

# PROJECT INTRO ANALYSIS FOR

---



*by*

Rector Upgrade Team

December 31, 2022

Hello sir Gleiss from Ares,

We've prepared the following analysis of the project. Thanks for sharing your goals on our call. We will make them easier to achieve by making the further development cheaper by making the codebase more maintainable.

You've shared with us few pain points that currently bother you:

- testing and CI quality
- old versions of packages and extensions
- making team more effective
- upgrade of Symfony 2.8 further
- upgrade of PHP 7.2 further
- deploy process optimization

Further, we will go through the most crucial pain-points we have found during the analysis. Each pain-point focuses on the reason why it is important for you and what is the impact of this problem.

# A. Introduction to Analysis

---

## 1. How to Read this Feedback

---

These are the first steps we see when we open this project; that makes us think. It might seem like a detail you don't think about anymore, like using a key when leaving the door. But it could be smoother for a new person coming to the project.

Imagine you're riding a bike to work every day. After the first trip, a tiny grain of sand gets to your bike bearings, and you don't notice. Every day you go to work, there is one new grain in the pack, and in a week, there are 7. As an effect, the wheel does not turn as quickly as before, and it takes you more energy to ride on a straight street.

In a month, there are 30 grains of sand on your bike. **It feels like you're riding against the wind.** In 2 months, you're riding downhill, but it feels uphill.

Take the following notes as steps to improve, not as a harsh critique. It's our job to spot weak places, kick in them and make the problems obvious. Only then can we make the structure solid and durable long-term.

## The Effort we Invest Daily will Bring Sustainable Growth

Imagine we go to a dentist to check our teeth after a 10 year break. He checks our mouth with a few mechanical tools, scans the teeth, and then tells us: "your 4th, 5th, and 7th tooth are damaged".

- We reply: "I know, it's because I don't have time to clean them as much as I want. I'm busy with my work and taking care of my family."
- Then the dentist says: "Ah, well, the fix will cost you 2000 € per each."
- We: "All right, thank you".

The dentist fixes our teeth, and after 3 hours, we're ready to go.

- We: "Thank you, bye".
- The dentist: "Thank you, bye... wait, if you keep neglecting your teeth like this, you will be back in 2 years again, paying 4000 € per each fix because they will be damaged even more. Is that worth it?".

Like visiting a dentist to tell them you don't have time to clean your teeth, and that's why there is a flaw.

**We're here to improve the code quality to a level that will be healthy, joyous, and bring happy feelings in the long term.** It will require self-reflection, discussion, and mostly hard work.

We prefer to be honest with you, sharing the problems we see. We focus on typical predictors of extensive legacy code growth. In the same way, many sweet cakes and candy are predictors of flawed teeth.

In the same way, healthy teeth are worth a little effort every day. Investing in a sustainable and effective codebase will return hundred folds in easy growth and adaptation.

## 2. Before we Start

---

### What did we do Before Analysis Itself?

- First, we read code from: <https://github.com/rectorphp/rector-src>
- Specific commit:  
<https://github.com/rectorphp/rector-src/commit/b23e1fb2ea7795a7f7bb3f6232e4fa82aa26b199>
- Then we look into the code and try to run dev tools that should be running in CI
- If something was a struggle to make work, we noted it. If it was apparent later, we removed it to focus only on friction spaces.

## B. The Analysis - Parts to Improve

---

Now it's time for the parts we can improve. They're ordered by complexity from simpler to more complex ones and grouped by shared area.

They often touch many areas of the project and need time and attention to resolve, so take it slow to grasp the story.

# 1. Bare Minimum

---

## 1.1 The project has `.editorconfig`

---

Without the `.editorconfig` file, every file has a different number of spaces, tabs, line-endings... and everything else we can't see.

Well, until you have errors like:

- YAML syntax error
- Mix of tabs and spaces

Or creatively structured code like:

```
<?php
```

```
class SomeClass
{
    public function someMethod()
    {
    }
}
```

We should not deal with these problems manually. It is the computers' job that we can automate by adding `.editorconfig`.

You can read more about it on <https://editorconfig.org/>.

## 2. Autoload and Composer

---

### 2.1 Composer defines PSR-4 autoload

---

PSR-4 autoloader was not found.

An autoloader enables your classes to be loaded in automatically, because a project follows some rules, these rules are called PSR4 ([PHP Standard Recommendation](#)) following PSR4 means you don't have to have endless includes in your php code and makes your code cleaner.

#### PSR4 Rules:

- 1 file per class, interface or trait
- class name must be the same as the file name
- The namespace must follow a naming convention <file-name>

### 2.2 Composer defines PHP version

---

The first thing we do when coming to a new project is to define the PHP version.

Based on this PHP version, we plan decisions about the upgrade process:

- what package can we use
- what security issue can we face between this and the latest stable PHP
- what is the priority on PHP / type upgrade
- what problems we might face - e.g., PHP 7.2 version deprecates the whole `mCRYPT` library, so we know we should check the project for any security code or check the dependencies for security-related packages that might have been deprecated too

If we know the PHP version, we also know what tooling we can use directly and where we'll have to hack around. E.g., PHP 7.2 is supported out of the box in the



latest versions of Rector, ECS, and PHPStan. That makes their usage effortless, and we can directly add new custom features on the fly.

If we're below this version, we have to prepare a split CI job or a Docker wrapper or have locally available 2 PHP versions to make them even run. Based on PHP versions, we also kick off the CI setup.

The PHP version should also go hand-in-hand with autoload and automated testing in the CI. This way, we can validate if this PHP version setup is accurate or not.

## How to find out this PHP version?

- check the server you're running on the PHP version
- check the compatibility tool

In the safest scenario, we can use %fill% and improve the version later based on CI/tooling feedback.

## 2.3 The explicit "require/include" for each file are replaced by the autoload

---

Classes should be loaded automatically, with custom `spl_autoload_register()` function.

There are a %count" of `include/require` variation calls, around ~300 cases all over the codebase.

```
include "some_file.php";
```

The reason is to "autoload" the class if it is used in the current file.

Since 2013, we have a standard to handle this in a single place via [PSR-4](#).

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

That's how the classes should be handled, with their own namespace and single include of `vendor/autoload.php` at the start of entry point files:

```
require_once __DIR__ . '/../../vendor/autoload.php';
```

```
// all classes are available here and unique
```

That way, we have a single way to load classes and know they're all unique.

## Resources

- Composer - basic autoloading  
<https://getcomposer.org/doc/01-basic-usage.md#autoloading>
- Composer - schema → autoload  
<https://getcomposer.org/doc/04-schema.md#autoload>

## 3. Testing and CI

---

### 3.1 PHPUnit dependency is in require-dev

---

phpunit/phpunit is the standard testing framework in PHP.

Unit tests cover a single class, but it is also useful for more complex testing like autoloading of files. We usually create a test battery to cover a certain topic, then add it to the CI test battery.

Only then we start to work on the change itself, when we know we're supported by test verification.

### 3.2 Composer script has ECS, Rector, and PHPStan

---

Add the commands to the scripts option in the `composer.json`

```
{  
    "scripts": {  
        "phpstan": "vendor/bin/phpstan analyse --ansi",  
        "check-cs": "vendor/bin/ecs check --ansi",  
        "fix-cs": "vendor/bin/ecs check --fix --ansi"  
    }  
}
```

### 3.3 CI Configs must exists for Continuous Integration

---

Before we start any code changes, we have to set up continuous integration that runs all the automated scripts for us.

Among automated scripts, we can find:

- unit tests
- composer check
- autoload check
- check of duplicated classes
- PHPStan
- coding standard
- Rector
- smoke testing etc.

All the code quality setup we work hard on is useless unless it runs on every single commit.

## Which to pick?

There are defacto 3 options for picking a CI service:

- Github - <http://github.com>
  - This one is the best, cheapest, and easy to setup even without deep knowledge of the server scripting thanks to prepared [Github Actions](#)
- Bitbucket - <https://bitbucket.org>
  - This one is somewhat harder to configure, yet still quite simple to use
- Gitlab - <https://gitlab.com>
  - This is the most cumbersome one, only suitable for Docker experts and DevOps geeks.

After we pick the service that suits you best, we set up config for bare CI run.

## 3.4 ecs.php config exists

---

[EasyCodingStandard](#) is PHP's most powerful and easy-to-use coding standard tool.

It checks the coding standard of the code and ensures it all looks the same. But before adding it, we have to deal with if/else brackets and short <? tags. Otherwise,

the coding standard tool can modify in a way it breaks the code.

## How to Add it?

Adding ECS is a bit tricky, as it requires PHP 7.2. Yet, with CI running, we can have 2 splits separate jobs, one for the project on PHP 5.3 and the other on PHP 7.2 for ECS:

```
composer require --dev symplify/easy-coding-standard
```

## 3.5 phpstan.neon config exists

---

[PHPStan](#) is a tool that runs static analysis on the code.

What is "static analysis"?

Let's say we have the following code:

```
$value = new SomeClass();
```

```
// thousands of lines
```

```
if ($someCondition) {
```

```
    echo $value;
```

```
}
```

We have a variable of `$value` that contains a `SomeClass` object.

If the condition passes, the PHP run continues to echo the object. This code would crash:

```
echo $value;
```

The static analysis can find this without running the code. It follows the same path as we humans do and detects these flaws on computer time speed.

## How to Add it?

Adding PHPStan is a bit tricky, like ECS, as it requires PHP 7.2. Yet, with CI running, we can have 2 splits separate jobs, one for the project on PHP 5.3 and the other on PHP 7.2 for PHPStan:

```
composer require --dev phpstan/phpstan
```

## 4. Standards

---

### 4.1 PHP files are located in /src directory

---

In old times the code was located randomly in any file or any directory we could type. Then [PSR-0 standard](#) came to make the location of source code standardised.

The following [PSR-4 standard](#) replaced the PSR-0 and took class location even further. Many tools depend on the PSR-4 standard now, and not only that.

The location of the project domain or business logic should be located in the `/src` directory.

That means only source code needed for production, so:

- no migrations
- no fixtures
- no test helpers
- no coding standard utils
- no templates
- no translations
- no configs
- no helpers bash scripts
- no git repository tricks

Once we rule in place know that every command, that works with PHP code will get only one argument: `/src`

```
vendor/bin/ecs check src
```

```
vendor/bin/phpstan analyse src
```

```
vendor/bin/rector process src
```

We can also easily exclude the rest in the configs.

## 4.2 There is define() in the code

---

### Move from define() to Encapsulated Config

There are now ~120 `define()` calls at the start of the project to define parameters. This can lead to hard-locked parameters that cannot be changed and make it impossible to create an isolated test environment in the CI.

Let me explain how.

### From Define to Container Parameters

The `define()` was used in the old times, before MVC frameworks. It sets globally available values everywhere after the `define()` call, and the global state is dangerous.

It's a race condition trap that uses the "who gets there first wins" approach.

One example for all:

The `PHP_CS_Fixer`, `CodeSniffer` and `php-parser` define their own `T_FN` constant. The first defines it as an `int` value, the other `string`, and the third crashes on requiring `string` but getting `int` because the first `define()` wins.

Modern codebases use dependency injection containers with scalar parameters to avoid this pitfall. Set them once in the booted config, and the dependency injection container will pass them via the constructor only to the required places.

Same we should do here. With 3rd party open-source DI container or a custom one. Only that way can we test it with different values and mainly separate production and test environments.



E.g.

```
final class ContainerFactory
{
    /**
     * @param array<string, mixed> $parameters
     */
    public function createWithParameters(array $parameters)
    {
        $container = new Container();
        $container->setParameters($parameters);

        return $container;
    }
}
```

Then we can test 2 different states quickly:

```
$containerFactory = new ContainerFactory();
```

```
$validationEnabledContainer =
$containerFactory->createWithParameters([
```

```
    'validation_enabled' => true
```

```
]);
```

```
$validationDisabledContainer =
$containerFactory->createWithParameters([
```

```
    'validation_enabled' => false
```

```
]);
```

## 4.3 PHP and HTML layers are separated

---

### Templating Engine

One of the most spread patterns that block automated code changes is a mix of PHP and HTML in the same file.

### What does Templating Handle For Us?

- keeps context, passed variable types, and the render template separated
- teaches the best practice of separating logic and presentation, like input and output
- handles security issues like escaping

E.g., the PHP code I write is dangerous because I'm not a security expert, and I don't think about possible `$_GET/$_POST/$_GLOBAL` misuse or their changes through PHP versions. We always use templating engines to protect our clients from raw language flaws.

### Mix of PHP and HTML Brings a few Problems

- impossible integration with a framework - would require back/forth mapper bridge, we've done it once, and it's money hole to maintain and error-prone
- race condition issues - when was the session set? by whom?
- direct access of "global" variables
- this might lead to coding in template files
- the template is probably included somewhere else via `include` `"template_file.php"`, so there is no way to find out what types are being passed in it - this breaks static analysis tools, automated upgrade with Rector, etc., it's like jumping from an airplane just your underwear
- to add more petrol to fire, the php-parser itself has troubles [parsing and printing such files](#)

That's why the templating engine is a must-have.

## How to Get it Separate?

The upgrade process is very nicely described here:

[https://symfony.com/doc/current/introduction/from\\_flat\\_php\\_to\\_symfony.html](https://symfony.com/doc/current/introduction/from_flat_php_to_symfony.html)

The choice of templating engine (framework) is up to you. We have the best experience with Twig and Symfony, mainly due to [long-term support](#) and stability through the last 8 years.

## 4.4 If/else has always brackets

---

The missing brackets at if/else conditions are tough to read. It can also lead to unexpected results in case of space or newline change:

```
if (...)
    echo 1;
    echo 2;
echo 3;
```

See <https://3v4l.org/8OIm0>

This is very dangerous! We should always use the brackets:

```
if (...) {
    echo 1;
} else {
    echo 2;
}
echo 3;
```

While using automated tools like Rector or coding standards, the code space can change. That's because these tools rely on specific minimal code quality context.

Did you know this simple bug crashed Apple? See [Reflections on Curly Braces – Apple’s SSL Bug and What We Should Learn From It](#).

## 4.5 There are no `$_GET`/`$_POST`/`$_GLOBAL` Variables

---

Accessing global variables can lead to state override or race-condition of value:

```
$_GET['value'] = 1;
echo $_GET['value'];

unset($_GET['value']);
echo $_GET['value'];
```

This code will output:

```
1
```

```
Notice: Undefined index: value in /in/qLbnu on line 9
```

See <https://3v4l.org/qLbnu>

This is very dangerous, mainly in configs that include each other and can influence each others' behaviour.

This is nicely explained in [Symfony and HTTP Fundamentals](#)

Instead, we should use a scope of an enclosed variable, e.g., `RequestFactory::create()` right at the start of the project, like [Symfony does](#). That way, we have an immutable object:

```
use Symfony\Component\HttpFoundation\Request;
```

```
$request = Request::createFromGlobals();

// the URI being requested (e.g. /about) minus any query parameters
$request->getPathInfo();

// retrieves $_GET and $_POST variables respectively
$request->query->get('id');

$request->request->get('category', 'default category');
```

Again, Symfony is just a best practice example. We can implement our own object and API in isolated scope. The goal is to have an immutable object that we can rely on and use in tests.

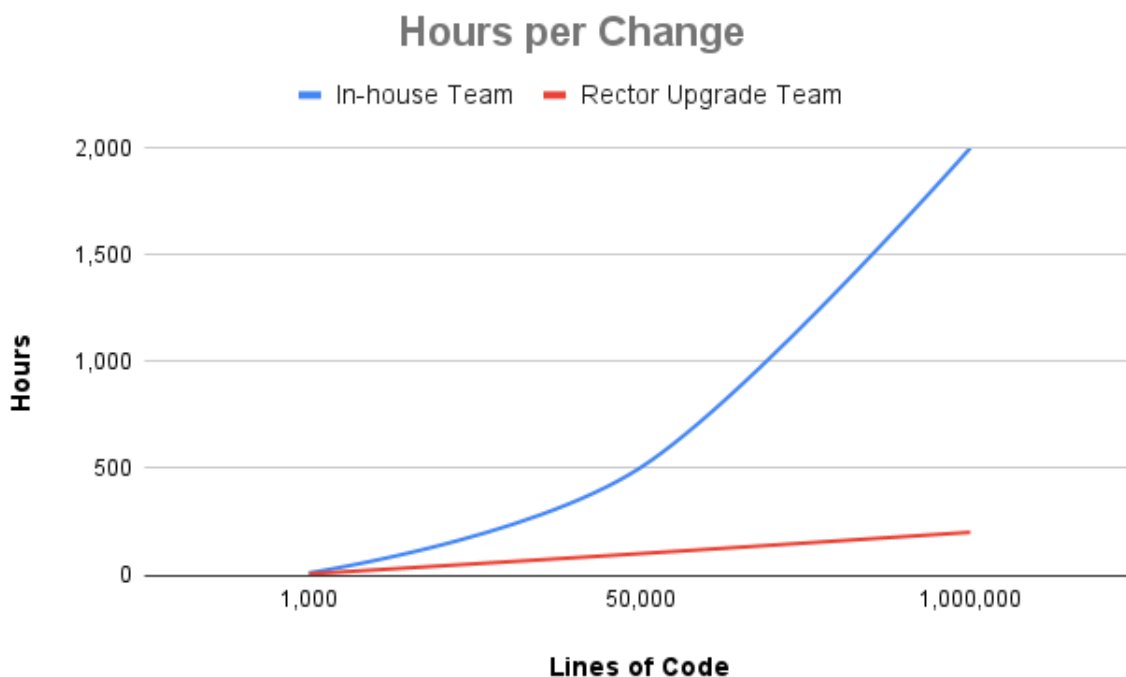
## C. How do We Start

Before starting building blocks, we have to prepare strong foundations. From our experience, it usually takes 6 months to build a strong foundation for further code improvement. Building the foundation cannot be skipped (see figure below).

- If we do it right, the rest of the work will feel like composing a few lego bricks.
- If we do it wrong, we can invest as much as we want in the higher levels, but they'll always fall on our heads.

You can implement the following steps yourself, that's all right. We can help you get there faster with our upgrade team expertise and known solutions to typical problems.

The bigger the project is, the more time and money we help you save:



Pillars required for solid foundations are listed in the intro analysis above.

With these foundations, we can move on to the subsequent phases. This is how the phases go one after another:

## 1. Building Foundations: Automate CI

---

- move to a more DX friendly CI like GitHub
- add bare ECS on CI
- add bare PHPStan on CI
- add bare PHP Linter on CI
- add bare Rector on CI
- start with basic PHPStan levels
- introduce small code improvements

**Rough estimate:**

**3-4 weeks**

## 2. Building Foundations: Code Quality & Cleanup

---

- move ECS to complete sets
- move PHPStan to higher level
- improve test coverage, introduce smoke tests and container test
- introduce tests for service registration, autowiring, dependency injection etc.

**Rough estimate:**

**2-3 weeks**

### 3. Upgrading: Start the PHP 7.4 Upgrade

---

We tried to run code on PHP 7.4 and composer install was flawless. It's unclear what the real project PHP version is, as the composer.json contains PHP 7.2.5. One way or another, we'll test this via CI with tooling from step 1, on multiple PHP versions.

There will be a lot of code deprecated in PHP 7.4. This includes server upgrades and fatal errors from removed features. The steps would be:

- go to PHP 7.3, then PHP 7.4
- upgrade packages that do not support such the PHP 7.3/7.4 versions yet
- make use of typed properties - determine types on previously unknown cases

**Rough estimate:**

**8-10 weeks**

### 4. Upgrading: Start the PHP 8.0 and 8.1 Upgrade

---

After we finish PHP 7.4, we can go directly to the PHP 8 and 8.1 upgrade. Those versions are very close, so we can handle them together. The PHP 8.0 actually removes the last PHP 7.x deprecation. That affects not only our PHP code, but also used dependencies and their upgrade.

- increase PHPStan level
- finalise PHP 7.4 property type declarations
- upgrade the core code itself
- upgrade depending packages that bring new version with new PHP

**Rough estimate:**

**4-6 months**



## 5. Improve Type Coverage as we Go

---

- this is parallel phase to all the previous ones
- the param/return/property types will be improved on the go
- the PHPStan baseline has to be narrowed as much as possible - we'd appreciate your help on this one, mainly with integration and API parts, in actions and forms
- the better type coverage we'll have, the faster we can move knowing everything is strictly checked

**Rough estimate:**

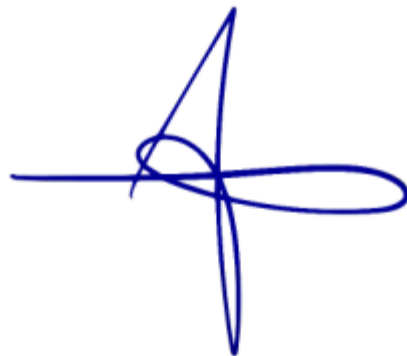
**4-10 weeks**

By nature, these phases influence and interact with each other. E.g. With the Symfony 6.x upgrade, we might need to add more types to properties and focus on type declarations of PHP 7.4 and the other way around.

We make these estimates more strict, to account for unexpected problems that arise on the way. They're based on our team expertise with 20+ Symfony/PHP project upgrades.

We look forward to making this happen together in a stable and safe fashion.

for the Rector Upgrade Team



Tomas Votruba

Founder of Rector