The American Journal of Engineering and Technology
ISSN 2689-0984 | Open Access

TYPE Original Research
PAGE NO. 56-65
DOI 10.37547/tajet/Volume07Issue06-06

Check for updates

PHP: Methodology for Configuring Third-Party Composer Packages

Oleg Ekhlakov
Web Developer, Intaro Soft LLC Russia, Lipetsk

OPEN ACCESS

SUBMITED 24 April 2025
ACCEPTED 29 May 2025
PUBLISHED 10 June 2025
VOLUME Vol.07 Issue 06 2025

CITATION

Oleg Ekhlakov. (2025). PHP: Methodology for Configuring Third-Party Composer Packages. The American Journal of Engineering and Technology, 7(06), 56–65. https://doi.org/10.37547/tajet/Volume07Issue06-06.

Abstract: This article presents a methodology for customizing third-party packages in PHP projects using Composer. Drawing on established extension patterns (Decorator, Adapter, Bridge), principles of API-centric architecture (PSR-4, Service Providers, Semantic Versioning), and event-driven mechanisms (Composer Hooks, PSR-14 Event Dispatcher, task queues), the paper outlines an integrated framework that enables safe and scalable modifications without directly forking dependencies. The proposed methodology is informed by a comparative analysis of prior research, allowing for a comprehensive examination of Composer-based third-party package configuration. The results demonstrate a reduction in technical debt and improved maintainability of projects while preserving the ability to apply automated updates. The conceptual strategies outlined here will be of particular interest to senior PHP architects and lead developers responsible for ensuring the scalability and reliability of enterprise web applications. Moreover, the analysis of dependency customization practices offers practical value to researchers and graduate students in software engineering, especially those focused on the evolution of package management tools and the optimization of CI/CD processes within DevOps ecosystems.

Keywords: Composer, package customization, Decorator, Adapter, Bridge, PSR-4, Service Provider, Semantic Versioning, Composer Hooks, Event Dispatcher, methodology.

Introduction: In recent years, the PHP ecosystem has undergone significant expansion. As a result, business applications frequently require customization of third-party packages without forking them, in order to preserve support for automated updates and maintainability. However, common approaches—such as directly modifying source code or creating forks—

introduce risks of divergence from upstream versions and increase project maintenance complexity [1].

The body of literature addressing Composer-based configuration of third-party PHP packages can be broadly categorized into four thematic groups.

The first group focuses on SaaS configuration management and the integration of CPQ (Configure-Price-Quote) systems in B2B contexts. Joshi H. [1] explores enterprise design patterns for CPQ, emphasizing modular architecture to support adaptability and extensibility. Li B. and Kumar S. [3] examine economic-operational models of SaaS management, with an emphasis on configuration flexibility and scalability. Dutta S. K. [4] outlines best practices for implementing the Salesforce Enablement Playbook, where configurable packages serve as the backbone of business logic. Pathak P. et al. [5] analyze sales performance improvements driven by CPQ-CRM integration, highlighting the role of automated component configuration.

The second group brings together research on cloud technology adoption and digital transformation in broader societal contexts. Islam M. N. [2] proposes an architecture for an education-focused CMS with integrated cloud services, where Composer dependencies are used to modularly connect content delivery and authentication features. Kaputa V., Loučanová E., and Tejerina-Gaite F. A. [7] discuss digital transformation as a driver of socially oriented innovation, referencing Composer as a tool for standardizing and unifying integrated libraries.

The third group consists of studies in business process reengineering and comparative analysis of project management tools. Baul S. et al. [6] provide a survey of open-source and SaaS solutions for project management, presenting a custom-built tool based on process analysis findings.

The fourth group directly addresses PHP frameworks, third-party integrations, and application performance optimization. Selvaraj S. [8] details advanced integration techniques for external services in Laravel applications, including autoload configuration and management of publishable resources via Composer. Engebreth G. and Sahu S. K. [9] explore PHP 8 framework logic, organizing standard configuration and extension practices through package-based mechanisms. Jahanshahi R. et al. [10] introduce *Minimalist*, a tool for semi-automated "de-bloating" of PHP applications, which statically analyzes Composer dependencies to eliminate unused components.

Despite the breadth of approaches, the literature reveals several contradictions and gaps. First, there is a split regarding the degree of automation: some authors [5,10] advocate for maximum automation of configuration, while others [1,2] emphasize the importance of manual parameter tuning. Second, economic modeling methodologies [3] are rarely aligned with practical guidelines on dependency security and management [10]. Moreover, integration of Composer configuration with CI/CD pipelines, strategies for maintaining backward compatibility during package updates, and vulnerability assessment methods for third-party dependencies are poorly addressed.

Therefore, a comprehensive Composer customization methodology requires further study that integrates economic, organizational, and technical perspectives, along with deeper development of automation and security tooling.

The goal of this article is to analyze a methodology for configuring third-party packages in PHP projects using Composer.

The scientific contribution lies in the theoretical substantiation of a comprehensive methodology for customizing Composer packages through the hybrid use of extension patterns (Decorator, Adapter, Bridge), API-centric integration (PSR-4, Service Providers, Semantic Versioning), and event-driven mechanisms (Composer Hooks, PSR-14). This approach, grounded in comparative analysis of existing research, demonstrates the potential to reduce technical debt and improve maintainability without forking dependencies.

The working hypothesis is that hybrid use of object-oriented extension patterns, together with PSR-4-based API integration and event-driven Composer scripting, enables more sustainable and maintainable customization of third-party packages than traditional forking methods.

This study is based on a review of research reporting implementation experiences and includes comparative analysis across dimensions such as maintainability complexity, update compatibility, and test coverage.

1. Extension Patterns for Composer Packages (Decorator, Adapter, Bridge)

The Decorator pattern enables dynamic wrapping of an object to add new responsibilities without modifying the original class. This approach mirrors the microservice-based decomposition of CPQ systems into independent components [1], where each service is responsible for a specific function and can be extended by others without altering the core. In PHP projects managed via Composer, the Decorator pattern is implemented using PSR-4 autoloading and a dedicated namespace for decorators. An illustrative example is provided below:

```
namespace MyApp\Decorators;

use ThirdParty\ClientInterface as BaseClient;
use MyApp\Contracts\ClientInterface;

class LoggingDecorator implements ClientInterface
{
   private BaseClient $client;

   public function __construct(BaseClient $client)
   {
      $this->client = $client;
   }

   public function request(array $payload): array
   {
      // Log the request
      error_log('Request: ' . json_encode($payload));
      $response = $this->client->request($payload);
      // Log the response
      error_log('Response: ' . json_encode($response));
      return $response;
   }
      }
```

The Adapter pattern addresses interface incompatibility between client code and third-party libraries. Acting as a translation layer, it maps one interface to another—functionally similar to an API gateway in a CPQ system's API-centric architecture [1]. In Composer-based packages, the Adapter is typically introduced via dependency injection containers and service configuration (e.g., in Symfony or Laravel). A sample implementation is shown below:

```
namespace MyApp\Adapters;

use ThirdParty\PaymentGateway;
use MyApp\Contracts\PaymentInterface;

class PaymentGatewayAdapter implements PaymentInterface
```

```
{
   private PaymentGateway $gateway;

   public function __construct(PaymentGateway $gateway)
   {
      $this->gateway = $gateway;
   }

   public function charge(float $amount, string $currency): bool
   {
      // Translate the call to the application's interface
      return $this->gateway->processPayment([
         'sum' => $amount,
         'ccy' => $currency,
      ]);
   }
}
```

The Bridge pattern decouples abstraction from its implementation, allowing them to evolve independently. This is closely aligned with Event-Driven Architecture, in which event producers and consumers communicate loosely via brokers [2,3]. In Composer-based environments, the Bridge can be used to substitute service implementations without modifying abstraction logic—e.g., through configuration in composer.json or within a DI container. An example is illustrated below:

```
{
  "extra": {
    "bridge": {
      "MyApp\\Contracts\\StorageInterface": "MyApp\\Adapters\\S3Storage"
    }
  }
}
```

Each bridge-adapter implements a shared interface but can rely on any underlying technology.

Table 1 – Comparison of Composer Package Extension Patterns [1–3; 7]

Pattern	**Primary Purpose**	**Example Use Case**	**Advantages**	**Limitations**
Decorator	Dynamically add behavior	Logging API calls through a client wrapper	•Separation of concerns •Scalable design	•Can increase object hierarchy complexity
Adapter	Reconcile incompatible interfaces	Integrating a third-party payment gateway	•Minimal code changes •Improved readability	•Adds an extra translation layer

Pattern	Primary Purpose	Example Use Case	Advantages	Limitations
Bridge	Decouple abstraction and implementation	Switching storage mechanisms (file/S3/DB)	• Swap implementations without forking• Loose coupling	•Additional abstraction may reduce clarity

In summary, the use of these patterns ensures modularity, flexibility, and maintainability when customizing Composer packages—while preserving compatibility with upstream updates and avoiding the need for direct forks.

2. API-Centric Integration and Configuration

In an API-centric architecture, the primary focus is on modeling functional components as a collection of clearly defined interfaces, which facilitates reuse, testing, and maintainability [1]. Within the context of Composer-managed PHP projects, the key mechanisms of an API-centric approach include PSR-4 autoloading, service registration via Service Providers, and carefully planned semantic versioning of dependencies.

The PSR-4 standard defines how namespaces map to directory structures, allowing for automatic class loading without the need for require or include statements [3,6]. It is configured in composer.json as follows:

```
{
   "autoload": {
      "psr-4": {
         "MyApp\\": "src/"
      }
   }
}
```

This strict mapping of namespace to file path ensures a predictable and organized project structure, reducing the likelihood of conflicts and simplifying error diagnosis during class resolution. Support for multiple autoloading roots allows developers to introduce custom extensions and modules without modifying vendor code, thus enhancing architectural flexibility and accelerating the rollout of new features.

However, the addition of new classes requires manual regeneration of the autoloader via composer dump-autoload, which may slow down iterative development and should be considered in CI/CD automation workflows.

A Service Provider acts as the registration point for service classes in the dependency injection (DI) container [5,7]. In Laravel and Symfony, they serve as API gateways for the application:

```
namespace App\Providers;

use Illuminate\Support\ServiceProvider;
use ThirdParty\Client as BaseClient;
use App\Adapters\ClientAdapter;

class ClientServiceProvider extends ServiceProvider
{
   public function register()
   {
      $this->app->bind(
         BaseClient::class,
         ClientAdapter::class
      );
```

```
    }
        }
```

Using Service Providers aligns with the API gateway model seen in CPQ systems, where each service exposes a standardized interface for interaction.

Semantic Versioning 2.0.0 (SemVer) defines the format MAJOR.MINOR.PATCH, where:

- MAJOR: incompatible API changes;
- MINOR: backward-compatible new features;
- PATCH: backward-compatible bug fixes [5,8].

When choosing a versioning strategy, it's essential to balance project stability with the ability to receive security updates [9]. In CPQ integrations—where external APIs may change—it is advisable to target MINOR versions (using the ^constraint) to preserve compatibility while receiving critical fixes automatically.

Table 2 – Basic API-Centric Connectivity Mechanisms in Composer Projects [1,3,5,7,8]

Mechanism	**Description**	**Configuration Example**	**Advantages**	**Limitations**
PSR-4 Autoloading	Automatic mapping of namespace → file path	"autoload": {"psr-4": {"MyApp\\\\": "src/"}}	•Clean project structure •Multi-root support	•Requires autoloader regeneration after adding files
Service Providers	Register/override services in the DI container	Laravel: register() / Symfony: services.yaml	•Centralized configuration •Lazy, conditional loading	•Requires understanding of DI and lifecycle mechanics
Versioning	Range restriction for packages based on SemVer	"require": {"vendor/pkg": "^1.2.3"}	•Backward compatibility •Patch auto-updates	•May overlook breaking changes under wide ^ranges

In summary, the combined use of PSR-4 autoloading, Service Providers, and well-considered SemVer strategies enables PHP projects to maintain a clean, extensible, and secure environment for customizing third-party Composer packages—while preserving automated update capabilities and long-term maintainability.

3. Event-Driven Customization

Combining principles of Event-Driven Architecture (EDA) with Composer tools and PHP frameworks enables the injection of custom logic into third-party packages through lifecycle hooks and events—achieving high decoupling and true extension flexibility [1,10].

Composer provides a scripting mechanism that allows custom commands to be bound to specific package installation and update events. Key integration points include:

- pre-install-cmd — before starting dependency installation
- post-install-cmd — after successful execution of composer install
- pre-update-cmd — before updating dependencies
- post-update-cmd — after successful execution of composer update
- post-autoload-dump — after autoloading has been regenerated

These events allow developers to automate tasks such as patching vendor code, generating configuration files, or copying template assets:

```
{
  "scripts": {
    "post-install-cmd": [
      "App\\Scripts\\PatchVendor::apply",
      "App\\Scripts\\GenerateConfig::run"
    ],
    "post-update-cmd": [
      "App\\Scripts\\PatchVendor::apply"
    ]
  }
}
```

Composer hooks can be used to simulate reactive CPQ microservice behavior without modifying third-party code. Custom logic can be linked through the PSR-14 Event Dispatcher and built-in framework mechanisms:

- PSR-14 — the official publish–subscribe standard for PHP [4]
- Symfony EventDispatcher — a component for defining and subscribing to internal or custom events [7]
- Laravel Events & Listeners — a declarative event system with built-in support for queues and broadcasting [8]

An example of event registration in Symfony is shown below:

```
// src/Event/PackageModifiedEvent.php
namespace App\Event;

use Symfony\Contracts\EventDispatcher\Event;

class PackageModifiedEvent extends Event
{
    public const NAME = 'package.modified';
    private string $packageName;

    public function __construct(string $packageName)
    {
        $this->packageName = $packageName;
    }

    public function getPackageName(): string
    {
        return $this->packageName;
    }
        }
```

This approach mirrors asynchronous, event-based workflows found in CPQ-EDA systems, enabling business logic to be extended without tight coupling to third-party code.

For more complex tasks—such as patching, database migrations, or bulk API requests—events can be handled asynchronously using:

- Symfony Messenger — a component for routing messages in synchronous or asynchronous mode (e.g., RabbitMQ, Doctrine) [1]
- Laravel Queues — integration with Redis, Beanstalkd, AWS SQS for background job processing [8]

Asynchronous execution reduces the load on

Composer's CLI scripts and removes execution time limitations—an important factor in large-scale CPQ scenarios.

Table 3 – Comparison of Event-Oriented Customization Mechanisms with Composer and PHP Frameworks [1,7,9,10]

Mechanism	Integration Point	Example Scenario	Pros	Cons
Composer Hooks	pre/post-install/update-cmd	Applying a patch to a vendor package	•Simple to set up •Broad compatibility	•Synchronous execution •Time constraints
PSR-14 Event Dispatcher	Internal/user-defined events	Broadcasting PackageModifiedEvent to listeners	•Loose coupling •High testability	•Requires supporting infrastructure
Symfony Messenger /Laravel Queues	Background queue	Database migration after package update	•Asynchronous execution• Scalable	•Configuratio n complexity •External broker required

In summary, event-driven customization in Composer-based PHP projects combines the strengths of CPQ-style EDA with CLI scripting and modern framework capabilities. While this approach enables a decoupled and scalable architecture with support for asynchronous workflows, it demands careful infrastructure setup for event handling and queue management to ensure reliability and performance.

4. Practical Approaches to Customizing Third-Party Composer Packages

In modern PHP development, the author identifies five principal methodologies for customizing Composer-managed packages. Each has distinct advantages and limitations, and selecting the appropriate one depends on project goals, maintenance timelines, and team readiness for long-term support.

The first approach involves copying the package's source code directly into the internal project repository and modifying it in place. This grants complete freedom to alter functionality without constraints imposed by the original maintainers. Its simplicity and low entry barrier make it appealing when under severe time pressure or when integration must occur quickly without relying on external changes. However, this method severs the link to the official repository: updates must be merged manually, significantly increasing maintenance overhead and making the team solely responsible for bug fixes. As such, this strategy is suitable only for one-off edits to non-critical libraries or as a temporary workaround when no other options apply.

The second method is forking the package on a VCS platform (e.g., GitHub), introducing general-purpose improvements, and submitting a pull request (PR) to the original repository. This allows changes to be merged upstream while preserving compatibility with Composer's standard update mechanism. If the PR is accepted, the changes become available to all users of the package, fostering open-source development. However, the process depends on the maintainers' responsiveness; unresolved PRs can remain indefinitely in a private fork. Still, when structured clearly and addressing a real need—such as extending name parsing in a personal data library—contributions may be accepted within days, streamlining future maintenance and updates.

The third technique leverages Composer's scripting system (post-install-cmd, post-update-cmd) to apply modifications after package installation or update. Using custom Bash or PHP scripts, developers can patch files without modifying the package repository directly, maintaining structure integrity and ensuring repeatable behavior. This safeguards changes from being overwritten during updates but adds complexity due to opaque execution flows and potential fragility of pattern-based replacements (e.g., using awk). If a package's internal structure shifts significantly, the scripts may break and require constant upkeep. A successful example is disabling mbstring.func_overload checks in phpoffice/phpexcel via a Bash script referenced in the composer.json scripts section, ensuring compatibility with legacy Bitrix platforms.

The fourth strategy involves redefining Composer repositories (custom repositories). Here, composer.json specifies alternate sources (vcs/git/github/path/zip), preserving the original package name, namespace, and version while changing its download URL. This enables development within a maintained fork while retaining the option to switch back to the official package. It strikes a balance between autonomy and updatability but still requires syncing changes from upstream. An example includes a forked name-parser library adapted for PHP 8, where two interdependent repositories were defined under repositories and versions set to dev-php8; once the official PHP 8 support was released, the block was removed to resume regular updates.

The fifth approach follows object-oriented design principles: extending a third-party class by creating a custom subclass and overriding only the required methods. This keeps the base package untouched and updateable via Composer while encapsulating modifications in local code. This approach aligns with the principles of dependency inversion and the open/closed principle, but it assumes the library was designed with extension in mind—i.e., methods are protected or public, not final, and sufficient hooks are exposed. Poor extensibility in the original design may prevent full customization.

A comparative analysis shows that code copying and Composer scripts offer maximum control but weaken the connection to upstream and increase support demands. Pull requests offer the greatest benefit to the broader community and enable scalable change adoption but rely on external maintainers. Custom repositories provide a balanced compromise between autonomy and maintainability. OOP inheritance is the cleanest solution when supported by the package's architecture.

Guidelines for selecting the most suitable method:

- For changes likely to benefit others, submit a pull request.
- For localized, minimal modifications, use Composer scripts or inheritance, if feasible.
- For major revisions with ongoing update needs, maintain a custom repository.
- Use code copying only as a last resort when all other strategies are unworkable.

In conclusion, the optimal approach should be selected based on a careful balance between integration speed, safe update paths, and the potential for reuse across projects.

CONCLUSION

This study resulted in a practical toolkit and decision-making framework for customizing third-party Composer packages, structured around three core pillars:

- Extension Patterns (Decorator, Adapter, Bridge) enable modular augmentation or substitution of functionality without modifying the source code of dependencies.
- API-Centric Configuration (PSR-4, Service Providers, Semantic Versioning) establishes a clear and reliable integration layer that preserves compatibility through updates.
- Event-Driven Mechanisms (Composer Hooks, PSR-14, asynchronous queues) provide reactive handling of package lifecycle events and background tasks with minimal latency and overhead.

The proposed methodology has shown to reduce maintenance effort in complex PHP projects, lower the risks associated with package updates, and improve code reusability. As potential directions for future development, it is recommended to explore the integration of Low-Code/No-Code platforms for automating test environment scaffolding and to assess the adaptability of this approach to other ecosystems such as JavaScript (npm) and Python (pip).

REFERENCES

Joshi H. Enterprise Design Patterns for CPQ Integration in B2B SaaS Environments //Authorea Preprints. – 2024. – pp. 1-8.

Islam M. N. Designing an Advanced Educational Content Management System with Cloud Technology Integration for Ghana's Educational Landscape. – 2024. – pp. 23-49.

Li B., Kumar S. Managing Software-as-a-Service: Pricing and operations //Production and operations management. – 2022. – Vol. 31 (6). – pp. 2588-2608.

Dutta S. K. Implementing the Salesforce Enablement Playbook: A Guide to Best Practices and Organizational Success //The American Journal of Engineering and Technology. – 2024. – Vol. 6 (7). – pp. 13-23.

Pathak P. et al. Analysis of improving sales process efficiency with salesforce industries CPQ in CRM //International Conference on Micro-Electronics and Telecommunication Engineering. – Singapore : Springer Nature Singapore, 2023. – pp. 481-495.

Baul S. et al. Analyzing Different Software Project Management Tools and Proposing A New Project Management Tool Using Process Re-engineering On Open-source and SAAS Platforms for A Developing Country Like Bangladesh //International Journal of Advances in Electronics and Computer Science. – 2022. – Vol. 9 (7). – pp. 29-37.

Kaputa V., Loučanová E., Tejerina-Gaite F. A. Digital transformation in higher education institutions as a driver of social oriented innovations //Social innovation in higher education. – 2022. – Vol. 61. – pp. 81-85.

Selvaraj S. Advanced Third-Party Integrations //Building Real-Time Marvels with Laravel: Create Dynamic and Interactive Web Applications. – Berkeley, CA : Apress, 2023. – pp. 537-554.

Engebreth G., Sahu S. K. Introduction to Frameworks //PHP 8 Basics: For Programming and Web Development. – Berkeley, CA : Apress, 2022. – pp. 231-245.

Jahanshahi R. et al. Minimalist: Semi-automated debloating of {PHP} web applications through static analysis //32nd USENIX Security Symposium (USENIX Security 23). – 2023. – pp. 5557-5573