Санкт-Петербургский государственный университет

Программная инженерия

Группа 24.М71-мм

Эволюция PHP-фреймворков как отражение развития экосистемы веб-разработки (2005–2025)

Альшаеб Басель

Отчёт по учебной практике

Преподаватель:
Басков Антон Андреевич

Санкт-Петербург
2025

Оглавление

1. Введение

PHP играет важную роль в веб-разработке на протяжении почти трёх десятилетий, оставаясь одним из наиболее распространённых серверных языков.

Согласно статистике W3Techs по состоянию на 1 ноября 2025 года, PHP используется в 72.9% веб-сайтов с определённым языком серверной части [?].

Широкая доступность хостинга и относительно низкий порог входа сделали PHP [?] основой для множества систем управления контентом и прикладных веб-решений.

Однако, устойчивость языка и его экосистемы на протяжении длительного периода в значительной степени связана с развитием общих архитектурных практик и стандартов, которые определили индустрию PHP [?]-разработки.

В начале 2000-х годов в ответ на растущую сложность веб-приложений начали формироваться первые полнофункциональные PHP-фреймворки.

CakePHP (2005) [?], Symfony (2005) [?] и CodeIgniter (2006) [?] заложили основы MVC-подхода и упорядочили разработку веб-приложений в условиях отсутствия единых стандартов.

Однако по мере усложнения проектов стало очевидно, что решения того периода сталкиваются с рядом системных ограничений, включая непрозрачную архитектуру, несовместимость компонентов, отсутствие унифицированных интерфейсов и недостаточную тестируемость.

В течение следующего десятилетия (2005–2015) PHP [?]-стек претерпел значительные изменения. Это было связано с разработкой PHP-FIG [?] и ряда стандартов PSR [?], появлением Composer [?], унификацией HTTP-модели [?], распространением принципов DI [?] и архитектурой middleware [?].

Тем не менее, между 2015 и 2025 годами произошли наиболее значительные структурные изменения в PHP-фреймворках. Эти изменения радикально изменили архитектурные решения, принципы проектирова-

ния и практики масштабирования.

Эти преобразования охватывали выход PHP 7 [?], внедрение массовой типизации и переход к строгой модели обработки ошибок, реорганизацию Symfony [?] в рамках компонентной архитектуры, стремительный развитие Laravel [?], а также упадок Zend [?] Framework и его последующую трансформацию в проект Laminas.

Цель настоящей работы — провести комплексный анализ ключевых архитектурных и технологических изменений, произошедших в PHP-фреймворках в период с 2015 по 2025 годы. В рамках исследования предполагается выявить причины указанных изменений, оценить эффективность принятых техническиатурных решений, а также определить, какие концепции и подходы были отвергнуты или модифицированы в ходе профессиональных дискуссий.

Для достижения поставленных целей используются следующие источники: материалы рассылок PHP-FIG [?], предложения RFC Internals PHP, официальные публикации Symfony [?] и Laravel [?], дискуссии GitHub о стандартах Composer и PSR, а также доклады разработчиков, представленные на профильных конференциях — в частности, SymfonyCon [?] и Laracon [?].

Таким образом, в статье рассматривается не только эволюция отдельных фреймворков, но и более широкий процесс формирования профессиональной PHP-экосистемы, в которой технологические инновации выступают в качестве ключевого фактора развития.

2. Особенности развития PHP-экосистемы до стандартизации (до 2015 года)

К 2015 году промышленная разработка на PHP [?] характеризовалась двойственным положением. С одной стороны, язык сохранял широкое распространение и технологическую зрелость как платформа для веб-приложений, обусловленную простотой развёртывания, высокой доступностью хостинговых решений и устойчивой обратной совместимостью. С другой стороны, PHP-экосистема страдала от существенных архитектурных ограничений, вызванных отсутствием унифицированных стандартов проектирования и низкой совместимостью между ключевыми компонентами. Это приводило к увеличению совокупной стоимости сопровождения проектов, а также замедляло темпы развития как сторонних библиотек, так и фреймворков.

2.1. Отсутствие единых интерфейсов

До появления PHP-FIG [?] в экосистеме PHP отсутствовали общепринятые технические соглашения по ключевым аспектам разработки, включая:

- организацию структуры каталогов;
- механизмы автоматической загрузки классов;
- унифицированное представление HTTP-запросов и ответов;
- интерфейсы контейнеров внедрения зависимостей (DI);
- стандартные контракты для middleware и модель обработки HTTP-запросов.

В результате каждый из ведущих фреймворков, включая Symfony [?], Zend [?], CakePHP [?] и CodeIgniter [?], реализовывал собственные, зачастую несовместимые подходы к решению этих задач. Это привело к эффекту фрагментации экосистемы PHP, при котором фреймворки

и библиотеки развивались как изолированные подсистемы с ограниченной совместимостью.

Например, Symfony [?] применял собственный автозагрузчик и строгие соглашения об именовании, тогда как Zend предлагал иную структуру каталогов. Многие сторонние библиотеки того времени подключались вручную и не содержали формализованных метаданных о зависимостях, что исключало автоматизированное управление ими.

Такая ситуация приводила к высокой степени связанности между фреймворками и их компонентами. Интеграционные решения часто основывались на неформализованных соглашениях и паттернах, а изменения в деталях реализации нарушали обратную совместимость.

2.2. Проблема совместимости и отсутствие компонентов повторного использования

Ключевая проблема данного периода заключалась в невозможности использовать одну и ту же библиотеку в разных фреймворках. Такие распространённые компоненты, как HTTP-клиенты, инструменты логгирования и шаблонизаторы, были тесно интегрированы в архитектуру конкретных фреймворков и зависели от их внутренних соглашений, стилей кодирования и точек расширения. Попытки переноса подобных решений между проектами зачастую приводили к конфликтам, обусловленным различиями в API, структуре кода и механизмах интеграции.

Symfony [?] стал одним из первых фреймворков, предпринявших системную попытку преодолеть эту фрагментацию за счёт внедрения компонентного подхода [?] (начиная с Symfony 2 [?]), однако в отсутствие общепринятых отраслевых стандартов такой подход оставался локальным решением, которое не обеспечивало гарантированной совместимости и стабильности интеграций.

2.3. Отсутствие стандартизированной модели HTTP

Одним из ключевых технологических ограничений PHP-экосистемы до появления общих стандартов являлось отсутствие унифицированного представления основных HTTP-компонентов: запроса, ответа, а также их атрибутов, заголовков и потоков данных.

Каждый из ведущих фреймворков реализовывал собственные, несовместимые между собой классы для работы с HTTP-слоем:

- HttpFoundation [?] в Symfony;
- Http [?] (Request / Response) в Zend;
- Http [?] в Slim;
- Собственные реализации в CakePHP [?].

Такая фрагментация затрудняла разработку взаимозаменяемых middleware-компонентов, снижала переносимость кода между проектами и препятствовала созданию единых решений для HTTP-клиентов, фильтров и других инструментов обработки запросов.

Глубина проблемы подтверждается продолжительностью обсуждения стандарта PSR-7 [?] в рабочих группах PHP-FIG: дискуссии по его формулировке велись почти три года. Это свидетельствует о фундаментальном характере задачи стандартизации HTTP-интерфейсов для всей экосистемы.

2.4. Отсутствие единых правил для автозагрузки и управления зависимостями

До появления Composer [?] и принятия стандарта PSR-4 [?] (2013–2014 гг.) подключение сторонних библиотек в PHP-проектах, как правило, осуществлялось вручную с использованием конструкций

`require/include`. Декларативное управление зависимостями фактически отсутствовало, а распространение кода происходило преимущественно через архивы в формате ZIP или посредством системы PEAR.

Отсутствие унифицированных правил автозагрузки классов обусловливало жёсткую привязку к структуре каталогов и способствовало частым конфликтам версий библиотек.

Ситуация принципиально изменилась с появлением Composer [**?**], который ввёл централизованный и декларативный механизм управления зависимостями и автозагрузкой. Однако его широкое внедрение в промышленную разработку началось лишь в 2014–2015 гг. Этот период стал отправной точкой для изменений, анализируемых в последующих разделах.

2.5. Проблемы тестируемости и внедрения зависимостей

До принятия стандарта PSR-11 [**?**] (Container Interface) каждый из ведущих PHP-фреймворков реализовывал собственный контейнер внедрения зависимостей (DI-контейнер), причём эти реализации не были совместимы между собой. К числу наиболее распространённых решений относились:

- Symfony Dependency Injection [**?**];
- Laravel Container [**?**];
- Zend Service Manager [**?**];
- Pimple [**?**].

Отсутствие унифицированного интерфейса контейнера создавало ряд системных ограничений:

- затрудняло разработку многократно используемых пакетов, не привязанных к конкретному фреймворку;

- делало невозможным перенос middleware-компонентов и других сервисов между проектами;
- замедляло развитие архитектур, основанных на принципах внедрения зависимостей и их управления.

Таким образом, архитектура внедрения зависимостей представляла собой одну из ключевых проблем экосистемы PHP, решение которой стало возможным лишь в последние 10 лет (2015–2025) в ходе стандартизации — в частности, с принятием PSR-11 [**?**].

3. Развитие стандартов и архитектурных решений в экосистеме PHP (2015–2025)

Период 2015–2025 годов стал для PHP-фреймворков временем глубокой технологической перестройки.

На протяжении предыдущего десятилетия (2005–2015) были заложены фундаментальные идеи: создание PHP-FIG [?], появление Composer [?], формирование первых PSR-стандартов [?], PSR-4 [?], PSR-7 [?], PSR-11 [?], PSR-15 [?].

Однако именно после выхода PHP 7 [?] и широкого принятия PSR-7 [?], PSR-11 [?], PSR-15 [?] начался качественный переход к современной архитектуре веб-приложений.

В этом разделе анализируются ключевые архитектурные изменения, причины их появления и влияние на экосистему.

3.1. Composer как основа модернизации экосистемы (2015–2025)

Хотя Composer [?] был выпущен в 2012 году, его массовое принятие произошло в период 2015–2017 годов: именно тогда большинство фреймворков и библиотек окончательно перешли на декларативное управление зависимостями.

До Composer управление зависимостями оставалось во многом ручным: библиотеки подключались через `require/include`, распространялись ZIP-архивами или через PEAR, а конфликты версий приходилось разрешать неформальными способами.

Это создавало типичную для крупных проектов проблему несовместимых зависимостей и существенно тормозило развитие фреймворков.

Composer предложил единый механизм описания зависимостей через composer.json [?], стандартный автозагрузчик на основе PSR-4 [?], практики семантического версионирования [?] и центральный репозиторий пакетов Packagist [?].

В результате фреймворки стали восприниматься не как монолитные

системы, а как композиции из независимых компонентов, которые можно выбирать, комбинировать и обновлять по отдельности.

К 2020-м годам Composer фактически стал инфраструктурным стандартом экосистемы PHP: снизилась привязка библиотек к конкретным фреймворкам, ускорился обмен компонентами между проектами и сформировалась массовая культура разработки пакетов, независимых от выбранного фреймворка.

3.1.1. PSR-4 и единая модель автозагрузки

PSR-4 [**?**] стал ключевым шагом от разрозненных соглашений к единой, предсказуемой модели автозагрузки.

Практически важным он стал в период 2015–2020 годов, когда фреймворки начали массово приводить внутренние структуры и пространства имён к PSR-4, а Composer получил устойчивую основу для генерации стандартизированного автозагрузчика.

До внедрения PSR-4 каждый крупный фреймворк имел собственные правила размещения классов и сопоставления имён с файлами, что усложняло интеграцию библиотек и повышало стоимость поддержки проектов при росте кодовой базы.

PSR-4 закрепил следующие ключевые требования:

- правила сопоставления пространств имён с каталогами;
- возможность автозагрузки классов без ручных `include`/`require`.

В результате библиотеки стали заметно более переносимыми, а сами фреймворки — более «компонентными»: уменьшилась доля собственного инфраструктурного кода и выросла совместимость экосистемы в целом.

3.2. PSR-7 и стандартизация HTTP-модели (2015)

Стандарт PSR-7 [**?**] (HTTP Message Interface) стал одним из наиболее значимых изменений за всё время развития экосистемы PHP-фреймворков.

До его появления каждый фреймворк представлял HTTP-запросы и ответы по-своему, что осложняло перенос middleware и делало практически невозможным создание взаимозаменяемых HTTP-компонентов (клиентов, роутеров, фильтров) между разными стеками.

Также отсутствовали единые интерфейсы для потоков, заголовков и других элементов HTTP-сообщений.

PSR-7 ввёл унифицированную модель HTTP-сообщений и набор интерфейсов для основных сущностей:

- `Request`;
- `Response`;
- `Stream`;
- `UploadedFile`;
- `URI`.

Практическим следствием стандарта стало развитие переносимых библиотек и middleware-слоя.

В экосистеме закрепились PSR-7-совместимые реализации (например Slim 3 [**?**] и Zend Diactoros [**?**]), а крупные фреймворки начали предоставлять адаптеры и мосты: Symfony добавил PSR-7-бриджи [**?**], а Laravel поддержал совместимость на уровне интеграций [**?**].

PSR-7 также подготовил почву для дальнейшей стандартизации middleware-контрактов в PSR-15 [**?**].

3.3. PSR-11 и унификация DI-контейнеров (2017)

Ещё одной ключевой проблемой PHP-фреймворков была несовместимость контейнеров зависимостей: Laravel, Symfony и Zend имели разные API, из-за чего библиотеки не могли запрашивать зависимости абстрактно и переносимо.

PSR-11 [**?**] ввёл минимальный общий контракт контейнера — два интерфейса:

- `ContainerInterface`;
- `NotFoundExceptionInterface`.

Это позволило библиотекам и компонентам требовать доступ к контейнеру зависимостей без привязки к конкретному фреймворку.

В результате DI стал общеэкосистемным механизмом (а не «внутренней особенностью» отдельных фреймворков), снизилась фрагментация пакетов и упростилось создание переносимых компонентов, которым достаточно соблюдения PSR-11.

3.4. PSR-15 и middleware-архитектура (2017–2018)

После появления PSR-7 [**?**] стало возможным стандартизировать не только представление HTTP-сообщений, но и саму модель обработки запросов через цепочки middleware.

До PSR-15 [**?**] разные фреймворки использовали несовместимые подходы (например, HttpKernel в Symfony и собственные middleware/фильтры в Laravel), а зрелая middleware-архитектура была характерна лишь для части стеков (Slim [**?**], Zend Expressive/Mezzio [**?**]).

PSR-15 [**?**] закрепил два базовых контракта:

- `MiddlewareInterface`;
- `RequestHandlerInterface`.

Это приблизило PHP к распространённой в других экосистемах модели middleware-цепочек (Rack в Ruby, WSGI в Python, Connect в Node.js).

Практически стандарт ускорил развитие переносимых `middleware` и позволил middleware-ориентированным фреймворкам (например Mezzio [**?**] и Slim 4 [**?**]) опираться на единый интерфейс.

Symfony и Laravel сохранили свои внутренние модели, но получили возможность совместимости через адаптеры и мосты [**?**, **?**].

В итоге middleware стала центральной архитектурной единицей для многих PHP-приложений.

3.5. Влияние PHP 7 и PHP 8 на архитектуру фреймворков

3.5.1. PHP 7 (2015)

- Существенный прирост производительности по сравнению с PHP 5.x.
- Строгая модель ошибок (движение от предупреждений к исключениям).
- Scalar type hints, return types.

3.5.2. PHP 8 (2020)

- Union types.
- Attributes.
- Match.
- JIT.
- Улучшенная типобезопасность.

3.5.3. Результаты для фреймворков

- Symfony 3/4 [**?**] переписали DI-контейнер под строгую типизацию.
- Laravel [**?**] стал массово переходить к типизированным сигнатурам.
- Появилась культура строгих DTO, Value Objects, Immutable объектов.
- Фреймворки сократили магию и усилили контрактность API.

Типизация стала центральным архитектурным трендом 2020-х годов.

3.6. Symfony: эволюция компонентной модели

Symfony стал главным драйвером стандартизации.

3.6.1. Основные изменения

- Переход от «полного фреймворка» к компонентам [?] (HttpFoundation, EventDispatcher, Console, Routing).
- Адаптация архитектуры под PSR-7/PSR-11 [?, ?].
- Внедрение autowiring и автоконфигурации.
- Появление Symfony Flex как современного пакета приложений.

3.6.2. Результаты

Компонентная модель Symfony привела к тому, что многие проекты начали использовать отдельные компоненты Symfony как «строительные блоки» независимо от полного фреймворка. Это усилило тенденцию к стандартной инфраструктуре вне ядра языка: такие компоненты, как HttpKernel и EventDispatcher, стали де-факто архитектурными ориентирами для построения расширяемых приложений и библиотек, включая экосистему Laravel [?].

3.7. Laravel: эволюция DX и влияние стандартов

Laravel [?] (2011) в 2015–2025 годах закрепился как фреймворк, ориентированный на удобство разработки (DX-first), единообразие API и практики convention over configuration, что сделало его особенно привлекательным для массовой разработки прикладных веб-систем.

В период 2015–2025:

3.7.1. Основные изменения

- Переход на PSR-4, Composer и частично PSR-11.
- Внедрение middleware-модели совместимой с PSR-15.

- Адаптация к PHP 7/8 (типизация, атрибуты).
- Появление Horizon, Octane, Sail, Pint — инфраструктурных компонентов.
- Более строгая структура маршрутизации и DI.

3.7.2. Результаты

Laravel [**?**] стал «массовым воплощением» стандартизированной экосистемы PHP:

- Он интегрирует PSR-совместимые библиотеки.
- Служит входной точкой для новых разработчиков.
- Влияет на индустриальные практики (DX, миграции, Eloquent как ORM-эталон).

4. Отвергнутые и отложенные изменения

Как и в случае с другими зрелыми технологиями, процесс стандартизации PHP-фреймворков сопровождался многочисленными предложениями, не вошедшими в финальные версии PSR-стандартов или отложенными на неопределённый срок.

Эти обсуждения позволяют понять, какие архитектурные решения были сочтены слишком узкими, слишком рискованными или концептуально несовместимыми с направлением развития экосистемы.

Ниже рассмотрены наиболее значимые инициативы, не приведшие к формированию стандарта.

4.1. Несостоявшийся PSR для маршрутизации (Router PSR)

Одним из регулярно поднимаемых предложений в рассылках PHP-FIG [?] начиная с 2015 года было создание стандарта для роутинга HTTP-запросов.

4.1.1. Причины появления инициативы

Фреймворки использовали разные архитектуры маршрутов:

- Laravel применяет декларативную синтаксическую модель (fluent API) [?].
- Symfony использует аннотации, YAML и PHP-конфигурации [?].
- Slim и Mezzio строят маршрутизацию вокруг middleware [?].
- FastRoute — чисто функциональная библиотека без привязки к фреймворку [?].

Отсутствие общего интерфейса приводило к невозможности создать:

- Единый набор middleware для маршрутизации.

- Универсальные инструменты тестирования маршрутов.
- Переносимые роутинговые DSL.

4.1.2. Причины отказа

В обсуждениях FIG (2016–2018) было выявлено несколько проблем:

Различие моделей маршрутизации

Одни фреймворки используют controller-based архитектуру (Laravel [**?**], Symfony [**?**]), другие (например Slim [**?**], Mezzio) используют middleware-based архитектуру. Приведение этих моделей к единому интерфейсу оказалось практически невозможным.

Слишком высокий уровень абстракции

Любой интерфейс становился либо:

- Слишком низкоуровневым (и не решал задачи).
- Или слишком высоким.

Стандарт рисковал закрепить устаревший подход

FIG избегает «Навязывать» архитектурные решения, чтобы не мешать инновациям.

Итог

Инициатива была закрыта как слишком сложная и недостаточно универсальная. Разработчики договорились, что роутинг останется частью каждого фреймворка, а интеграции будут строиться через PSR-7 [**?**] и PSR-15 [**?**].

4.2. Попытка создать PSR для ORM и абстракции работы с базами данных

Регулярно обсуждалось создание стандарта для доступа к данным аналога JDBC для Java. Попытки разработать единый интерфейс для ORM поднимались в рассылке PHP-FIG с 2014 по 2020 год [**?**].

Мотивация

- Множество несовместимых ORM: Doctrine ORM [**?**], Eloquent ORM [**?**], Propel [**?**], RedBeanPHP [**?**].
- Попытки создать стандартный QueryBuilder или EntityManager обсуждались с 2014 по 2020 год.

Причины отказа

- Слишком разные философии данных.
  - Doctrine — ориентирована на DDD и Unit of Work.
  - Eloquent — ActiveRecord и stateful-модель.
  - Propel — XML-генерация моделей.
  - RedBean — динамические схемы.
- Разработчики ORM не готовы к унификации.
  Doctrine имеет строгую архитектуру, Laravel — гибкую, Eloquent использует «магические» свойства.
- Большая часть индустрии предпочитает свободу.
  FIG не хотел повторить опыт Java EE, где стандарты замедляли эволюцию ORM.

Итог

PSR для ORM был признан нежизнеспособным.

4.3. PSR-14 (Event Dispatcher): стандарт, который останется частичным

PSR-14 [?] был принят в 2019 году, но его разработка сопровождалась огромным количеством противоречий в рассылках PHP-FIG [?].

Проблемы

- Фреймворки используют разные event models.
  - Symfony: синхронный диспетчер событий, на основе объектных слушателей [?].
  - Laravel: разделение событий и слушателей + очередь + broadcast [?].
  - Zend Framework: агрегаторы событий [?].

  Найти общую модель оказалось крайне трудно.

- Глубокие различия в семантике слушателей.
  Например, прекращение обработки событий отсутствует во многих системах.

- Слишком узкий охват стандарта.
  PSR-14 определяет слишком абстрактный интерфейс:

```
interface EventDispatcherInterface {
    public function dispatch(object
```

Итог

PSR-14 принят, но в экосистеме остаётся «вторичным» стандартом. Существенная часть сообществ его игнорирует.

4.4. Попытка создания PSR для валидаторов и форм

Это обсуждение велось эпизодически с 2016 по 2021 годы [?].

Мотивация

- Symfony Form + Validator имеют сложную, но зрелую модель [**?**, **?**].
- Laravel Validation — декларативная модель со строковыми правилами [**?**].
- Respect/Validation — функциональная библиотека [**?**].

Наличие множества несовместимых подходов порождало предложение создать переносимый стандарт.

Причины отказа

- Слишком разный уровень абстракции.
- Слишком разный DSL.
  Сравните:
  - `'required|min:6'` (Laravel),
  - `new Length(['min'=>6])` (Symfony),
  - `v::stringType()->length(6)` (Respect).
- Нежелание ограничивать инновации.
  Фреймворки активно экспериментируют со схемами данных.

Итог

PSR для валидаторов и форм был признан нежизнеспособным.

4.5. Споры вокруг PSR-18 (HTTP Client)

PSR-18 [**?**] был принят в 2019 году, но с серьёзными дискуссиями [**?**].

Причины споров

- Разные модели ошибок (исключения vs error responses).
- Различия в реализации Guzzle [?], HTTPlug [?], Symfony HttpClient [?].
- Споры о синхронности vs асинхронности.

Некоторые разработчики хотели:

- Асинхронный интерфейс (в стиле Promise).
- Unified Streaming API.
- Поддержку cancellation.

FIG решил ограничиться минимальным синхронным интерфейсом, что вызвало критику, но позволило стандартизировать общие ожидания.

Итог

Стандарт был принят как «минимально необходимый», а остальные аспекты намеренно не стандартизированы.

4.6. Предложение о расширении PSR-11 (унифицированная конфигурация контейнера)

Иногда обсуждался стандарт для [?]:

- Регистрации сервисов.
- Определения параметров.
- Описания factories.

Это сделало бы DI-контейнеры полностью совместимыми.

Причины отказа

- Разные модели конфигурации контейнера.
  - Symfony использует YAML/PHP/XML [**?**].
  - Laravel использует bindings и closures [**?**].
  - Laminas: Массивы-конфигурации [**?**].

  Невозможно привести к общему знаменателю.

- Опасение закрепления устаревших подходов Стандартизация могла заморозить развитие DI.

Итог

PSR-11 [**?**] остался минималистичным.

5. Заключение

За период 2015–2025 годов экосистема PHP-фреймворков претерпела глубокую трансформацию, в результате которой веб-разработка на PHP фактически перешла от фрагментированного набора несовместимых архитектур к единому технологическому пространству, основанному на стандартах и переносимых компонентах.

Ключевую роль в этом процессе сыграли стандарты PHP-FIG (PSR-4, PSR-7, PSR-11, PSR-15), Composer и переход языка к строгой модели типизации в версиях PHP 7 и 8.

Эти изменения оказали системное влияние как на внутренние архитектуры фреймворков, так и на методологию разработки приложений.

Принятие PSR-4 и Composer стало отправной точкой для унификации структуры проектов, что позволило разрушить жёсткие границы между экосистемами разных фреймворков.

Стандартизация HTTP-модели (PSR-7) и middleware-контрактов (PSR-15) сформировала общий слой interoperability, дав толчок развитию кросс-фреймворковых библиотек и middleware-стека.

PSR-11 обеспечил совместимость контейнеров зависимостей, благодаря чему переносимость сервисов и компонентов значительно увеличилась.

Совокупно эти стандарты сформировали основу современной PHP-инфраструктуры, в которой логика приложения теперь отделена от конкретного фреймворка.

Наряду со стандартами FIG значительное влияние оказали изменения в самом языке: строгая типизация, исключительная модель ошибок, увеличение производительности PHP 7, а также атрибуты и JIT-компиляция в PHP 8.

Эти изменения стимулировали фреймворки к переработке внутренних механизмов и улучшению архитектурных практик.

Symfony в этот период окончательно утвердился как компонентный фреймворк, определяющий технические ориентиры для всей PHP-индустрии; Laravel, напротив, стал укреплять свою позицию как высо-

коуровневый фреймворк, фокусирующийся на удобстве разработки и интеграции с современными DevOps-практиками.

Современные версии обоих фреймворков демонстрируют высокую степень согласованности с PSR-стандартами и активно используют возможности нового PHP.

Особое значение имеют технологии и стандарты, которые не были приняты.

Несостоявшийся PSR для роутинга, отсутствие стандарта для ORM, частичная применимость PSR-14 — всё это демонстрирует, что стандартизация не может и не должна охватывать все аспекты PHP-экосистемы.

Слишком разнообразные архитектуры и различные философии разработки делают некоторые стандарты непрактичными.

В этом смысле отказ от стандартизации отдельных областей оказался не менее ценным, чем успешные инициативы: он позволил индустрии сохранить гибкость и конкурентное разнообразие.

В целом, архитектурная эволюция PHP-фреймворков в 2015–2025 годах может быть охарактеризована как движение к стандартизации на ключевых уровнях абстракции при сохранении свободы в реализации высокоуровневых концепций.

Этот период стал временем консолидации и зрелости: фреймворки перестали быть самостоятельными островами и стали частями единой экосистемы, в которой интерфейсы важнее реализаций, а архитектура — важнее конкретных технологий.

PHP, часто считавшийся устаревающим, за это десятилетие подтвердил свою жизнеспособность, адаптировавшись к современным требованиям производительности, типобезопасности и модульности.

Эта трансформация стала возможной благодаря открытому процессу разработки, публичным обсуждениям в PHP-FIG и активной роли сообществ Symfony и Laravel.

Таким образом, изменения 2015–2025 годов можно считать одним из самых успешных этапов в истории PHP: язык и его фреймворки не только сохранили позиции, но и стали примером того, как открытая

стандартизация и согласованные архитектурные решения способны преобразовать зрелую технологию в соответствующую требованиям нового времени.