI/CD, где зависимости меняются редко. npm выполнил горячую установку за 10.4 с — приемлемое время, учитывая, что менеджер заново обходит дерево и восстанавливает отсутствующую папку node_modules. Неожиданно высокий результат rnpm — 31.8 с — объясняется отсутствием в контейнере предварительно прогретого глобального стора. В таких условиях rnpm вынужден воссоздавать все жёсткие ссылки с нуля, что требует больше времени, чем простое копирование у npm. В реальных же пайплайнах, где глобальное хранилище монтируется как том или кэшируется слой Docker, горячая установка rnpm обычно укладывается в 2–5 с, вплотную приближаясь к Yarn Berry.

Размеры дисковых артефактов также показательны. npm и rnpm хранят зависимости в папке node_modules схожего объёма — 154 и 156 МБ соответственно, при этом rnpm внутри делает это за счёт жёстких ссылок, не дублируя одинаковые версии пакетов, тогда как npm физически размещает каждую зависимость в иерархии каталогов, что на масштабе крупных проектов может приводить к заметной разнице. Yarn Berry полностью устранил node_modules, заменив его на кэш zip-архивов размером 65 МБ, что более чем вдвое компактнее. Это напрямую отразилось на итоговом размере образа: 175 МБ у Yarn против 222 МБ у npm и 218 МБ у rnpm. Лёгкий образ Yarn особенно ценен в микросервисной архитектуре, где каждый мегабайт влияет на скорость загрузки и масштабирования контейнеров. Компактный lock-файл rnpm (122 КБ) по сравнению с npm (205 КБ) также уменьшает размер передаваемого контекста в CI/CD.

Проведённый эксперимент подтверждает, что выбор менеджера пакетов — это компромисс между скоростью холодного старта, быстротой повторных установок и занимаемым местом. rnpm с его самой быстрой холодной сборкой и компактным lock-файлом предпочтителен для конвейеров с полными пересборками и проектов, где важен каждый мегабайт размера репозитория. Yarn Berry оправдан в средах с частыми повторными установками при стабильных зависимостях, а его нулевой node_modules упрощает аудит безопасности и избавляет от проблемы «призрачных зависимостей». npm, будучи стандартным менеджером, встроенным в Node.js, обеспечивает наивысшую совместимость и предсказуемость, оставаясь разумным выбором для проектов, где стабильность важнее экстремальной производительности. Важно также отметить ограничение данного теста: в реальных CI/CD-системах глобальное хранилище rnpm обычно кэшируется, что существенно улучшает его показатели горячей установки и делает конкурентоспособным с Yarn Berry. Дальнейшие

исследования могут быть направлены на сравнение менеджеров в условиях кэшируемого стора и в проектах с числом зависимостей на порядок больше.

4 Выводы

1. `pnpm` обеспечивает минимальное время холодной установки (279 с) и самый компактный `lock`-файл (122 КБ), что делает его предпочтительным для CI/CD-конвейеров с изолированными сборочными окружениями и частыми полными сборками.

2. `Yarn Berry` с `Plug'n'Play` лидирует по скорости горячей установки (2.1 с) и экономии дискового пространства (0 МБ `node_modules`, образ 175 МБ), но имеет наибольшее время холодной установки (540 с) и потенциальные проблемы совместимости, поэтому рекомендован проектам с высокой частотой повторных установок и готовностью к модели `PnP`.

3. `pnpm` демонстрирует сбалансированные, хотя и не рекордные, показатели (холодная 345 с, горячая 10.4 с) и максимальную совместимость; подходит для проектов, где стабильность и отсутствие сюрпризов важнее абсолютной скорости.

4. Размер `lock`-файла коррелирует с детализацией графа зависимостей: `pnpm` (205 КБ) хранит наиболее подробное описание, `pnpm` (122 КБ) — наиболее компактное, `Yarn Berry` (159 КБ) занимает промежуточную позицию.

5. Практическая стратегия выбора: для новых проектов с активным CI/CD и требованиями к минимальному размеру `lock`-файла — `pnpm`; для проектов, где нужна рекордная скорость повторных установок и допустима адаптация под `PnP`, — `Yarn Berry`; для `legacy`-проектов и максимальной совместимости — `pnpm`.

Библиографический список

1. Алуев А. С. Решение вопросов безопасности в приложениях `node.js`: экономические последствия повышения безопасности //Международный журнал гуманитарных и естественных наук. 2024. №. 9-1 (96). С. 94-98. URL: <https://cyberleninka.ru/article/n/reshenie-voprosov-bezopasnosti-v-prilozheniyah-node-js-ekonomicheskie-posledstviya-povysheniya-bezopasnosti> (дата обращения: 24.05.2026)
2. Галатенко В. А., Дзабраев М. Д., Костюхин К. А. Выбор пакетного менеджера для многоверсионных приложений //Программные продукты и системы. 2018. Т. 31. №. 3. С. 469-474. URL: <https://cyberleninka.ru/article/n/vybor-paketnogo-menedzhera-dlya-mnogoversionnyh-prilozheniy> (дата обращения: 24.05.2026)
3. Горячкин Б. С. и др. Эффективность фронтенд-разработки на основе анализа сборщиков //Инженерный вестник Дона. 2025. №. 5 (125). С. 42-58. URL: <https://cyberleninka.ru/article/n/effektivnost-frontend-razrabotki-na-osnove-analiza-sborschikov> (дата обращения: 24.05.2026)
4. Литвяков Д. С. Разработка конфигурации CI/CD для автоматизации

- развёртывания и управления приложениями //StudNet. 2021. Т. 4. №. 6. С. 1089-1097. URL: <https://cyberleninka.ru/article/n/razrabotka-konfiguratsii-ci-cd-dlya-avtomatizatsii-razvyortyvaniya-i-upravleniya-prilozheniyami> (дата обращения: 24.05.2026)
5. Данилин Д. А., Зиновьев Я. В., Кузьмин К. М. Анализ технологий веб-программирования для создания модулей визуализации и выгрузки данных информационных систем //Вестник Пензенского государственного университета. 2019. №. 3 (27). С. 76-79. URL: <https://cyberleninka.ru/article/n/analiz-tehnologiy-veb-programmirovaniya-dlya-sozdaniya-moduley-vizualizatsii-i-vygruzki-dannyh-informatsionnyh-sistem> (дата обращения: 24.05.2026)