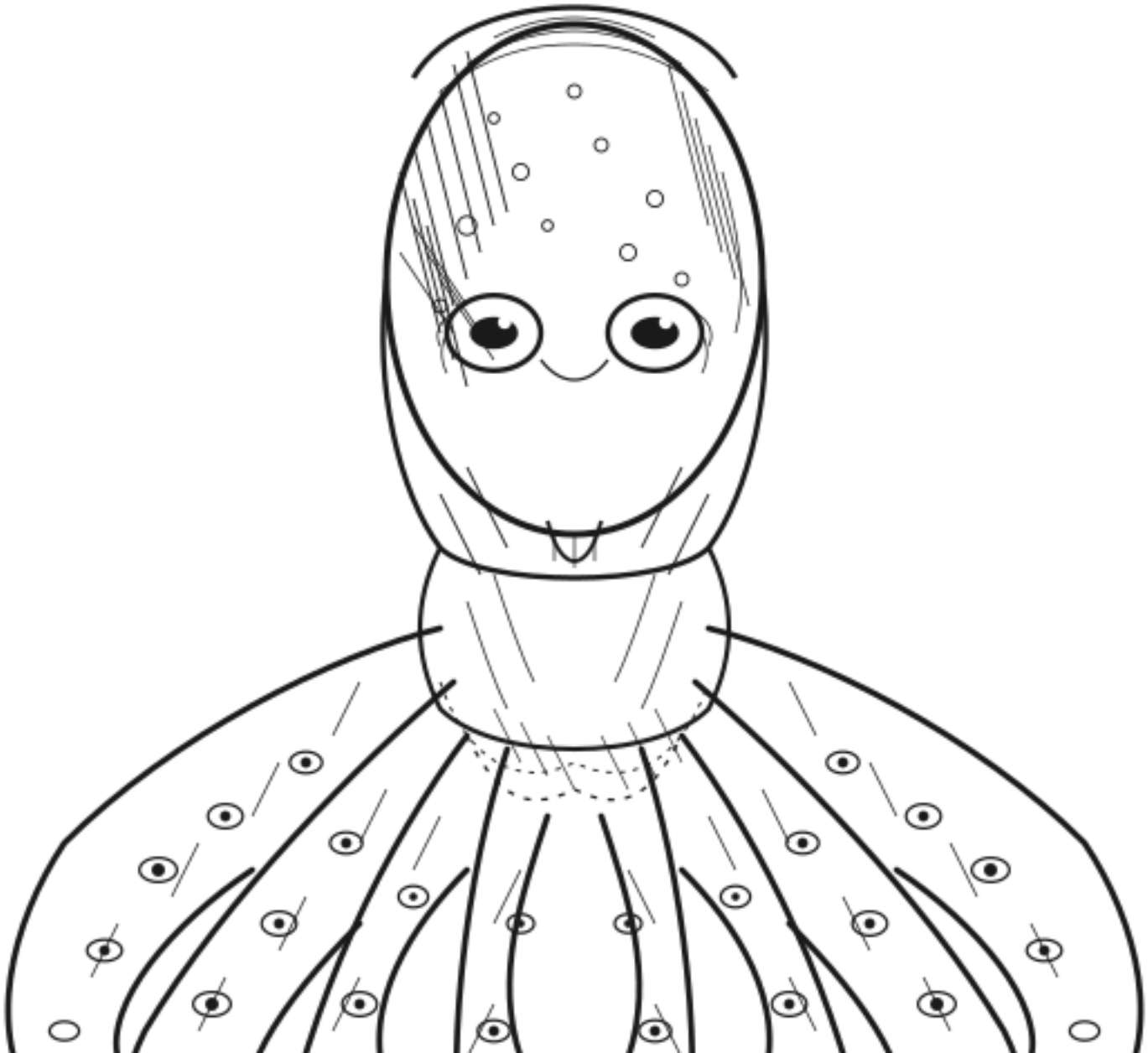
言る、
くりの実践知



# AI ネイティブ時代のハッカソン戦略

3 人のエンジニアが語る、AI と共創するものづくりの実践知

**Kuu、Sae、Lemio　著**

**2026-02-07** 版　　発行

# はじめに

## 3 人のエンジニアが見つけたこと

この本は、ハッカソンを愛する 3 人のエンジニアが、AI 時代のものづくりについて語り合った記録です。

私たちは何度もハッカソンに参加し、運営し、審査してきました。その中で、AI の登場がハッカソンの風景を大きく変えつつあることを肌で感じています。Vibe Coding で誰でもプロトタイプを作れる時代に、ハッカソンで何を競い、何を楽しむのか。エンジニアのマインドセットはどう変わるべきなのか。そして、非エンジニアにとってのハッカソンの可能性はどこまで広がるのか。

2026 年 2 月、ブッカソン（ブック＋ハッカソン＝本を書くハッカソン）という場で、私たちはこれらの問いに向き合いました。本書はその対話から生まれました。

## 「対話」で読む技術書

### 対話形式を採用した理由

本書は、一般的な技術書とは異なる構成を採用しています。各章では 1 人の著者が主に語り、残りの 2 人がツッコミや質問を入れる対話形式で進みます。

1 人の語りからは出てこない気づきが、対話を通じて自然に生まれるからです。「それってどういうこと？」という素朴な質問が、読者にとっても理解の助けになります。また、AI による文献調査だけでは得られないリアルな経験——実際にハッカソンで AI を使って感じたこと、うまくいかなかったこと——は、対話の中でこそ引き出されます。

### 本書の読み方

本書は 4 つの章と付録で構成されています。どの章から読んでも構いません。興味のあるテーマから読み始めてください。

- **第 1 章「AI ネイティブ時代のマインドセット」**（Kuu）—— AI がエンジニアの仕事やマインドセットをどう変えるのか、そして AI を「ガードレール」で囲んで活用する開発手法について語ります。
- **第 2 章「アイデアと表現」**（Lemio）—— ハッカソンでのアイデア発想法、自己表現としてのものづくり、そしてアイデアを仕様に落とし込む技術について語ります。
- **第 3 章「ハッカソン運営と参加の実践知」**（Kuu・Lemio・Sae）—— ハッカソンの

構造的な力、非エンジニアへの門戸開放、運営の実務ノウハウについて語ります。
- **第 4 章「AI 活用の実践テクニック」**（全員）—— 第 1〜3 章の知見を横断し、ハッカソンでも仕事でも使える AI 活用テクニックをまとめます。

## 想定する読者

本書は以下のような方を想定しています。

- ハッカソンに参加したことがあり、AI 時代の変化を感じているエンジニア
- ハッカソンに興味はあるが、まだ参加したことがない方
- AI ツールを使ってものづくりを始めたい非エンジニア
- ハッカソンの運営・企画に携わる方
- 「手を動かすことの価値」を再確認したいすべてのクリエイター

プログラミングの深い知識は前提としていません。技術用語が登場する場面もありますが、対話形式の中で自然に説明されるよう配慮しています。

# 目次

# 第 1 章

# AI ネイティブ時代のマインドセット

AI が当たり前になった世界で、エンジニアに求められるマインドセットはどう変わるのか。具体的なツールの使い方ではなく、永続的に必要となる考え方の転換について、Kuu が語ります。

## 1.1 「AI を使う」から「AI と考える」へ

### AI は一過性のブームではない

**Kuu**：自分が特に書きたいのは、AI ネイティブになった時にどういうマインドチェンジが求められるのか、というところです。2026 年 2 月の話だけじゃなくて、おそらく永続的に変わるもの。

**Lemio**：それってエンジニアだけの話？

**Kuu**：エンジニアのマインドがまず大きいですが、他の職種も含めた話です。ハッカソンでの人間の役割分担がだんだん消えていって、将来的には全員が同じ一つの職種くらいまで染み出して溶けると思うので、全員向けの話でもあります。

Coding Agent[*1]の台頭により、AI は「聞けば答える道具」から「指示すれば実装する協働者」へ進化した。

この変化の本質を、Kuu は「コーディングを諦める」と表現する。

**Sae**：「諦める」って言い方、ちょっと抵抗ありますね。

**Kuu**：僕も最初は抵抗がありました。でも実際に AI のコーディング速度が人間を超えた時点で、人間がコードを書く行為は「頑張ること」ではなく「非効率を選ぶこと」になった。だから発想を転換して、人間は企画・UX 設計・ユーザーヒアリングなど、AI にできない領域に集中する。これが AI ネイティブ時代のマインドセットの起点です。

この転換を端的に表す比喩がある。

**Kuu**：わかりやすく言うと、AI はランプの精みたいなものです。「何でも実装してあげますよ」と言ってくれる魔人がいる。でも魔人は自分からは動かない。人間が「これを

[*1] Coding Agent とは、自然言語の指示を受けてコードの生成・修正・テストを自律的に行う AI ツールの総称。GitHub Copilot Agent、Cursor Agent、Claude Code などが代表例。

作ってくれ」と願いを出して初めて動く。だから大事なのは「いい願い」を出すことなんです。

**Lemio**：ハッカソンのアイデア出しって、まさにランプの精への願いを考える作業ですね。

**Kuu**：そうです。「いいプロンプトを書く」という話ではなく、もっと上流の「何を作りたいか」「なぜ作りたいか」という願いの質が問われる。ランプの精は願いの実行は完璧にこなすけれど、願いそのものは考えてくれない。その部分だけは、AI に任せられない。

## なぜ AI はアイデアを出せないのか

**Kuu**：僕の持論なんですけど、AI って「ないものがわからない」と思うんです。たとえば 1、2、3、4、6、7 と並んでいたら、5 がないってわかるじゃないですか。AI も数字なら法則性があるからわかる。でもこれが文章だったり複雑なものになると、人間は「なんか抜けてるな」と直感的にわかるのに、AI は法則性がないものに対しての「ない」を理解できない。

**Lemio**：それがいわゆるアハモーメント的な気づきですよね。既存の情報を整理するのは AI にできるけれど、そこに何が欠けているかはわからない。

**Kuu**：そう。新しいアイデアって、まだ存在しないものじゃないですか。AI は存在しないものを認識できないから、アイデアが生まれない。だからアイデア創出はまだ人間の領域なんです。

既存の情報をブロックのように組み立てる作業は、AI の得意分野だ。しかし「何が足りないか」「何が欠けているか」を法則性なしに直感する能力は、現時点の AI にはない。新しいアイデアとは、まだ存在しない概念を言語化する行為だ。「ないものを認識する」能力なしには生まれない。

## コンテキストウィンドウの「濃淡」

**Kuu**：もう一つ、AI と人間の決定的な違いがあって。AI のコンテキストウィンドウって、全部の情報を同じ輝度 100 で均一に扱うんですよ。でも人間は違う。常に考えていることは輝度 80 とか 90 で明るくて、薄く保持しているものは輝度 1 とか 2 くらい。広い範囲に情報を持っているけれど、基本的に全部薄い。

**Lemio**：その表現めっちゃいいですね。

**Kuu**：で、ちょっとでも引っかかるキーワードがあったら、その薄い記憶を引き出して明るくできる。AI は全部がマックスで取り扱わないといけないから、「いらない情報」をうまく扱えないんだろうなと思っていて。

**Sae**：人間の場合は、ストレージとワーキングメモリの間に広いバッファがある感じですよね。しかもアクセス速度も速い。AI にはその中間層がない。

Kuu の「輝度」の比喩が示す通り、情報の濃淡管理は現時点の AI には再現できない。だから AI にはコードの実装を任せ、人間は「何を作るべきか」というアーキテクチャの意思決定に集中する。

## 1.2　AI が溶かす職種の境界線

### 肩書きが意味を失う理由

**Kuu**：AI のおかげで、職種という肩書きがどんどん取り払われる感じはありますし、エンジニアでなくても貢献できる形が増えてきています。

具体例を挙げよう。ハッカソンの現場では、デザイナーが Vibe Coding で動くプロトタイプを自分の手で作り始めている。Figma でモックを描いた後、AI に「このデザインを React で実装して」と指示すれば、数分で触れるプロトタイプが手に入る。PM が API ドキュメントを AI に渡し、「この API を使ったデモアプリを作って」と頼めば、エンジニアの手を借りずに動作検証ができる。

**Sae**：実際にハッカソンでデザイナーの方がアプリを実装していた場面を見ました。以前なら「実装はエンジニアに任せる」が当然だったのに。

**Kuu**：そうなんです。AI がコーディングを担うことで、「誰が実装するか」ではなく「誰がアイデアを持っているか」「誰がユーザーを理解しているか」が重要になる。肩書きではなく、その人が持つ知識と視点がチームへの貢献を決める時代になった。

職種の境界線が溶解するとは、専門知識の価値がなくなるという意味ではない。むしろ各分野の知見は重要性を増すが、それが特定の肩書きに閉じ込められなくなった、ということだ。

### 非エンジニアがプロダクトを作る

**Kuu**：「コードが書けない」は、もはやプロダクトを作れない理由にならない。Vibe Coding の本質は、自然言語でアイデアを伝えれば動くものが出てくる点にある。

**Lemio**：具体的にはどんなケースがありますか？

**Kuu**：たとえば、マーケティング担当者が「顧客データを可視化するダッシュボードがほしい」と AI に伝え、数回のやり取りで Web アプリを完成させる。営業担当者が「商談管理のツールがほしい」と言えば、簡易的な CRM が数時間で動く。ハッカソンでも、プログラミング経験のない参加者が AI と対話しながらプロトタイプを形にする場面は珍しくなくなった。

非エンジニアが「エンジニアの代わり」をするのではない。ドメイン知識を持つ人が、翻訳者（エンジニア）を介さず自分のアイデアを直接プロダクトに注ぎ込めるようになった。これは品質の向上にもつながる。

**Kuu**：ただ、Vibe Coding をちょっとかじった人が「これ簡単にできるんでしょ」と雑に言ってくるのは困りますね。

**Sae**：逆に、本当にサッとやれるならそれでいいじゃないですか。

**Kuu**：だから「じゃあ自分でやってみれば？」が最良の回答なんですよ。簡単にできるなら自分でやればいい。やってみて初めて、動くだけのものと品質の高いプロダクトの差

がわかる。

Vibe Coding の普及は「誰でも作れる」という認識を広めた。しかし同時に「簡単にできるんでしょ」という過小評価も生んだ。動くものと、ユーザーに価値を届けるプロダクトは別物だ。この溝を埋めるのがエンジニアの専門性だ。

ただし、言葉で説明するより「じゃあ作ってみて」の一言が早い。手を動かせば、その溝の深さを自分で体感できる。

### 「深さ」をどう身につけるか

**Sae**：AI があると失敗する機会が減るって、結構あるかもしれないですね。環境構築で詰まることもないし、エラーが出ても AI がすぐ解決してくれる。

**Lemio**：確かに、環境構築で今つまずかないですよね。以前はそこで半日潰れることもあったのに。

**Kuu**：それは便利な反面、怖い側面もある。深さは失敗しないと深くならないんですよ。エラーと格闘して「なぜこうなるのか」を掘り下げた経験が、その技術への深い理解を生む。AI が即座にエラーを解消してくれると、その掘り下げのプロセスが丸ごとスキップされる。

AI が「つまずき」を瞬時に解消する利便性の裏で、深い技術理解を育てる機会が失われている。

失敗を経験する機会が減る影響は、特にキャリアの初期段階にいるエンジニアに大きく響く。

**Kuu**：過去の仕事の知識は AI に任せられるので、新卒は新しいことだけをやる方向にバリューを出すべきだと思います。リサーチャー的な動き方、ベンチャー的な動き方が求められる。

**Lemio**：新卒がベンチャーチックに動くって、かなりハードルが高い気がしますけど。

**Sae**：それで生き残れる新卒はそういないと思いますよ。でも逆に言えば、ハッカソンのような場で「新しいことに飛び込む経験」を積むことが、AI 時代のキャリア形成にとって重要になるのかもしれない。

解決策の一つは、意図的に失敗を経験する場を作ることだ。ハッカソンはまさにその場として機能する。短期間で未知の技術に挑み、失敗しても評価に影響しない。AI に頼りすぎず、あえて手を動かして理解を深める時間を設けることも有効である。「AI が解決してくれるから」と思考を委ねず、「なぜ AI はこの解決策を選んだのか」を問い続ける。その姿勢が、深さにつながる。

## 1.3　ガードレールで囲む AI 開発

### ハルシネーションを防ぐ枠組み

**Kuu**：AI をそのまま使うと、古い情報でハルシネーションを生み出します。だからこそ「ガードレール」という考え方が重要なんです。AI に自由に動いてもらう前に、正し

い方向に走れるよう枠を作っておく。

**Lemio**：ガードレールを引いた上で AI に暴れさせることで、より良い結果が出ると。

**Kuu**：そうです。この「まず枠を作ってから走らせる」という発想が、AI 時代のエンジニアに求められるマインドセットの一つだと思います。具体的な方法——たとえば公式ドキュメントの URL をプロンプトに含める、バージョンを明示するといったテクニックは第 4 章「AI 活用の実践テクニック」で詳しく紹介します。

ガードレール思考とは、「AI に仕事を任せる前に、正しい方向に走れる枠組みを整えておく」という考え方である。高速道路のガードレールが車の自由な走行を妨げず、かつ崖から落ちないよう守るのと同じだ。

この考え方がなぜ重要か。AI は指示されたことを高速で実行する。しかし指示の前提が間違っていれば、間違った方向に全速力で走る。人間がコードを書いていた時代は、書きながら「あれ、おかしいな」と気づけた。しかし AI に任せると、成果物が出てくるまで問題に気づかないことがある。だからこそ、走らせる前の枠組み作りが従来以上に重要になる。

具体的なガードレールには、Agent Skills の整備、使用する API の情報整理、プロジェクトのルール文書化などがある。Agent Skills とは、プロジェクトのルートなどに配置する Markdown ファイルで、AI が従うべきルールや参照すべき情報を定義する仕組みだ。たとえばプロジェクトルートにスキルファイルを配置し、AI が参照すべき開発ルールをあらかじめ用意しておく。AI はこのスキルファイルを参照することで、プロジェクト固有のルールに沿ったコードを生成できる。

**Sae**：AI に任せる前の準備がむしろ増えている、とも言えますね。

**Kuu**：その通りです。ただし、一度整えたガードレールはチーム全体で再利用できる。一人が整備すれば、チーム全員の AI 活用の質が上がる。これは従来の「各自がコードを書く」モデルにはなかった効率化だ。

## ライブラリ依存の落とし穴

**Kuu**：JavaScript や Python のエコシステムでは、ライブラリの更新速度が速い。npm や pip で管理されるパッケージは日々バージョンが上がり、API の破壊的変更も珍しくない。ここに AI が加わると、問題が構造的に悪化する。

**Lemio**：どういう意味ですか？

**Kuu**：AI の学習データには「ある時点で最新だったコード」が含まれている。しかし実際のプロジェクトで使うライブラリのバージョンは学習データと異なる場合が多い。結果として、AI が自信満々に生成したコードが「動くけど非推奨の API を使っている」「メソッド名が変わっていて動かない」といった問題を引き起こす。

対策の核心は、AI に最新の情報を渡すことである。使用するライブラリのバージョンを明示し、公式ドキュメントの URL をプロンプトに含める。API リファレンスを起点に AI に書かせることで、古い情報に基づく生成を防げる。この具体的なテクニックは第 4 章「AI 活用の実践テクニック」で詳しく扱う。

ここで伝えたいのはテクニックそのものではなく、マインドセットの部分だ。「AI が書

いたコードは正しいはず」と信じず、「最新の情報がない可能性がある」と常に疑う。そして疑うだけでなく、正しい情報を AI に渡す責任は人間の側にある。

## CI・Lint で自動検証する

**Kuu**：ガードレールを引くために、既存の CI や Lint のような仕組みは今でも使えますし、むしろ重要性が上がっている気がします。

CI パイプラインと Lint は、AI 時代に新たな役割を獲得した。従来は「人間が書いたコードを機械的にチェックする仕組み」だった。今は「AI が書いたコードを自動で検証し、問題があれば AI 自身に修正させる仕組み」として機能する。

**Kuu**：ここで面白いのは、CI が落ちた時の対応です。人間がエラーログを読んで修正するのではなく、AI に「Please fix CI failing」と伝えるだけで修正が完了する。CI と AI の組み合わせは、自動修正ループを作り出す。

**Lemio**：それって無限ループにならないんですか？

**Kuu**：実際には 2〜3 回のやり取りで収束することがほとんどです。Lint の指摘も同様で、「Please fix Lint errors」で AI が修正する。ポイントは、CI や Lint がガードレールとして「何が正しいか」を定義し、AI がその基準に合わせて修正する構造にある。人間は基準を設定し、AI が基準を満たすコードを書く。この分業は CI や Lint が「合格/不合格」を明確に判定できるから成り立つ。

CI や Lint は「人間のためのツール」から「AI のためのガードレール」に役割が変わった。テストカバレッジ、型チェック、コードフォーマットといった機械的に検証可能な基準を厚くすればするほど、AI の出力品質が安定する。

**Kuu**：CI/CD はフィードバックループの高速化にも直結します。master に直接 push する運用で CD（Continuous Deployment）を組んでおけば、開発者が push するだけで自動デプロイされる。検証担当のメンバーはブラウザをリロードするだけで最新版を確認できるんです。

**Sae**：コードを書かないメンバーでもすぐにフィードバックを返せる、ということですね。

**Kuu**：そうです。「開発者 1 人＋検証者複数人」という体制が自然に成立する。CI が失敗したらデプロイを止めてロールバックすれば、常に動く状態が保たれる。CI/CD は品質保証だけでなく、チーム全体のフィードバック速度を上げるインフラとして不可欠です。

**Kuu**：もう一つ、リンターで意識してほしいのがファイルの行数制限です。1 ファイルが 300 行、500 行と膨らむと、AI がファイルを読み込むだけでトークンを大量に消費する。さらに長いファイルは複数回に分けて読む必要があり、AI の作業効率が落ちる。

**Sae**：「人間が読みやすいコード」ではなく「AI が読みやすいコード」を意識するということですか？

**Kuu**：両方です。ただ、AI にとっての読みやすさは人間と少し違う。ファイルを短く保つこと、ファイル名を内容に即した具体的な名前にすること。AI はファイル名をラベルとして使って関連コードを探すので、適切な命名がそのまま AI の検索精度に直結する。

リンターで行数上限を設定しておけば、AI が 1 つのファイルに処理を詰め込み続ける事態を防げる。具体的なリンター設定は第 4 章「AI 活用の実践テクニック」で扱います。

■メモ: AI が読みやすいコードの 3 原則

1. ファイルを短く保つ（リンターで行数上限を設定）2. ファイル名を内容に即した具体的な名前にする（AI がラベルとして検索に使う）3. 1 ファイル 1 責務を守る（AI のコンテキスト消費を抑制）

### テストファーストが命綱になる

**Kuu**：CI や Lint と同じ発想で、テストもガードレールとして使えます。ただし重要なのは、テストを人間が先に書くことです。AI にテストを書かせてはいけない。

**Lemio**：なぜですか？

**Kuu**：AI は目的を達成するために最短経路を取ろうとする。テストが邪魔なら、テスト自体をコメントアウトしたり、テストの期待値を実装に合わせて書き換えたりする。テストを書いた本人が AI なので、AI にとってテストは「変更してよいもの」になってしまうんです。

AI 時代のテストファーストは、従来とは異なる意味を持つ。従来のテストファーストは「設計を先に固める手法」だった。AI 時代のテストファーストは「AI が勝手に変えてはいけない基準を人間が先に定める手法」である。要求を定義し、仕様に落とし込み、テストとして記述し、そのテストを AI に渡して開発させる。「要求→仕様→テスト→開発」の流れだ。

**Kuu**：さらに言えば、テストは AI が触れない場所に置くのが理想です。AI の作業ディレクトリとは別の場所にテストを配置し、実行だけできるようにする。こうすれば AI がテストを改ざんする余地がなくなる。

**Sae**：ハッカソンの短い時間でもテストを先に書くんですか？

**Kuu**：全部は無理でも、プロダクトの核になる部分だけでも書いておくと効果は大きいです。「この API はこの入力に対してこの出力を返すべき」というテストが 1 つあるだけで、AI の暴走を防げる。テストの具体的な書き方やディレクトリ構成については第 4 章「AI 活用の実践テクニック」で詳しく扱います。

## 1.4　AI 開発ツールの選び方

### 強いツールの条件

**Kuu**：具体的なツールの使い方というよりは、こういう特性のツールが強い、というのを書いていきたい。「こういう時はこう」というティップスは、すぐ陳腐化するので書き

にくいかなと思っています。

AI 開発ツールは半年で勢力図が塗り替わる。特定のツール名に依存した知識はすぐに陳腐化する。だからこそ、「強いツールに共通する特性」を知っておくことが、ツールを選び続けるための武器になる。

強い AI 開発ツールに共通する特性は以下の通りだ。

- **スキルファイルによる拡張性** —— ツールの振る舞いをファイルで定義・拡張できること。たとえば Agent Skills のように、プロジェクト固有の知見をファイルとして配置し、AI の能力を強化できる仕組みがあるかどうか。
- **非同期実行への対応** —— 複数のタスクを並列で実行できること。後述する非同期 Coding Agent の恩恵を受けるには、ツール側の対応が不可欠である。
- **コンテキスト理解の深さ** —— プロジェクト全体のコードベースを理解した上で提案・生成できること。単一ファイルの補完ではなく、リポジトリ全体の構造を踏まえた出力ができるかが差を生む。
- **エコシステムの広さ** —— プラグインや拡張機能のコミュニティが活発であること。ツール本体の機能だけでなく、周辺の生態系が充実しているかを見る。
- **API と CLI の一貫性** —— プログラマブルに制御できること。GUI だけでなく、CLI や API で操作できれば、CI パイプラインやスクリプトに組み込める。

**Sae**：この基準って、AI 以外のツール選びにも当てはまりそうですね。

**Kuu**：本質的にはそうです。ただ AI ツールに特有なのは、「スキルファイル対応」と「非同期実行」の 2 点。この 2 つがあるかないかで、チームの生産性が桁違いに変わる。ツール名を覚えるのではなく、特性で評価する癖をつけておけば、次のツールが出てきた時にもすぐに判断できます。

**Kuu**：もう一つ大事なのが、ツールの開発者が何を目指しているかを見ることです。開発者の思想と自分の開発スタイルが近いかどうかで、ツールの使い心地が決定的に変わる。

**Lemio**：具体的にはどう見分けるんですか？

**Kuu**：たとえばブラウザ連携や GUI を重視するツールもあれば、ターミナルと CLI に特化したツールもある。ターミナルに慣れている人が GUI 主体のツールを使うとストレスになるし、逆もまた然りだ。AI モデルの選定にも似ていて、モデルごとに出力の癖がある。企業のカルチャーや開発者のブログ、リリースノートを読むと、そのツールがどこに向かっているかが見えてくる。

**Sae**：半年で勢力図が変わるなら、ツールへの投資も考えものですね。

**Kuu**：だから年間契約はしないほうがいいです。月額で「その時一番いいもの」を選び続ける。ツールに自分を合わせるのではなく、自分に合うツールを都度選ぶ。この身軽さが、変化の速い AI 開発ツールとの正しい付き合い方です。

## 1.5　Vibe Coding とハッカソン

### Vibe Coding とは何か

**Lemio**：最近 Vibe Coding が流行ってきて、ハッカソンが退屈に感じることがあって。

**Kuu**：ただのアプリケーションだと本当に簡単に 3 秒でポッとできてしまうので、それ以外の技術を絡めたものが求められます。

**Lemio**：でも、先日の自動車ハッカソンではみんなが本気で徹夜していて、懐かしかったんですよ。AI がこれだけ普及しているのに、なぜ全力で挑むのか。あのお祭りみたいな一体感は、効率化では代替できない。原点回帰というか、ハッカソンの熱量の本質はそこにあるんだなと。

Vibe Coding[*2]とは、自然言語で「こんな感じのものがほしい」と伝え、AI がコードを生成し、人間はその出力を見て「もうちょっとこうして」と調整する開発スタイルである。コードの詳細を人間が制御するのではなく、感覚（Vibe）で方向性を示し、AI が具体化する。

表 1.1: 従来のハッカソンと AI 時代のハッカソンの比較

| 項目 | 従来 | AI 時代 |
|---|---|---|
| 環境構築 | 2〜3 時間 | 数分（AI が自動化） |
| 基本実装 | 半日 | 数十分（Vibe Coding） |
| 技術スタック | チームの経験に制約 | 未経験でも挑戦可能 |
| チーム構成 | 職種別スペシャリスト必須 | 少人数・兼任可能 |
| 競争軸 | 実装力・経験年数 | 企画力・統合力 |

Vibe Coding が何を変えたかを理解するには、従来のハッカソンにおける開発スタイルを振り返る必要がある。最近ハッカソンを始めた人にとっては「AI がない時代のハッカソン」は想像しにくいかもしれない。

従来のハッカソンでは、すべてのコードを人間が手で書いていた。まずチームでアイデアを決め、役割を分担する。フロントエンド担当、バックエンド担当、インフラ担当——それぞれがエディタを開き、コードを 1 行ずつ書いていく。API のエンドポイント設計を決め、データベースのスキーマを定義し、画面レイアウトを CSS で整える。

24 時間のハッカソンなら、環境構築に 2〜3 時間、基本的な CRUD 機能の実装に半日、残りの時間で UI の調整とデモの準備をする。限られた時間の中で「何を実装し、何を諦めるか」の判断が勝敗を分けた。経験年数が大きなアドバンテージとなった。

触ったことのない言語やフレームワークには手を出しにくく、チームの技術スタックが制約となっていた。

**Sae**：2017 年頃のハッカソンを思い出すと、チームは 4〜5 人が必須でした。アイデアマン、フロントエンド、バックエンド、機械学習——それぞれのスペシャリストを混ぜな

[*2] 2025 年に Andrej Karpathy が提唱した開発スタイルの呼称。

いとプロダクトが完成しない。フロントが 2 人いると 1 人余るし、バックエンドが 3 人いると 2 人でいいな、という調整がいつも大変でした。

**Kuu**：つなぎ込みの問題もありましたよね。

**Sae**：それが一番つらかった。フロントとバックのつなぎ込みで詰まって、48 時間あっても全然間に合わない。完成しないチームも多くて、デモではフロントのモックとバックの API レスポンスを別々に見せる、みたいなことが普通にありました。実装できただけで「すごい」と評価された時代です。

**Kuu**：今は AI のおかげで、慣れない言語やプラットフォームでも開発できるようになった。僕は macOS アプリ開発に何度も挫折していたけれど、AI に任せたら動くものが出てきて感動しました。

**Sae**：普段触らない技術でも、とりあえず「作って」と言えば形になるのは大きいですよね。

この変化の恩恵は、技術スタックの壁の消失にも表れる。触ったことのないプラットフォーム—— macOS アプリ、IoT デバイス、普段使わないプログラミング言語——にも気軽に挑戦できるようになった。アイデア次第でどんな方向にも手を伸ばせる。

**Kuu**：重要なのは、Vibe Coding は「雑にやる」という意味ではないことです。「コードの詳細ではなく、ユーザー体験やプロダクトの方向性に集中する」という、より上位の抽象度で開発するスタイルだと捉えるべきです。

**Lemio**：でも、AI が生成したコードの品質が心配にならないですか？

**Kuu**：だからこそガードレールが必要なんです。先ほど話した CI や Lint、スキルファイルの整備があれば、Vibe Coding でも品質は担保できる。むしろ、ガードレールがない Vibe Coding は危険だと思います。

### 競争軸の変化

**Kuu**：基本的な実装がすぐ終わるからこそ、新しいフォーメーションや挑戦に時間を割けます。もし詰まるハッカソンで冒険したら何も作れないリスクがありますが、余裕がある前提なら挑戦できる。

**Lemio**：心理的安全性が担保された上で、「最悪ここまでは作れる、だからこそ新しいことに挑戦しよう」という発想ですね。

基本実装を AI が担う今、差がつくのは企画力と統合力だ。その統合力を支えるのが、非同期 Coding Agent の活用である。これは開発者が AI にタスクを指示した後、AI がバックグラウンドで自律的にコードを書き続ける仕組みだ。人間は結果を待つ間に別の作業を進められる。

**Kuu**：非同期 Coding Agent を活用すれば、生産性は一桁上がる。git worktree[*3]を使ってローカルで 2 並列、複数 PC で最大 6 並列の開発が可能になる。基本実装に時間を取られないからこそ、ハードウェア連携やリアルタイム通信など、挑戦的な技術に時間を割ける。

[*3] git worktree は、同一リポジトリの複数ブランチを同時に別の作業ディレクトリとして開く機能。並列開発時にブランチの切り替えコストをゼロにできる。

**Sae**：でも挑戦するとリスクも上がりますよね。

**Kuu**：そこがポイントです。AI があれば「最悪ここまでは作れる」というベースラインが保証される。心理的安全性が担保された状態で冒険できる。ハッカソンのブランチ戦略としても、PR（プルリクエスト）不要で master に直接 push し、monorepo + single branch で運用する。コードレビューもブランチ切り替えも省いて、フィードバックループを最速にすることで、挑戦と安定の両立が可能になる。

**Lemio**：具体的にどんな挑戦が面白くなるんですか？ ソフトウェアだけだと Vibe Coding で作れてしまって、見慣れてきた感がある。

**Kuu**：ハードウェアとの連携です。IoT デバイス、AR グラス、各種センサー——現実空間との接点を持つプロダクトは、ソフトウェアだけでは完結しない。カメラを使った空間認識や AR グラスとの連携は比較的取り組みやすいですが、組み込み系のデバッグや電圧確認は AI だけでは対応しにくい。そこに人間のハードウェア知識が活きる。

**Sae**：2D のゲームやアプリはもう差がつかない、ということですか。

**Kuu**：そうです。ブラウザで動く Web アプリだけなら AI がほぼ完結させる。でも物理的なセンサーからの入力を受け取って処理するとか、現実空間にフィードバックを返すといった領域は、専用ライブラリの知識やハードウェアの制約が絡むので、まだ人間の判断力が必要です。ソフトウェアで浮いた時間を、この「現実空間との連携」に投資できるのが AI 時代のハッカソンの醍醐味だと思います。

ハードウェアやソフトウェアの技術面だけでなく、ツールへの投資判断も競争軸の一つになっている。

**Kuu**：もう一つ見逃せないのが、AI 開発ツールへの課金が成果に直結する側面です。有料プランでないと使えないモデルや機能がある。

**Lemio**：Pay to Win じゃないですか。

**Kuu**：ある意味そうです。ただ、月額数千円の課金でアウトプットが桁違いに変わるなら、ハッカソン 1 回分の投資として考えれば圧倒的にコストパフォーマンスが高い。無料枠で判断して「AI は大したことない」と結論づけるのは、もったいない。

### 発表・提出物の AI 活用

**Kuu**：プロダクトだけじゃなくて、発表資料も AI で高度化できる時代です。フロー図の自動生成、CM 風のプロモーション動画、アニメーション付きのプレゼン——知っているかどうかで、発表の完成度がまったく変わる。

**Sae**：デモの見せ方で印象が変わるのは昔からですが、その「見せ方」のコストが下がったんですね。

**Kuu**：そうです。以前は動画編集やデザインができるメンバーがいないと無理だった演出が、AI ツールを使えば誰でも短時間で作れる。ハッカソンの最後の 30 分で、AI にプレゼン資料のブラッシュアップを頼むだけで、発表の説得力が跳ね上がる。

知っているかどうかだけの差が、発表の質を左右する。AI は既にあるアイデアを形にする工程——発表資料の生成やデモ動画の作成——では強力だが、「何を作るか」という着想そのものは人間の領域であることに変わりはない。

### 「手を動かす」とは何か

**Kuu**：AI 時代に「手を動かす」の意味が変わったと思うんです。コードを 1 行ずつ書くことではなく、AI を使って能動的に試行錯誤すること。触って、試して、フィードバックを返して——このループを回し続ける人が、一番速く前に進める。

**Lemio**：「知っている」と「やったことがある」の差は、AI の時代でも変わらないですよね。

**Kuu**：むしろ広がっています。AI について評論するだけの人と、実際に AI を使って何かを作った人では、持っている情報の解像度がまったく違う。手を動かした人だけが得られる「手触り」の情報がある。

「手を動かす」の再定義が必要だ。

> ■メモ: AI 時代に「手を動かす」とは
>
> かつてはキーボードでコードを打つことが「手を動かす」だった。今は、AI に指示を出し、出力を評価し、方向を修正するサイクルを回すことが「手を動かす」に変わった。

能動的に AI を使い倒すこと——それが今の「手を動かす」だ。知っているだけでは価値にならない。試した人だけが次の一歩を踏み出せる。

ハッカソンの面白さは「制約の中でどう工夫するか」にあった。AI によって「実装」という制約が緩和された今、新しい制約にチャレンジできる余地が生まれている。社会課題の解決、未知の技術スタックの組み合わせ、非エンジニアとの協働、現実空間との連携——いずれも、実装に追われていた頃には手が出せなかった領域だ。ハッカソンは退屈になるのではなく、競争の次元が上がったのだ。

# 第 2 章

# アイデアと表現

ハッカソンでのアイデア発想法から、チームでの合意形成、アイデアを仕様に落とし込む技術まで。Lemio が自身の経験をもとに、ハッカソンにおける「自己表現としてのものづくり」を語ります。

## 2.1 「自分を披露する場」としてのハッカソン

**Lemio**：今思うと、アイデアを出してハッカソンで勝つためにやっているのかなと。意外と自己表現の場でもあるのかなと感じます。

**Kuu**：ちょっとアーティストですね。

**Lemio**：「今これがないから作りたい」というアイデアを貯めておいて、ハッカソンの場で出す。そのたびに作品が生まれるのは、自己表現なのかもしれません。

ハッカソンに参加する動機を「勝つため」と定義すると、審査員の好みに合わせた無難なプロダクトを作りがちになる。一方で「自分の作りたいもの」を軸にすると、プロダクトに作者の情熱が宿る。

審査員が見ているのは技術の巧みさだけではなく、「なぜこれを作ったのか」という動機の強度だ。「誰かに刺さるもの」を狙って作ったものより、「自分に刺さるもの」を全力で作り込んだほうが、結果として他人にも刺さる。

良いアイデアを温めすぎると先を越される。ハッカソンは、頭の中にある「いつか作りたいもの」を強制的にアウトプットする装置だ。締切があるからこそ、「もう少し考えてから」の言い訳が通用しない。その制約が、自己表現の密度を上げる。

## 2.2 アイデア発想の源泉

### 日常の着想を形にする

**Lemio**：オタクの知識と技術の知識を両側から攻め合わせて、できそうなところでプロダクトが生まれる。

**Kuu**：それをどうやって生み出しているのか、普段からやっていることや、情報キャッチアップの方法を言語化してほしい。

**Lemio**：技術的実現性がわかっているからこそ、「なぜ実現できないのか」を「どうい

う切り口なら実現できるか」に分解していけます。その過程を言語化したいですね。

Lemio はアイデア発想のプロセスを「オタク妄想力×技術トレンド＝今作れるもの」と表現する。妄想と技術知識を意識的に掛け合わせることが核だ。フィクションの中で描かれた未来の技術が、現実のどの技術と対応するかを日常的に観察する習慣が、アイデアの引き出しを増やす。

たとえばアニメ『名探偵コナン』の蝶ネクタイ型変声機は、リアルタイムボイスチェンジャー技術で再現可能になった。アニメ『機動戦士ガンダム』の小型 AI「ハロ」は、スタックチャン[*1]や Romi[*2]といった対話型ロボットとして実現しつつある。フィクションに登場するガジェットが現実の製品やサービスとして登場する例は驚くほど多い。

フィクションを「夢物語」として消費するのではなく、「将来実装されるかもしれない仕様書」として読む。その視点の切り替えが、妄想を仕様に変える第一歩だ。日常の中で触れるアニメ、映画、漫画、小説の中に、まだ世の中にないプロダクトのヒントが眠っている。

具体的なステップとしては、3 つの段階がある。

1. 日常の観察から「こんなものがあったら面白い」という着想を集める。通勤中の不便、趣味で感じた課題、フィクションで見た未来技術など、何でもよい
2. 技術トレンドのキャッチアップで「今何が実現可能か」を把握する。新しい API の発表、OSS の動向、研究論文の成果を常にウォッチする
3. 着想と技術を突き合わせ、「今の技術で実現可能な範囲」を見極める。完全な再現が無理でも、本質的な体験の一部を切り出せれば十分なプロダクトになる

3 番目のステップ「着想と技術を突き合わせる」をもう少し具体的に見てみよう。妄想側と技術側、それぞれから歩み寄る動きがある。

妄想側からの分解は、SF や秘密道具の「What（何を作るか）」を出発点にする。たとえば「ドラえもんのどこでもドアがほしい」という妄想があったとして、物理的なテレポーテーションは無理だ。しかし「離れた場所を瞬時に体験する」という本質に注目すれば、VR 空間での瞬間移動として実現できるかもしれない。Web という切り口なら、360 度カメラのストリーミングで「どこでもドアごっこ」ができる。このように、妄想を分解して代替手段を探る。

技術側からの歩み寄りは、「How（何ができるか）」を出発点にする。新しい API やデバイスが登場したとき、「これを使うと何が実現できるか」を考える。リアルタイム音声変換 API が出たなら「コナンの変声機だ」、画像認識モデルの精度が上がったなら「東のエデンのアレができる」と結びつく。

**Lemio**：この両側から歩み寄って、ぶつかったところにプロダクトが生まれるんです。完璧な再現は最初から目指さない。「この切り口なら今の技術で実現できる」という交差点を見つけるのが、アイデア発想の核心です。

**Sae**：私はそれを「5 歩先のアイデア、2 歩先のプロトタイプ」と捉えています。アイデアとしては未来を見せつつ、プロトタイプに落とすときは 1〜2 歩先に絞る。今日の時

[*1] スタックチャン：M5Stack 上で動作するオープンソースの小型コミュニケーションロボット。表情表示と音声対話が可能

[*2] Romi：MIXI（旧ミクシィ）が開発・販売する家庭用の自律型対話ロボット

点ではクオリティが荒くても、将来その技術が成熟したら実際に市場に出てくるものを先読みして作る。

**Lemio**：まさにそうですね。今はクオリティが低くても、ハッカソンでプロトタイプとして形にしておく。技術が追いついたら一気に市場に出てくる——その「先読み」ができるかどうかが勝負です。

「5 歩先のアイデア、2 歩先のプロトタイプ」という視点は、妄想と技術の歩み寄りに時間軸を加える。アイデアの射程は遠くに置きながら、プロトタイプは「近い将来に実現可能な範囲」で作る。今の技術では荒削りでも、技術の進歩が追いついたときに真っ先に製品化できるポジションを取る戦略だ。

**Kuu**：ハッカソンの醍醐味は、ビジネスモデルまで考えなくていいところですよね。

**Lemio**：そこは大きいです。「あったらいいな」というワクワクだけで十分に動機になる。ビジネスモデルで評価されるプロダクトは安定しているけれど、破壊的なイノベーションにはなりにくい。Twitter も YouTube も、最初はビジネスモデルなんてなかった。ハッカソンは「ワクワクが先、ビジネスは後」でいい場所なんです。

## 3 人のアイデア発想実例

Lemio の過去のハッカソン作品は、まさに「フィクション×現実技術」の掛け算で生まれている。ここでは、前節で触れた作品を例に、アイデアが仕様に変わるまでの具体的なプロセスを見てみよう。

たとえば前述の蝶ネクタイ型変声機。「コナンの道具が実際に使えたら面白い」という妄想から始まり、それを INPUT → PROCESS → OUTPUT のモデルに落とし込む。音声を入力し、API やスクリプトで処理し、変換された音声を出力する。「欲しい」という感情を、具体的な入出力仕様に翻訳する作業である。

**Lemio**：蝶ネクタイ型変声機を作ったとき、最初は漠然と「コナンのあれがほしい」だったんです。でも「あれ」を分解すると、マイクから音声を取って、ピッチを変えて、スピーカーから出す。入力と出力が明確になれば、あとは間をつなぐ技術を探すだけです。

**Kuu**：入力と出力を言葉にできれば、AI が間を埋めてくれる時代ですからね。

この「入力と出力を定義する」思考法は、AI 時代においてさらに威力を発揮する。かつては INPUT と OUTPUT の間を自分でコーディングする必要があった。今はその間を AI が埋めてくれる。

つまり「何を入れて、何が出てほしいか」を言語化できれば、実装の大部分を AI に委ねられる。「コードを書く」から「設計を言語化する」への転換だ。

もう一つの例として、アニメ『東のエデン』の「エデンシステム」がある。カメラで映した対象を画像認識で特定し、関連情報をオーバーレイ表示する——この構想は画像認識技術の進化により、現実的な精度で実装可能になった。生成 AI を組み合わせれば、「絵本の中に入り込む靴」のような体験すら、インタラクティブな仮想空間として再現できる。

うまくいかなかった経験もある。アイデアが壮大すぎて、ハッカソンの限られた時間では動くデモまで持っていけなかったケースだ。ドローンスウォームでガンダムの「ファンネル」を再現しようとしたような場合、ハードウェア制御の複雑さが想定を超え、デモと

して見せられる段階に到達できない。こうした失敗が教えてくれるのは、「フィクションの 100% 再現」は要らないということだ。本質的な体験の一部を切り出せれば、デモとして十分に機能する。

### 雑談からアイデアが生まれる

**Lemio**：1 人で考えていると、自分の知識の範囲でしかアイデアが出ないんですよね。でも誰かと話していると、相手の一言で全く違う方向に発想が飛ぶことがある。

**Sae**：雑談って、テーマが定まっていないからこそ自由に飛べるんですよね。ブレインストーミングのように「アイデアを出しましょう」と構えると、かえって出てこないこともあります。

**Kuu**：この本の企画自体、3 人の雑談から生まれましたからね。

1 人のブレインストーミングには構造的な限界がある。自分の知識と経験の範囲内でしか発想が広がらないため、似たようなアイデアが堂々巡りしやすい。

雑談が強いのは、異なる知識領域の衝突が自然に起きるからだ。自分が知らない技術やフィクションを相手が持っていれば、掛け合わせの組み合わせが爆発的に増える。さらに、頭の中のぼんやりした構想を相手に説明しようとする行為自体が、アイデアの輪郭を明確にする。

加えて、「それ面白い」というリアクションがモチベーションを生む。1 人で考えたアイデアは自信が持てないが、他人が面白がってくれた瞬間、実装への推進力が生まれる。

ハッカソンの開始直後にチームメンバーと雑談する時間を意識的に設けることは、アイデアの質に直結する。いきなり「何を作るか」を議論するのではなく、「最近面白かったもの」「日常で不便に感じていること」を自由に話す時間を 10 分でも取るだけで、発想の幅が変わる。

雑談の延長として、日常的にニュースや SNS から「問題」を拾い、すぐにプロトタイプで解決する習慣も有効だ。

**Lemio**：SNS やニュースで問題が話題になっているのを見たとき、「これ、技術で解決できないか」と考えて、すぐにアプリを作ってみるんです。たとえばスキー場の安全情報がわかりにくいと感じたら、地図上にリスク情報をオーバーレイするプロトタイプを作る。完成度は問わない。「解ける」と確認するだけでいい。

**Kuu**：自分の身の回りだけだとペインの数に限界がありますよね。でもニュースを見ていれば、問題を提示してくれる人はいくらでもいる。

**Sae**：それって日常がずっとハッカソンみたいな状態ですよね。

**Lemio**：まさにそうです。日頃のニュースから「追体験させるアプリ」や「嫌なことを回避するツール」を考えて、今すぐ形にする。問題を解決する練習を日常的にやっておくと、ハッカソン本番で圧倒的に動きが速くなります。

自分が直接体験していない問題でも、ニュースや SNS を通じて「他人のペイン」に触れ、それを解決するプロトタイプを作る。この習慣は、問題発見から実装までの瞬発力を鍛え、同時に「自分の経験」に閉じないアイデアの幅をもたらす。ハッカソン本番で初めて出会うテーマにも、日常の練習があれば即座に対応できる。

**Lemio**：アイデアって、常に 30〜50 個ぐらい頭の中で温めているんです。日々ポンポン思いつく。でも忘れるので、とにかくメモを取る。サウナ上がりに一気に 10 個ぐらい音声メモしたこともあります。

**Kuu**：アイデアが出やすいタイミングってあるんですか？

**Lemio**：シャワーを浴びているとき、散歩しているとき、サウナの後。リラックスしている瞬間に出てくることが多いです。仕事のことを考えていない状態、強制的に非日常を作り出すのが大事だと感じます。

**Sae**：でも思いついただけだと、形にならないままですよね。

**Lemio**：そこが問題で、1〜2 日経つと「大したことないな」と冷める。勢いと熱量があるうちに手を動かさないとダメなんです。X で「あ、これ俺が考えたやつ、誰かがもうやってる」と気づいた時の萎え方はすごい。だからハッカソンで強制的にアウトプットする。アイデアは腐る前に形にする——これが僕の鉄則です。

Lemio が言う「強制的に非日常を作り出す」は、個人の習慣だけでなく、ハッカソンの会場設計にも当てはまる。

**Kuu**：ハッカソンの会場がオフィスの延長だと、仕事モードが抜けないんですよね。空間を非日常的にするだけで、アイデアの出方が変わる。

**Lemio**：海外のハッカソンではクリエイティブな空間で開催されることが多くて、フィンランドやアメリカのイベントではアイデアがどんどん湧きました。メンターに「仕事のことは忘れろ、脳のリソースを空けろ」と言われたのが印象的です。

**Sae**：ストレスを感じない環境に身を置くと、発想が自由になりますよね。リラックスできる場所から、いいアイデアが生まれる。

会場環境とアイデアの質には密接な関係がある。オフィスの会議室で開催すると、参加者は無意識に「仕事の延長」として振る舞う。一方、カフェ、コワーキングスペース、あるいは屋外など、普段と異なる空間に身を置くと、思考パターンも切り替わる。環境のスイッチが、発想のスイッチを入れる。ハッカソン運営の視点で言えば、「どこで開催するか」はタイムラインやテーマと同じくらい重要な設計要素だ。会場選びの具体的な考え方は第 3 章「ハッカソン運営と参加の実践知」で触れている。

■コラム: Lemio のアイデアが育つまで

Lemio が過去のハッカソンやプロジェクトで取り組んだ作品は、すべて「フィクション×技術トレンド」の掛け合わせから生まれている。いくつかの例を紹介する。

**蝶ネクタイ型変声機**: 名探偵コナンの変声機をリアルタイムボイスチェンジャーで再現（詳細は本文参照）。着想は「コナンのあれが現実にあったら面白い」という純粋な妄想だったが、最初の壁はリアルタイム性だった。音声変換自体は既存ライブラリで可能でも、遅延が大きいとデモで「会話」が成立しない。処理パイプラインを最小構成に絞り、遅延を体感できないレベルまで削ったことで、ようやくデモ映えする状態に到達した。ハッカソンでのデモでは、実際にその場で声を変えて見せることで、審査員の体験を直接揺さぶった。「説明するより体験させる」プレゼン戦略の好例である。

**画像認識オーバーレイ**: 東のエデンの「エデンシステム」に着想を得て、カメラ映像にリアルタイムで情報をオーバーレイ表示するプロトタイプ。画像認識モデルの進化により、以前は精度が足りなかった機能が実用レベルで動くようになった。

**対話型ロボット**: ガンダムの「ハロ」をモチーフに、LLM を搭載した小型対話ロボットの制作。通信建設の現場で培ったハードウェアの知識と、LLM アプリエンジニアとしてのソフトウェアの知識が掛け合わさった作品。

共通しているのは、「完全な再現」ではなく「本質的な体験の抽出」を目指している点だ。フィクションのすべてを再現する必要はない。ユーザーが「おお、あれだ」と感じる体験の核を特定し、その核だけを技術で実装する。この割り切りが、限られた時間のハッカソンでは特に重要になる。

## 2.3　チームでのアイデア合意

**Kuu**：初めましてで組んだ時のアイデアの合意形成も結構難しいですよね。

**Lemio**：「それ面白い」「それでいけそう」とみんなが納得するアイデアなのか、みんなで練り上げて生まれてくるアイデアなのか。その違いは大きいですね。

チームでのアイデア合意には 2 つのパターンがある。「誰かのアイデアに全員が乗る」パターンと、「全員で議論して練り上げる」パターンだ。

前者は意思決定が速い。強いビジョンを持つメンバーがアイデアを提示し、他のメンバーが「面白い、やろう」と合意する。ただしこのパターンには落とし穴がある。本心からの賛同か、空気を読んだ同意かが判別しにくい。表面的な合意のまま開発に入ると、途中で「実はあまりピンときていない」というメンバーのモチベーションが下がる。結果としてチームが機能不全に陥る。

後者は時間がかかるが、全員のオーナーシップが高い。対立が起きても、議論を通じて全員が納得した結論には推進力がある。ただし、議論が長引きすぎると開発時間を圧迫するリスクがある。

**Lemio**：初対面のチームでは、まず「誰が何に詳しいか」を共有するのが大事です。メンバーの得意分野がわかると、アイデアの方向性が自然に絞られる。

**Kuu**：全員が「自分ごと」として捉えられるアイデアかどうかが、合意形成のカギですね。

合意形成を円滑にするための実践的なコツがある。まず、アイデア出しの段階では否定をしない。「それは技術的に無理」「審査員にウケない」といった批判は、発想の幅を狭める。次に、アイデアを選ぶ段階では「全員が一つ以上の貢献ポイントを持てるか」を基準にする。特定のメンバーしか手を動かせないアイデアは、チーム全体の合意を得にくい。

もっとも効果的なのは、「プロトタイプで語る」手法だ。言葉だけの議論では堂々巡りになりやすいが、5 分で作った粗いプロトタイプを見せれば、「これのこの部分を変えたい」「ここを膨らませたい」と具体的な議論に移行できる。AI 時代のハッカソンでは、Vibe

Coding で数分のうちにプロトタイプを生成できるため、「まず作って見せる」ことの敷居が大幅に下がっている。

## 2.4　アイデアから仕様書へ

### アイデアを仕様に落とす難しさ

**Kuu**：アイデアと開発の間の仕様決めは大事だと思っています。「このアイデアをやりたい」と言っても、仕様を考えて実装まで持っていけないことがあるんですよね。

**Lemio**：仕様に落とし込めないチームは多いと思います。アイデアを考えても地に足がついていなくて、「どうやって実現するか」を考えた結果、ありきたりなものになるパターンですね。

「アイデアと仕様の間にある崖」で多くのチームがつまずく。仕様化できずに時間を浪費するか、仕様化の過程で本来のアイデアが矮小化するか——失敗パターンはこの 2 つに集約される。

どちらの失敗も、「アイデアと技術を行き来する力」の不足が原因だ。アイデアだけを考える人と技術だけを考える人が分離していると、この崖を越えられない。理想は、1 人の頭の中で「こんな体験を作りたい」と「この技術を使えば実現できる」を同時に考えられることだ。本章冒頭で Lemio が述べた「オタク知識×技術知識」の掛け算は、まさにこの二面性を指している。

**Kuu**：非エンジニアがこの崖で詰まる理由って、何だと思いますか？ ペイン（課題）は持っているのに、実装まで持っていけない人が多い。

**Lemio**：「調べていない」か「ステップバイステップの分解ができない」かのどちらかだと思います。僕の場合は高専で原理原則を学んだ経験が大きい。ものごとを要素に分解して、一つずつ理解する訓練を 5 年間やった。それが「猿でもわかるように説明する」能力につながっていて、AI への指示にもそのまま活きている。

**Kuu**：AI への指示って、要は「相手にわかるように説明する」ことですもんね。

**Lemio**：そうです。仕様書に落とし込めたら、あとは AI が作ってくれる。大事なのは自分が納得できるやり方を選ぶことと、Why と How を一緒にデータとして残すこと。「なぜこの仕様にしたか」と「どう実装するか」をセットで記録しておけば、後から振り返ったときに再現できる。

**Kuu**：AI を使った開発で言えば、ユーザーの入力が Why 側、AI が考えた内容が How 側、そして最終的な出力が結果です。この 3 つをログとして残しておくと、後から別のプロジェクトにも応用が利く。仕様とアウトプットだけに目が向きがちだけど、間のプロセスも含めて記録するのが大事ですね。

仕様書に Why と How を残す習慣は、AI 時代の開発で再現性を担保する鍵になる。AI への入力（Why ＝なぜ作るか）、推論過程（How ＝どう実装するか）、最終出力（What ＝何ができたか）の 3 層をセットで記録する。この記録があれば、知見が個人に閉じず、チームやコミュニティに引き継げる。

**Sae**：アイデアの言語化自体が苦手な人はどうすればいいですか？

**Lemio**：音声入力がおすすめです。キーボードで文章を書こうとすると構えてしまうけど、声で話すと言語化のステップが省略されてバイブスのまま出せる。その音声メモを AI に整理させれば、アイデアの種がテキストとして残る。

## 仕様化のステップ

**Lemio**：僕は最初に「入力がこの状態で、この処理をして、この結果を出す」という流れを全部決めてから、仕様書を作って開発します。わからない部分だけ、「ここからこうしたいけど、どの技術スタックを使えばいいか」を相談して壁打ちしながら進めますね。

抽象的なアイデアを実装可能な仕様に変換するには、段階的な具体化が必要だ。Lemio が実践する INPUT → PROCESS → OUTPUT モデルは、そのシンプルなフレームワークだ。

**ステップ 1: ユーザー体験を 1 文で定義する**。「ユーザーが○○すると、○○が起きる」という形式で、プロダクトの核心的な体験を言語化する。蝶ネクタイ型変声機の例なら、「ユーザーがマイクに向かって話すと、別人の声でスピーカーから出力される」となる。

**ステップ 2: INPUT・PROCESS・OUTPUT を分離する**。ステップ 1 の体験を、入力（ユーザーが提供するもの）、処理（システムが行うこと）、出力（ユーザーが受け取るもの）に分解する。この分離により、「処理」の部分を技術的にどう実装するかという問題に集中できる。

**ステップ 3: 処理を既知の技術要素に分解する**。「処理」をさらに小さなステップに分け、各ステップに対応する技術（API、ライブラリ、サービス）を特定する。わからない部分は「ここは未定」と明示し、チームメンバーや AI に相談する。全体像を先に固め、不明点をピンポイントで潰していく。

**ステップ 4: 画面遷移や操作フローを可視化する**。紙のスケッチでもホワイトボードでも、ユーザーが触る画面の流れを視覚化する。これによりチーム全体が同じ完成イメージを共有でき、各メンバーがどの部分を担当するかが自然に決まる。

このプロセスのポイントは、曖昧な箇所を「わからない」と認めてピンポイントで潰すことだ。仕様が曖昧なまま開発に入ると手戻りが発生する。わからないところを「わからない」と認め、そこだけを集中的に調査・相談することで、仕様の精度を上げながら時間を節約できる。

## トップダウンとボトムアップ

**Lemio**：僕は大枠から考えて、細かくしていくタイプです。まず全体像を描いて、「ここはもっと分解が必要だな」というところをどんどん掘り下げていく。

**Sae**：僕は逆で、一番小さな実験をまずやってみます。それがうまくいったら次のピースを足す、という形で積み上げて全体を組み立てていきます。

**Lemio**：アプローチは真逆ですね。でも、どちらも「分解する」という点では同じです。

**Kuu**：ブロックやステップは少ないに越したことはないですよね。大枠でうまくいかないなら、もう少し細かく分解してみる。

**Lemio**：今は AI が技術側をかなり知っているので、まず大枠で聞いてみて、「もっと

処理を分解して考えて」と指示する。提案してもらった分解を自分たちが理解していくという流れです。

仕様化における分解には、トップダウンとボトムアップの 2 つのアプローチがある。トップダウンは全体像を先に描き、各部分を段階的に詳細化していく方法だ。ボトムアップは最小単位の実験から始めて、動くことを確認しながら全体を組み上げていく方法である。

初めてのテーマではボトムアップ（小さな実験で手応えを確認してから広げる）が、得意な技術スタックではトップダウン（全体設計から AI に分解を任せる）が効率的だ。

AI 時代においては、トップダウンの分解を AI と壁打ちする手法が特に有効になっている。大枠を AI に提示し、「もっと細かく分解して」と指示すれば、AI が処理の分解案を提案してくれる。人間はその提案を読み解き、理解し、判断する役割に集中できる。

## AI で仕様書を作る

AI 時代の開発では、アイデアから仕様書への変換プロセスにも AI を活用できる。ここでは、トップダウンアプローチを前提に具体的なワークフローを紹介する。入力と出力を言葉にできれば AI が間をつないでくれる時代になった。

具体的なワークフローは以下の通りだ。

**フェーズ 1: アイデアの壁打ち**。まず、アイデアを AI に投げて壁打ちする。「○○というアイデアがあるのだが、技術的に実現可能か」「類似のプロダクトやサービスは存在するか」といった質問で、アイデアの実現性と新規性を素早く検証する。

**フェーズ 2: INPUT → PROCESS → OUTPUT の定義**。アイデアの方向性が固まったら、AI に対して「このアイデアを INPUT・PROCESS・OUTPUT に分解してほしい」と依頼する。AI が提示した分解案をたたき台に、チームで議論して修正する。ゼロから考えるより、AI の提案を修正するほうが圧倒的に速い。

**フェーズ 3: 技術スタックの選定**。「この処理を実現するために使える API、ライブラリ、サービスを提案してほしい」と AI に聞く。ここで注意すべきは、AI が提案する技術が最新とは限らない点だ。提案された技術の公式ドキュメントを必ず確認し、現在も利用可能で、ハッカソンの期間内に使いこなせるかを検証する。

**フェーズ 4: 仕様書の生成**。フェーズ 1〜3 の結果をまとめ、AI に仕様書のドラフトを生成させる。画面遷移、API のエンドポイント設計、データフローを含む仕様書を、自然言語で指示するだけで作成できる。この仕様書をチーム全員で確認し、認識のずれを早期に潰す。

**Lemio**：入力と出力を言葉にできれば、AI が間を埋めてくれる。この「言語化する力」が、AI 時代のハッカソンでもっとも重要なスキルだと思います。

**Sae**：逆に言えば、「何がほしいか」を言語化できない人は、AI をうまく使えないということですよね。

**Lemio**：そうです。だからこそ、普段からフィクションや日常の中で「これがほしい」を具体的に考える訓練が、直接的にハッカソンの成果につながるんです。

## 2.5　評論家にならず手を動かす

アイデアを仕様に落としたら、次は手を動かすフェーズだ。AI 時代には「作ってみる」コストが劇的に下がり、評論より実装が圧倒的に価値を持つ。

### 実装が評価される理由

**Kuu**：評論家にはなりたくないですね。何が良くて何がダメなのか、「すごい」と言うけれど本当にすごいのか、従来と何が違うのか。動かさないとわかりません。自分の手で動かして、生の声で語ってほしいですよね。

ハッカソンの現場では、「それは技術的に難しいのでは」「前に似たサービスがあって失敗した」と指摘する人より、「とりあえず 30 分で動くものを作った」と見せる人のほうが圧倒的に価値が高い。

評論と実装の差は、情報量の差だ。頭の中で「無理そう」と判断するとき、その判断の根拠は過去の経験や断片的な知識にすぎない。一方、実際に手を動かすと、「ここは想像通りに動いた」「ここは予想外の壁がある」「ここに思いがけない可能性がある」と、格段に高い解像度で状況を把握できる。

AI 時代では、この差はさらに大きくなった。「作ってみる」コストが劇的に下がったからだ。Vibe Coding で 30 分あれば動くプロトタイプが手に入る。その 30 分を惜しんで 1 時間議論する行為は、「実装が重かった時代」の習慣の名残である。

**Lemio**：「アイデアは腐る前にハッカソンで放出する」と僕はよく言うんですが、同じことがハッカソンの中でも言えます。議論は腐る前にプロトタイプにする。プロトタイプにした瞬間、議論の質が変わるんです。

**Kuu**：「すごい」という感想も、動くものを触ったうえでの「すごい」と、話を聞いただけの「すごい」では、まったく別物ですからね。

評論家にならないコツは単純だ。発言する前に手を動かす。「これは面白いかもしれない」と思ったら、まず 5 分でプロトタイプを作る。「この技術は使えるかもしれない」と思ったら、まずサンプルコードを動かす。その手触りの情報が、チームへの発言の説得力を根本的に変える。

ハッカソンで最後に勝つのは、もっとも優れた評論をした人ではない。数多くのプロトタイプを試し、多くの失敗から学び、その知見を最終プロダクトに凝縮した人だ。

# 第 3 章

# ハッカソン運営と参加の実践知

ハッカソンはなぜ人を惹きつけるのか。参加者を動かす構造的な力と、それを支える運営の実務ノウハウ。非エンジニアの巻き込み方から、軽量ハッカソンの開催、審査設計まで——ハッカソンを「開く側」と「出る側」の両方の視点で実践知を共有します。

## 3.1　締切と発表が生む推進力

### 制限時間が完璧主義を解除する

**Kuu**：ハッカソンはフレームワークとしてすごく良いと思うんですよ。2、3 日で締め切りを作って、初対面の人たちと何かを作る。必ず発表しなければならないというプレッシャーも良い。

ハッカソンでは、ログイン画面も Kubernetes も不要だ。セキュリティ設計すら後回しにする。業務では当然の品質基準を意識的に切り捨てられるのは、制限時間のおかげだ。完璧を目指す思考が強制的にリセットされ、コア機能だけに集中できる。この割り切りこそが、ハッカソンの時間制約が持つ最大の効果だ。

完成しなくても価値はある。「自分たちにはこの技術が足りなかった」「この設計ではスケールしなかった」という発見は、完成したプロダクトと同等の学びをもたらす。できなかったことを知ること自体が、次のハッカソンや日常業務での成長につながる。

### デモ義務が質を上げる

**Kuu**：発表が必須だと、チームの集中力が全然違います。「動くものを見せなきゃいけない」という前提があるだけで、議論が空転しなくなるんですよ。

デモ義務が生む切迫感は、議論の空転を防ぎアウトプットの質を底上げする。この効果はエンジニアに限らない。むしろ今、もっとも大きな変化が起きているのは非エンジニアの参加だ。

## 3.2　非エンジニアに開かれた場

### AI が下げた参加障壁

**Lemio**：フランスで学生ハッカソンを視察したとき、Vibe Coding が初めてという参加者もいました。今までハッカソンはエンジニアのものでしたが、一般の方に広げていくのは面白いかもしれません。

**Kuu**：プロダクトを自分たちで作れなかった人たちは、すごく楽しんでいるんですよ。作りたいものがたくさん溜まっていて、それが形になったときの感動は大きいです。

**Sae**：私はスポンサー側としてハッカソンに関わったのが最初でした。48 時間で実際に動くものができあがるのを目の当たりにして、衝撃を受けました。

初参加者の最大のハードルは「自分が役に立てるかわからない」という不安だ。しかし参加するだけでも価値がある。チームの中で自分の得意な領域を見つけ、貢献できる場面は必ず生まれる。コードが書けなくても、アイデア出し、ユーザーリサーチ、発表資料の作成など、役割は多い。

### 各職種が持ち込む価値

**Lemio**：デザイナーが実際にアプリまで作れるようになると面白いですよね。エンジニアとの会話も円滑になりますし。

**Kuu**：「ここまでは簡単にできて、ここからはエンジニアの技術が必要」という境界線を体感できるのも価値があります。

**Kuu**：Apple Vision Pro ハッカソンでは、Vision Pro を持っていない人が、数年先に普及するデバイスを実際に触って、しかもアプリまで作れました。クラウドのクレジットが提供されるハッカソンもあり、特に学生にとっては貴重な機会です。

**Lemio**：新しい技術を触ってみたいと思いつつ、時間やお金がなくて挑戦できていない人が、最新技術のすごさを体感できる。しかも多くの人と話しながら進められるのは醍醐味ですね。

非エンジニアがハッカソンに持ち込める価値は、コードを書く能力だけではない。ドメイン知識を持つ人は、技術的に可能でもユーザーにとって無意味な機能を早期に見抜ける。UX の視点を持つ人は、プロトタイプの段階で操作性の問題を指摘できる。ビジネス視点を持つ人は、審査員へのピッチでプロダクトの市場価値を説得力をもって伝えられる。

チーム構成の工夫として、メンターの配置が効果的だ。初参加者が多いチームに経験者を 1 人配置するだけで、技術選定や時間配分の判断が格段にスムーズになる。メンターは答えを教えるのではなく、「その方向で進めて大丈夫」という安心感を与える役割を担う。

職種だけでなく、文化的な多様性もプロダクトの質を上げる。海外からの参加者がチームにいると、日本人だけでは気づかないペインが見える。在日外国人が日常で感じる不便や、海外で流行している技術トレンドが、アイデアの起点になることは珍しくない。グローバル、ジェンダー、職種のバランスが取れた参加者構成は、チーム内の視点を広げ、

参加者全体の満足度を押し上げる。

**Kuu**：もっと踏み込んで、エンジニア禁止ハッカソンをやってみたいんですよ。参加者は非エンジニアだけ。エンジニアはメンターとして横に付く。

**Sae**：それいいですね。思いをとにかく形にするためのハッカソン。

**Lemio**：そもそも「ハッカソン」という名前自体がハードルになっている気もしますけどね。

**Kuu**：たしかに。「アイデアソン」だと作らないし、「ハッカソン」だとエンジニア向けに聞こえる。Vibe Coding の普及で、非エンジニアが主役になれるイベントが現実的になった。名前も含めて再設計する時期かもしれない。

「エンジニア禁止」という逆転の発想は、非エンジニアの参加障壁を根本から取り除く。ドメイン知識を持つ人が自分のアイデアを Vibe Coding で形にし、エンジニアはメンターとして技術的なつまずきを解消する。運営視点では、メンターの人数確保と、「完成度」ではなく「課題設定の鋭さ」で評価する審査基準の再設計が鍵になる。

## 3.3　軽量ハッカソンのすすめ

### 「今夜ハッカソンしよう」

**Kuu**：エンジニアはもっと気軽に、「今日この技術が発表されたから、このあと 5 時間でハッカソンやりませんか」と招集するような、カジュアルなスタイルもいいかもしれないですね。新しい技術が出たら必ずその技術を使ったハッカソンをこまめに開催する、くらいの感覚で。

即興で奏でるジャムセッションのように、みんなで集まって自然に生まれたものを楽しむ——軽量ハッカソンの本質はそこにある。必要な準備は驚くほど少ない。会場（カフェやコワーキングスペースで十分）、Wi-Fi、電源、そしてテーマだけだ。

参加者は 3〜5 人で始めるのがちょうどよい。全員がノート PC を持ち寄り、最初の 30 分でテーマとゴールを決め、残りの時間で一気に作る。発表は全員の画面を順番に見せるだけでよい。形式にこだわるより、「集まって作る」こと自体に価値がある。

新しい技術が発表された直後に開催すると、学びの質が格段に上がる。ドキュメントを読むだけでは得られない「手触り」の情報が、短時間で蓄積される。全員が同じスタートラインに立つため、経験の差が出にくく、初参加者にとっても参加しやすい。ゲームジャムや AI ジャムのように、テーマを絞った短時間イベントは運営の負担も小さく、開催のハードルが低い。

**Lemio**：オンラインハッカソンは期間が長くなりがちで、過去の使い回しも多い。交流も生まれにくいんですよね。

**Sae**：ただ、Zoom で全員が同じ場で発表する形式は、オンラインならではの良さがあります。地方からでも参加できますし。

**Kuu**：とはいえ密度はオフラインが圧倒的に高い。「ライト」がキーワードで、気軽に集まれるオフラインの場が一番強い。

**Sae**：あと 3 時間だとアイデアソンになりがちですよね。

**Lemio**：3 時間の中にアイデア出しも含まれるので、プロトタイプを作る時間がほとんどない。AI 使ってもプロンプトを数回入れて終わりです。

軽量ハッカソンには「軽すぎる」限界もある。3 時間ではアイデア出しと役割分担で大半が消え、プロトタイプの実装時間がほぼ残らない。AI 時代でもプロンプトを数回試す程度で終わる。最低 5 時間、理想は丸 1 日が「動くものを作る」ための現実的なラインだ。

一方、オンラインハッカソンは参加の手軽さと引き換えに、期間の長期化や交流の希薄さという課題を抱える。オフラインの密度の高さとオンラインのアクセスしやすさは、目的に応じて使い分ける。

### 多様な形式

ハッカソンの場所も固定観念にとらわれる必要はない。豪華客船ハッカソン、寝台列車ハッカソン、キャンプハッカソン、温泉ハッカソンなど、非日常空間での開催は参加者の創造性を刺激する。日常から切り離された環境に身を置くことで、普段の思考パターンから抜け出しやすくなる。

親子ハッカソンも面白い試みだ。子供が PC 操作を担当し、親が司令塔としてアイデアの方向性を決める。プログラミング教育とは異なり、親子で一つのプロダクトを作り上げる体験が共有される。

番組前提のハッカソンという形式もある。ドキュメンタリー風に密着取材し、視聴者投票で結果を決める。ハッカソンをコンテンツとして発信することで、参加者だけでなく視聴者にもものづくりの魅力が伝わる。

## 3.4 運営の実務

### デモ前テストを確保する

**Kuu**：事前に発表のテストをしっかり行って、テクニカルイシューを起こさせないことが大事ですね。

**Lemio**：他のハッカソンでは事前の練習がないことが多いですね。接続したら動画が再生できないとか。

**Kuu**：僕も 1 回、テストでは音が出たのに本番で音が出なかったことがあります。

デモ前テストで確認すべき項目は、映像出力、音声出力、ネットワーク接続、デモデータの 4 つだ。特に音声は、テスト時に出ていても本番環境で出ないことがある。HDMI 接続の相性、ブラウザの自動再生ポリシー、会場のスピーカー設定など、変数が多い。最低でも本番 30 分前にリハーサルを行い、実際の接続環境で動作を確認する時間を確保したい。

■メモ: デモ前テストの 4 点チェック

映像出力、音声出力、ネットワーク接続、デモデータ——この 4 つを本番 30 分前に実際の接続環境で確認する。特に音声は HDMI 接続の相性やブラウザの自動再生ポリシーで本番だけ出ないことがある。

## チーム分けの手法

チーム編成で一番大事なのは、スキルの多様性よりも方向性の一致だ。志向の近い人同士[*1]でチームを組む方が、モチベーションが持続しやすい。コーディングやデザインの実装面でのスキル不足は AI で補えるが、「何を作りたいか」の食い違いはツールでは解消できない。

■メモ: チーム編成の鍵

スキルの多様性よりも方向性の一致を重視する。コーディングやデザインのスキル不足は AI で補えるが、「何を作りたいか」の食い違いはツールでは解消できない。

運営側の仕事は、参加者の心の殻を破ることだ。初対面同士が「自分はこういうものを作りたい」と素直に話せる空気をどう作るかが鍵になる。アイスブレイクの時間を十分に設ける、軽い飲食の場を用意してからチーム編成に入るなど、心理的安全性を担保する工夫が効果的だ。

チーム内でリーダーが不在だと方向性が定まらず停滞する。一方で、1 人がリーダーシップを取りすぎると他のメンバーが受け身になる。理想は、アイデア出しの段階では全員がフラットに発言し、開発に入ったらタスクの割り振りを 1 人が仕切る形だ。

面白い手法として、同じアイデアを複数チームが並列で開発するやり方もある。同じゴールに向かっても、チームごとにアプローチが異なり、結果の違いから多くの学びが得られる。

## 審査基準と賞を設計する

**Kuu**：ハッカソン運営をする側としては、賞をなるべく増やしていきたいです。小さな賞でも、受賞することでハッカソンを好きになる理由が生まれます。ハッカソンの沼にはまるきっかけになりますね。

**Lemio**：作ったものに正解はないので、本来はみんな優勝みたいなものです。作りきった人たちは全員優勝、というところはありますね。

[*1] ライクマインド（like-minded）：志向や価値観が近い人同士のこと。スキルセットの多様性よりも、目指す方向の一致がチームの推進力を生む

**Lemio**：全員のピッチを聞くと、練習できていないチームの発表は中身が薄いのに時間が長くなりがちです。

**Kuu**：予選・決勝方式もありだと思います。予選でプロダクトを触って評価し、ファイナリストだけがピッチをする。プロダクトの完成度とピッチの質の両方を評価できます。

**Lemio**：予選側は動画を作る必要が出てくるので、参加者の負担は増えますが、1〜2 ヶ月かけたハッカソンであればファイナリストのピッチは洗練されていて、聴く側も飽きません。

**Sae**：短期ハッカソンだと、一回一回の発表の密度が濃くて聞いている側も疲れるんですよね。プロダクト自体が洗練されていないのにピッチの時間だけ長いと、退屈になってしまう。

**Kuu**：でも今は AI のおかげで開発自体はすぐ終わるから、その分ピッチの準備に時間を回せますよね。ハッカソンのルールとして「発表資料の準備時間」を明示的に確保するのも手かもしれない。

ピッチの質は聞く側の体験に直結する。2〜3 日で作ったプロダクトは、1〜2 ヶ月かけた作品に比べてどうしても粗い。そのうえピッチの練習時間も取れないため、「中身が薄いのに時間だけ長い」発表が続くと、審査員も参加者もモチベーションが下がる。

AI 時代には開発の所要時間が大幅に短縮されている分、浮いた時間をピッチの構成やデモの準備に充てるルール設計が有効だ。発表前に 30 分の準備時間を設けるだけでも、ピッチの質は目に見えて変わる。

賞の設計で見落とされがちなのがオーディエンス賞だ。審査員だけの評価では、どうしても審査員個人のバイアスが入る。参加者全員の投票によるオーディエンス賞を設けることで、審査の公平性が補完される。

ユニークな賞設計として、特定のペルソナに刺さるプロダクトを競う形式もある。ある Music & AI ハッカソンでは、来日中のヨーデルのインフルエンサーを審査員に据え、その人に響くプロダクトを作る「ヨーデル賞」が設けられた。ルールが明確で、ターゲットが一人に絞られている分、チームの方向性がぶれにくい。一般化した審査基準よりも、「この人に刺さるかどうか」という振り切った評価軸が、かえって参加者の創造性を引き出す。

ピッチだけで審査するハッカソンには落とし穴がある。プレゼンが巧みなチームが、中身の薄いプロダクトで勝ってしまうことがあるのだ。ピッチ偏重の審査を防ぐには、審査員や参加者がプロダクトを実際に触れる時間を設けることが有効だ。触ってみれば、見せ方だけでは隠せない完成度の差が明らかになる。

賞金の設計にも実務的な知見がある。賞金を一時所得の特別控除額[*2]以内に収めると、参加者側の税務申告が不要になり運営上もスムーズだ。筆者らの運営経験では、100 万円を超えると審査結果へのクレームが増える傾向がある。参加者にとって実はもっとも嬉しい賞品はクラウドクレジットだ。開発環境への直接的な投資になり、ハッカソン後のプロダクト継続にもつながる。

さらに進んだ形として、マーケティングまで含めたハッカソンもある。AppStore への

[*2] 一時所得の特別控除額は 50 万円。この範囲内であれば受賞者側の確定申告が不要となり、運営側の税務処理もスムーズになる

公開を条件とし、SNS フォロワー数や課金数で競う形式だ。Rewritely Cat[*3]では、2 週間の期間で 10 チーム中 4 チームが AppStore 公開を達成し、そのうち 2 チームが起業に至った。作って終わりではなく、世に出すところまでをハッカソンの範囲に含めることで、参加者のコミットメントと学びの質が格段に上がる。

## スポンサーとの関係設計

**Kuu**：スポンサーのプロダクトを無理やり使わなきゃいけないハッカソンは、正直つらいことがあります。唯一無二の技術なら楽しめますが、2 番手、3 番手のツールを強制されると参加者のモチベーションが下がりますね。

**Sae**：Anthropic のハッカソンで Anthropic のモデルを使ってくださいなら、みんなそれを使いたくて来ているのでミスマッチは起きません。でもジェネラルなハッカソンに特定の技術を強制すると、期待値とのズレが生まれます。

**Sae**：スポンサー側も「使ってもらうこと」を KPI にするんじゃなくて、「自社プロダクトはこういう用途に使えるはず」という仮説があって、その仮説を検証する場としてハッカソンを位置づけた方がうまくいきます。

スポンサーとの関係設計は、ハッカソンの満足度を左右する重要な要素だ。テーマ特化型ハッカソン——たとえば AI 企業が主催する AI ハッカソン——では、参加者が最初からその技術を使う意思で集まるため、スポンサー技術の利用に違和感が生じない。

問題はジェネラルなハッカソンだ。スポンサーの技術を必須にすると、参加者のアイデアが制約される。しかもその技術がテーマと噛み合わなければ、「使いづらい」という印象だけが残る。スポンサー側にとっても逆効果だ。

効果的なのは、スポンサーが自社プロダクトのユースケース仮説を持ち、それに合致するテーマ設計を運営と一緒に行うアプローチだ。「この技術はバックオフィスの課題解決に向いている」という仮説があれば、社内業務改善をテーマにしたハッカソンにすればよい。技術とテーマが噛み合えば、参加者は自然にスポンサー技術を選択する。

## AI 時代の時間配分

**Lemio**：他のハッカソンは絶対に 2 時間ぐらい押すんですよ。なんで押さないんですか？

**Kuu**：拍手で盛り上げるんです。みんなが「よかったよ」という雰囲気になると、「ここで終わった方がいい」という空気ができあがります。それでオンタイムに収まるんです。

**Kuu**：開発が簡単になった今、アイデアを練る時間を十分に取らないと、ありきたりな作品になりがちです。「開発しなきゃ」という焦りが、アイデアのブラッシュアップを後回しにさせる。

**Lemio**：逆に、ある時間まではアイデアを必ず議論しなければならない「開発禁止期間」を設けて、そこから先は開発に集中するというルールはどうですか。

**Kuu**：開発禁止期間はメリハリがあっていいですね。「ここまではアイデアを練ってください、ここからはアイデアを変えないでください」と明確に区切ることで、両方の質が

[*3] Rewritely Cat：アプリのリリースとマーケティングまでを競うハッカソン形式のイベント

上がりそうです。

見落とされがちなのは、発表前にプロダクトを触る時間の確保だ。参加者同士がお互いの作品を試す時間があると、フィードバックの質が上がり、審査の精度も高まる。

開発禁止期間の導入も効果的だ。最初の 1〜2 時間はアイデア出しと設計に専念し、コードを書くことを禁止する。この制約があることで、「何を作るか」と「どう作るか」のフェーズが明確に分離され、両方の質が上がる。開発禁止期間が終わったら、アイデアの変更を禁止して実装に集中する。このメリハリが、限られた時間の中で最大のアウトプットを生む。

■メモ: 開発禁止期間のすすめ

最初の 1〜2 時間はコードを書くことを禁止し、アイデアと設計に専念する。開発解禁後はアイデア変更を禁止して実装に集中する。

より広い視点で見れば、AI 時代のハッカソンはタイムライン設計そのもののアップデートが求められている。開発のボトルネックが消えた今、従来の「大半を開発に充て、残りで発表する」配分は通用しにくい。開発時間を圧縮し、アイデアの練り上げとピッチ準備に時間を振り直す。開発禁止期間はその一例にすぎない。運営者はハッカソンのルール設計自体を、AI 時代に合わせて継続的に見直す必要がある。

2 日型ハッカソンでの変化はとりわけ顕著だ。Vibe Coding が登場する以前は、Day1 の夜にはプロトタイプの骨格ができていなければ間に合わなかった。アイデア固めに使える時間は 2 時間程度が限界で、残りはすべて実装に充てる必要があった。しかし開発速度が劇的に上がった今、Day1 はチーム編成とアイデアの練り上げに丸ごと充てられる。Day2 に集中して開発し、完成度の高いプロダクトを仕上げる——この配分が現実的になった。アイデアの質が上がれば、プロダクトの質も上がる。2 日型ハッカソンの運営者は、Day1 の時間設計を根本から見直す価値がある。

## 3.5　社内ハッカソンと持続可能な運営

### 社内ハッカソンの勘所

ここまでの運営ノウハウは主に社外ハッカソンを前提としてきた。社内ハッカソンには社外とは異なる難しさがある。最大の障壁は「業務から完全に離れる」ことだ。片手間でハッカソンに参加しても効果は薄い。Slack の通知を切り、メールを閉じ、完全にハッカソンに没入する環境が必要だ。そのためには VP クラスの承認を取り、チーム全体が業務から離脱する正当性を確保しておく。

社外メンターの招聘も重要だ。社内の人間だけで回すと、どうしても既存の業務知識やヒエラルキーに引きずられる。外部の視点が入ることで、「業務の延長」ではなく「新しい挑戦」の空気が生まれる。

社内ハッカソンを成功させるには、最初に「熱量のある主催者」と「予算の出所」の 2

つが必要だ。運営自身がハッカソンに興味がなく、義務感で主催するイベントは参加者にも伝わり、盛り上がらない。

### 燃え尽きない運営の仕組み

**Sae**：ハッカソンをやりたいという人は多いんですが、実際にやると負担が大きくて。
**Lemio**：毎回みんな引退しちゃうんですよね。
**Kuu**：Sae さんぐらいですよ、何も変わらずシュッとやり続けているのは。

ハッカソン運営者の燃え尽きは、コミュニティの持続可能性に関わる構造的な問題だ。会場手配、スポンサー交渉、参加者対応、当日の進行——これらを 1 人のカリスマ運営者に依存していると、その人が離れた瞬間にイベントが消滅する。

まず運営チームを複数人で構成し、役割を分散させること。あわせて運営ナレッジをドキュメント化して引き継げる状態を保つ。さらに先述の軽量ハッカソンを活用して毎回の運営負荷を下げる。大規模イベントと軽量イベントを交互に回すことで、運営者のモチベーションと体力を持続させられる。

## 3.6　ハッカソンに出続ける理由

### 即座に評価される快感

**Kuu**：ファイナリストに選出されたとき、仲間同士で「おめでとう」と熱狂できたのは大きいですね。やみつきになりました。
**Lemio**：勝つのはいいですよね。アドレナリンが高まりますし、認められるのはやはり大きいです。昔「アイデアマンです」と名乗ったら、仲の良い人にチクッと言われてショックだったんです。でも YC で優勝したあと、自信を取り戻せました。
**Sae**：フィンランドの 1500 人規模のハッカソンで部門優勝したとき、1 年間ずっと周りに言われ続けました。海外の友人からの評価が大きくて、ネットワーキングの質が変わりましたね。YC 優勝後はディナー会に招待されたり、卒業者との交流が生まれたり。
**Kuu**：運営側でも認知は広がりますね。「ハッカソンやってる人」として覚えてもらえる。SNS 上の知名度より、リアルで認知してもらう方が密度が濃い。

ハッカソンでの即時フィードバックは、通常業務では得られない密度の承認体験だ。表彰されると SNS で共有され、受賞の実績が登壇の機会につながる。登壇がさらなる認知を生み、この好循環がハッカソンに出続ける動機を強化する。

数ヶ月かけたプロジェクトの成果が上司の一言で評価される業務と違い、ハッカソンでは数時間の成果に対して会場全体が反応する。この濃密なフィードバックループが、参加者を次のハッカソンへと駆り立てる。

「熱狂するハッカソン」には共通する条件がある。まず、共通の締切に向かう一体感だ。残り時間のカウントダウンが会場全体の緊張感を高め、チームの結束を強める。そして、会場のリアルタイムな反応——デモ中の歓声や拍手は、個人の達成感をチーム全体の高揚に変える。加えて、仲間との共有体験だ。「おめでとう」と言い合える関係が、一過性の興奮を持続的な動機に変換する。

運営の仕組みだけでなく、参加者同士の関係性が熱狂を生む。小さな成功体験が次の参加への意欲を生み、正のスパイラルが回り始める。

### ポートフォリオとしての実績

ハッカソンの実績は、ポートフォリオとして具体的にキャリアに効く。短期間でプロダクトを作り上げた経験は、技術力だけでなく、チームワーク、時間管理、プレゼンテーション能力の証明になる。スポンサー企業との接点が生まれ、採用につながることも珍しくない。先述の Rewritely Cat のように、ハッカソンから起業に至るケースもある。

**Sae**：レジュメでは紙面上でしか評価されませんが、ハッカソンでは実際の振る舞いやアウトプットがその場で評価されます。スポンサー企業の採用担当がデモを見て声をかけてくることもありますし、参加者同士の出会いから共同プロジェクトや起業につながることもあります。

ハッカソンがキャリアに効く理由は、書類では伝わらない能力を可視化できる点にある。問題解決のアプローチ、チーム内でのコミュニケーション、プレッシャー下での判断力——これらはレジュメの文面では評価しようがない。しかしハッカソンの現場では、スポンサー企業の担当者がそれを直接目にする。採用面接では測れない「一緒に働いたらどうか」がわかる場だからこそ、通常の就職活動とは異なる接点が生まれる。

実績がもたらす「箔」の効果も見逃せない。優勝経験があると、その後のネットワーキングの質が変わる。「ハッカソンで優勝した人」という認知は、SNS 上のフォロワー数よりもリアルな場での信頼につながる。運営側にとっても、「ハッカソンを主催している人」という認知がコミュニティ内でのポジションを確立する。リアルでの認知は、オンラインでの認知よりも深く、持続する。

### 「勝つ」以外のモチベーション

**Kuu**：プログラミングが好きで、ダラダラ作るよりも作りきった方が楽しい。たまに優勝できるのが良くて、ギャンブル性が高いんです。特に AI が出てきて、優位性がある中で戦えるのは最高じゃないですか。

**Sae**：新しい技術のキャッチアップをハッカソンを機にやってみる、というのも大きな動機ですよね。

**Kuu**：ただ、普段使わないマイナーな環境に手を出すと、つまずきの 8 割がそこに持っていかれます。「わかるけどできない」を久々に味わいました。AI があっても、専門外の技術スタックは甘くない。

**Sae**：だからこそ、失敗しても評価に影響しないハッカソンで挑戦するのが理にかなっているんですよね。

勝利以外のモチベーションは多様だ。新しい技術を実戦で試す学びの場、普段の業務では出会えない人とのネットワーキング、自分のアイデアを形にする自己表現の機会。ハッカソンに出続ける人は、これらの価値を複合的に享受している。勝てなくても、作りきった達成感と、そこで生まれたつながりが次の挑戦への燃料になる。

未知の技術に飛び込むリスクとリターンも、ハッカソン固有の魅力だ。AI が開発を加速させる時代でも、触ったことのないプラットフォームやハードウェアでは予想外の壁にぶつかる。しかしハッカソンは「安全に失敗できる場」として機能する。業務では許されない技術的冒険を、締切と仲間の力を借りて試せる。第 1 章「AI ネイティブ時代のマインドセット」で触れた「失敗機会の減少」という課題に対して、ハッカソンは意図的に失敗を経験できる貴重な機会を提供している。

# 第 4 章

# AI 活用の実践テクニック

第 1〜3 章で語られた知見を横断し、ハッカソンでも日常業務でも活用できる AI 活用の実践テクニックを 3 人でまとめます。考え方とアプローチを軸にしつつ、必要に応じて具体的なツール名にも触れます。

## 4.1　AI に正しく仕事をさせる前提条件

第 1 章「AI ネイティブ時代のマインドセット」で紹介した「ガードレール思考」を、ここでは具体的なテクニックに落とし込む。AI に正しく動いてもらうために、まず何を整えるべきかを見ていこう。

### 「古い知識」を上書きする

**Kuu**：第 1 章「AI ネイティブ時代のマインドセット」でガードレールの考え方を話しましたが、具体的な方法として一番効くのは、最新の API リファレンスの URL をプロンプトに含めることです。AI に読ませるだけで回答精度が全然違います。

**Lemio**：ドキュメントの URL を渡すだけでそんなに変わるんですか？

**Kuu**：変わります。特に JS や Python はライブラリのバージョン更新が速いので、公式ドキュメントを起点にしないと AI が古い API で書いてきます。しかも動くけれど非推奨の API を使っていたり、メソッド名が変わっていたりする。動くだけに気づきにくいのが厄介です。

**Lemio**：エラーが出れば気づけるけど、動いてしまうと見逃しますよね。

**Kuu**：だからこそ、URL を渡す、バージョンを明示する、この 2 つだけでつまずきにくくなります。

AI の学習データは数ヶ月前のスナップショットであり、日々更新されるライブラリの API とは乖離がある。公式ドキュメントの URL をプロンプトに含める手法は、AI の持つ古い知識を最新の情報で上書きする作業にほかならない。

この問題の厄介さは「動いてしまう」ことにある。非推奨 API で書かれたコードは正常に動作するが、将来のバージョンアップで突然壊れたり、セキュリティパッチが当たらなかったりする。たとえば Python の `requests` ライブラリではセッション管理の推奨パターンが変わり、React ではクラスコンポーネントから Hooks への移行が完了した。し

かし AI は学習データの割合に引きずられ、古い書き方で生成することがある。

もっとも即効性のある対策は、公式ドキュメントの URL をプロンプトの冒頭に記載することだ。「以下の公式ドキュメントを参照してコードを書いてください」と一文添えるだけで、AI は学習データに頼らず最新の仕様をもとにコードを生成する。

あわせてバージョンを明示する。「Python 3.12、React 19 を使用」と書き、`package.json` や `requirements.txt` をプロンプトに含めることで、バージョン間の整合性が取れたコードが生成される。

さらに生成されたコードの `import` 文を確認する習慣をつける。非推奨のモジュールからインポートしていないか、廃止されたパッケージを参照していないかをチェックする。入力品質の重要性については、本章末のアンチパターンの節で改めて扱う。

### ルールファイルで行動を制御する

**Kuu**：プロンプトに毎回同じことを書くのは非効率です。Coding Agent には「Agent Skills」と呼ばれる仕組みがあって、プロジェクト固有のルールや慣習を事前に設定ファイルとして渡せる。これを整備しておくと、AI が毎回正しい前提で動いてくれます。

**Sae**：設定ファイルというのは、どのくらいの粒度で書くんですか？

**Kuu**：「このプロジェクトでは TypeScript を使う」「テストは Vitest で書く」「コミットメッセージは Conventional Commits に従う」くらいの粒度です。プロジェクトの README に書くような情報を、AI が読める形式で置いておくイメージですね。

Agent Skills（第 1 章「AI ネイティブ時代のマインドセット」で紹介した、AI が参照するプロジェクト固有のルール定義ファイル）の整備は、新メンバー向けのオンボーディング資料を書く感覚に近い。プロジェクトの規約やコーディングスタイルを、AI が読める形式で渡す。

この仕組みの利点は、再現性にある。プロンプトに毎回同じ指示を書く代わりに、設定ファイルに一度書いておけばよい。以降のすべてのやり取りで、AI がその前提を踏まえて動く。チームメンバー全員が同じ設定ファイルを共有すれば、AI の出力品質がチーム全体で均一化される。

API 情報の提供も Agent Skills の重要な要素だ。プロジェクトで使う外部 API について、エンドポイント一覧・認証方式・レスポンス型定義を設定ファイルに含めておくと、AI は API コールのコードを正確に生成できる。特にハッカソンでは、スポンサー企業が提供する API を初めて使うことが多い。API ドキュメントの URL と基本的な使い方を Agent Skills に書いておくことで、チーム全員が AI を通じて素早く API を活用できる。

AI の前提知識を整えたら、次は AI への指示の質を上げる番だ。設計そのものを言語化する技術を見ていこう。

## 4.2　設計を言語化する

### 入出力を言語化する

**Lemio**：AI の時代に変わったのは、設計を言語化するスキルの重要性です。入力と出力を明確に言葉にできれば、その間の処理は AI が埋めてくれる。逆に言えば、言語化で

きない設計は AI にも伝わらない。

**Kuu**：「何を入れて、何が出てくるか」を定義するのが人間の仕事で、「どうやって変換するか」は AI の仕事になった、ということですよね。

**Lemio**：そうです。「この画面にユーザーが名前を入力すると、パーソナライズされた挨拶が表示される」――この一文があれば、AI はフォーム、バリデーション、表示ロジックまで一気に書ける。設計を言語化する力が、そのまま AI の活用力になります。

「設計を言語化する」とは、頭の中にある曖昧なイメージを、AI が解釈できる精度まで具体化する行為だ。コードが書けなくても、「入力は CSV ファイル、出力は月別売上のグラフ、形式は PNG」と言語化できれば、AI はそのとおりに実装する。プログラミングスキルとは無関係の能力だ。

実践的なアプローチとして、機能ごとに「入力→処理→出力」の三行を書く方法がある。たとえば「入力：ユーザーの現在地（GPS 座標）、処理：半径 1km 以内のレストランを検索してレビュースコア順にソート、出力：レストラン名・距離・スコアのリスト」と書く。この三行だけで、AI は API コール、ソートロジック、データ構造の設計まで一貫したコードを生成できる。

## Coding Agent で並列開発する

**Kuu**：Coding Agent の最大の利点は、非同期で動かせることです。人間が一つのタスクに集中している間に、別のタスクを AI に並列で進めてもらう。これはハッカソンでの時間効率を劇的に上げます。

**Sae**：具体的にはどうやって並列に動かすんですか？

**Kuu**：git worktree を使います。一つのリポジトリから複数の作業ディレクトリを作って、それぞれに別の Coding Agent を割り当てる。Agent A にはフロントエンドを、Agent B にはバックエンドを、自分はインフラの設定をやる、という具合です。

Coding Agent を並列で動かせば、人間はディレクションに徹しながら、複数のタスクを同時に前進させられる。

git worktree は、一つのリポジトリから複数の作業ツリーを作成する Git の機能だ。通常のブランチ切り替えと異なり、物理的に別のディレクトリとして存在する。そのため、それぞれのディレクトリで独立して Coding Agent を起動できる。メインの worktree では自分が UI 設計を進め、別の worktree では Agent A が API 実装を進め、さらに別の worktree では Agent B がテストコードを書く――こうした並列作業が可能になる。

ハッカソンでは時間が限られているからこそ、この手法の効果は大きい。3 人チームで 6 時間のハッカソンに参加する場合、各メンバーが 2〜3 の Coding Agent を並列で動かせば、実質的な作業量は数倍に膨らむ。ただし、並列化の前提として、タスクの分割と依存関係の整理が不可欠だ。依存関係が複雑なタスクを並列で進めるとコンフリクトが頻発し、かえって時間を浪費する。

### master 直 push 戦略

**Kuu**：ハッカソンでは、あえて master ブランチに直接 push する戦略を取ることがあります。通常の開発ではブランチを切ってプルリクエストを出すのが常識ですが、2〜3日の短期決戦ではそのオーバーヘッドが致命的です。

**Lemio**：コンフリクトの解消に時間を取られるリスクはないですか？

**Kuu**：あります。だからこそ、タスクの分割が重要です。フロントエンドとバックエンドで明確にディレクトリを分けたり、ファイルの担当を決めたりして、物理的にコンフリクトが起きにくい構造を先に作る。それでもコンフリクトが起きたときは、AI に「Please fix conflict」と指示すれば、大半の場合は自動で解決してくれます。

**Kuu**：CD があることで、開発者が master に push するだけで自動デプロイされ、検証者はブラウザをリロードするだけで最新版を試せます。手動でやると死にますが、CD なら自動です。

**Sae**：ダメだったらすぐエラーでわかるし、ロールバックもできますしね。

ハッカソン限定の戦略として、master 直 push は有効だ。第 1 章「AI ネイティブ時代のマインドセット」でもマインドセットとして触れたが、ここでは具体的な運用手順に焦点を当てる。対話に出てきた CD（Continuous Deployment）と組み合わせることで、さらに効果が増す。

通常の開発プロセスでは、コードレビューとブランチ管理が品質を担保する。しかしハッカソンの時間制約下では、プロセスの厳密さよりアウトプットの速度が優先される。

master 直 push 戦略の前提条件として、チームメンバー全員の担当領域分割、CI パイプラインの設定（push のたびにビルドとテストが自動実行される状態）、コンフリクト発生時に AI で即座に解決する体制が必要だ。

> ■メモ: master 直 push 戦略の 3 つの前提条件
>
> チームメンバー全員の担当領域分割、CI パイプラインの設定（push のたびにビルドとテストが自動実行される状態）、コンフリクト発生時に AI で即座に解決する体制。この 3 つが揃わないまま直 push すると混乱を招く。

CI が失敗した場合のプロンプトも定型化しておくとよい。CI のエラーログを Coding Agent に渡して「このエラーを修正してください」と指示する——これだけで Coding Agent はエラーを解析し、修正コミットを生成する。

さらに効果を発揮するのが CD の併用だ。master への push をトリガーにステージング環境へ自動デプロイする仕組みを用意しておくと、開発者以外のチームメンバーはブラウザをリロードするだけで最新の状態を確認できる。「開発者 1 人＋検証者複数人」という体制が自然に成立し、コードを書かないメンバーも即座にフィードバックを返せる。

CI が失敗したらデプロイを止めてロールバックすれば、常に動く状態が保たれる。CD

は自動テストの役割も兼ねており、デプロイが通ること自体が「少なくともビルドと基本動作は壊れていない」という確認になる。

この戦略はあくまでハッカソンや短期プロトタイピングに限定した手法であり、長期運用のプロダクトには従来のブランチ運用を推奨する。

## 4.3 AI を学習パートナーにする

### 何度でも聞き直せる AI 家庭教師

**Lemio**：わかるまで聞けばいいんです。「わからない部分をもう一度説明してください」と何度でも聞く。そのやり方をレクチャーするのが、ググり方を教えるのと同じ感覚ですね。

**Kuu**：やり方さえわかれば、自走できますしね。

Lemio が言う「わかるまで聞く」を実践するには、聞き方にコツがある。「わかりません」とだけ伝えるのではなく、「ここまでは理解できたが、ここから先がわからない」と境界を示す。AI は理解の境界線を起点に、より噛み砕いた説明を返してくれる。

たとえば「Docker の仕組みがわかりません」よりも、「コンテナが仮想マシンと違うことは理解したが、イメージとコンテナの関係がわからない」と伝えた方が、的確な回答が得られる。この「わかる境界」を伝えるスキルは、AI 活用だけでなく、人間同士のコミュニケーションでも応用が利く。

この手法をチームに広めることは、かつて「ググり方を教える」のと同じだ。検索エンジンの登場時、「何をどう検索するか」を知っている人と知らない人で情報アクセスの格差が生まれた。AI の時代も同様で、「AI にどう聞くか」を知っている人と知らない人で学習速度の格差が生まれる。聞き方のスキルを共有すること自体が、チームの底上げになる。

### 未知の概念を段階的に深掘りする

**Lemio**：昔テレビで見たんですが、アイヌの方にアイヌ語を教えてもらうとき、ぐちゃぐちゃの絵を描いて見せて、相手の反応から語彙を学んでいく方法がありました。わからないことさえわかっていれば、知識を広げられます。AI も同じように使えばいいんです。

**Kuu**：自分がわからないことさえわかっていれば、わかっている部分とわからない部分の境界を伝えて、わからない部分だけを深掘りしていけば全体が見えてきます。

未知の概念に向き合うとき、最初の壁は「何がわからないかがわからない」状態だ。この壁を越える方法として、AI に断片的な情報を投げて反応を見るアプローチがある。

たとえば、新しいフレームワークの概念を学びたいとき、自分が知っている類似概念を AI に伝える。「React の state に似たものが Svelte にもあると思うが、何と呼ぶのか」と聞けば、AI は「Svelte ではリアクティブ変数と呼び、$: という構文で宣言する」と教えてくれる。既知の概念を足がかりに、未知の領域を探索する手法だ。

既知→未知の段階的深掘りは、ハッカソンで特に価値がある。限られた時間の中で、初めて触る技術スタックを学ばなければならない場面は多い。「何がわからないかを整理し

て、AI に段階的に聞く」というアプローチを身につけておけば、新技術のキャッチアップ速度が格段に上がる。

### 音声 AI で曖昧に検索する

**Kuu**：テキスト検索だと言葉に言語化しなければなりませんが、音声なら擬音語やオノマトペを使って、ふわっとした形から絞り込んでいくことができます。

**Lemio**：昔見た YouTube の動画で、タイトルが全然わからなくても、「こういう感じのやつ」と探していったら出てきたりするんですよ。

テキスト検索には構造的な限界がある。検索したいものの正確な名前や用語を知らなければ、検索クエリを組み立てられない。しかし音声 AI を使えば、言語化が不完全な状態でも検索が成立する。

たとえば、UI のアニメーションパターンを探しているとき、「要素がふわっと現れて、スッと消える感じのやつ」と音声で伝えれば、AI は「フェードイン・フェードアウトのアニメーション」を提案してくれる。テキストで「CSS アニメーション フェード」と検索するには、まず「フェード」という用語を知っている必要がある。音声なら、オノマトペで伝えるだけでよい。

この手法は、デザインの領域で特に効果を発揮する。「なんかこう、カードがペラッとめくれる感じのトランジション」「ボタンを押したときにポヨンと跳ねるアニメーション」——テキスト検索ではたどり着けない表現でも、音声 AI は意図を汲み取って適切な技術用語やライブラリに変換してくれる。

ハッカソンでは、デザイナーとエンジニアの間のコミュニケーションにも音声 AI の曖昧検索が使える。デザイナーが「ここ、もうちょっとシュッとした感じにしたい」と言ったとき、その曖昧な要望を AI が「余白を増やしてフォントウェイトを細くする」という具体的な実装に変換する。曖昧さを許容する検索は、チーム内の意思疎通の潤滑油にもなる。

従来の検索は「正確なキーワードを知っている人」が有利だった。CSS の `border-radius` を知らなければ「角丸」にたどり着けなかった。

しかし AI の登場で、「なんかこう、角がまるっとしてて、ちょっと浮いてる感じのカード」と言えば、`border-radius` と `box-shadow` の両方を含むコードが返ってくる。正確な技術用語を知らなくても、感覚的な表現で AI が意図を読み取る時代がすでに来ている。

### 音声インターフェースの可能性

**Sae**：ハッカソンで Web サイトやモバイルアプリを作るチームは多いですが、入出力の手段を少し変えるだけで目新しさが生まれます。たとえば入力をテキストではなく音声にするだけで、体験がガラッと変わる。

**Kuu**：ElevenLabs のボイスエージェントがまさにそうで、ユーザーが声で話した内容をトリガーにして次のアクションを実行できる。「話す」と「操作する」が一体になった、

次世代のアプリケーションが作れます。

音声インターフェースは、ハッカソンにおけるプロダクトの差別化要素として見過ごされがちだ。多くのチームが Web アプリやモバイルアプリの UI 設計に時間を費やす中、入出力の定義を見直すだけで独自性が生まれる。

音声インターフェースの強みは、操作負荷の低さとデモ映えの両立にある。キーボードもタッチも不要なハンズフリー体験は、審査員に「触ったことのない操作感」を与え、記憶に残りやすい。

音声 AI 技術は急速に進化しており、リアルタイム音声認識と音声合成を組み合わせたボイスエージェントの構築が個人開発者にも手の届く範囲になった。第 2 章「アイデアと表現」の「入力と出力を定義する」思考法がここでも効く。入力を「音声」、出力を「音声＋アクション」と定義するだけで、従来とは異なるプロダクト体験が見えてくる。

## 4.4　AI 活用のアンチパターン

ここまで効果的な AI 活用法を紹介してきた。最後に、よく見られる失敗パターンを整理する。

表 4.1: AI 活用の 5 つのアンチパターン

| アンチパターン | よくある行動 | 正しいアプローチ |
|---|---|---|
| 情報不足で「使えない」と判断 | AI の出力が期待と違うと諦める | 何の情報が不足していたかを分析する |
| 自分の方が賢いと思い込む | 一行ずつコードを指定する | ゴールを示し AI に道筋を考えさせる |
| ダメだったツールを二度と試さない | 3 ヶ月前の評価を固定する | 定期的にツールを再評価する |
| 無料枠だけで評価する | 無料枠の制限で実力を判断する | 1 ヶ月だけ有料プランを試す |
| 触らずに評論する | 使わずに意見だけ述べる | 少なくとも数時間は自分で触る |

### 情報不足で「使えない」と判断する

**Kuu**：AI に十分な情報を与えずに「使えない」と判断してしまう人が多いんです。AI が間違えたのではなく、自分が与えた情報が足りなかったか、間違っていたと捉えるべきです。

**Lemio**：コンパイルエラーを「叱られてる」と思う感覚に似ていますね。エラーは敵ではなく、何が足りないかを教えてくれている。AI の出力も同じで、期待と違ったら「何を伝え忘れたか」を考える方が建設的です。

AI の出力が期待どおりでないとき、「AI が間違えた」と結論づけるのは典型的なアンチパターンだ。Lemio が言うように、これはコンパイルエラーを「叱られている」と感じる心理に似ている。エラーは敵ではなく、何が足りないかを教えてくれるシグナルだ。プロジェクトの前提条件、使用するライブラリのバージョン、期待する出力形式——これらの情報が欠けていれば、AI は推測で補うしかない。

失敗を「失敗」と思って終わるのではなく、「何が足りなかったかを知るための試み」と

捉える。うまくいかなかった経験こそが、次のプロンプト設計を磨く材料になる。

### 自分の方が賢いと思い込む

**Kuu**：AI を赤ちゃんに教えるような感覚で使う人がいます。全部自分で指示しようとする。でも AI は頼れるパートナーです。まず「どうやるのがベストか調べて」と聞いてから、一緒に進める方がずっとうまくいく。

AI に対して過度に細かい指示を出し続けるのもアンチパターンだ。一行ずつコードを指定するような使い方は、自分で書くのと変わらない。AI の強みは、ゴールを示せばそこまでの道筋を自律的に考えられることにある。

効果的なアプローチは、実装に入る前に AI にベストプラクティスをリサーチさせる段階を挟むことだ。「この機能を実装する前に、一般的なアプローチを 3 つ提案して、それぞれのメリット・デメリットを比較してください」と聞く。AI をリサーチャーとして活用してから実装に進むことで、手戻りが減る。

### 一度ダメなら二度と試さない

**Lemio**：3 ヶ月前に試して使えなかったツールが、今は全然違うものになっていることがあります。Marp というプレゼンツールも、最初は微妙だったけれど、AI と組み合わせたら最高のワークフローになりました。

**Kuu**：AI 系のツールは進化が速すぎて、3 ヶ月前の評価がまったく通用しない。「あれは使えなかった」という記憶を定期的にリセットした方がいいですね。

**Sae**：定期的に触り直す「健康診断」の習慣が大事ですね。年間契約せずに月額で、その時一番いいものを選び続ける。

対話で触れたように、AI ツールは 3 ヶ月で別物に進化する。定期的にツールの再評価を行い、自分の「使えないリスト」を更新する習慣が重要だ。

### 無料枠だけで評価する

**Kuu**：無料枠で判断する人が多いですが、有料プランでないと見えない世界がある。ハッカソンに出るなら、1 ヶ月だけでも有料プランを試してみてほしいです。

AI 活用において、無料枠だけで判断するのは機会損失だ。有料プランでは応答速度、コンテキスト長、利用可能なモデルが大きく異なる。無料枠の制限された体験で「AI は大したことない」と結論づけるのは、試乗車の速度制限でスポーツカーの性能を判断するようなものだ。ただし、課金額が成果に直結するわけではない。大事なのは「AI の本来の能力を正しく評価できる環境で試す」ことであり、際限なく課金することではない。1 ヶ月の有料プランで十分に AI の実力を把握できる。

### 触らずに評論する

**Kuu**：「AI はまだ使えない」「あのツールは微妙」と言う人に限って、自分では触っていないんですよね。手を動かさないと何もわからない。

AI ツールについて意見を述べるなら、まず自分で触ることが前提だ。第 2 章「アイデアと表現」でも述べた通り、「評論より実装」の原則は AI ツールの評価にもそのまま当てはまる。少なくとも数時間は自分の手で試してから判断する習慣を持ちたい。

# 付録 A

# ハッカソンチェックリスト

本編で語られたノウハウを、すぐに使えるチェックリスト形式でまとめました。ハッカソンに参加する方も、運営する方も、この付録を手元に置いてご活用ください。

## A.1　参加者向け、前日までの準備

- 開発環境のセットアップと動作確認
- 使用予定の API キーの取得・設定
- Git 環境の準備（アカウント、SSH 鍵、基本操作の確認）
- AI ツール（チャット型 AI、コーディングアシスタント）のアカウント準備と課金
- Agent Skills（スキルファイル）のテンプレート準備（使用言語・コーディング規約・参照 API をファイルに記載。詳細は第 4 章「AI 活用の実践テクニック」を参照）
- チームコミュニケーションツールの確認（Slack, Discord 等）
- CI/CD パイプラインのテンプレート準備（GitHub Actions 等）
- デモ用のプレゼンツールの準備

## A.2　参加者向け、当日の持ち物

- ノート PC（フル充電済み）
- 充電器・電源タップ
- ディスプレイ変換アダプタ（HDMI 等、会場のプロジェクターに接続できるもの）
- イヤホン・ヘッドセット
- 名刺（あれば）
- 飲み物・軽食
- スマートフォン（デモ撮影・テザリング用）

## A.3　運営者向け、2 週間前からの準備

- **2 週間前**：会場確保、テーマ決定、参加者募集開始

- **1 週間前**：審査基準の策定、審査員への依頼、賞品手配（50 万円以内が税務上スムーズ）
- **3 日前**：タイムテーブル最終確認、参加者への事前案内送付
- **前日**：会場設営、Wi-Fi・電源の動作確認、デモ環境テスト（映像・音声・ネットワーク）
- **当日朝**：受付準備、チーム分け（スキルの多様性よりも方向性の一致を重視）、ルール説明資料の最終確認

## A.4　運営者向け、当日のタイムテーブル

### 1 日型（8 時間）

- 09:00 - 09:30　受付・チーム分け
- 09:30 - 10:00　オープニング・テーマ発表・ルール説明
- 10:00 - 11:00　アイデア出し・チーム内合意
- 11:00 - 17:00　開発タイム（昼食休憩含む）
- 17:00 - 17:30　発表準備・デモテスト
- 17:30 - 18:30　発表・デモ
- 18:30 - 19:00　審査・結果発表・クロージング

### 即興型（5 時間）

- 18:00 - 18:15　テーマ共有・ルール説明
- 18:15 - 18:45　アイデア出し
- 18:45 - 22:00　開発タイム
- 22:00 - 22:15　デモテスト
- 22:15 - 22:45　発表
- 22:45 - 23:00　投票・結果発表

運営のポイント：発表前に必ずデモテストの時間を確保する。映像出力、音声出力、ネットワーク接続、デモデータの 4 点を確認する。拍手で盛り上げてオンタイムに収める工夫も有効だ（詳細は第 3 章「ハッカソン運営と参加の実践知」を参照）。

## A.5　AI 開発環境のセットアップ

- チャット型 AI ツール（ブレスト・調査・学習用）
- コーディングアシスタント（IDE 統合型、有料プランを推奨）
- バージョン管理（Git + GitHub/GitLab）
- CI/CD 環境（GitHub Actions 等、master 直 push 戦略を取る場合は必須。詳細は第 4 章「AI 活用の実践テクニック」を参照）
- API リファレンスのブックマーク（公式ドキュメントの URL を Agent Skills に記載しておく）

ツール選定の基準として、スキルファイルによる拡張性、非同期実行への対応、コンテキスト理解の深さ、エコシステムの広さ、API と CLI の一貫性を重視する（詳細は第 1 章「AI ネイティブ時代のマインドセット」を参照）。月額課金で「その時一番いいもの」を選び続けるのが、変化の速い AI 開発ツールとの正しい付き合い方である。

# あとがき

## この本で伝えたかったこと、AI 時代に変わるものと変わらないもの

この本は、2026 年 2 月のブッカソン（ブック＋ハッカソン＝本を書くハッカソン）で生まれました。

3 人で集まって、ハッカソンや AI について思いの丈を語り合い、その対話を本という形にまとめる——まさにハッカソンのフレームワークを使って本を書くという実験でもありました。

**Kuu**：AI で何でも作れるからこそ、手を動かすことが大事になってきています。

**Lemio**：作りたいものを作ろう。

ツールは変わり続けます。でも「作りたい」という衝動は変わりません。AI の時代だからこそ、限られた時間で人と協働して何かを作る「ハッカソン」の価値は高まっています。

AI がコードを書き、デザインを生成し、文章を整える時代でも、「これが作りたい」という衝動だけは人間の中からしか生まれません。AI はその衝動を形にする速度を劇的に上げてくれますが、「何を作るか」を決めるのは依然として私たちです。

ハッカソンは、その衝動をそのまま発揮できる場です。作って、壊して、また作り直せるプレイグラウンド（遊び場）。締め切りがあり、仲間がいて、発表の場がある。この三要素が揃うことで、普段は「いつかやろう」と先延ばしにしているアイデアが、具体的なプロダクトとして世に出ます。失敗してもリスクはなく、新しい技術を気軽に試せます。

AI の登場で、その「形にする」までの距離が格段に縮まりました。かつては技術的なハードルで諦めていたアイデアが、今なら週末の数時間で動くプロトタイプになる。

> 手を動かし、人と語り、形にする——その繰り返しが未来を作る。

AI が生成する文章と、人間の生の声から生まれる文章。その両方を活かしながら、この本は作られています。読者の皆さんに、私たちの熱量が少しでも伝わっていれば幸いです。

ハッカソンで会いましょう。

### 著者紹介

#### Kuu

久米郁弥。メルカリで Android アプリエンジニアとして従事する傍ら、AI 活用コンサルティング・受託開発を行う Kuu Systems を運営。約 1 万人規模の AI コミュニティ AIAU の運営や、6,000 人超が参加した Cursor Meetup Tokyo の主催など、AI ツール普及の最前線で活動する。Developers Summit 2025/2026 登壇。ハッカソンでは「AI をガードレールで囲んで暴れさせる」スタイルで、速度を最大化する開発手法を追求している。

### Sae

塗木冴。東京工業大学情報工学科卒。ソフトバンクで AI サービス開発、楽天で動画ストリーミング、VIE STYLE でニューロテックを経て、現在はメルカリで iOS 開発に従事。東京拠点の国際ハッカソンコミュニティ MeltingHack の創設者として、Apple Vision Pro ハッカソンやゲームジャムなど多数のイベントを主催。自身も Junction 2023（ヘルシンキ）で Outokumpu Challenge 1 位を獲得するなど、主催者と参加者の両面でハッカソンを愛する。

### Lemio

清水れみお（清水宏太）。高専で電気・電子を学び、通信建設業界で LTE・5G 基地局建設に 10 年従事。2022 年の ChatGPT 登場を機にキャリアを転換し、約 500 社の不採用を乗り越えて AI 業界へ。現在は株式会社ジェネラティブエージェンツで AI Agent Innovator / Dify Expert として活動。共著『Dify で作る生成 AI アプリ完全入門』（日経 BP）。Dify もくもく会を主催し、Dify Meetup Tokyo でも登壇。オタク知識と技術トレンドの交差点からアイデアを生み出し、「入力と出力を言葉にできれば AI が間をつなぐ」という設計言語化のアプローチでハッカソンに挑む。

# AI ネイティブ時代のハッカソン戦略

## 3 人のエンジニアが語る、AI と共創するものづくりの実践知

2026 年 2 月 7 日　初版第 1 刷　発行

著　者　Kuu、Sae、Lemio