

PHP実行環境の歴史

PHP-FPMからFrankenPHPの誕生へ



Together, we create a Sustainable NET-ZERO future.

私たちは、人類史上最大の課題である「気候変動 / 持続可能性」の解決に挑み、NET-ZERO、Sustainabilityリーダーの SX.GX を加速させるTechnology パートナーです。

本日のテーマ

PHPの実行環境の発展から見る
「プロセス分離型」と「Webサーバー統合型」の遷移

1. Apache + CGI + PHP
2. Apache + mod_php
3. Nginx + php-FPM
4. FRANKENPHP

自己紹介



HN : ma_me (twitterも同じ)

Webエンジニア歴 : 10年ちょい

主な言語 : 

所属 : 

簡単な用語説明

より理解を深めてもらうために、
頻繁に出てくる用語の簡単な概要説明

1.CGI

2.プロセス

CGIの概要説明

1. Common Gateway Interface

- よくCGIと呼ばれるもの
- Apache, NginxなどのWebサーバーと
スクリプト言語間（例：PHPなど）で
データを受け渡すための標準的なインターフェース

CGIのRFC

<u>4.</u>	The CGI Request	<u>10</u>
<u>4.1.</u>	Request Meta-Variables	<u>10</u>
<u>4.1.1.</u>	AUTH_TYPE.	<u>11</u>
<u>4.1.2.</u>	CONTENT_LENGTH	<u>12</u>
<u>4.1.3.</u>	CONTENT_TYPE	<u>12</u>
<u>4.1.4.</u>	GATEWAY_INTERFACE.	<u>13</u>
<u>4.1.5.</u>	PATH_INFO.	<u>13</u>
<u>4.1.6.</u>	PATH_TRANSLATED.	<u>14</u>
<u>4.1.7.</u>	QUERY_STRING	<u>15</u>
<u>4.1.8.</u>	REMOTE_ADDR.	<u>15</u>
<u>4.1.9.</u>	REMOTE_HOST.	<u>16</u>
<u>4.1.10.</u>	REMOTE_IDENT	<u>16</u>
<u>4.1.11.</u>	REMOTE_USER.	<u>16</u>
<u>4.1.12.</u>	REQUEST_METHOD	<u>17</u>
<u>4.1.13.</u>	SCRIPT_NAME.	<u>17</u>
<u>4.1.14.</u>	SERVER_NAME.	<u>17</u>

[RFC 3875 - The Common Gateway Interface \(CGI\) Version 1.1](#)

プロセスのざっくり概要

2. プロセス

OS が管理する実行単位

- 親（メイン）プロセス
- 子（サブ）プロセス

が存在する

親が落ちると子も落ちる

プロセスのざっくり概要



注) 本当はブラウザのプロセスはもっと複雑です。説明のため省略

前置き終了



1 : Apache + CGI + PHP



+ CGI + 時代背景

- 仮想化技術が未発達
- 一台のサーバーで
どれだけリクエストを捌けるかが勝負の時代



Apacheの特性

- 1 リクエストに対して 1 プロセスを生成するPre-Fork型
- プロセス数 > CPUのコア数になると、処理待ちが発生する



+ CGI +  プロセス分離

1. Apache (mod_cgi)
2. PHP (php-cgi)

Apache側がPHPをCGI経由で呼び出す
「プロセスを分離」

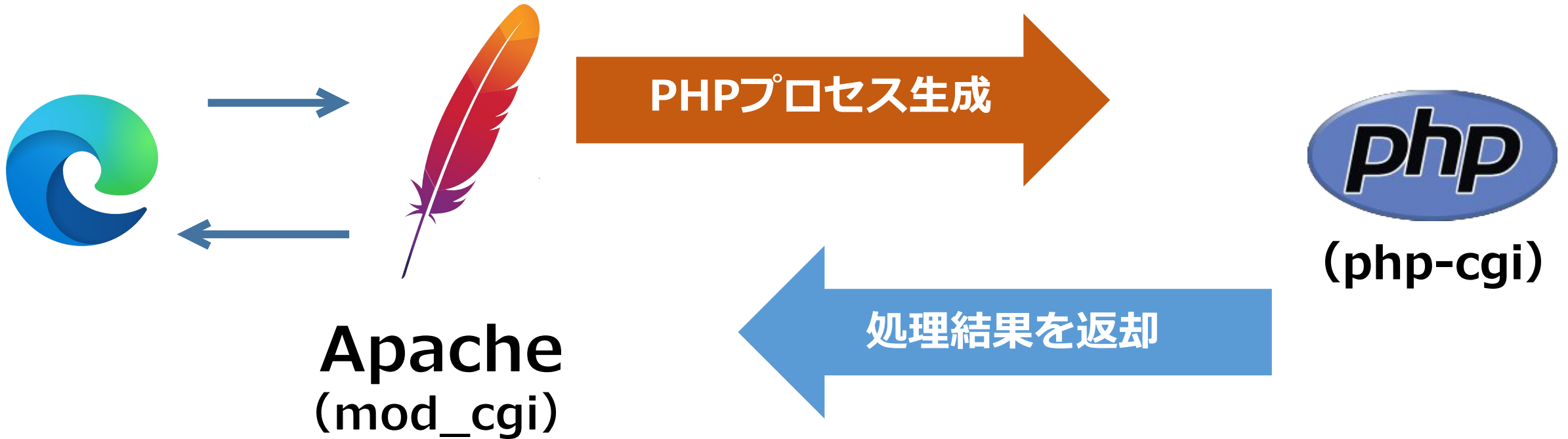


Apache





+ CGI +  プロセスのフロー





+ CGI +  構成のメリット

- ApacheとPHPが疎結合で
お互いのバージョンや設定に影響を受けにくい
- Apacheから見た場合
CGI実装であればPHP以外でも差し替えた



+ CGI +



構成のデメリット

リクエストのたびに

- Apacheが1プロセスを利用

- PHPプロセスが生成・破棄される

高コスト

(CPU & メモリ負荷が高い)



USER	PID	%
www-data	12345	
www-data	12346	

~

COMMAND
/usr/sbin/apache2 -k
/usr/bin/php-cgi



USER	PID	%
www-data	12345	
www-data	12347	
www-data	12348	

~

COMMAND
/usr/sbin/apache2 -k
/usr/bin/php-cgi
/usr/bin/php-cgi

破棄PID : 12346

NewPID : 12347,12348



+ CGI +  構成のデメリット

リクエストのたびに

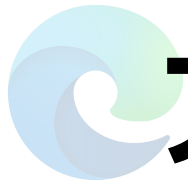
- Apacheが1プロセスを利用

- PHPプロセスが生成 破棄される

高コスト

(CPU & メモリ負荷が高い)

「Webサーバー統合」の



アプローチで、解決

→ Apache + mod_phpへ



USER	PID %
www-data	12345
www-data	12346

COMMAND
/usr/sbin/apache2 -k
/usr/bin/php-cgi

USER	PID %
www-data	12345
www-data	12347
www-data	12348

COMMAND
/usr/sbin/apache2 -k
/usr/bin/php-cgi
/usr/bin/php-cgi

破棄PID : 12346

NewPID : 12347,12348



2. Apache + mod_php (PHP Module)



mod_php 時代背景

かわらず

- 仮想化技術が未発達
- 一台のサーバーで
どれだけリクエストを捌けるかが勝負の時代



Apacheの特性

- 1 リクエストに対して 1 プロセスを生成するPre-Fork型
- プロセス数 > CPUのコア数になると、処理待ちが発生する



mod_php Webサーバーに統合

Apacheにモジュールとして、PHPを統合した

= mod_php

実装言語がC同士なので、比較的容易だった



実装言語



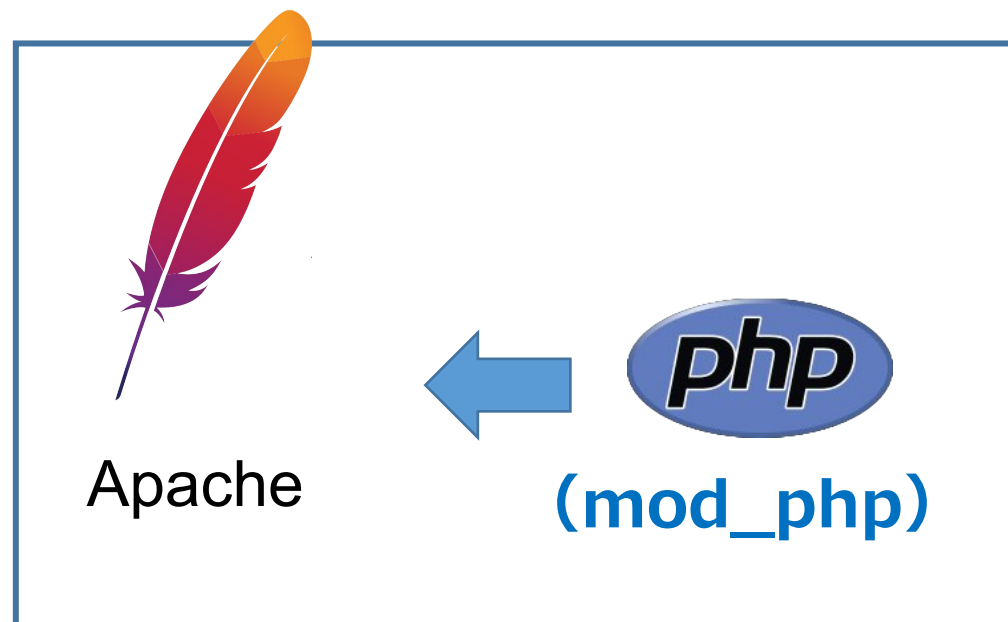


mod_php イメージ

Apache + CGI + PHP
(プロセス分離のアプローチ)



Webサーバー統合のアプローチ
(Apache + mod_php)

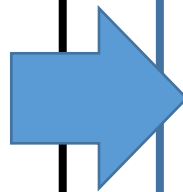




mod_php プロセス

Apache + CGI + PHP

USER	PID	%CPU	COMMAND
www-data	12345	0.2	/usr/sbin/apache2 -k start
www-data	12346	0.1	/usr/sbin/apache2 -k start
www-data	12347	0.1	~ /usr/sbin/apache2 -k start
www-data	12348	0.1	/usr/sbin/apache2 -k start
www-data	23456	0.3	/usr/bin/php-cgi
www-data	23457	0.2	/usr/bin/php-cgi
www-data	23458	0.2	/usr/bin/php-cgi



Apache + mod_php

USER	PID	COMMAND
www-data	12345	~ /usr/sbin/apache2 -k start
www-data	12346	/usr/sbin/apache2 -k start
www-data	12347	/usr/sbin/apache2 -k start

Apacheだけに
phpのプロセスは無くなった



mod_php メリット

- CGIのオーバーヘッドがなくなり、
処理が高速化
- 1 リクエストごとの
PHPプロセスの生成・破棄が無くなった



mod_php デメリット

- すべてのリクエストに PHP モジュールが付随
例：HTML・画像など
- メモリ使用量の増大とスケーラビリティ低下



mod_php デメリット

メモリをより大量に消費しやすい環境に
同時接続を処理する際のスケーラビリティ問題が発生

C10k問題の発生確率 **増** ↑



mod_php デメリット

メモリをより大量に消費しやすい環境に

同時接続を処理する際のスケーラビリティ問題が発生

再び「プロセス分離」

することで、解決へ

→ Nginx + PHP-fpmへ



3. Nginx + PHP-FPM



+



時代背景

- クラウド技術の発展とともに仮想化が進み、コンテナ（Docker など）が定着
- **Nginx** がイベント駆動型のアーキテクチャを採用
Apache の Pre-Fork 方式と比べてスケーラビリティが向上
- C10k問題から、高負荷環境やコンテナ環境で
Nginx の採用が加速



改良点

イベント駆動アーキテクチャ

Pre-fork型（従来のApache）	イベント駆動型（Nginx）
同期的・ブロッキング	非同期・ノンブロッキング
1プロセス・1リクエスト	1プロセス・複数リクエスト
プロセスごとにメモリを多く消費	リソースを効率よく使用



c10k問題（大量の同時接続処理）にも
対応しやすい

補足 今ではApache も イベント駆動に対応している

Apacheはイベント駆動への取り組みが**Nginx**と比べて遅かった
今では様々な実行形式を選べるようになっている

現在選択できるMPM

1. event（イベント駆動形式）
2. prefork（旧来の形式）
3. worker（マルチプロセス・マルチスレッド形式）

補足 今ではApache も イベント駆動に対応している

- スライドではスペースの都合上、**Nginx**で表記
- 現在ではphp-fpm なら **Nginx**ではなく
php-fpm なら **Apache** でも **Nginx** でも可能



OK



OK



改良点

PHP **F**astCGI **P**rocess **M**anagement

- **F**astCGI

CGIが進化したFastCGIを採用して、CGI の欠点を克服した

- **P**rocess **M**anagementの独立

PHP のプロセス管理を独立させた



php-fpm (FastCGIの実装) による改良点

	CGI	FastCGI	改良点
通信方式	環境変数 + stdin/stdout	バイナリプロトコル (ソケット通信)	データ転送効率向上
プロセス管理	1リクエスト = 1プロセス	プロセスを再利用	高速化 & 負荷軽減
レスポンス	一括送信	ストリーミング可能	リアルタイム処理可能
環境変数	使用する	使用しない (バイナリパケット)	OS の制限を回避



Process Managementの独立

ps -a

```
USER      PID
root       1
www-data   6
www-data   7
```

~

```
TIME  COMMAND
0:00  php-fpm: master process (/usr/local/etc/php-fpm.conf)
0:00  php-fpm: pool www
0:00  php-fpm: pool www
```

php-fpm.d/www.conf

(pm = プロセスマネージャーの略)

```
[www]
user = www-data
group = www-data
listen = 127.0.0.1:9000
pm.max_children = 5
pm.start_servers = 2
```



プロセス再利用のイメージ



USER	PID
root	1
www-data	6
www-data	7

~

COMMAND
php-fpm: master process (/usr/local/etc/php-fpm.conf)
php-fpm: pool www
php-fpm: pool www



USER	PID
root	1
www-data	6
www-data	7

~

COMMAND
php-fpm: master process (/usr/local/etc/php-fpm.conf)
php-fpm: pool www
php-fpm: pool www

+ 改善点

プロセスの再利用による効率化

➡ **1つのプロセスを複数のリクエスト処理に活用**

プロセスの生成を最小限に抑制

➡ **より少ないリソースで多数の接続を処理可能**

非同期処理によるスケーラビリティ向上

➡ **高負荷環境でもスムーズに動作**



+

phpfpm

改善点

プロセスの再利用による効率化

→ 1つのプロセスを複数のリクエスト処理に活用

今でも長く使われる

→ より少ないリソースで多数の接続を処理可能

環境に

非同期処理によるスケーラビリティ向上

→ 高負荷環境でもスムーズに動作



4. ここまでの遷移まとめ

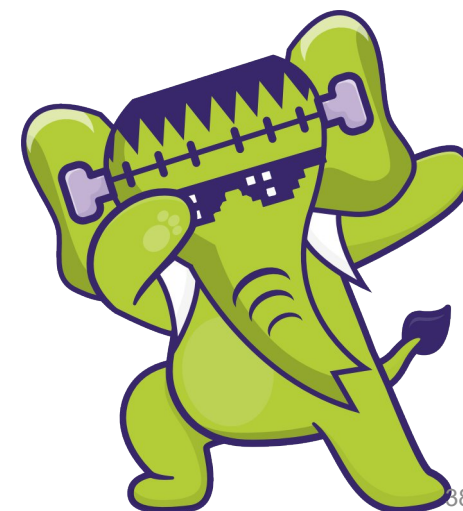
ここまでの遷移まとめ

プロセス分離：Apache + CGI + PHP

→ Webサーバー統合：Apache + mod_php

→ プロセス分離：Nginx (Apache) + php-fpm

→ ??? (次は?)



ここまでの遷移まとめ

プロセス分離 : Apache + CGI + PHP

→ Webサーバー統合 : Apache + mod_php

→ プロセス分離 : Nginx (Apache) + php-fpm

→ Webサーバー統合 :


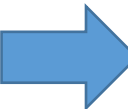




4. Franken PHP

時代の遷移

時代が進むにつれて、コンテナ技術が進化・普及した

- 1つのWebアプリケーションに対して
複数台のコンテナを利用することが当たり前になった
- スケーリングの切り替わり
垂直方向  (1台のパフォーマンスを增強) の拡張から
水平方向  (台数を増やして負荷分散) の拡張へ



+

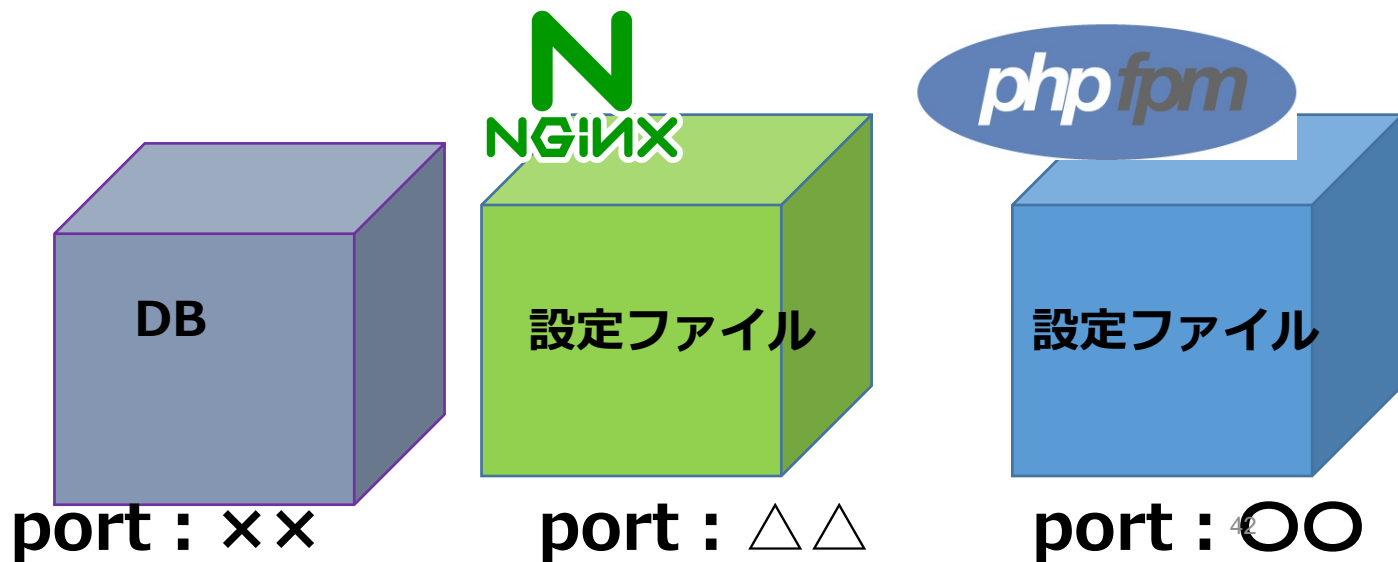


デメリット

PHPでWebサービス作る時の最低限のセット数

1. php-fpm
2. Webサーバー
3. DBサーバー

最低限 **3 台** はコンテナが必要





+



デメリット

水平スケールする時に

- 1. 各コンテナ・サービスの設定ファイル
- 2. 各コンテナ間のファイル同期
- 3. 各コンテナ間のポート調整
- 4. 各コンテナ間の通信調整

(1～4) * N回必要
かつ コンテナ間の管理がより煩雑に



+



デメリット

水平スケールする時に

1. 各コンテナ・サービスの設定ファイル

2. 各コンテナ間の同期

3. 各コンテナ間のポート調整

4. 各コンテナ間の通信調整

もっと単純に

かつパフォーマンスを出そう！

→ FrankenPHPへ

(1～4) * N回必要

かつ コンテナ間の管理がより煩雑に



Franken PHPとは

Go言語製のWebサーバー Caddy



- CaddyにPHP実行環境を組み込み、
PHPの実行環境を備えたWebサーバー
- PHPのプロセスはCaddy内部で直接処理されるため
FastCGIプロトコルを使用しない



特徴

- Go 言語で実装されていて、
組み込みのマルチスレッド & 非同期 I/O に強い
- ゴルーチンによる非同期処理と
効率的なマルチスレッド処理を活用して、
高並列なリクエスト処理が可能
- 後発なだけあって、**Apache**や**Nginx**より高性能
~~（Go 言語に強くないので、サイトの引用）~~



FrankenPHPにおける PHPの立ち位置

直接Webサーバーに統合

php-fpmも利用しないし、FastCGIも実装していない



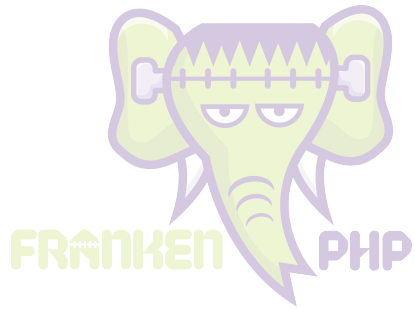
統合



未使用



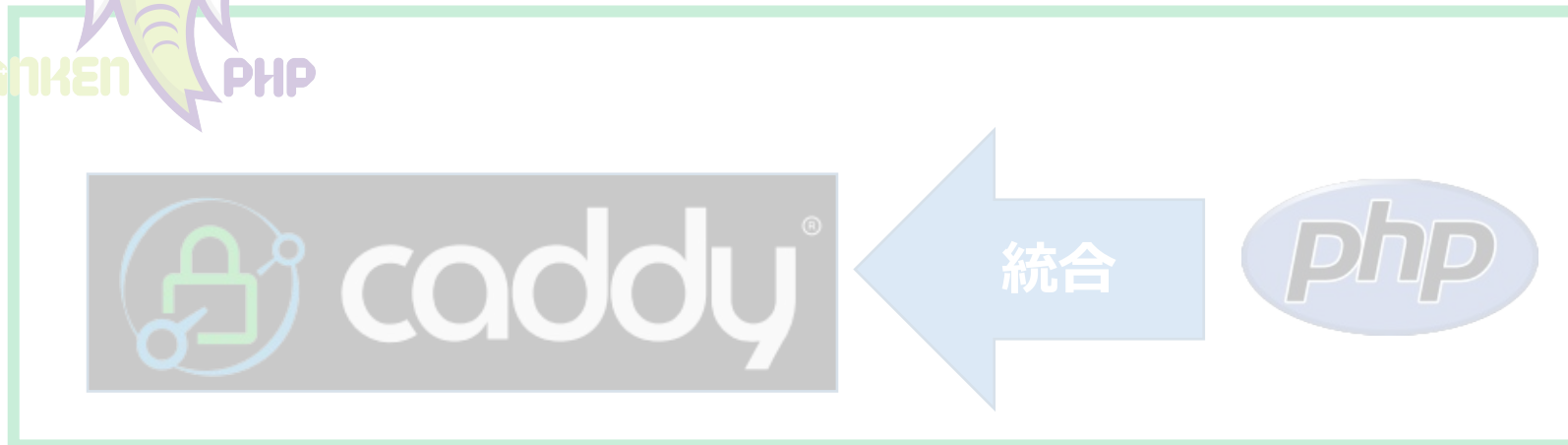
FastCGI



FrankenPHPにおける PHPの立ち位置

直接Webサーバーに統合

php-fpmも利用しないし FastCGIも実装してない
php-fpmもFastCGIも使わない
既視感があるアプローチ



未使用



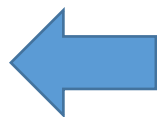


アプローチ比較

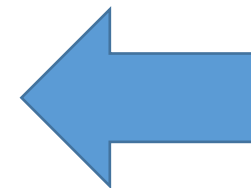
Apache + mod_php もとった

Webサーバーに統合するアプローチ

Apache



(mod_php)

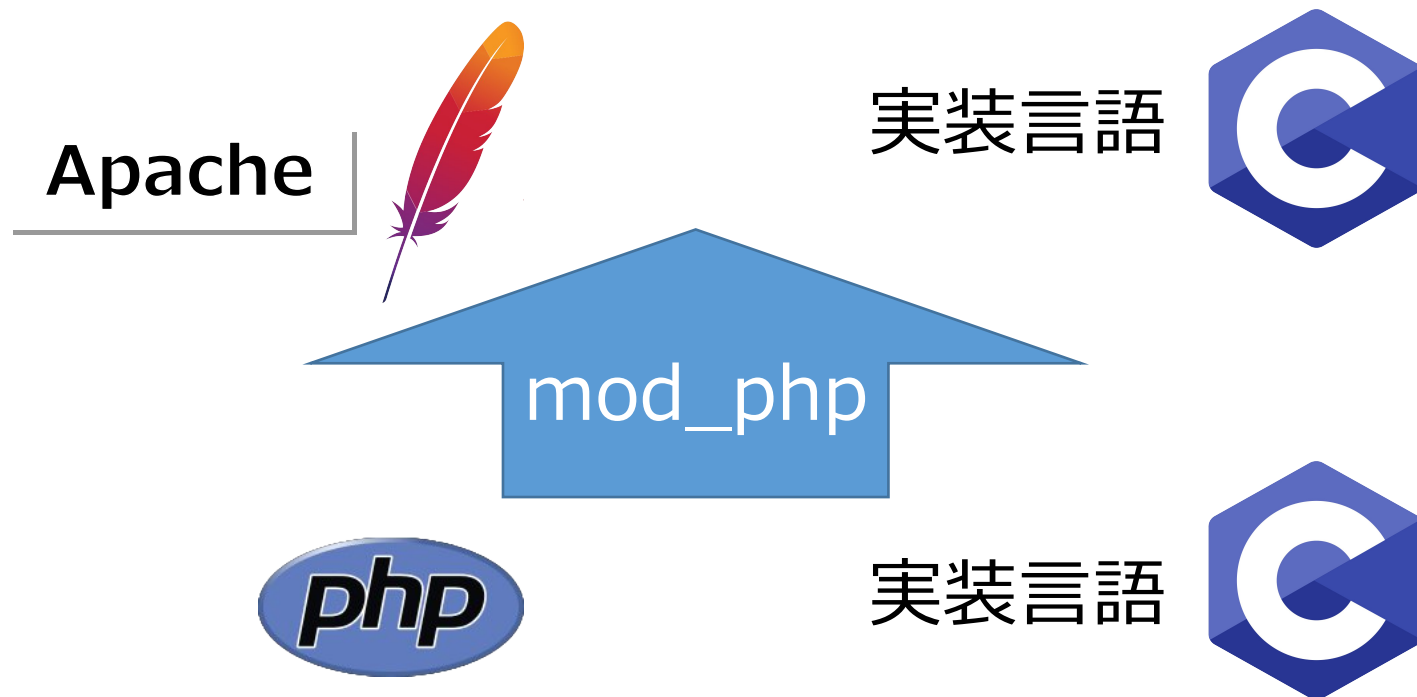




Franken PHPの黒魔術

Apache + mod_phpの場合

ベースが同じC言語なので組み込みやすい





Franken PHPの黒魔術

Franken PHPの場合

言語が違うのに、どうやって組み込む？



実装言語



実装言語





Franken PHPの黒魔術

GolangのコードからC言語のコードを呼び出すことが可能
Golangが持つ「cgo」というメカニズム





Franken PHPの黒魔術

もしかしてC言語で実装されたPHPも
cgoで直接呼び出せるのでは？



実装言語





黒魔術 完成の瞬間

Call the function from Go!

```
// #include "main.c"
import "C"
import "unsafe"

func main() {
    fileName := C.CString("my-script.php")
    defer C.free(unsafe.Pointer(fileName))

    C.frankenphp_execute_script_cli(fileName)
}
```

出典 : [FrankenPHP: A modern app server for PHP, written in Go - Speaker Deck](#)



黒魔術 完成の瞬間

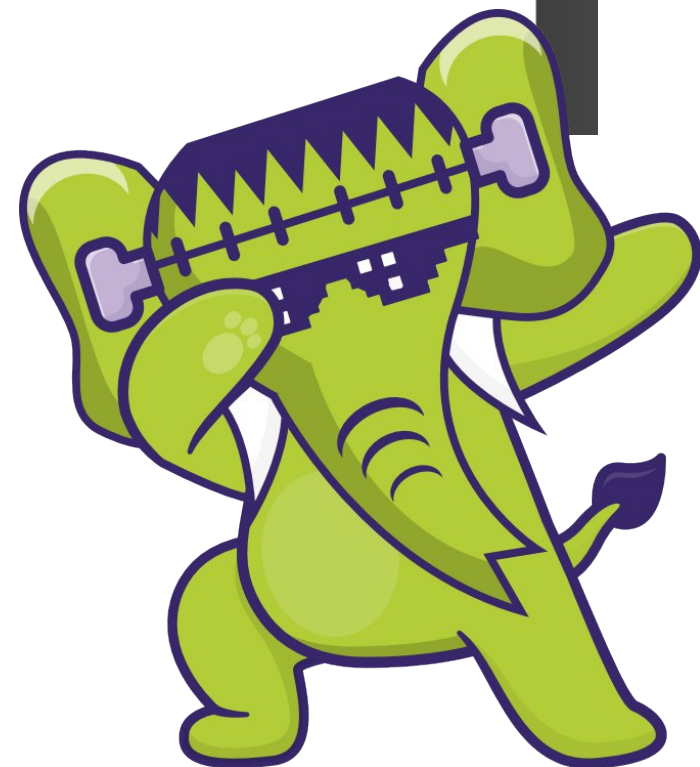
Call the function from Go!

これは確かに
Franken (stein)

```
// #include "main.c"
import "C"
import "unsafe"

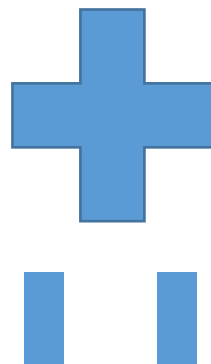
func main() {
    fileName := C.CString("my-script.php")
    defer C.free(unsafe.Pointer(fileName))

    C.frankenphp_execute_script_cli(fileName)
}
```

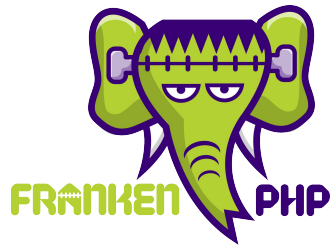




誕生



誕生



何故インターフェースを かまさないのか？

中々無茶なことをやっているように見えます

- APIやCGIを経由する
- 別サービスに切り出す
- etc

何らかのインターフェースを噛ますことが1つの解決策

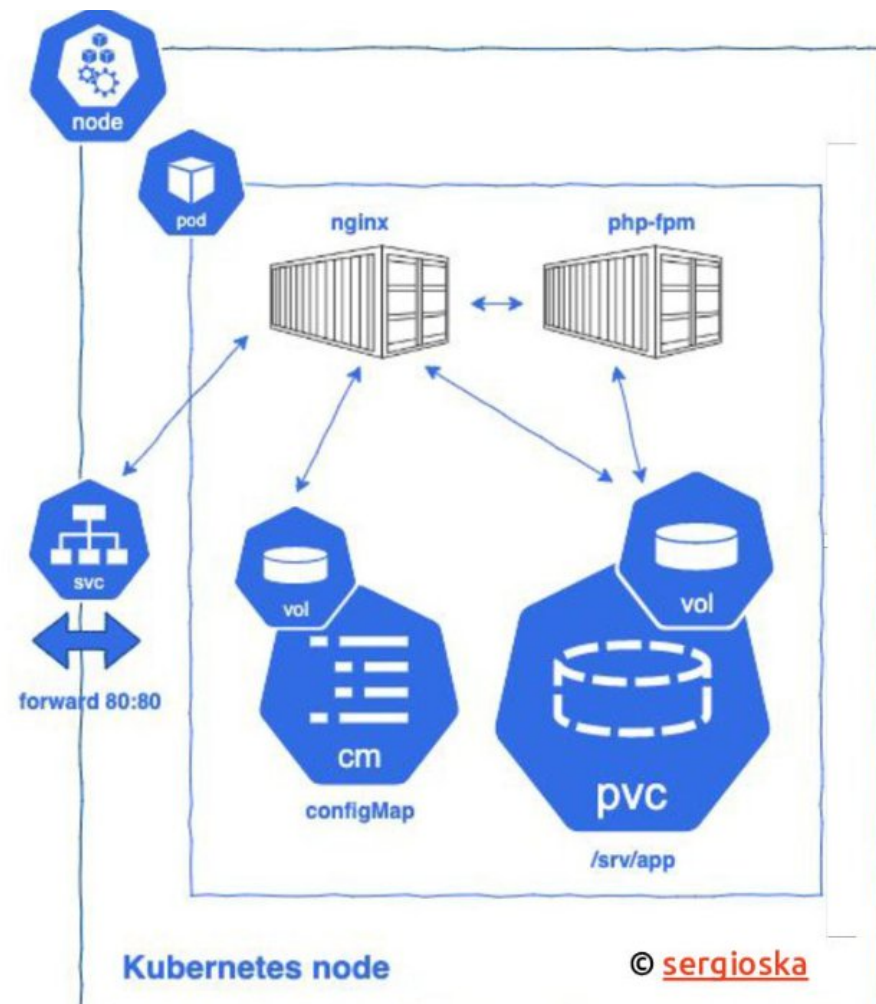
では何故そうしないのか？

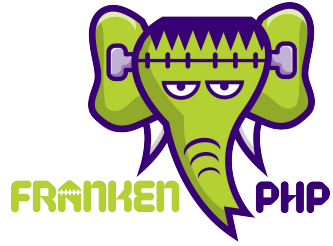


Q & A

A : 何らかのインターフェースを介すると

**サーバーやコンテナの構成など、
いろんな面で複雑になるから
(例 : PHP-FPM)**





Q & A

組み込んで柔軟性を捨てたことで
どれだけの**メリット**を得られたかが大切



どれだけメリットを得られたか

caddyに組み込んでいるため、直接やり取りできる
caddyの持っている機能・メリットを全部享受することに成功

Caddyのメリット

- 1.非同期I/O、イベント駆動アーキテクチャ
 - 2.103 Early Hints status code
 - 3.HTTP2,3 (Quick) 標準対応
 - 4.JSライブラリやSDKは不要のイベントプッシュ
- etc...

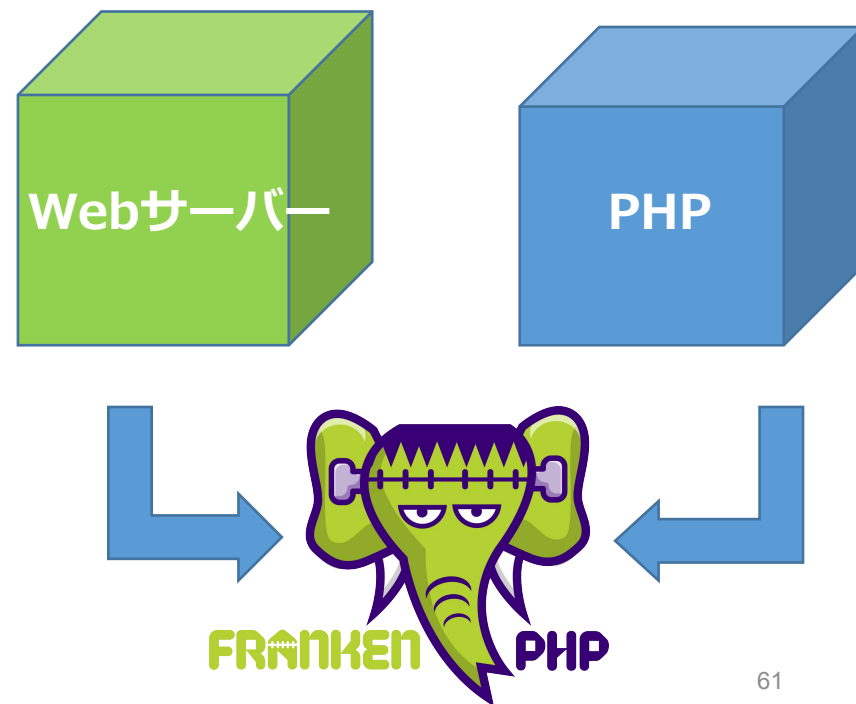


どれだけメリットを得られたか

そして何より

1. コンテナが1つで済む

2. 設定が楽





組み込んだデメリット

1. Webサーバーと蜜結合ゆえに実質**Apache** or **Nginx**から乗り換えることになる
2. 付随して、学習コストや情報源の問題が発生
3. プロダクト採用実例も殆どない

この**メリット** **デメリット**を
どう感じるかが、採用の決めて



5. 余談

プロセス分離と Webサーバー統合を比較して

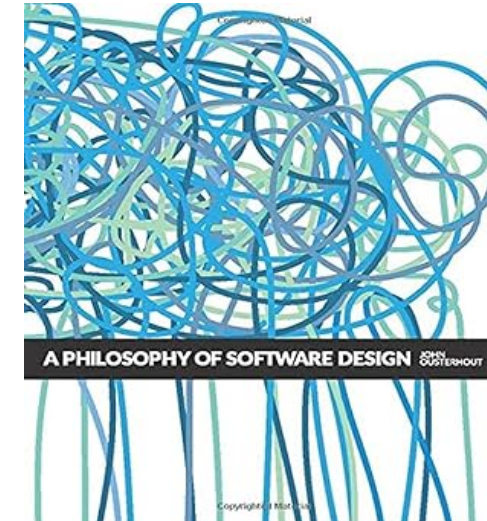
プロセス分離の特徴

依存関係を制御し、変化に強い設計で、
長年使えるソフトウェアを創る

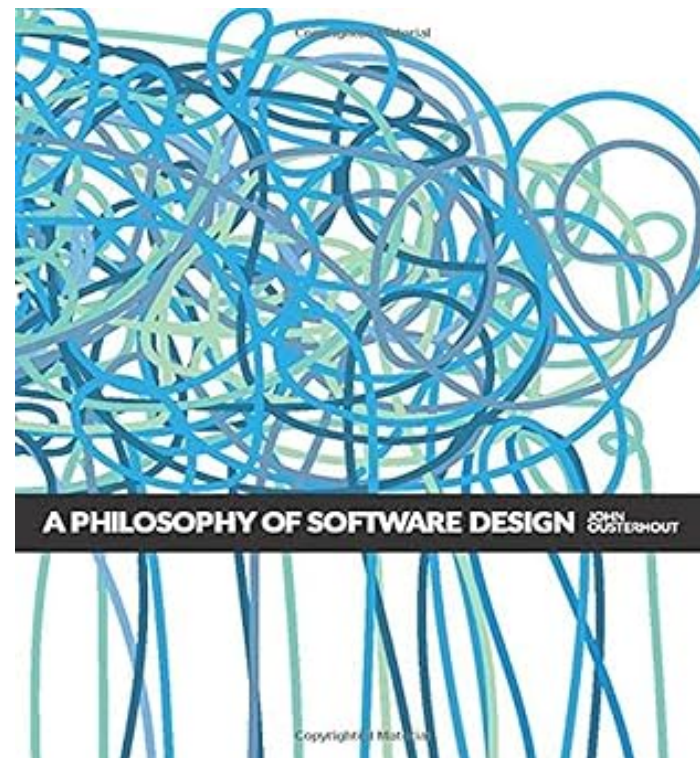


Webサーバー統合の特徴

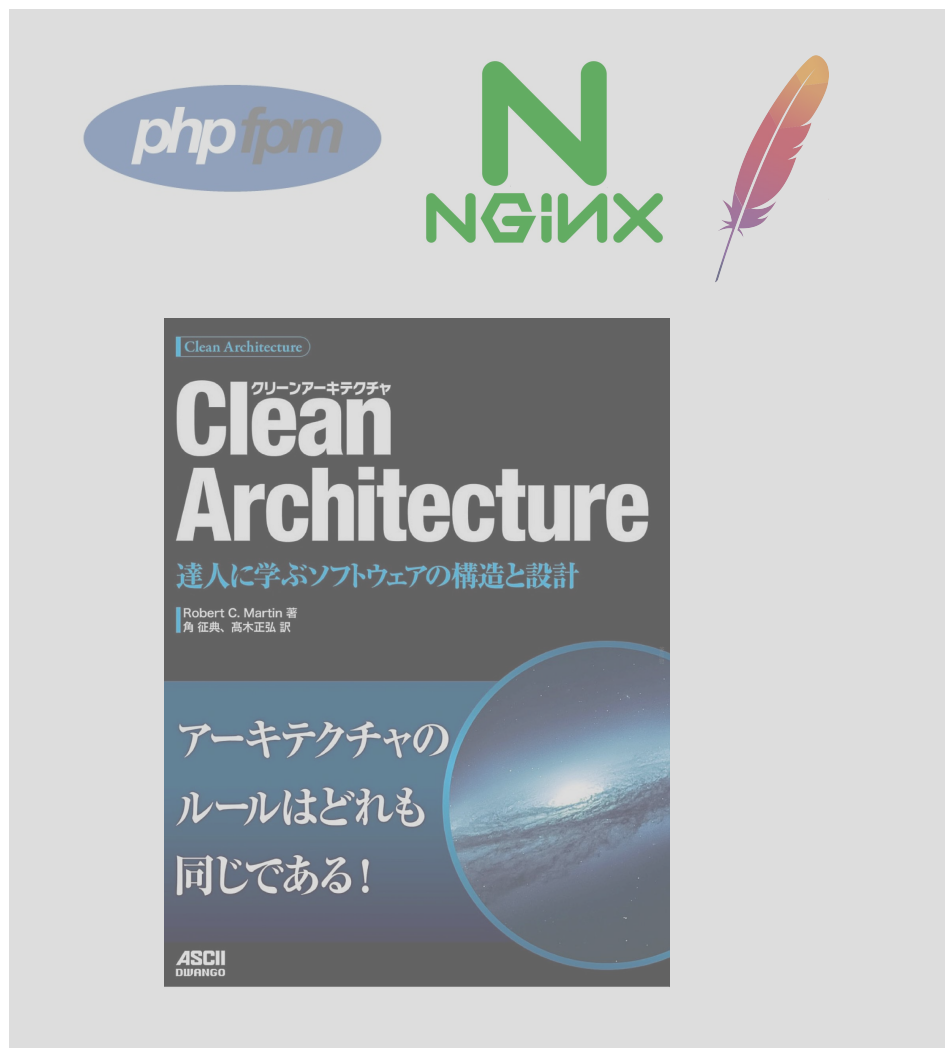
深いモジュールで複雑さを封じ込め、
使用者に単純さを提供する



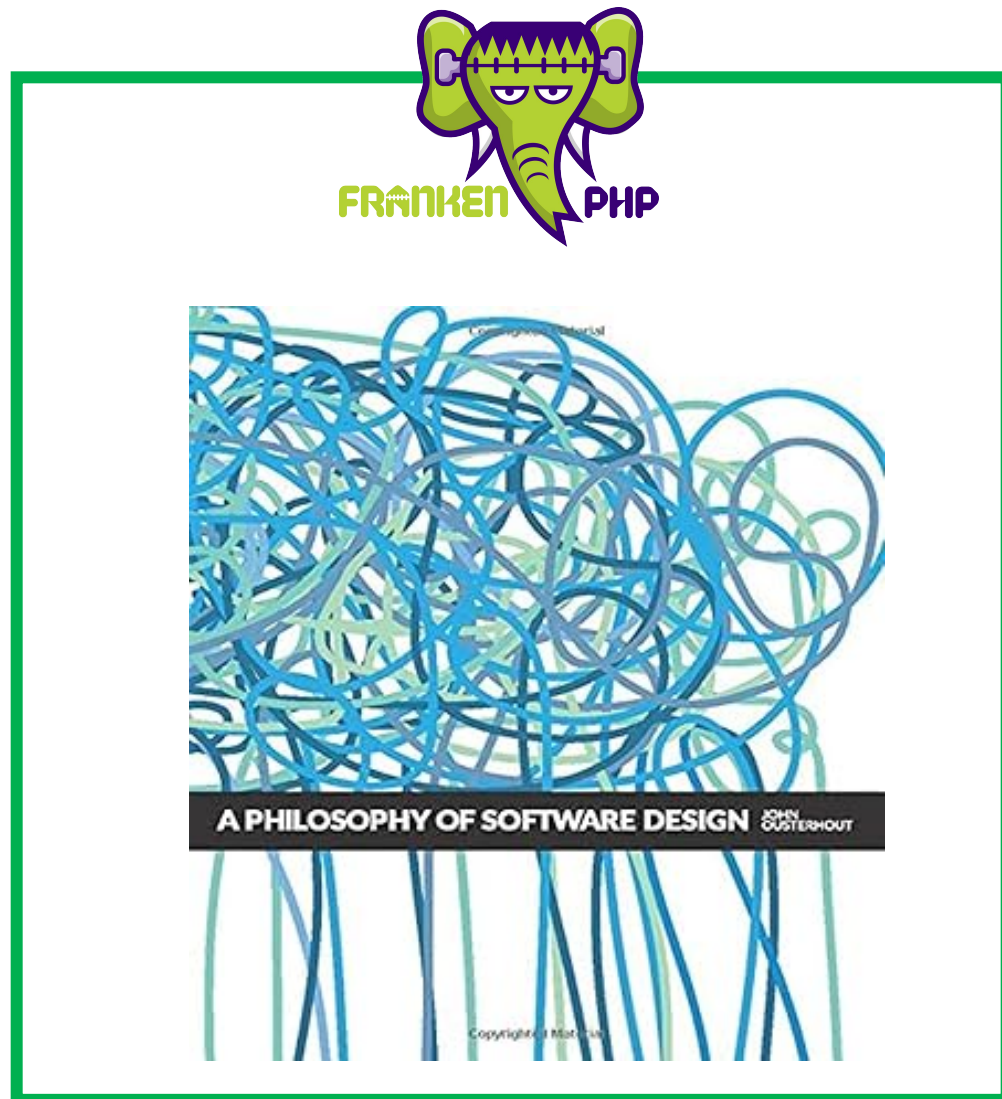
既視感 どちらのアプローチが好きですか？

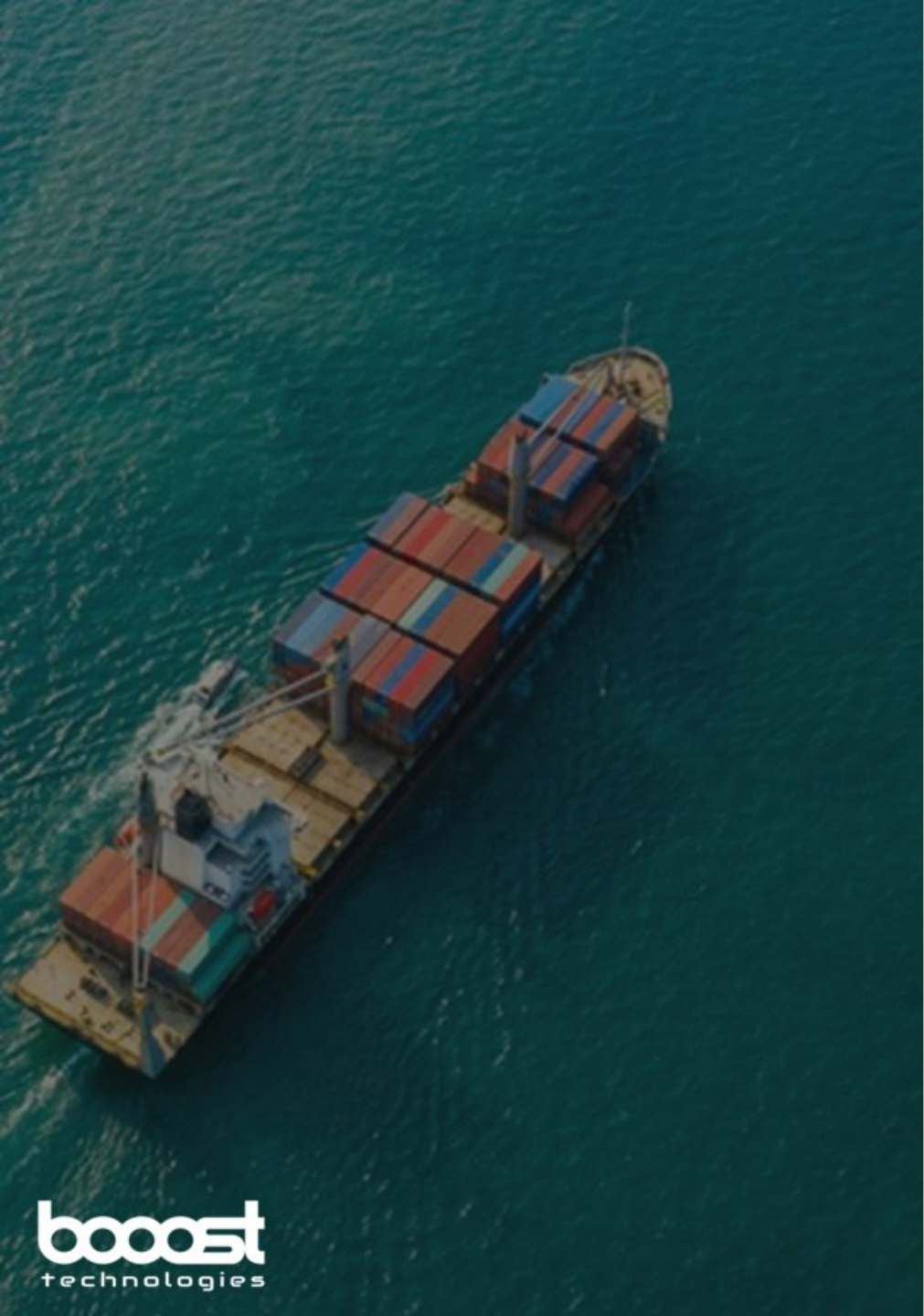


個人の感想



めっちゃ好き





6. まとめ

まとめ アーキテクチャ分類

プロセス分離型



CGI



N
NGINX



サーバー統合型



mod_php



	プロセス分離型	サーバー統合型
安定性	高い	低い
スケーラビリティ	高い	低い
設定	複雑	簡易
デプロイの複雑さ	複雑	簡易

どれがハイパフォーマンスなのか？

FrankenPHPがどこまでパフォーマンスが出せるのかは
試せていない…

次回機会があれば、各構成のパフォーマンス比較検証したい



VS



VS



最後に宣伝



PHPエンジニアを絶賛募集中

採用情報 | <https://boost-tech.com/recruit>



ご清聴ありがとうございました