

# もう一度考える 10x プログラマー

ValueUp株式会社

株式会社コルモアナ

谷口幸宏



# 自己紹介

- 株式会社コルモアナ
- ソフトウェアエンジニア
- ValueUp株式会社
- 谷口幸宏
- 寿司
- 肉
- 酒
- Twitter: [@nasjp\\_dev](#)
- GitHub: [@nasjp](#)



	ユーザー数	エンジニア数
	4 億 5000 万人	32 名
	3000 万人	13 名
	10 万人	1 名

Refs: [10x エンジニアの幸福な終焉 \(a16z\)](#)

Refs: [Dropbox の歴史 | Dropbox はこうして 5 億人超のユーザー支持を得た！](#)



# なんですか？

- LLM を使ってエンジニアとしての生産性を上げる話をみんなでしたい
- 本当に人の何十倍も生産性の高いエンジニアは存在するのか？
  - 存在する
- まずはエンジニアの生産性について考える必要がある





# 10x プログラマーという神話 by antirez

「一流のプログラマーは、普通のプログラマーの 10 倍の生産性を持つ」

- antirez さんのブログエントリー
- antirez さん
  - Salvatore Sanfilippo (通称 antirez)
  - オープンソースソフトウェアの開発者
  - Redis の Founder
  - 6 歳の頃からプログラミング

Refs: [The mythical 10x programmer](#)



# 10x プログラマー

- 1968 年の研究
- 最も優れたプログラマーが他のプログラマーよりも 10 倍生産的であることが示された
- 「人月の神話」で紹介され、広く知られるようになった
- プログラミングの複雑さやチームワークの重要性はあまり考慮されていない



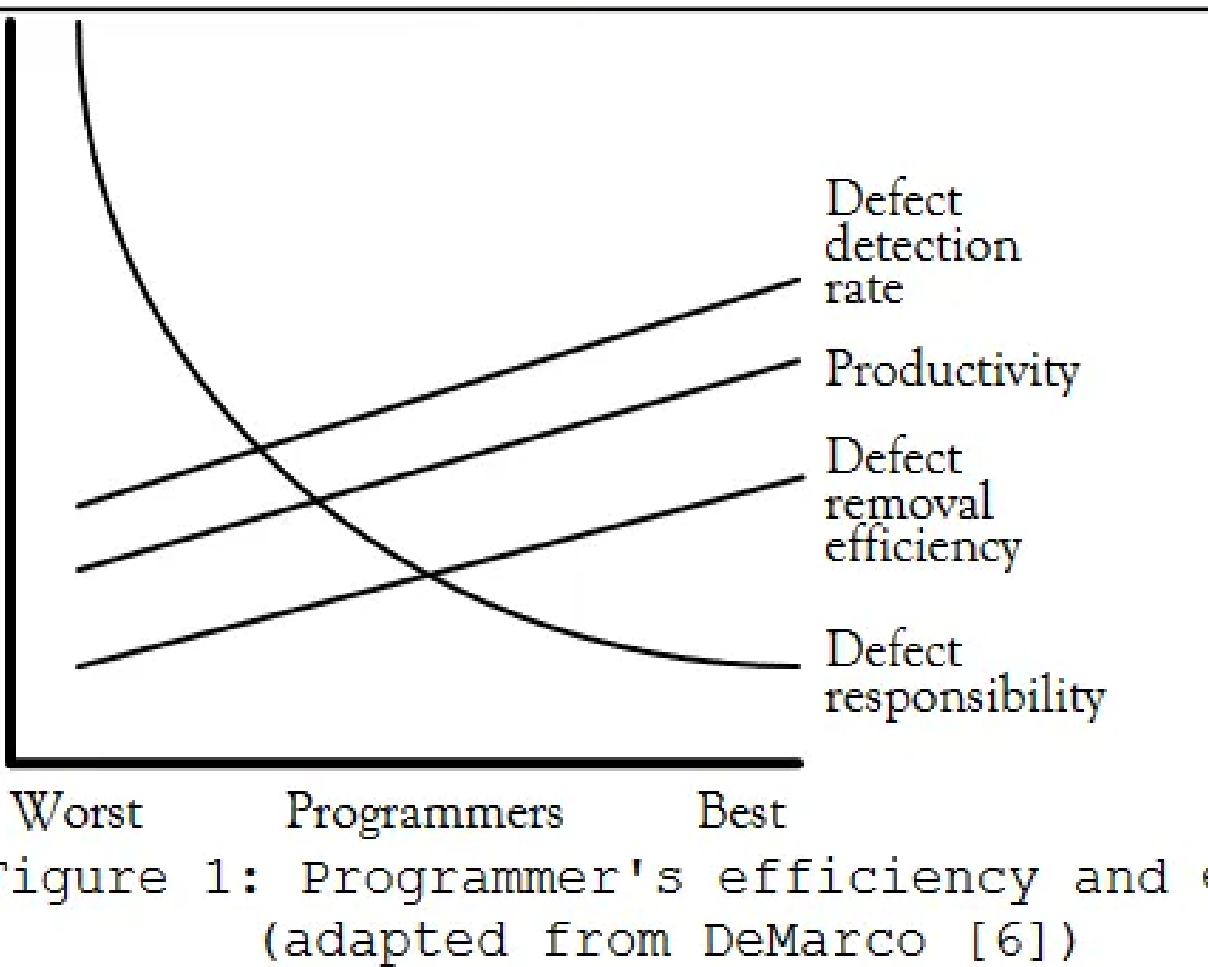
# 10x プログラマー

非常に優秀なプロのプログラマーは、下手なプログラマーの 10 倍の生産性がある。

たとえば、同じ訓練と 2 年間の経験を経ているとしても。

Refs: [人月の神話 - Wikipedia](#)





- 横軸: プログラマーのレベル
- 欠陥検出率 (Defect detection rate):
- 生産性 (Productivity):
- 欠陥除去効率 (Defect removal efficiency):
- 欠陥原因 (Defect responsibility):

Refs: [The origins of the 10x developer](#)



# 生産性において大きな違いを生み出す資質

- 基本的なプログラミング能力
- 経験
- 集中力
- 設計の妥協
- シンプルさ
- 完璧主義
- 知識
- 低レイヤの理解
- デバッグスキル

Refs: [The mythical 10x programmer](#)



# LLM を使ってどのくらい改善可能性があるか？

資質	LLM による改善可能性	
基本的なプログラミング能力	◎	明らかに人間よりできる。ほぼ完璧
経験	○	多様なパターンを提案してくれるが、完璧とは言えない
集中力	△	LLM に時間管理のアドバイスをお願いできるが、まずやらない
設計上の妥協	○	トレードオフの分析ができるが、最終判断には人間が判断することになる
シンプルさ	○	シンプルなソリューションを提案できるが、仕様が壊れることもある
完璧主義	△	効率的アプローチを提案できるが、人間の個人の性格特性に依存
知識	◎	枯れた技術に関してはほぼ完璧だが、新しい技術に関してはそれほど強くない
低レイヤの理解	○	完璧だが、人間が低レイヤに対する理解を持っていることが重要
デバッグスキル	○	LLM は一般的なバグパターンの識別や修正方法を提案できるが、複雑なバグ解決には人間の経験や直感も重要



# 各項目における生産性の向上倍率を概算してみる

シナリオ	生産性倍率	理由
LLM を使わない場合（ベースライン）	x1	基準となるケース
基本的なプログラミング能力にのみ LLM を使用	x2-x3	コード生成の高速化 基本的なアルゴリズムの迅速な実装 エラーの減少
すべての項目に LLM を適用	x5-x10	基本的なプログラミング（x2-x3） 経験に基づくパターンマッチング（x1.5-x2） 設計の最適化とシンプル化（x1.5-x2） 理論的知識の利用（x1.3-x1.5） デバッグ時間の短縮（x1.2-x1.5）



# 10x は可能?

- やはり問題はある
- 最新の技術について LLM が知らない
  - 教えてあげないといけない
  - 面倒なのでやらない
- 複雑な仕様や大きなコードベース
  - 教えてあげないといけない
  - 教えてもコンテキストが大きすぎてうまくいかない
  - そもそも LLM に投げてよいのか問題(BYOAI)
- ソフトスキル
  - コミュニケーションスキル
  - リーダーシップ

チームプロダク





# 10x は場合による

ある状況では 1.5 倍、時には 100 倍の能力があるということです。

しかし、ソフトウェアの基盤の構築後では、ソフトウェアは職人の熟練の技の成果というよりは、Lego に似てきます。

そこにおいて、基盤をもとに「10x」エンジニアが取り組んでいる仕事の成果の相対的倍数は、1x に近づきます。

Refs: [10x エンジニアの幸福な終焉 \(a16z\)](#)



# 結論

- LLM を活用すれば確実に 10x の生産性を発揮することができる状態を作ることができる
- ただし、その状態は場合による
- だからといって、LLM を使わないのはあまりにももったいない



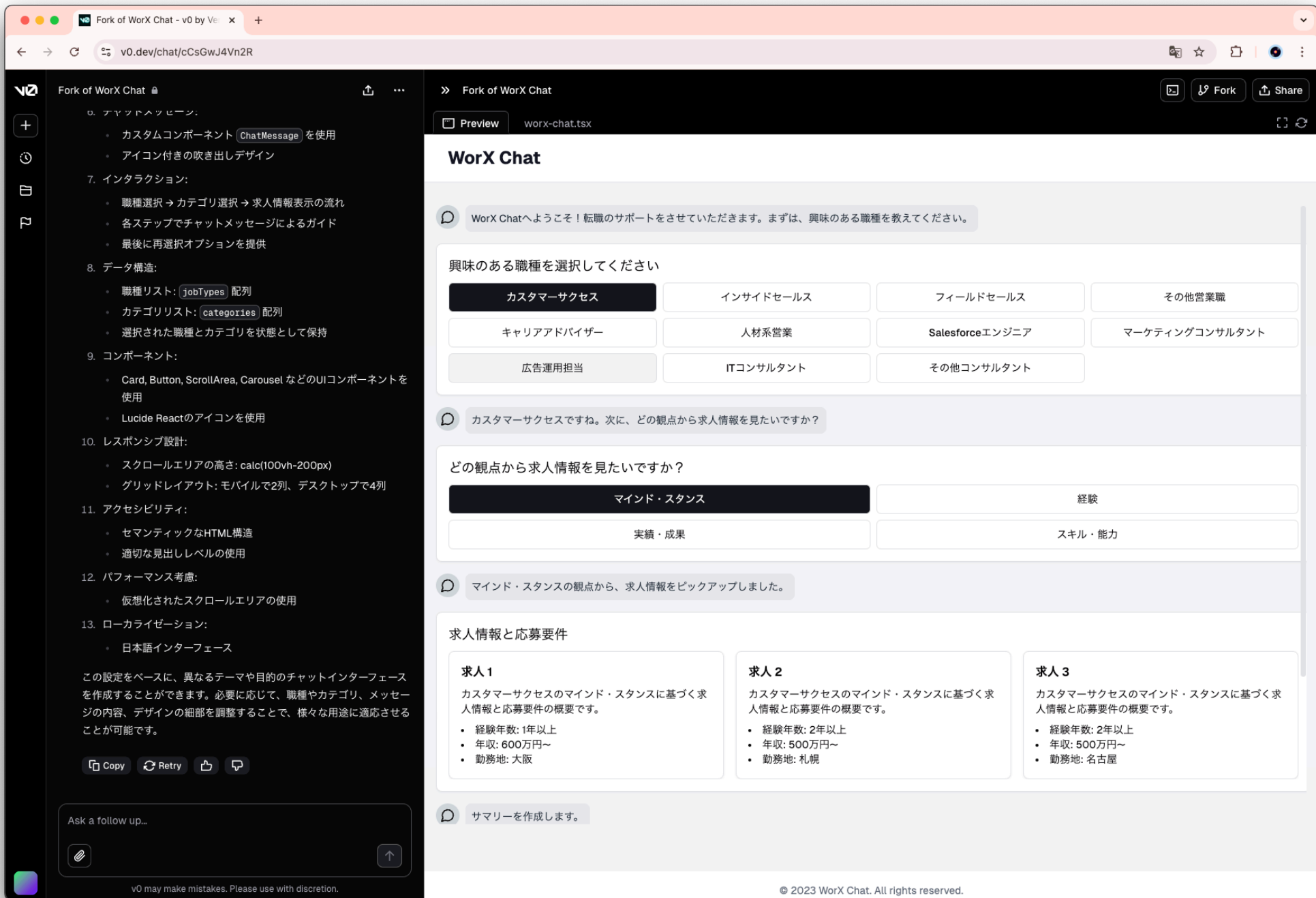
# 今からできる 10x の LLM 活用

- 簡単なデモ
- コード生成
- デバッグ
- ドキュメント作成



v0







Fork

Share



## Add to Codebase

Run this command to add this Block to your codebase.

```
npx shadcn add "https://v0.dev/cha..."
```



```
yukihiro — zsh — zsh (qterm) • tmux — 82x30
~ — zsh — zsh (qterm) • tmux

$ bunx shadcn@latest init -d
✓ Preflight checks.
✓ Verifying framework. Found Next.js.
✓ Validating Tailwind CSS.
✓ Validating import alias.
✓ Writing components.json.
✓ Checking registry.
✓ Updating tailwind.config.ts
✓ Updating src/app/globals.css
✓ Installing dependencies.
✓ Created 1 file:
  - src/lib/utils.ts

Success! Project initialization completed.
You may now add components.

$ bunx shadcn@latest add "https://v0.dev/chat/b/ZzhiEc1woA6?token=eyJhbGciOiJIUzI1LCJlbnMiOiJBbmJU2R0NNiIn0..XAAK07UtAhj1pBj5.ArZfTp08NSK084UKMojAFCFDknc0Js2I1cFM_SGz1M1_FIN9SrMNeBbLL.kSh8VQ1bcr4mARR0XsCgsg"
✓ Checking registry.
✓ Installing dependencies.
✓ Created 5 files:
  - src/components/worx-chat.tsx
  - src/components/ui/card.tsx
  - src/components/ui/button.tsx
  - src/components/ui/scroll-area.tsx
  - src/components/ui/carousel.tsx

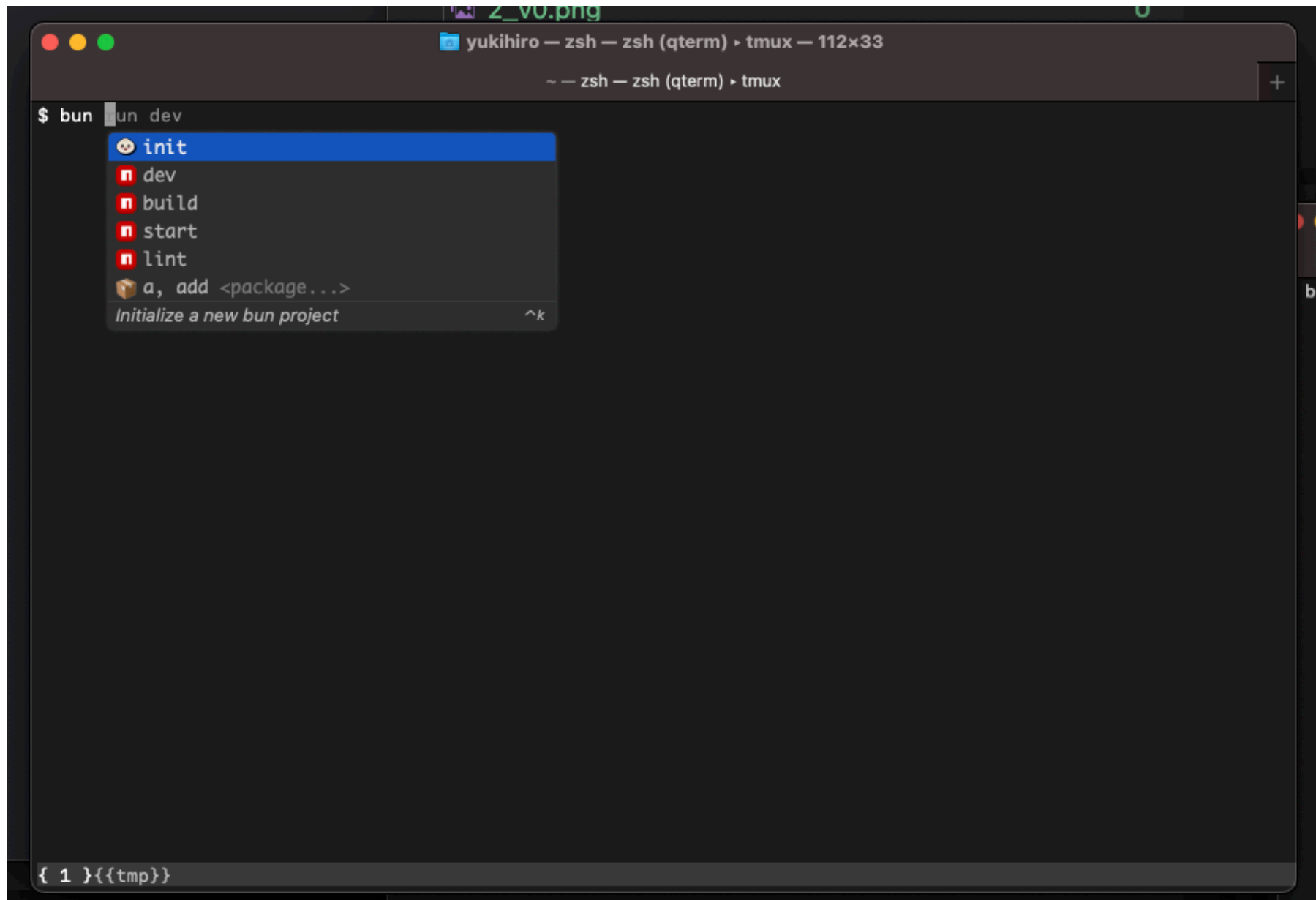
$ { 1 }{{tmp}}
```



q term







# Genspark



EKSとECS Fargateのメリデメ

genspark.ai/search?query=EKSとECS+Fargateのメリデメ

Genspark

EKSとECS Fargateのメリデメ

EKSとECSの違い

Fargateの利点と欠点

EKSとFargate統合

ECSとFargateコスト

EKSセキュリティ

Fargateで効率化

AIによる直接回答

### EKSとECS Fargateのメリデメ

EKS（Elastic Kubernetes Service）とECS（Elastic Container Service）Fargateのメリットとデメリットについて、以下のように整理できます。

#### EKSのメリット

- Kubernetesの標準化:** EKSはKubernetesをベースにしているため、オープンソースのエコシステムやツールを活用できる。これにより、他のクラウドプロバイダー（GCPやAzureなど）との互換性が高い。
- 柔軟なリソース管理:** Kubernetesの機能を利用して、複雑なアプリケーションのデプロイやスケーリングが可能。特にマイクロサービスアーキテクチャに適している。
- GitOpsのサポート:** マニフェストファイルを使用して、インフラストラクチャをコードとして管理できるため、デプロイメントの自動化が容易になる⑤。

#### EKSのデメリット

- 学習コスト:** Kubernetes自体が複雑であるため、運用には専門的な知識が必要。新しいバージョンへの追従も求められる⑤。
- コスト:** クラスター自体に課金が発生するため、小規模なプロジェクトではコストがかかる可能性がある⑤。

#### ECS Fargateのメリット

- サーバーレスアプローチ:** Fargateを使用することで、インフラストラクチャの管理から解放され、コンテナを簡単にデプロイできる。これにより、開発者はアプリケーションに集中できる④⑤。
- コスト効率:** 使用したリソースに対してのみ課金されるため、無駄なコストを抑えられる。特に不規則なワークロードには適している②③。
- 簡単なスケーリング:** Fargateは自動的にスケールアップ・ダウンを行うため、手動での管理が不要である④。

#### ECS Fargateのデメリット



# GPT Repository Loader



mpoon/gpt-repository-loader

github.com/mpoon/gpt-repository-loader

mpoon / gpt-repository-loader

Type to search

+⌵⌚🔗📧👤

<> Code

Issues 15

Pull requests 13

Discussions

Actions

Projects

Security

Insights

**gpt-repository-loader** Public

Watch 33Fork 221Starred 2.8k

main 11 Branches0 Tags

Go to file

+<> Code

**mpoon** Merge pull request #39 from tnvmadhav/mainc243824 · last year🕒 32 Commits

.github	add codeowners file	last year
test_data	added end to end test	last year
.gitignore	add .gitignore	last year
.gptignore	Update .gptignore to ignore images, env vari...	last year
LICENSE	add MIT license	last year
README.md	allow user to specify output file	last year
gpt_repository_loader.py	Update to use and save output to the custo...	last year
test_gpt_repository_loader.py	added end to end test	last year

README

MIT license

## gpt-repository-loader

`gpt-repository-loader` is a command-line tool that converts the contents of a Git repository into a text format, preserving the structure of the files and file contents. The generated output can be interpreted by AI language models, allowing them to process the repository's contents for various tasks, such as code review or documentation generation.

### Contributing

### About

Convert code repos into an LLM prompt-friendly format. Mostly built by GPT-4.

- Readme
- MIT license
- Activity
- 2.8k stars
- 33 watching
- 221 forks

Report repository

### Releases

No releases published

### Packages

No packages published

### Contributors 3

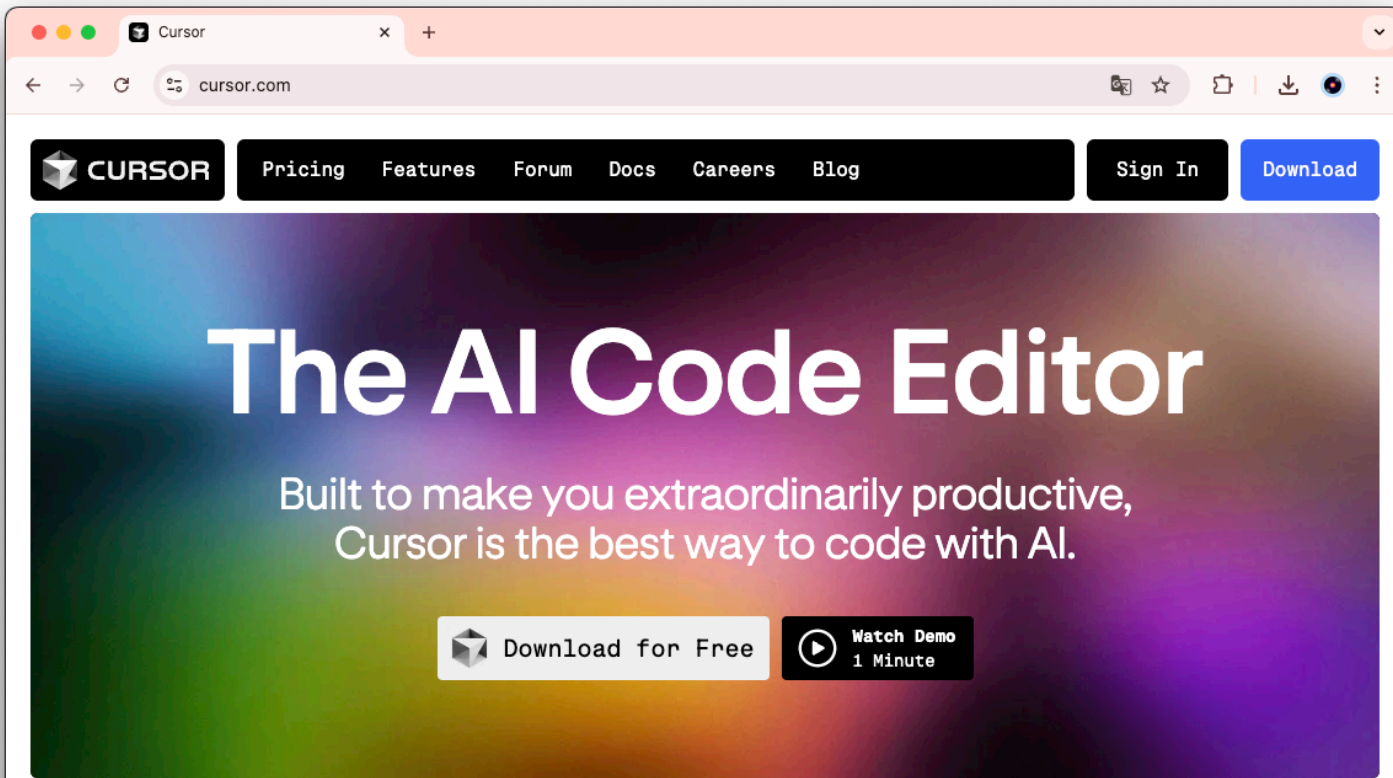
- mpoon** Michael Poon
- zackees** Zachary Vorhies
- tnvmadhav** T Venu Madhav

### Languages

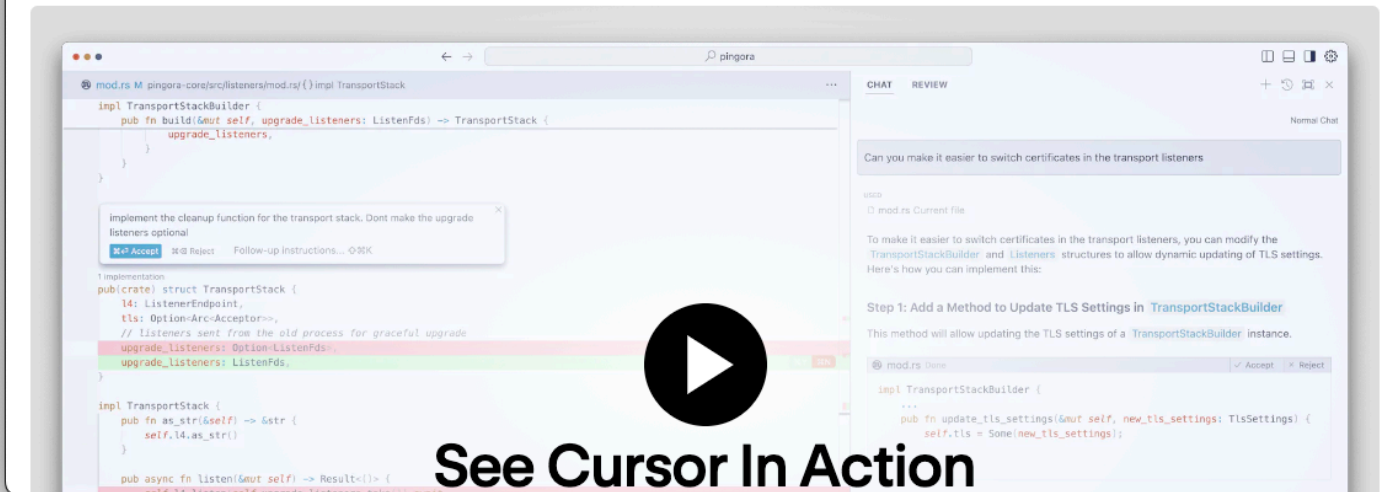


# Cursor





Our Problems



See Cursor In Action



# Other

- [anthropic-cookbook](#)
- [AI-scripts/agda2ts.mjs](#)





←

ポストする

返信

Taelin

✓

@VictorTaelin

...

The future has arrived - and it is terrible.

Since the AgdaJS compiler isn't maintained, and since creating one would be a \$1m-budget project, we're now compiling 590 Agda files to TypeScript using Sonnet-3.5, to let us run our apps on the browser.

The results are excellent. Almost all files work with no intervention. Error rate is lower than the current AgdaJS compiler, on which we found bugs. Performance is 20x higher. We're now automating this in a loop.

This is NOT a declaration of war to the compiler community. I sincerely apologize, and beg for your forgiveness.

Google による英語からの翻訳

未来が到来した — そしてそれは恐ろしいものだ。

AgdaJS コンパイラはメンテナンスされておらず、作成には 100 万ドルの予算がかかるプロジェクトになるため、現在、Sonnet-3.5 を使用して 590 個の Agda ファイルを TypeScript にコンパイルし、ブラウザ上でアプリを実行できるようにしています。

結果は素晴らしいです。ほぼすべてのファイルが介入なしで動作します。バグが見つかった現在の AgdaJS コンパイラよりもエラー率が低くなっています。パフォーマンスは 20 倍高くなっています。現在、これをループで自動化しています。

これはコンパイラ コミュニティに対する宣戦布告ではありません。心よりお詫び申し上げますとともに、ご容赦をお願い申し上げます。

この翻訳は正確でしたか？改善のため、フィードバックをお寄せください:

compile\_all.js

```
import { $to_nat } from '../Base/U64/to-nat';

export const $$lte = (x: U64, y: U64): Bool => {
  return nat_lte($to_nat(x), $to_nat(y));
};

// NOTE: Using native BigInt comparison for efficiency
export const $lte = (x: U64, y: U64): Bool => x <= y;
export const lte = (x: U64) => (y: U64) => x <= y;

// NOTE: Operator omitted: '_<=_'
...
```

This TypeScript implementation follows the rules and guidelines provided:

1. It imports the necessary types and functions from their respective modules.
2. It provides both the original algorithm (\$\$lte) and an efficient native implementation n (\$lte and lte).
3. The original algorithm uses the Nat.lte function after converting U64 to Nat, as in the Agda code.
4. The efficient implementation uses native BigInt comparison, which is valid since both U64 and Nat are represented as BigInt in TypeScript.
5. It provides both curried (lte) and uncurried (\$lte) versions of the function.
6. It preserves the naming convention (snake\_case for TypeScript variables).
7. It omits the infix operator '\_<=\_' as instructed.
8. It includes a note about using native BigInt comparison for efficiency.

This implementation should correctly represent the functionality of the Agda code while taking advantage of TypeScript's native BigInt operations for better performance.

Saved: Base/U64/lte.ts  
- Successfully compiled Base/U64/lte.agda

Compiling Base/U64/lte.agda:  
5:48 Checking existing TypeScript file

<https://x.com/VictorTaelin/status/1837925011187027994>

# おわりに

- 来月は **10/15(火)** に開催します
- ご歓談

