17

並列 Claude のチームで C コンパイラを作る

— Building a C compiler with a team of parallel Claudes —

公開日	2026-02-05
原題	Building a C compiler with a team of parallel Claudes
著者	Anthropic Engineering Team
原文	https://www.anthropic.com/engineering/building-c-compiler
翻訳	Claude（機械翻訳／Anthropic）
編集	2026-04-09

並列 Claude のチームで C コンパイラを作る

執筆: Nicholas Carlini、Safeguards チームの研究者。

私は、私たちが「エージェントチーム（agent teams）」と呼ぶ、言語モデルを監督する新しいアプローチを実験してきました。

エージェントチームでは、複数の Claude インスタンスが共有コードベース上で、人間の能動的介入なしに並列に働きます。このアプローチは、LLM エージェントで達成可能な範囲を劇的に広げます。

これをストレステストするため、16 エージェントに Linux カーネルをコンパイルできる Rust ベースの C コンパイラをスクラッチから書かせました。およそ 2,000 回の Claude Code セッションと API コスト $20,000 で、エージェントチームは x86、ARM、RISC-V で Linux 6.9 をビルドできる 10 万行のコンパイラを生み出しました。

このコンパイラはそれ自体が興味深いアーティファクト ですが、ここでは長時間の自律エージェントチーム用のハーネスを設計することについて学んだことに焦点を当てます——人間の監督なしにエージェントを軌道に保つテストの書き方、複数エージェントが並列に進捗を出せるように作業をどう構造化するか、そしてこのアプローチがどこで天井を打つか。

長時間の Claude を可能にする

Claude Code のような既存のエージェント足場は、オペレーターがオンラインで共働できることを要求します。長く複雑な問題の解を求めると、モデルは一部を解くかもしれませんが、結局は続きの入力——質問、ステータス更新、明確化の要求——を待って停止します。

持続した自律的な進捗を引き出すため、私は Claude をシンプルなループに置くハーネスを作りました（Ralph-loop を見たことがあるなら、これに見覚えがあるはず）。1 タスクを終えると、即座に次を拾います。*（実際のマシンではなく、コンテナで走らせること。）*

```
#!/bin/bash

while true; do
    COMMIT=$(git rev-parse --short=6 HEAD)
    LOGFILE="agent_logs/agent_${COMMIT}.log"

    claude --dangerously-skip-permissions \
           -p "$(cat AGENT_PROMPT.md)" \
           --model claude-opus-X-Y &> "$LOGFILE"
done
```

エージェントプロンプトでは、解くべき問題を伝え、問題を小さなピースに分解し、何に取り組んでいるかを追跡し、次に何に取り組むかを判断し、完璧になるまで効果的に走り続けるよう求めます。(この最後の点について、Claude に選択肢はありません——ループは永遠に走ります。もっとも 1 度、Claude が誤って `pkill -9 bash` をして自分自身を殺しループを終えるのを見ました。おっと!)

Claude を並列に動かす

複数インスタンスを並列に動かすことは、単一エージェントハーネスの 2 つの弱みに対処できます。

- 1 つの Claude Code セッションは一度に 1 つのことしかできません。プロジェクトの範囲が広がると特に、複数の問題を並列にデバッグする方がはるかに効率的です。
- 複数の Claude エージェントを動かすと専門化が可能になります。いくつかのエージェントが目下の問題解決を担う一方で、専門エージェントが(たとえば)ドキュメントの保守、コード品質の監視、専門サブタスクの解決を行えます。

並列 Claude の私の実装は最小限です。新しいベア git リポジトリを作り、各エージェント用に Docker コンテナを立ち上げ、リポジトリを `/upstream` にマウントします。各エージェントはローカルコピーを `/workspace` にクローンし、終わると自分のローカルコンテナから upstream にプッシュします。

2 つのエージェントが同じ問題を同時に解こうとするのを防ぐため、ハーネスはシンプルな同期アルゴリズムを使います。

1. Claude は `current_tasks/` にテキストファイルを書いてタスクに「ロック」を取る(たとえばあるエージェントが `current_tasks/parse_if_statement.txt` をロックし、別のエージェントが `current_tasks/codegen_function_definition.txt` をロックする)。2 つのエージェントが同じタスクを取ろうとすると、git の同期により 2 つ目のエージェントは別を選ばざるを得なくなる。
2. Claude はタスクに取り組み、その後 upstream から pull し、他のエージェントからの変更をマージし、自分の変更を push し、ロックを取り除く。マージ衝突は頻発するが、Claude はそれを解くのに十分賢い。

3. 無限エージェント生成ループが新しいコンテナで新しい Claude Code セッションを spawn し、サイクルが繰り返される。

これはごく初期のリサーチプロトタイプです。エージェント間の通信手段を他に実装していませんし、高レベル目標を管理するプロセスを強制してもいません。オーケストレーションエージェントも使いません。

代わりに、各 Claude エージェントがどう行動するかは任せます。ほとんどの場合、Claude は「次に最も明白な」問題を拾います。バグで行き詰まったとき、Claude はしばしば失敗したアプローチと残タスクの進行中ドキュメントを維持します。プロジェクトの git リポジトリ では、履歴を辿って、エージェントが様々なタスクにロックを取っていく様子を読めます。

Claude エージェントチームでのプログラミングから学んだこと

足場は Claude をループで走らせますが、そのループが役に立つのは Claude が進捗の仕方を分かる場合だけです。私の労力のほとんどは、Claude 周りの環境——テスト、環境、フィードバック——を設計することに費やされ、それによって Claude は私なしで状況を把握できました。これが複数の Claude インスタンスをオーケストレーションする際に最も役立ったアプローチです。

極めて高品質のテストを書く

Claude は私が与えたどんな問題でも自律的に解くために働きます。ですから、タスクの verifier がほぼ完璧であることが重要です——さもなければ Claude は誤った問題を解いてしまいます。テストハーネスの改善は、高品質のコンパイラテストスイートを見つけ、オープンソースソフトウェアパッケージの verifier とビルドスクリプトを書き、Claude が犯したミスを観察し、その失敗モードを特定しながら新しいテストを設計することを必要としました。

たとえばプロジェクトの終盤近く、Claude は新機能を実装するたびに既存機能を壊すようになりました。これに対処するため、継続的インテグレーションパイプラインを作り、Claude が自分の仕事をよりよくテストできるようにより厳格な強制を実装しました。これにより、新しいコミットが既存コードを壊せなくなりました。

Claude の立場で考える

このテストハーネスを書いているのは自分のためではなく Claude のためだと、常に自分に思い出させる必要がありました。つまり、テストがどう結果を伝えるべきかについての多くの前提を見直すことでした。

たとえば、各エージェントは文脈なしで新しいコンテナに落とされ、特に大きなプロジェクトでは状況把握にかなりの時間を使います。テストに到達する前に、Claude が自分を助けられるよう、現在のステータスを頻繁に更新する詳細な README と progress ファイルを維持する指示を含めました。

また、言語モデルには固有の制限があり、この場合はそれを設計で回避する必要があるということも念頭に置きました。以下のようなものです。

- **コンテキストウィンドウの汚染**: テストハーネスは数千バイトの無用な出力を表示すべきではありません。せいぜい数行の出力を表示し、すべての重要情報はファイルにログするべきで、Claude が必要なときに見つけられるようにします。ログファイルは自動処理しやすくあるべきです——エラーがあれば、Claude は `ERROR` を書き、理由を同じ行に置いて grep で見つけられるようにすべきです。Claude が再計算しなくて済むよう、集約サマリー統計を事前計算しておくのも有効です。
- **時間感覚の欠如**: Claude は時刻を知ることができず、放っておくと進捗を作る代わりに何時間もテストを走らせることを喜んでやってしまいます。ハーネスは（文脈を汚染しないよう）漸進的な進捗を稀にしか表示せず、1% または 10% のランダムサンプルを走らせるデフォルトの `--fast` オプションを含みます。このサブサンプルはエージェントごとに決定論的ですが VM を横断するとランダムで、Claude は全ファイルをカバーしつつ各エージェントが回帰を完璧に特定できます。

並列性を簡単にする

失敗している独立したテストがたくさんあれば、並列化は簡単です——各エージェントが異なる失敗テストを選んで取り組みます。テストスイートが 99% 合格率に達した後、各エージェントは異なる小さなオープンソースプロジェクト（例: SQLite、Redis、libjpeg、QuickJS、Lua）をコンパイルさせる作業に取り組みました。

しかしエージェントが Linux カーネルのコンパイルを始めると、行き詰まりました。数百の独立したテストを持つテストスイートと違い、Linux カーネルのコンパイルは 1 つの巨大なタスクです。各エージェントが同じバグに当たり、そのバグを直し、互いの変更を上書きしました。16 エージェントが走っていても、それぞれが同じタスクを解こうとして行き詰まっているので、助けになりません。

修正は、オンラインの既知良好なコンパイラオラクルとして GCC を使って比較することでした。新しいテストハーネスを書き、カーネルの大半を GCC でランダムにコンパイルし、残りのファイルだけを Claude の C コンパイラでコンパイルしました。カーネルが動けば問題は Claude のファイルサブセットの中にはなく、壊れたらこれらのファイルのさらに一部を GCC で再コンパイルして絞り込めます。こうして各エージェントは並列に働き、異なるファイルの異なるバグを直して、やがて Claude のコンパイラがすべてのファイルをコンパイルできるようになりました。（これが動いた後も、単独では動くが一緒だと失敗するファイルのペアを見つけるために、delta debugging の手法を適用する必要がありました。）

複数のエージェント役割

並列性は専門化も可能にします。LLM が書いたコードは既存機能を再実装することが多いので、あるエージェントには発見した重複コードを統合するタスクを与えました。別のエージェントにはコンパイラ自体の性能改善を、3 つ目には効率的なコンパイル後コードの出力を担当させました。別のエージェントには Rust 開発

者の視点からプロジェクト設計を批判し、全体のコード品質を改善する構造的変更を加えるよう依頼し、さらに別のエージェントにはドキュメントを担当させました。

エージェントチームの限界をストレステストする

このプロジェクトは能力ベンチマークとして設計しました。私は、モデルが将来確実に達成することに備えるため、LLM が今日 *ぎりぎり* 達成できるものの限界をストレステストすることに興味があります。

私は Claude 4 モデルシリーズ全体のベンチマークとしてこの C コンパイラプロジェクトを使ってきました。以前のプロジェクトと同様に、まず望むもののドラフトを書きました——依存なしのスクラッチからの最適化コンパイラ、GCC 互換、Linux カーネルをコンパイルできる、複数バックエンドをサポートするよう設計されている。設計のいくつかの側面（たとえば複数の最適化パスを可能にする SSA IR を持つべきこと）は指定しましたが、どう実現するかの詳細は指定しませんでした。

以前の Opus 4 モデルはかろうじて機能するコンパイラを生成できる程度でした。Opus 4.5 は、大きなテストスイートを通せる機能するコンパイラを生成できるしきい値を越えた最初のモデルでしたが、それでも実際の大きなプロジェクトをコンパイルすることはできませんでした。Opus 4.6 での私の目標は、再び限界をテストすることでした。

評価

2 週間にわたる約 2,000 回の Claude Code セッションで、Opus 4.6 は 20 億入力トークンを消費し、1.4 億出力トークンを生成しました。総コストは $20,000 弱。最高額の Claude Max プランと比べても、これは極めて高価なプロジェクトでした。しかしその合計は、自分自身で生み出すのにかかるコストの一部に過ぎません——ましてやチーム全体で行う場合に比べれば。

これはクリーンルーム実装でした（Claude は開発中いかなる時点でもインターネットアクセスを持ちませんでした）。依存は Rust 標準ライブラリのみです。10 万行のコンパイラは x86、ARM、RISC-V で起動可能な Linux 6.9 をビルドできます。QEMU、FFmpeg、SQLite、Postgres、Redis もコンパイルでき、GCC torture test suite を含む多くのコンパイラテストスイートで 99% の合格率を持ちます。開発者の究極のリトマステストも通ります——Doom をコンパイルして実行できるのです。

ただしコンパイラには制限もあります。

- Linux をリアルモードから起動するのに必要な 16 ビット x86 コンパイラを持ちません。これには GCC を呼びます（x86_32 と x86_64 コンパイラはそれ自身のもの）。
- 独自のアセンブラとリンカを持ちません。これらは Claude が自動化を始めた最後の部分で、まだ少しバグがあります。デモ動画は GCC アセンブラとリンカで作られました。

- コンパイラは多くのプロジェクトをビルドできますが、すべてではありません。本物のコンパイラのドロップイン置き換えにはまだなっていません。
- 生成コードはあまり効率的ではありません。すべての最適化を有効にしても、GCC ですべての最適化を*無効*にしたものより効率が悪いコードを出力します。
- Rust コードの品質は妥当ですが、熟練 Rust プログラマが生み出すであろう品質にはまったく及びません。

結果のコンパイラは、Opus の能力の限界にほぼ達しました。私は上記の制限のいくつかを直そうと（必死に）試みましたが、完全には成功しませんでした。新機能とバグ修正は既存機能を壊すことが頻発しました。

特に難しかった例として、Opus は 16 ビットリアルモードで起動するのに必要な 16 ビット x86 コードジェネレータを実装できませんでした。コンパイラは 66/67 オペコードプレフィックスで正しい 16 ビット x86 を出力できますが、結果のコンパイル出力は 60kB 超で、Linux が強制する 32kB のコード制限を大きく超えます。代わりに Claude はここで単純に手抜きをして、このフェーズに GCC を呼び出します（これは x86 だけの話です。ARM や RISC-V なら Claude のコンパイラは完全に自力でコンパイルできます）。

コンパイラのソースコードは公開しています。ダウンロードして、コードを読み、お気に入りの C プロジェクトで試してみてください。言語モデルに何ができるかを理解する最良の方法は、限界まで押して、どこで崩れ始めるかを研究することだと一貫して感じています。今後の日々、制限への対処を続ける Claude の試みを追いたければ、Claude に新しい変更を push させ続けます。

今後の展望

言語モデルの世代が変わるごとに、新しい働き方が開きます。初期のモデルは IDE のタブ補完に有用でした。すぐに、モデルは docstring から関数本体を完成させられるようになりました。Claude Code のローンチはエージェントを主流に持ち込み、開発者が Claude とペアプログラミングできるようにしました。しかし、これら製品はすべて、ユーザーがタスクを定義し、LLM が数秒〜数分走って答えを返し、その後ユーザーがフォローアップを提供するという前提で動いています。

エージェントチームは、複雑なプロジェクト全体を自律的に実装する可能性を示します。これにより、これらのツールのユーザーとして、私たちはゴールに関してより野心的になれます。

私たちはまだ初期にいて、完全自律開発には実際のリスクが伴います。人間が開発中に Claude のそばにいれば、一貫した品質を保証し、リアルタイムでエラーを捕まえられます。自律システムでは、テストが通ったのを見て仕事が終わったと思い込みやすいですが、実際にはそうでないことが多いです。私は以前ペネトレーションテストで働き、大企業が生み出した製品の脆弱性を悪用していましたが、プログラマが自分で検証したことのないソフトウェアをデプロイするという考えは本物の懸念です。

ですから、この実験は私を興奮させる一方で、不安にもさせます。このコンパイラを作るのは最近で最も楽しかったことの 1 つでしたが、2026 年初頭でここまで可能になるとは予想していませんでした。言語モデルと、対話する足場の両方の急速な進歩は、膨大な量の新しいコードを書くドアを開きます。ポジティブな応用がネガティブなものを上回ると予想していますが、安全に navigating するには新しい戦略を要する新世界に入りつつあります。

謝辞

Josef Bacik、Edwin Chen、Bernardo Meurer Costa、Jake Eaton、Dan Kelley、Felix Klock、Jannet Park、Steve Weis、そして Anthropic 全体の多くの方々からの支援と貢献に特別に感謝します。