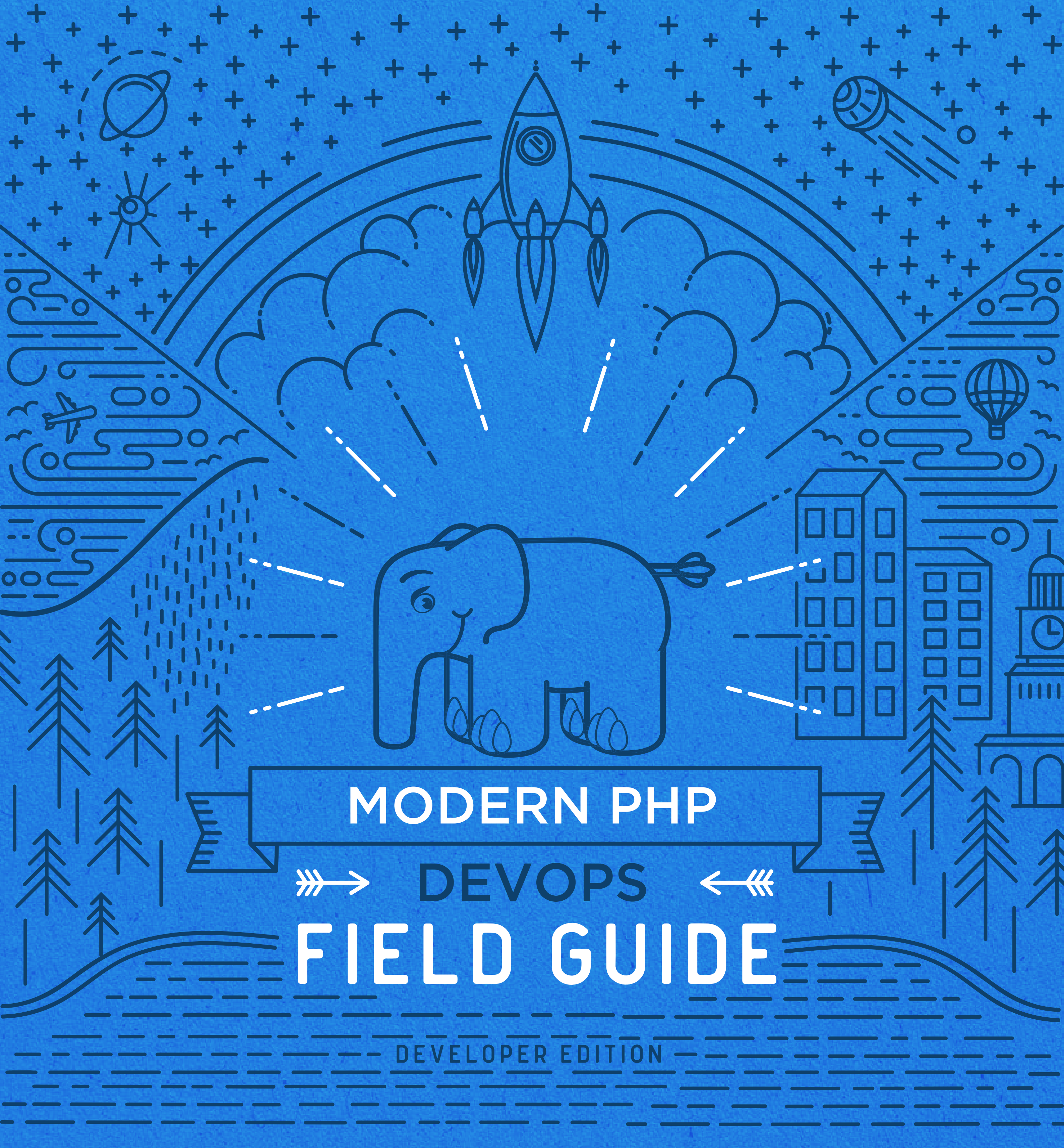


MODERN PHP DEVOPS FIELD GUIDE

DEVELOPER EDITION

platform.sh

TABLE OF CONTENTS

PHP today is not the same as PHP of yesterday.

Modern DevOps practices can have a huge impact on project lead time, cost, maintenance burdens, and failure rates:

- Hot fixes and ongoing development don't conflict
- Customers can test each feature in isolation for faster sign-off
- Switch from one development branch to another with no loss of momentum
- Common, simple toolchains get out of your way and let you work

Implementing and integrating DevOps processes can seem complex, but with the right tools and techniques PHP teams can be more productive than ever. We have gone through the journey ourselves and have developed guidelines for our own internal PHP projects, as well as received feedback from our numerous clients and agency partners.

INTRODUCTION

Companies who embrace DevOps have seen much added value to their business as a result:

- Up to 60% overall cost reduction
- 3x higher developer productivity
- 14x faster feature and UAT sign-off

To get you started on DevOps best practices, here are six steps that you can follow.

1

Embrace Change

The development model of PHP based applications has changed in recent years and your teams need to change with it. Dependency management tools like Composer have rendered the old model of committing all your 3rd party packages – modules, plugins, libraries, and the like – to version control a thing of the past. Ideally, you should only be committing the code that belongs to you and your development team.

Depending on the framework you use, this is either going to be mostly trivial or somewhat complex. A Symfony project will have many “best practices” built-in. Doing consistent builds with dependency management on Wordpress is more complex, and subject to some diverging opinions.

If your shop is using multiple technologies, try to define a common framework that applies to all.

Here are five ways to prepare and adapt to change with ease:

Use Composer. A Composer-based workflow is better suited to a continuous delivery infrastructure and makes leveraging a wider variety of existing code far easier. If you’re using a legacy framework that wasn’t built with Composer in mind, see if it has a Composer-friendly variant.

Be build-oriented. Use dependency and configuration management exclusively in your development process.

- Make every PHP project’s Git repository as lean as possible by only including the project-specific code and configuration.
- Never commit libraries to your repository. Use Composer and ignore the vendor directory in your .gitignore.

- Never hack your framework. Never hack contributed libraries either. Use clean patches with the cweagans/composer-patches plugin. Do not forget to contribute upstream. Learn to use the more advanced features of Composer.
- Use NPM or Yarn for your JavaScript dependencies, use webpack to pull in CSS and JS dependencies.
- Use Gulp, Grunt, or another task runner to compile static assets. Again, do not commit compiled assets to the repository. Commit only what your development team needs to do their job.
- Always commit lock files to your repository. This will tremendously speed up your builds as the dependency graphs for your project's libraries will have been worked out beforehand.
- Never assume database connectivity during build. Building your app should be standalone.

Wherever possible, export configuration to code. Get configuration out of the database and into version control. Avoid environment specific configuration in your database (for example do not have fully qualified URLs in the database, either use relative URLs or be sure to export those as environment variables).

Promote internal reuse:

- If you repeat the same type of projects create an internal distribution with an installation profile as a starting point.
- Do not commit internal modules directly to your distribution repository. Use a private Composer repository instead (either your own with Satis or via a service like Private Packagist).

2

Leverage modern infrastructure elements

LAMP had its day. A more modern infrastructure can save you a lot of time in the development cycle and a lot of heartache when you need to scale.

LEVERAGE MODERN INFRASTRUCTURE ELEMENTS

Use HTTP/2.0. It's faster. Use SSL everywhere, and redirect HTTP to HTTPS.

Use the latest version of PHP wherever possible. Never, ever, use the unmaintained PHP versions below 5.6. PHP 7.x can give you 2X better memory and CPU usage, with almost no code impact. And new versions of the language make development more enjoyable, too.

Move caching out of your SQL database to a dedicated caching service such as Redis. Prefer Redis to Memcached in most use-cases. (The richer API can offer better gains with less effort). Either one will give you more control and keep your database clear of large, transient blobs of data.

Do not put application logic in the HTTP caching layer (e.g. Varnish custom VCLs). Instead, make sure you are correctly setting cache headers in the application. That ensures your application is compatible with any HTTP cache or CDN, and even the browser cache.

Do not rely on MySQL search. Use Apache Solr or Elasticsearch if you need search functionality. Consider PostgreSQL if you want to limit the number of moving parts, as it has trigram search functionality available natively via an extension.

LEVERAGE MODERN INFRASTRUCTURE ELEMENTS

Move any background tasks to a dedicated queue server such as RabbitMQ, with a separate queue worker. This will perform faster and more scalably than Cron based behavior.

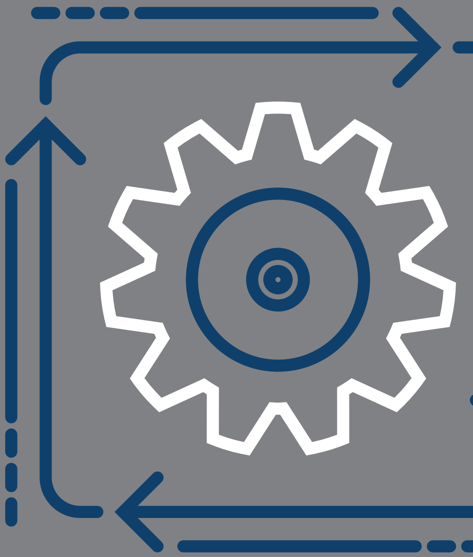
Do not rely on insecure services or unencrypted services such as FTP. Only allow SSH access to your servers.

Embrace openness to other tools. The framework you are using doesn't have to solve all of your problems. Resolve hard problems through infrastructure capabilities. If you need complex aggregations and visualization use Elasticsearch and Kibana. If you need to implement some simple machine learning use Python.

Split your application into logical micro-services. Even if you use the same framework for each component, allowing different parts of the application to evolve independently can free up your development teams to experiment.
Deploy micro-services together as a single deploy of your entire application. That ensures you are always testing and deploying compatible versions.

Make sure your backups contain everything, not just one application or service that's out of sync with another.

3



Automate all your testing environments

To make the business happy, developer teams must be agile enough to develop, test, and ship quality code continuously. To make that happen it’s necessary to move beyond the old traditional model of single prod, staging, dev environments.

Align your development and testing clusters 100% with production.

- Same versions of all components (code as well as infrastructure).
- Do not test on small datasets; test at production scale.

Create ephemeral testing environments for every Git branch or pull request; test features in isolation.

Test deployment and migration processes as well as the features themselves.

Write tests and run them on every Git push

Make sure all servers are immutable (read-only). Do not allow changes on production other than through a version control system.

Manage developer credentials centrally. Make sure you know who has access to which environment. Restrict access to the production branch.

Use protected Git branches and enforce code review and sign-off before release. The production branch should be stable and deployable to production at any moment.

Embrace continuous performance regression testing on every new feature. Tools like Blackfire.io can save you a lot of heartache. Avoid production surprises.

Embrace continuous security testing on every new feature. Run static analysis across your codebase on every Git push. Avoid production surprises.

4

Automate deployments and updates

Wouldn’t it be great if developers could spend more time coding new features instead of worrying whether each feature will ship successfully? If your tests are done right in the first place, then you should be able to deploy automatically with no fear.

AUTOMATE DEPLOYMENTS AND UPDATES

Deployments should be repeatable and fully automated. No-one should have access to production servers. Avoid having root access.

Use a configuration management tool for your infrastructure. Make sure OS configuration and firewall rules are managed centrally and have an audit trail.

Deployments should be immutable You should always be able to destroy an existing cluster and create a new one that is precisely the same. Always commit lock files of versions. Know precisely what version of what infrastructure element you are running (MySQL, nginx etc).

Automate security updates for your infrastructure (operating system, database, web server).

Make sure anything that can be run by the web server is read-only (use a read-only filesystem). Remove anything from the web root that can be removed. Use an unprivileged user for any service that is running. Make sure that writable mounts (file uploads, caches) are not executable on the command line and won’t be interpreted by the web server.

AUTOMATE DEPLOYMENTS AND UPDATES

Implement log-rotation and temp files garbage collection to make sure you don’t run out of disk space.

Consider implementing services redundancy so security updates can be applied with no downtime.

Make sure your backups can be restored. Prefer cluster-wide, automated, consistent backups (including MySQL, writable mounts, Solr...).

5

Implement observability

Make sure every service is healthy by implementing proactive monitoring.

IMPLEMENT OBSERVABILITY

Monitor CPU, Memory, Disk usage, and global latency (as seen by the user). Configure alerting at levels less than critical.

Integrate alerting with chatops (through Zapier, Hipchat, Slack) implement time-based rules for escalation (like PagerDuty).

Centralize logging, and make sure log access is protected.

Instrument production for app performance monitoring, use tools like Blackfire.io or NewRelic.

Keep a global audit log for changes to the application, the infrastructure, and user access rules.

Monitor cloud resources cost by implementing an optimization strategy for instance reservations.

6

Implement high availability and scaling

Sometimes things will go wrong, no matter how good your code or deployment plan is. Prepare for things to break and make sure plan B is ready. You don’t need your own Chaos Monkey, just the right level of active redundancy.

IMPLEMENT HIGH AVAILABILITY AND SCALING

Implement high availability and load balancing by deploying every service (including MySQL with Galera) to a cluster. Prefer active-active replication when you can.

Implement automated failover. When a cluster member fails or misbehaves, have an automated procedure to kill it and create a new one.

Consider triple redundancy as the minimum per service to allow for zero-downtime scaling and security updates (so you can take one element offline and update it while still having redundancy).

Prefer the public cloud. Only do on-premise deployments when there is no other choice. Fight “compliance theater” as you would “Security theater”. Verify whether your cloud provider can provide data protection guarantees as required by your use-case. Minimize internal systems and connect to those through a SSH tunnel. If you deploy on-premises, prefer vanilla OpenStack distributions to align as much as possible your internal systems.

Deploy to a cloud provider that offers instant creation of new machines, automate the creation of machines and their cluster configuration. Consider scaling first vertically (making each cluster member bigger) and only secondly horizontally (adding more members to each service cluster).

Reduce cloud provider lock-on. Beware of using cloud specific features and APIs that would make your application non-portable to another cloud provider.

Serve all of your traffic through a CDN, consider using a multi-tiered approach (cheap CDN for static assets, full-featured CDN with global instant purging and tag based purging for content served through the application).

Sounds complicated?
Not if you do one thing.

What if you could follow all six steps at once? Use Platform.sh. It gives you everything, out of the box, ready to use, with zero upfront investment.

Companies using Platform.sh have seen their developer teams achieve better quality results at a faster pace:

Task	What clients typically say
New system setup time	Hours to <30mins
DevOps & ticket reduction	3x less
Deployment time reduction	15x faster
Deployment frequency	Up to 20x faster

With Platform.sh you don’t need to be overly concerned with planning all of the details out in advance, and you can start slowly. Platform.sh can run a traditional PHP project in the manner that you’ve always been running it - committing everything to Git, including “vendor”, the minified assets, the entire production artifact essentially.pen When you’re ready we make it simple for you to adapt your project to these modern practices piece by piece, whether over months or a weekend.