不安定な価値の世界（Unstable Value World）におけるシステム設計論：VCDesignと境界指向アーキテクチャ（BOA）の包括的分析レポート

1. 序論：コード生成時代における価値の転換点

1.1「Unstable Value World (uvw)」の前提と定義

現代のソフトウェアエンジニアリングは、アセンブリ言語から高級言語への移行以来、あるいはインターネットの普及以来とも言える巨大なパラダイムシフトの只中にある。この変革の中心にあるのが、「Unstable Value World（uvw、不安定な価値の世界）」と呼ばれる概念的前提である[1]。

かつて、ソースコードは「資産（Asset）」であった。人間が論理を構築し、記述し、レビューを行い、その堅牢性を担保するために多大なコストが支払われた。この時代において、コードは「安定した価値」を持ち、一度書かれたロジックは（バグがない限り）未来永劫、決定論的に動作することが保証されていた。しかし、大規模言語モデル（LLM）と生成的AIの台頭は、この前提を根本から覆した。

uvwにおいて、コード生成の限界費用は限りなくゼロに近づいている。AIは自然言語による曖昧な指示（プロンプト）から、瞬時に実行可能なコードを生成する。これにより、コードは「磨き上げられるべき資産」から、消費され、廃棄され、再生成される「流動的な素材（Fluid Material）」へと変質した[2]。この世界では、ロジックはもはや静的なものではなく、確率的に生成される「揺らぎ」を含んだ不安定な存在となる。

この変化は、エンジニアリングにおける主要な制約条件を「生産（Production）」から「検証（Verification）」へと劇的にシフトさせた。Hironobu Arakawa氏が提唱するVCDesignの哲学によれば、コードを書くコストがゼロになれば、その正しさを検証するコストは対照的に高騰する[2]。人間はAIが生成する膨大な量のコードを、従来の「Code Review」のプロセスで精査することは不可能となる。この「検証のボトルネック」こそが、uvwにおける最大の課題であり、新たなアーキテクチャが求められる根源的な理由である。

1.2「Vibe Coding」と「Context Gap」の脅威

この不安定さを象徴する現象が「Vibe Coding（雰囲気コーディング）」である[3]。AIは文脈（Context）の深層、すなわちビジネスの厳密な制約や暗黙知を完全に理解することなく、表層的なパターンの確率的結合によって「それらしい」コードを出力する。これをVCDesignでは「Context Gap」と呼び、システムに対する重大なリスク要因として位置づけている[4]。

従来のアーキテクチャ、特にMVC（Model-View-Controller）などは、内部のロジック（Controller）が信頼できる静的なコードで記述されていることを前提としている。しかし、Vibe Codingによって生成さ

れたコードは、確率的な「幻覚（Hallucination）」を含む可能性があり、これを信頼済み領域（Trusted Zone）で実行することは、システム全体を汚染するリスクを伴う[3]。

1.3 本レポートの目的と構成

本レポートは、ユーザーの問いに基づき、uvwを前提とした場合、VCDesignおよびその実装形態であるBOA（Boundary-Oriented Architecture）が、次世代のシステム制作の原則的デザインとなり得るかを検証するものである。

分析は以下の観点から多角的に行われる：

1. **VCDesign/BOA**の構造的分析: その哲学「Responsibility Closure（責任の閉包）」と、アーキテクチャ「Triad（三層構造）」の詳細。
2. 比較アーキテクチャ論: MVC、Guardrails AI、Agentic Patterns、DSPyなど、主流および新興アーキテクチャとの比較優位性と欠点。
3. 安全性領域（**Safety Domain**）での適用: 銀行、医療など、失敗が許されない領域（Mission Critical）における決定論的ガードレールの価値。
4. 人間負担とエンジニア像の変容: 「Coder（統治者）」から「System Legislator（システム立法者）」への役割転換。
5. 課題とより良いアプローチの探求: BOAの限界（レイテンシ、過剰設計）と、Language Oriented Programming（LOP）などを組み合わせた発展的アプローチの可能性。

---

2. VCDesignと境界指向アーキテクチャ（BOA）の深層分析

2.1 哲学：**Value Continuity**（価値の連続性）と責任の閉包

VCDesign（Value Continuity Design）は、単なる設計パターンではなく、「AIによる継続的な高速最適化の圧力の中で、システムの価値の連続性をいかに維持するか」という問いに対する設計哲学である[5]。

その核心概念が「Responsibility Closure（責任の閉包）」である[2]。Arakawa氏は、AIが生成する論理（Meaning）を「強力だが有害な産業廃棄物」や「有毒な流体」に例えている。環境法が有害物質の土壌への漏洩を防ぐために厳格な「封じ込め容器」を義務付けるように、AIの思考プロセスもまた、データベースやユーザーの信頼という「清浄な環境」から隔離された「閉包（Closure）」の中で行われなければならないとする。

これは、「AIをCopilot（副操縦士）として信頼する」という現在主流のMicrosoft/GitHub的なアプローチとは対極にある。VCDesignにおいて、AIは「信頼できない外部者」であり、その出力は構造的に隔離されなければならない。

2.2 アーキテクチャ：**BOAのTriad**（三層構造）

BOAはこの哲学を具現化するために、システムを物理的に分断された3つの領域（Triad）に再編す

る。従来のMVCが「機能」で層を分けていたのに対し、BOAは「責任と信頼の所在」で層を分ける[3]。

層（**Zone**）	名称	定義と役割	**AI**の権限	データの性質
Fact	事実（**The Anchor**）	ユーザー入力、DBレコード、ログなどの「不変の事実」。	**Read-Only**。AIは参照のみ可。改変不可。	Immutable（不変）
Meaning	意味（**The Fluid Zone**）	解釈、計算、推論、コード生成が行われる領域。AIの主戦場。	**Full Access**。生成、破棄、再生成が自由。副作用なし。	Fluid（流動的）、Ephemeral（一時的）
Responsibility	責任（**The Protected Zone**）	DB書き込み、APIコール、決済などの「副作用」を伴う実行領域。	**Forbidden**。AIは接触禁止。	Deterministic（決定論的）、Stateful（状態を持つ）

2.2.1 **Fact**（事実：アンカー）

Fact層はシステムの「グラウンド・トゥルース（Ground Truth）」である。ここにあるデータは、AIによる解釈が入る前の生の事実（Raw Fact）である。例えば、「ユーザーがボタンを押した」というイベントログや、「口座残高が1000円である」というDBの記録がこれに当たる。BOAでは、AIがこの層を直接書き換えることを構造的に禁止する。これにより、AIが「事実を捏造（幻覚）」するリスクを根絶する[3]。

2.2.2 **Meaning**（意味：流動領域）

ここはAIのためのサンドボックスである。AIはこの領域内で、Factを材料にして「推論」や「計画」を行う。例えば、「返金処理をすべきか？」という判断や、「返金額の計算」はここで行われる。重要なのは、この領域が副作用（**Side-Effect**）から完全に切り離されている点である[3]。AIがここで「1億円を返金する」という誤った計算（幻覚）を生成しても、それは単なる「文字列」や「オブジェクト」として存在するだけであり、実際の送金処理は走らない。この「安全な思考空間」の確保こそが、uvwにおけるVibe Codingを許容するための必須条件となる。

2.2.3 **Responsibility**（責任：保護領域）

ここは人間（または検証済みの決定論的コード）が支配する領域である。決済APIを叩く、DBを更新するといった「取り返しのつかない（Irreversible）」アクションはすべてここに集約される[6]。BOAの鉄則は、**「AIはいかなる場合もResponsibility層のコードを直接実行してはならない」**という点にあ

る。AIの役割はあくまで「提案（Proposal）」の作成までであり、その提案を実行に移す「トリガー」を引く権限は持たない。

2.3 制御メカニズム：IDGとRP

Meaning（流動的な思考）をResponsibility（堅固な実行）に変換するために、BOAは2つの厳格なプロトコルを用意している。

1. **IDG (Interface Determinability Gate)**: これは「意味」を「責任」へ渡す際の検問所である。IDGは単なるバリデータではなく、型（**Type**）の強制装置である [2]。AIが出力した「返金提案」オブジェクトに対し、IDGは厳密なスキーマチェック（金額は正の整数か？ IDは存在するか？ 理由は定義済みリストにあるか？）を行う。確率的なAIの出力を、決定論的なデータ構造へと「崩壊（Collapse）」させる役割を持つ。
2. **RP (Responsibility Promotion)**: IDGを通過した「提案」を、実際の「実行命令」へと昇格させる手続きである [2]。これには通常、人間による承認（Human-in-the-Loop）や、自動化されたテストスイートのパス（System-in-the-Loop）が署名として要求される。このプロセスを経ることで、AIの出力に対する「責任の所在」が、AIから「承認した人間/テスト」へと移転する。これが「Responsibility Closure」の実装詳細である。

---

3. 比較アーキテクチャ分析：主流および新興モデルとの対比

VCDesign/BOAが「原則デザイン」となり得るかを評価するために、既存の主流アーキテクチャ（MVC）およびAI時代の新興パターン（Guardrails, Agentic, DSPy）と比較分析を行う。

3.1 BOA vs. MVC (Model-View-Controller)

MVCは過去20年間のWeb開発のデファクトスタンダードであった。しかし、uvwの前提に立つと、MVCには致命的な構造的欠陥が存在する。

特徴	**MVC (Model-View-Controller)**	**BOA (Boundary-Oriented Architecture)**
コードの前提	静的・信頼（人間が書き、テスト済み）	流動的・非信頼（AIが生成、幻覚の可能性）
ロジックの所在	Controller / Service層	Meaning層
権限の分離	Controllerが計算とDB更新の両方を行うことが多い（密結合）	計算（Meaning）と実行（Responsibility）が物理的に分断（疎結合）

AI導入時のリスク	AIがControllerを書くと、バグが直接DB破壊や誤送金につながる	AIのバグはIDGで弾かれ、実行層には到達しない（Fail-Safe）
適合性	人間中心のコーディング（Copilot時代以前）	AI中心の自動生成（Vibe Coding時代）

分析: [3]や[3]で指摘されるように、MVCにおけるControllerは「計算」と「実行」が混在しやすい場所である。人間がコードを書く場合、これは効率的だが、AIにコードを書かせる場合、これは「危険地帯」となる。AIが誤って delete_user() を呼び出すコードを生成すれば、MVCではそれを防ぐ構造的障壁がない。BOAは、この「計算」と「実行」の間にてこでも動かない「壁（Boundary）」を設けることで、AI時代の安全性を担保している。

3.2 BOA vs. Guardrails AI / NeMo Guardrails

「Guardrails」は現在、生成AIアプリケーション開発において最も一般的な安全性確保のアプローチである [7]。NVIDIAのNeMo GuardrailsやオープンソースのGuardrails AIなどが代表例である。

比較軸	**Guardrails AI (Library/Framework)**	**BOA (Architecture Pattern)**
定義	LLMの入出力を検証・修正するためのライブラリ・ツール	システム全体の設計思想・構造（トポロジー）
適用箇所	パイプラインの一部（入力前、出力後）	システム全体（DB、API、UIを含む構造）
アプローチ	戦術的（**Tactical**）：特定の出力をフィルタリングする	戦略的（**Strategic**）：危険な操作が不可能な構造を作る
限界	Text-to-SQLなどでAIがDBに直結している場合、フィルタ漏れが致命的になる	AIがDB層から物理的に隔離されているため、フィルタ漏れのリスクが低い
関係性	BOAのIDG（ゲート）を実装するための手段として利用可能	Guardrails AIを包含する上位概念

分析: Guardrails AIは「ツール」であり、BOAは「設計図」である。Guardrails AIを使っても、アーキテクチャが「AIにDBの直接操作権限を与える」形になっていれば、プロンプトインジェクション等で突破

されるリスクが残る [9]。BOAは、そもそもAIがいる場所（Meaning）からDB（Responsibility）への直接的なパスが存在しない構造を強制するため、より根本的な安全性を提供する。しかし、BOAのIDGを実装する際に、Guardrails AIのようなバリデーションライブラリを使用することは極めて有効であり、両者は対立するものではなく補完関係にある。

3.3 BOA vs. Agentic Patterns (Autonomous Agents)

AutoGPTやReAct（Reason + Act）パターンなどの自律エージェントモデルは、AIに「道具（Tool）」を使わせ、自律的にタスクを完遂させることを目指す [11]。

比較軸	**Agentic Patterns (ReAct)**	**BOA (Boundary-Oriented)**
行動原理	思考（Reason）→ 行動（Act）の自律ループ	思考（Meaning）→ 提案 → 承認 → 実行（Responsibility）
AIの権限	APIやツールを直接呼び出す権限を持つ	ツールを呼び出す「提案」をする権限のみ持つ
信頼モデル	AIの推論能力を信頼する	AIの推論能力を信頼しない（Verify first）
主な課題	ループの暴走、ツールの誤用、無限課金 [13]	人間/システムの承認待ちによるレイテンシ、自律性の制限

分析: Agentic Patternは「自律性」を最大化するが、信頼性と安全性が犠牲になりやすい。特に金融や医療などの領域では、AIが勝手にAPIを叩くことは許されない。BOAは、Agenticな動きをMeaning層の中に閉じ込め、外部への作用（Act）をIDGで厳格に管理することで、「安全なエージェント」を実現するための檻（Cage）として機能する。これを「Unstable Value World」における現実的な落とし所とするならば、VCDesignのアプローチは、無制限なAgenticモデルへのアンチテーゼとして機能する。

3.4 BOA vs. DSPy (Declarative Self-improving Python)

DSPyは「プロンプティングではなくプログラミング」を標榜し、LLMパイプラインを最適化するフレームワークである [14]。

比較軸	**DSPy**	**BOA**

目的	LLMの出力精度と信頼性の向上（**Optimization**）	LLMの出力が誤っていてもシステムを守る封じ込め（**Containment**）
アプローチ	コンパイル時にプロンプトや重みを自動調整し、正解率を高める	実行時に出力を厳格に型チェックし、不正なものを弾く
関係性	Meaning層の内部ロジックを高品質化するために使用	システム全体の安全性を担保する外枠

分析: DSPyは「AIを賢くする」アプローチであり、BOAは「AIが賢くなくても大丈夫なようにする」アプローチである。両者は極めて相性が良い。BOAのMeaning層の中でDSPyを用いて高品質な提案を生成させ、それをIDGでチェックするという構成が、現時点で考えうる「より良いアプローチ」の有力候補である。

---

4. 安全性領域（Safety Domain）での利用価値

VCDesign/BOAのアプローチは、特に失敗が許されない「Safety Critical」な領域において、その真価を発揮する。

4.1 金融・銀行領域における「決定論的ガードレール」

銀行システムにおいては、AIによる「確率的な判断」をそのままトランザクションに反映させることは規制上もリスク管理上も不可能である。[16]の研究によれば、金融機関ではGenAIを「確率的なランタイム判断（例：不正検知のスコアリング）」に利用しつつも、その実行は「決定論的なガードレール（例：制裁対象国への送金は無条件で拒否）」で上書きするハイブリッドアーキテクチャが採用されている。

BOAはこの構造を理想的にサポートする。

- **Meaning**層: GenAIが取引パターンを分析し、「不正の疑いあり（スコア0.85）」という提案を生成。
- **IDG/Responsibility**層: 決定論的なルールエンジンが、「スコア0.8以上、かつホワイトリストに含まれない」という条件を厳密に評価し、取引停止を実行。
  このように、確率と決定論を明確に分離することで、説明責任（Accountability）とコンプライアンスを両立できる。

4.2 医療領域（CDSS）における「臨床的ガードレール」

医療判断支援システム（CDSS）においても同様の課題がある。[19]の研究では、AI（データ駆動モデ

ル）の出力を、オントロジーやルールベース（知識駆動モデル）による「臨床的ガードレール（Clinical Guardrails）」で検証するハイブリッドモデルが提案されており、これにより禁忌肢の選択率を18%から6%に低減させている。

BOAの視点では：

- **Fact**: 患者の電子カルテデータ（不変）。
- **Meaning**: AIによる診断推論や処方提案（流動的）。
- **Responsibility**: 処方オーダの発行（保護領域）。
- **IDG**: 薬物相互作用データベースと照合し、禁忌がある場合はAIの提案を却下するゲート。

この構造により、AIがどれほどもっともらしい（Vibe Coding的な）誤診を生成しても、それが患者への処方として実行されるリスクをシステムレベルで遮断できる。これは「Primum non nocere（第一に、害をなすなかれ）」[20]という医療倫理をアーキテクチャレベルで実装することに他ならない。

4.3 説明責任と「責任のギャップ」の解消

AIの自律性が高まるにつれ、「誰が責任を負うのか」という法的・倫理的な問題（Responsibility Gap）が浮上する [21]。完全自律型エージェント（Autonomous Agents）は法的な主体になり得ないため、事故時の責任追及が困難になる。

BOAの「RP（Responsibility Promotion）」プロセスは、この問題に対する明確な回答を提供する。AIの提案を実行に移すには、必ず「承認」のステップが必要であり、そこには「人間の承認（Human Approval）」か「事前定義されたテストセットのパス（System Approval）」という署名が残る。これにより、事故が起きた際の責任は、「承認ボタンを押した人間」または「不十分なテストセットを定義したエンジニア（System Legislator）」に帰属することが明確化される。

---

5. 人間負担の意味とこれからのエンジニア像

VCDesignは、エンジニアの役割を「コードを書く人（Coder）」から「システムの法律家（System Legislator）」へと再定義することを提唱している [2]。

5.1 検証のパラドックスと「溺れる」リスク

uvwでは、コード生成コストがゼロになるため、検証コストが爆発的に増大する（Verification Skyrocket）。もしエンジニアが、AIが生成した大量のコードを従来の「コードレビュー」のように一行一行目で追って確認しようとすれば、彼らは情報の洪水に「溺れて（Drowning）」しまうだろう [2]。人間の認知能力はAIの生成速度に追いつけないため、「マイクロマネジメント（行単位の管理）」は破綻する。

5.2 「System Legislator（システム立法者）」への転換

この状況下で人間が生き残る唯一の方法は、抽象度を上げることである。Arakawa氏はこれを「統

治者（Ruler）」から「立法者（Legislator）」へのシフトと表現する [2]。

- **Ruler**（旧来のエンジニア）: 「Xの場合にYせよ」という命令（Imperative Code）を詳細に記述し、実行フローを完全に支配する。
- **Legislator**（これからのエンジニア）: 市民（AI）の個々の行動には干渉しないが、「憲法（不変の制約）」と「国境（責任の境界）」を設計する。「どのようなロジックが生成されても構わないが、このIDGのスキーマだけは遵守せよ」という境界条件を定義する。

5.3 求められる新たなスキルセット: **Evaluation Engineering**

このシフトに伴い、エンジニアに求められるスキルは「アルゴリズムの実装」から「評価系の設計（Evaluation Design）」へと移行する [23]。これからの開発は「Evaluation-Driven Development (EDD)」となり、以下の能力が核心的価値となる:

1. スキーマ設計力: ビジネス要件を厳密なPydanticモデルやJSON Schemaに落とし込み、AIの出力を縛る能力。
2. 評価指標の策定: 「有用性」「安全性」「正確性」を定量化し、AI判事（LLM-as-a-Judge）を構築・運用する能力 [25]。
3. バウンダリ分析: システムのどこにIDGを設置すれば、最小の労力で最大のリスクを封じ込められるかを見極めるアーキテクチャ設計能力。

---

6. 課題と「より良いアプローチ」の探求

VCDesign/BOAは強力な原則であるが、万能ではない。分析から浮かび上がる課題と、それを超える可能性のあるアプローチについて考察する。

6.1 課題: レイテンシと過剰設計

- レイテンシ（**Latency**）: すべてのAI思考（Meaning）を一度IDGでシリアライズし、検証し、デシリアライズするプロセスは、リアルタイム性を損なう可能性がある [26]。特に音声対話など、数百ミリ秒の遅延がUXを破壊するケースでは、厳格なBOAは適用しにくい場合がある。
- 過剰設計（**Over-Engineering**）: クリエイティブな執筆支援ツールや、リスクの低いチャットボットにおいて、厳密な三層構造と承認プロセスを導入することは、開発スピードを殺ぐ過剰設計となり得る [28]。

6.2 課題: **Context Gap**の残存

BOAは「システムが壊れないこと（安全性）」は保証するが、「ビジネス的に正しいこと（有用性）」までは保証しない。IDGを通過した「構文的に正しい返金提案」が、ビジネス戦略的には「今は返金すべきでない顧客」に対するものである可能性（Context Gap）は排除できない [4]。これを防ぐには、Fact層にビジネスコンテキストを極めて高密度に注入する必要がある。

6.3 「より良いアプローチ」の提言: **Hybrid & Language Oriented**

Programming (LOP)

ユーザーの「より良いアプローチはあるか」という問いに対し、本レポートでは**Language Oriented Programming**（言語指向プログラミング）と**BOA**の融合を提言する。

提言：**LOP-BOA**融合モデル

[30]で言及されているLOPのアプローチは、AIに汎用言語（Python/Java）を書かせるのではなく、そのドメイン専用に設計された**安全なDSL（ドメイン特化言語）**を書かせる手法である。

- 従来**BOA**: AIがPythonコードやJSONを書き、IDGがそれをチェックする。
- **LOP-BOA**:
  1. エンジニア（Legislator）は、副作用や不正操作が物理的に記述不可能なDSLを定義する（例：refund()関数はあるが、delete_db()関数が存在しない言語）。
  2. AIはこのDSLのみを用いてロジック（Meaning）を記述する。
  3. DSLのコンパイラ/インタプリタがIDGの役割を果たし、安全に実行（Responsibility）する。

このアプローチは、ガードレールを「門番」として外付けするのではなく、「言語仕様」として内包するため、より堅牢で、かつランタイムの検証コストをコンパイルタイム（または生成時）にシフトできる可能性がある。さらに、DSLは自然言語に近いため、Context Gapを埋める共通言語としても機能し得る[32]。

---

7. 結論

VCDesignはコード自動生成時代の原則デザインとなり得るか？

結論として、VCDesignおよびBOAが提唱する「Fact/Meaning/Responsibilityの分離」と「Responsibility Closure」の概念は、**uvw**（**Unstable Value World**）における生存のための必須要件であると評価できる。AIの出力を「確率的な流体」として扱い、決定論的なシステムから隔離するという思想は、金融・医療などのSafety Criticalな領域での先行事例とも合致しており、普遍的な妥当性を持つ。

従来のMVCなどのアーキテクチャは、信頼できる人間がコードを書くことを前提としており、Vibe Coding時代にはリスクが高すぎる。一方で、単なるツールとしてのGuardrails AIだけでは、構造的な脆弱性をカバーしきれない。BOAはその間を埋める「構造的な安全性（Structural Safety）」を提供する。

より良いアプローチへの展望

しかし、BOAを教条的に適用するだけでは不十分である。レイテンシや開発効率とのトレードオフを解消するために、**DSPy**による内部ロジックの最適化や、**LOP**（言語指向プログラミング）による安全な記述空間の提供といった技術を、BOAのフレームワークに統合していくことが、次世代の「System Legislator」に求められる真の最適解（Better Approach）となるだろう。

エンジニアはもはや「ビルダー」ではない。荒れ狂う確率の海（AI）に対し、決して崩れない堤防（Boundary）を築く「守護者」としてのアイデンティティを確立する時期に来ている。

参照資料（**Sources**）: .[1]

引用文献

1. 1月 1, 1970にアクセス、https://github.com/unstable-value-world/unstable-value-world
2. From Coders to Legislators: Designing Boundaries in the Age of AI | by Hironobu Arakawa, 1月 30, 2026にアクセス、https://medium.com/@arakawa.hiro/title-from-coders-to-legislators-designing-boundaries-in-the-age-of-ai-d729f2a079cc
3. Is MVC Dead? Why AI Needs “Boundary-Oriented Architecture ..., 1月 30, 2026にアクセス、https://medium.com/@arakawa.hiro/is-mvc-dead-why-ai-needs-boundary-oriented-architecture-boa-631d7bdbce16
4. 1月 1, 1970にアクセス、https://medium.com/@hironobu_arakawa/why-vc-ad-the-context-gap-12345
5. vcdesign-core-prompts/README.md at main - GitHub, 1月 30, 2026にアクセス、https://github.com/VCDesign-org/vcdesign-core-prompts/blob/main/README.md
6. hironobu-arakawa/predictability-gate - GitHub, 1月 30, 2026にアクセス、https://github.com/hironobu-arakawa/predictability-gate
7. guardrails-ai/guardrails: Adding guardrails to large ... - GitHub, 1月 30, 2026にアクセス、https://github.com/guardrails-ai/guardrails
8. Introduction to Generative AI, Second Edition - DOKUMEN.PUB, 1月 30, 2026にアクセス、https://dokumen.pub/introduction-to-generative-ai-second-edition.html
9. AI Guardrails: Tutorial & Best Practices - Patronus AI, 1月 30, 2026にアクセス、https://www.patronus.ai/ai-reliability/ai-guardrails
10. AI guardrails | Thoughtworks United States, 1月 30, 2026にアクセス、https://www.thoughtworks.com/en-us/insights/decoder/a/ai-guardrails
11. Choose a design pattern for your agentic AI system | Cloud Architecture Center, 1月 30, 2026にアクセス、https://docs.cloud.google.com/architecture/choose-design-pattern-agentic-ai-system
12. Agentic AI patterns and workflows on AWS - AWS Prescriptive Guidance, 1月 30, 2026にアクセス、https://docs.aws.amazon.com/prescriptive-guidance/latest/agentic-ai-patterns/introduction.html
13. 7 Must-Know Agentic AI Design Patterns - MachineLearningMastery.com, 1月 30, 2026にアクセス、https://machinelearningmastery.com/7-must-know-agentic-ai-design-patterns/
14. stanfordnlp/dspy: DSPy: The framework for programming ... - GitHub, 1月 30,

2026にアクセス、https://github.com/stanfordnlp/dspy
15. Is It Time To Treat Prompts As Code? A Multi-Use Case Study For Prompt Optimization Using DSPy - arXiv, 1月 30, 2026にアクセス、https://arxiv.org/html/2507.03620v1
16. Deterministic AI vs. Probabilistic AI: Scaling Securely - Moveo.AI, 1月 30, 2026にアクセス、https://moveo.ai/blog/deterministic-ai-vs-probabilistic-ai
17. AI-Powered Cybersecurity Mesh for Financial Transactions: A Generative-Intelligence Paradigm for Payment Security - MDPI, 1月 30, 2026にアクセス、https://www.mdpi.com/2813-0324/12/1/10
18. Probabilistic and Deterministic Logic | by Val Huber - Medium, 1月 30, 2026にアクセス、https://medium.com/@valjhuber/probabilistic-and-deterministic-logic-9a38f98d24a8
19. From engineering principles to healthcare practice: A hybrid reasoning framework for transparent clinical decision support - AccScience Publishing, 1月 30, 2026にアクセス、https://www.accscience.com/journal/AIH/articles/online_first/6109
20. Ethical Responsibility in the Design of Artificial Intelligence (AI) Systems - JMU Scholarly Commons, 1月 30, 2026にアクセス、https://commons.lib.jmu.edu/cgi/viewcontent.cgi?article=1114&context=ijr
21. AI accountability | Carnegie Council for Ethics in International Affairs, 1月 30, 2026にアクセス、https://carnegiecouncil.org/explore-engage/key-terms/ai-accountability
22. A Call for Collaborative Intelligence: Why Human-Agent Systems Should Precede AI Autonomy - arXiv, 1月 30, 2026にアクセス、https://arxiv.org/html/2506.09420v1
23. Generative AI and the Transformation of Software Development Practices - arXiv, 1月 30, 2026にアクセス、https://arxiv.org/html/2510.10819v1
24. From TDD to EDD: Why Evaluation-Driven Development Is the Future of AI Engineering, 1月 30, 2026にアクセス、https://medium.com/@nimrodbusany_9074/from-tdd-to-edd-why-evaluation-driven-development-is-the-future-of-ai-engineering-a5e5796b2af4
25. AI Engineering and Evals as New Layers of Software Work | Towards Data Science, 1月 30, 2026にアクセス、https://towardsdatascience.com/ai-engineering-and-evals-as-new-layers-of-software-work/
26. Performance | Your Enterprise AI needs Guardrails, 1月 30, 2026にアクセス、https://www.guardrailsai.com/docs/concepts/performance
27. No Free Lunch With Guardrails - arXiv, 1月 30, 2026にアクセス、https://arxiv.org/html/2504.00441v2
28. The Hidden Cost of Over-Engineering in Software Development - DEV Community, 1月 30, 2026にアクセス、https://dev.to/alisamir/the-hidden-cost-of-over-engineering-in-software-development-4dnk
29. Microservices: is over-engineering always the best approach? - Metyis, 1月 30, 2026にアクセス、

https://metyis.com/impact/our-insights/is-over-engineering-always-the-best-approach
30. Language-oriented development using functional programming fundamentals - Medium, 1月 30, 2026にアクセス、https://medium.com/@duane.edmonds/language-oriented-development-using-functional-programming-fundamentals-4aed0d752835
31. Neh-Thalggu : Language Oriented Programming for Vibe Coders - ClojureVerse, 1月 30, 2026にアクセス、https://clojureverse.org/t/neh-thalggu-language-oriented-programming-for-vibe-coders/11513
32. Natural Language-Oriented Programming (NLOP): Towards Democratizing Software Creation Corresponding author - arXiv, 1月 30, 2026にアクセス、https://arxiv.org/html/2406.05409v1
33. VCDesign-org/boa-core: Boundary-Oriented Architecture core — separating fact, meaning, and responsibility in AI-assisted systems - GitHub, 1月 30, 2026にアクセス、https://github.com/VCDesign-org/boa-core