

Output-Driven Development: A Paradigm Shift in AI-Assisted Software Engineering

Author: Fuyi (ODDFounder, fuyi.it@live.cn)

Date: 2026-01-12

Version: v7.1 (English Edition)

Abstract

The code generation capabilities of Large Language Models (LLMs) underwent a qualitative transformation between 2023–2025, creating an unprecedented engineering dilemma: **AI generates code far faster than humans can review it.** Traditional software development methodologies (Agile, TDD, BDD, etc.) all assume "human code review" as the final line of defense for quality assurance—an assumption that fails in the AI era.

This paper proposes **Output-Driven Development (ODD)**, a new paradigm designed specifically for AI-assisted software engineering. ODD's core innovations include:

1. **Artifact-centric:** The goal of software development is not generating code, but generating artifacts that satisfy human needs
2. **Contract-driven:** Using precise contracts to define artifact specifications, making requirements quantifiable, verifiable, and testable
3. **Mutation testing trust:** Replacing human review with mutation testing as the foundation of trust
4. **Sealing protection:** Protecting verified code through auditable, traceable version management

We argue that ODD is not merely a methodological innovation, but a **systematic restructuring of production relations** in software development: achieving the historic separation of intellectual labor from executive labor, propelling the software industry from "craft workshop mode" to "intelligent factory mode."

Keywords: ODD, Output-Driven Development, Artifact, Contract, AI-assisted development, Software engineering, Paradigm shift, Mutation testing

Part I: Core Concepts

1. Artifacts: The True Goal of Software Development

1.1 What is an Artifact?

Definition: An Artifact is a verifiable output produced during software development that can satisfy specific human needs.

Essential Definition of Artifact

Artifact = Verifiable Output + Satisfies Specific Need + Has Use Value

Three core properties of artifacts:

1. Verifiability

- Correctness can be confirmed through testing
- Compliance can be checked against contracts

2. Need-fulfilling

- Corresponds to explicit human needs
- Solves concrete business problems

3. Use-value

- Can be directly used by users
- Produces actual utility

1.2 The Relationship Between Artifacts and Human Needs

Philosophical insight: Humans fundamentally need **results**, not processes.

Analogy 1: Sausage and the Hundred-Dollar Bill

Imagine this scenario: You're hungry and want sausage. You have \$100.

The Essence of the Sausage Transaction

`[$100] → [Sausage]`
`(Input) Do you care what happens in between? (Artifact)`

`The meat factory's machines? The chef's cooking? The cashier?`

`Answer: You don't care at all. You just want sausage.`
`If a magic box could turn money directly into sausage,`
`you'd use it without hesitation.`

`Insight: Process is developer self-indulgence;`
`Artifact is the customer's real need.`

Analogy 2: The Official Stamp and Blank Paper

You go to a government office. You hold a paper filled with text (an application).

What's your purpose? Not "queuing," not "talking to the clerk," not "watching them press down on the paper."

Your only purpose is: **To get a red official stamp on this paper.**

Artifact State Transition

`State A: Paper without stamp`

`Value = 0`

`↓`

`[Queue → Submit → Review → Stamp]`

`↓`

`State B: Paper with stamp`

`Value = Permission/Rights`

`Insight: Work is essentially artifact state transition.`

`All processes exist only to transform artifacts`
`from State A to State B.`

Software domain mapping:

A client gives you \$1 million for an e-commerce system. They don't care if you use Java or Go, microservices or monolith. They only care about:

- Can users place orders?

- Can payments succeed?
- Can shipping be tracked?

These are artifacts—things that can be used and create value.

1.3 Artifact Classification System

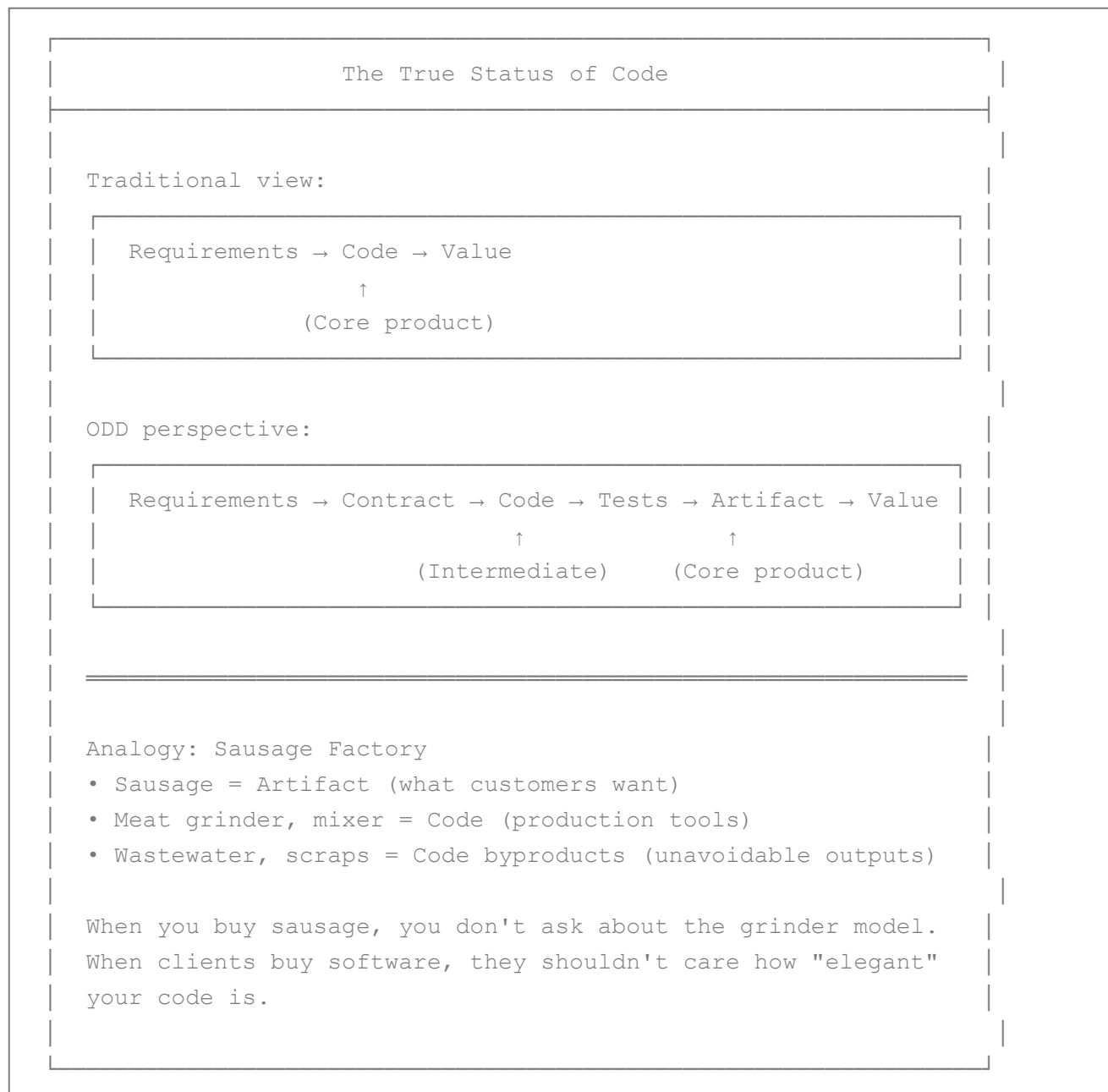
ODD categorizes software development artifacts into 698 types, each with clear definitions, templates, and acceptance criteria.

| 698 Artifact Types Classification (Excerpt) | |
|---|-----------------------------|
| 01. Functional Artifacts (~300 types) | |
| 01.01 API Endpoints | 01.02 Business Services |
| 01.03 Data Processors | 01.04 User Interfaces |
| 01.05 Background Tasks | 01.06 Integration Adapters |
| 02. Verification Artifacts (~150 types) | |
| 02.01 Unit Tests | 02.02 Integration Tests |
| 02.03 E2E Tests | 02.04 Performance Tests |
| 02.05 Security Tests | 02.06 Mutation Test Configs |
| 03. Configuration Artifacts (~100 types) | |
| 03.01 App Config | 03.02 Environment Config |
| 03.03 Build Config | 03.04 Deployment Config |
| 03.05 Monitoring Config | 03.06 Security Config |
| 04. Documentation Artifacts (~80 types) | |
| 04.01 API Docs | 04.02 Architecture Docs |
| 04.03 User Manuals | 04.04 Operations Manuals |
| 04.05 Changelogs | 04.06 Decision Records |
| 05. Contract Artifacts (~68 types) | |
| 05.01 Feature Contracts | 05.02 API Contracts |
| 05.03 Data Contracts | 05.04 Performance Contracts |
| 05.05 Security Contracts | 05.06 Integration Contracts |
| Why 698 types? | |
| • Finer classification = More accurate AI understanding | |
| • Each type has dedicated templates, reducing AI "creativity" | |
| • AI understands these categories; humans define once | |
| • Enables automated acceptance and quality metrics | |

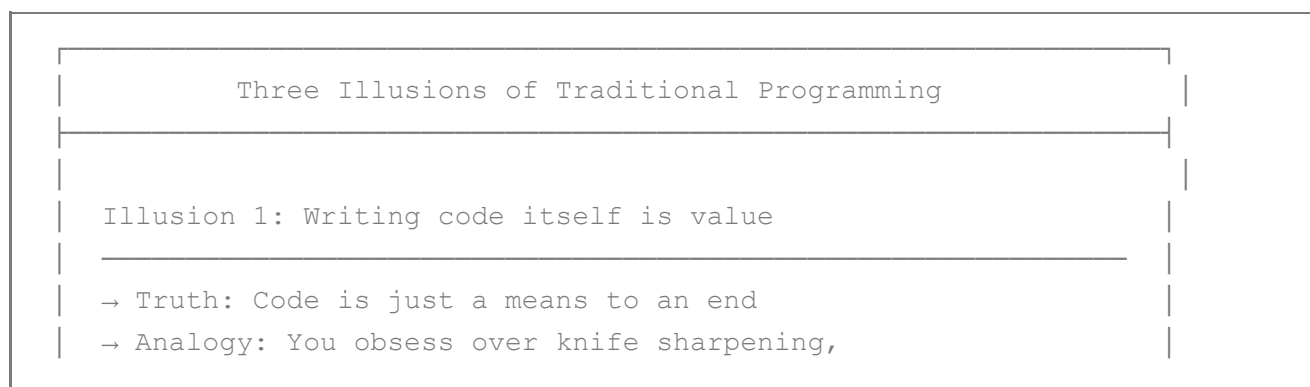
2. Why Generate Artifacts, Not Code?

2.1 The True Status of Code

Traditional thinking holds that programmers' work is "writing code," and code is the programmer's "creation." But this is a **historical misunderstanding**.



2.2 Three Illusions of Traditional Programming Thinking



but the customer just wants the dish

Illusion 2: Code is an asset

→ Truth: Code is a liability (more = harder to maintain,
harder to understand, more bugs)

→ Analogy: More factory machines = higher maintenance costs

Illusion 3: Code quality = Software quality

→ Truth: Artifact correctness = Software quality

→ Analogy: Whether sausage tastes good doesn't depend on
how clean the meat grinder is

Common programmer "self-indulgences":

- Knife skills (code style) → Customer just wants the dish
- Plating (architecture) → Customer only cares about taste
- Cookware (framework) → Customer just wants to be fed

These matter to professional chefs, but for hungry customers,
they're secondary.

2.3 Why ODD Focuses on Artifacts

Why ODD Focuses on Artifacts

Reason 1: Artifacts are verifiable

- Whether code is "elegant" is subjective
- Whether artifacts are "correct" is objective fact
- We can test artifacts; we can't test "code aesthetics"

Reason 2: Artifacts correspond to human needs

- Code corresponds to "implementation approach"
- Artifacts correspond to "user stories"
- Clients accept artifacts, not lines of code

Reason 3: Artifacts can be sealed and protected

- Code can be accidentally modified or overwritten
- Once accepted, artifacts can be sealed
- Sealed artifacts are stable "building blocks"

Reason 4: AI excels at generating code, but acceptance requires humans

- AI can rapidly generate vast amounts of code
- But AI can't judge "is this what the user wanted?"
- Artifact definition and acceptance is human work
- ODD: Humans focus on "define what"; AI focuses on "how"

2.4 Paradigm Shift: From "Generating Code" to "Generating Artifacts"

Paradigm Shift: From "Code" to "Artifacts"

Traditional AI-assisted development (e.g., GitHub Copilot):

```
Human writes → AI completes → Human reviews → Human tests
    ↑             ↑             ↑
  (Human-led)   (AI assists)  (Bottleneck!)
```

Problem: AI generates 100x faster than humans,
but review speed remains human speed

Result: Faster AI = Busier humans = Bigger bottleneck

ODD approach:

```
Human defines contract → AI generates → System verifies → Auto-seal
    ↑                               ↑
  (Human-led)                   (No human bottleneck)
```

Key shifts:

- Humans no longer review code—they define contracts
- System verifies via mutation testing, not human eyes
- Artifacts are central; code is just means to produce them

Part II: Problem Definition

3. The Core Contradiction of the AI Era

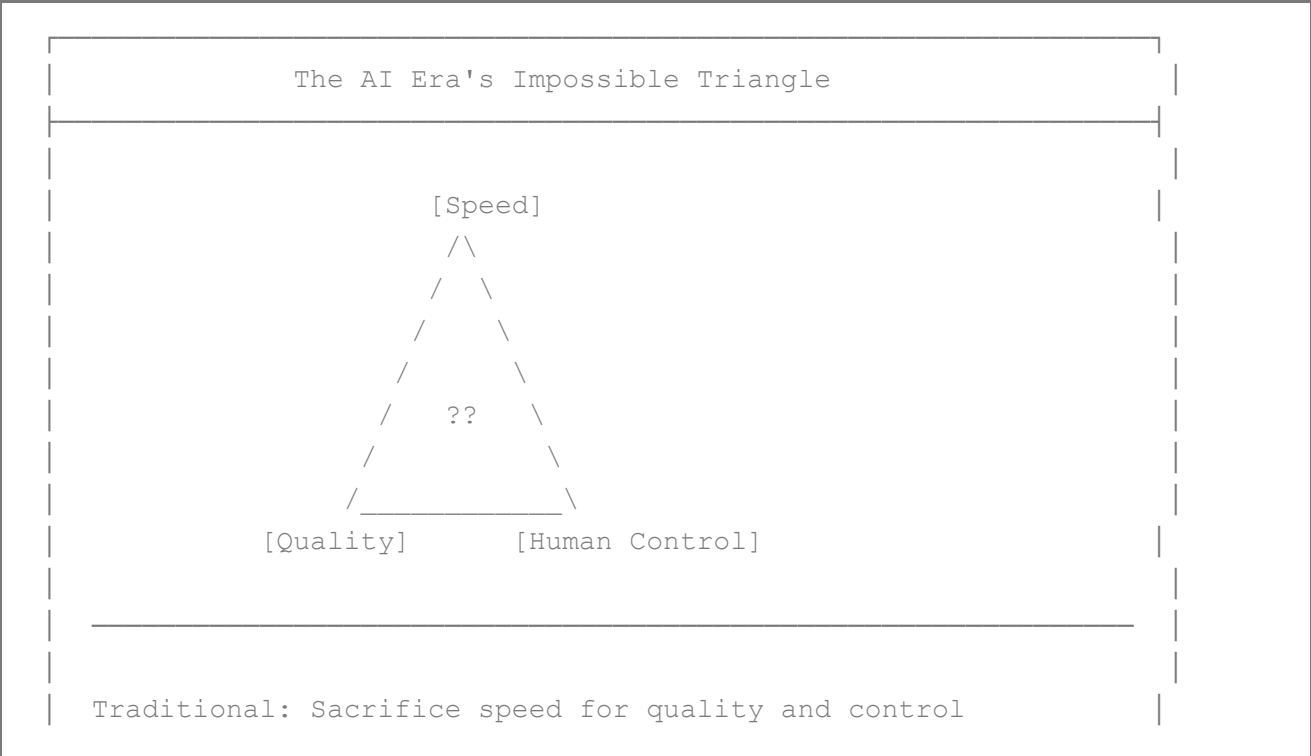
3.1 Qualitative Leap in Productivity

AI code generation technology underwent a qualitative leap between 2023–2025:

| Qualitative Leap in AI Code Generation | | | |
|--|----------------------|---------------------|-------------|
| Dimension | Pre-2023 | 2025 | Change |
| Generation Speed | Human: 1 day/feature | AI: minutes/feature | 100x+ |
| Token Cost | \$0.03/1k tokens | \$0.001/1k | 30x↓ |
| Code Quality | "Barely usable" | "Production" | Qualitative |
| Context | Single file | Entire codebase | Qualitative |
| Multi-language | Limited | Almost all | Qualitative |
| Conclusion: AI can now generate production-grade code at 100x+ human speed | | | |

3.2 The Impossible Triangle

This creates an **impossible triangle**:



- Humans write code, humans review code
- Slow but controllable

Uncontrolled AI: Sacrifice quality and control for speed

- AI generates code, deploy directly
- Fast but uncontrollable, quality not guaranteed

ODD solution: Achieve all three

- Replace human review with mutation testing
- Fast, quality assured, human control via contracts

Core question: How can we trust AI-generated code without reviewing it?

4. Why Traditional Methods Fail

4.1 Hidden Assumptions of Traditional Methodologies

All traditional software development methodologies share a common **hidden assumption**:

Hidden Assumptions of Traditional Methods

| Method | Hidden Assumption | Why It Fails in AI Era |
|------------------|---------------------------|-----------------------------|
| TDD | Human writes tests+code | "Self-grading" untrusted |
| BDD | Human defines+implements | AI still needs review |
| DbC | Contract embedded in code | Contract-code coupling |
| Code Review | Human reviews human code | AI volume exceeds review |
| Agile | Team understands context | AI can't read between lines |
| Waterfall | Document-driven | Docs can't be auto-verified |
| Spec Programming | Markdown specs | Vague, untestable |
| Vibe Coding | Natural language | No precise acceptance |

Shared assumption: Human code review is the final defense

This assumption fails in the AI era.

Because:

- AI generation speed >> Human review speed
- AI generation volume >> Human review capacity
- Human review of AI code << Human review of human code
(AI's logic may differ completely from human thinking)

4.2 Comprehensive Comparison: ODD vs Traditional Methods

| ODD vs Traditional Methodologies Comparison | | | | |
|---|-------------|--------------|-------------|-------------|
| Dimension | Waterfall | Agile/Scrum | TDD | ODD |
| Requirements | Documents | User Stories | Test Cases | Contracts |
| Code Author | Human | Human | Human | AI |
| Test Author | QA Team | Developers | Developers | AI |
| Quality Assurance | Manual QA | CI | Coverage | Mutation |
| Test Review Mechanism | Code Review | Code Review | Code Review | System |
| Verify Trust Foundation | Human Judge | Human Judge | Tests Pass | Math Proof |
| Scaling Method | Add People | Add People | Add People | Add Compute |
| Iteration Cycle | Months | Weeks | Days | Hours |
| Knowledge Store | Documents | Wiki | Tests | Contracts |
| Precision | Low | Medium | Medium-High | High |
| Verifiability | Low | Low | Medium | High |
| AI Era Fitness | X | △ | △ | ✓ |

Legend:

X = Not adapted (core assumptions fail)

△ = Partially adapted (requires heavy human involvement)

✓ = Fully adapted (designed for AI era)

Why Spec Programming and Vibe Coding aren't enough?

Specification Programming:

- Describes requirements in Markdown
- Problem: Vague, non-quantifiable, not auto-testable
- Example: "System should respond quickly" → How fast? Unverifiable

Vibe Coding:

- Natural language interaction with AI
- Problem: No precise acceptance criteria, unpredictable results
- Example: "Write me a login feature" → AI may understand differently

ODD Contracts:

- Precisely define inputs, outputs, boundaries, acceptance criteria
- Quantifiable, testable, verifiable
- Example: Contract defines "response time < 200ms", auto-verifiable

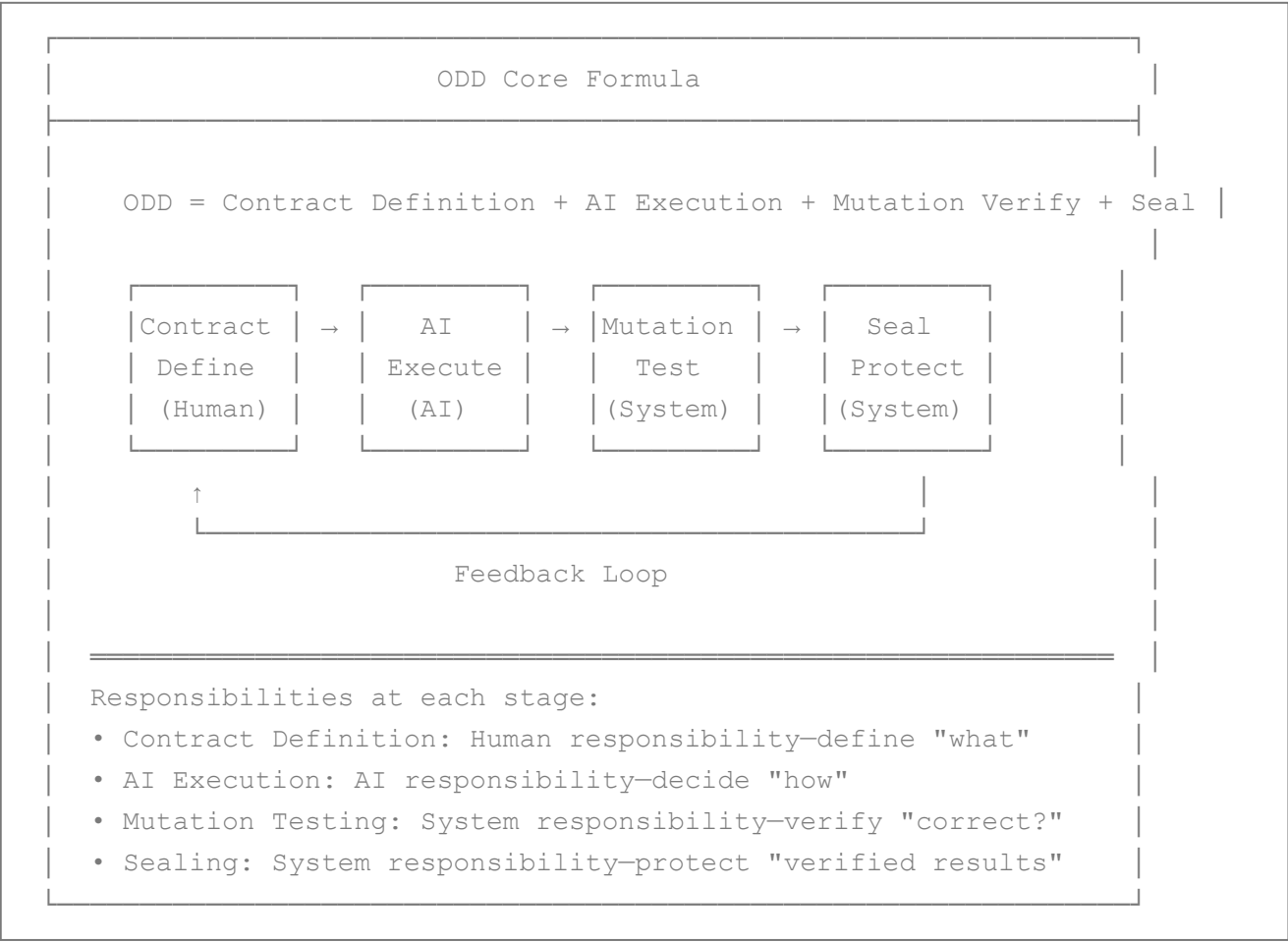
Part III: What is ODD

5. ODD Definition

5.1 One-Sentence Definition

*ODD (Output-Driven Development) is a software development paradigm for the AI era:
Humans define artifact specifications (contracts), AI generates implementation code, the system verifies correctness through mutation testing, and correct artifacts are protected through sealing.*

5.2 ODD Core Formula



5.3 Five Core Characteristics of ODD

| Five Core Characteristics of ODD | |
|---------------------------------------|---|
| ① Humans Don't Write Code | <ul style="list-style-type: none">Humans only define contracts; 100% code by AIAll complexity delegated to AIDomain experts can participate directly (no coding skills)Human value is "defining what," not "implementing how" |
| ② Humans Can Skip Code Review | <ul style="list-style-type: none">Mutation testing provides trust foundation (math, not intuition)"Is code correct?" verified by system, not human judgmentFrees human bandwidth to focus on defining valueHuman review goes from "mandatory" to "optional" |
| ③ Sealed Code Can't Be Modified by AI | <ul style="list-style-type: none">Verified code is protected from accidental AI changesPrevents AI from breaking B while fixing AAuditable: Every seal has complete records (who, when, why)Traceable: Any version can be restoredSystem has "regret capability": Errors can be rolled back |
| ④ Infinite Parallel Scaling | <ul style="list-style-type: none">AI "worker" count limited only by compute and LLM speedDistributed development scales infinitely1 person + ODD \approx Traditional small team (5-8 people)No interpersonal communication cost, no meeting overhead |
| ⑤ Define on Phone, Produce in Cloud | <ul style="list-style-type: none">Support contract definition on mobileInvoke cloud computing resources to generate artifactsDeliver use-value humans needAnytime, anywhere, on-demand production |

5.4 ODD is Tool-Agnostic

Important declaration: ODD is a **methodology**, not a product. It can be implemented with any tools.

| ODD Tool Independence | |
|-----------------------|------------------------|
| Component | Implementation Options |

| | |
|------------------|--|
| LLM Engine | Claude, GPT-4, Gemini, LLaMA, Qwen, DeepSeek, Local models, Any code-gen LLM |
| Mutation Testing | Stryker (JS/TS), Pitest (Java), mutmut (Python), Mull (C++), Custom tools |
| Version/Sealing | Git + Custom extensions, Database + Code mgmt, SVN, Mercurial, Even manual management |
| Contract Storage | Database + Code, JSON, YAML, XML, Natural language + Structured templates |
| Execution Env | Cloud, Local, Hybrid, Edge computing |

Progee is a reference implementation of ODD, but ODD itself is an open methodology. Anyone can practice ODD with any tools.

ODD's value is in the ideas, not specific tools:

- Artifact-centric
- Contract-driven development
- Mutation testing verification
- Sealing protection mechanism

Part IV: Contract System

6. Contracts: Precise Agreements Defining Artifacts

6.1 Contract Definition

Traditional understanding (incomplete): A contract is an "agreement" between humans and AI.

ODD definition (complete):

***Contract** is a precise agreement defining artifacts—a specification that transforms requirements into utility.*

Contracts can be used for collaboration between humans, between humans and AI, or between AI agents.

Contracts are particularly suitable for AI understanding because they are structured, quantifiable, and testable.

Complete Definition of Contract

Contract = Precise Artifact Definition + Requirements as Utility + Quantifiable & Testable

Three core characteristics of contracts:

1. Precision

- Defines explicit inputs, outputs, boundary conditions
- No gray areas, no "roughly," no "close enough"
- Machine-parseable and understandable

2. Utility-oriented

- Focuses on artifact use-value
- Defines "what to do," not "how to do it"
- Acceptance based on utility, not implementation

3. Verifiability

- Every stipulation can be verified through testing
- Acceptance criteria are executable
- Success or failure is binary-no "partial success"

Contract applicability:

- Human ↔ Human: Clear division among team members
- Human ↔ AI: Human defines requirements, AI implements
- AI ↔ AI: Multi-agent collaboration
- System ↔ System: API contracts between microservices

6.2 Contracts vs Other Requirement Expression Methods

Contracts vs Other Requirement Expression Methods

| Method | Characteristics | Problems |
|--------|-----------------|----------|
|--------|-----------------|----------|

| | | |
|------------------|---------------------------------|----------------------|
| Natural Language | "System should respond quickly" | Vague, untestable |
| User Stories | "As a user I want..." | Lacks precise bounds |
| Markdown Docs | Structured natural lang | Still vague |
| UML Diagrams | Graphical description | Hard to auto-verify |
| Test Cases | Executable verification | Lacks full picture |

ODD Contracts Precise+Utility+Verifiable ← Best for AI era

Concrete example:

Natural language:

"When users fail login too many times, lock the account"

Problems: How many is "too many"? How long locked? How unlock?

Markdown document:

"## Account Locking Feature

- Lock when failures exceed threshold
- Notify user when locked"

Problems: What threshold? What notification content?

ODD Contract:

```
{
  "feature": "account_lock",
  "trigger": {"failed_attempts": 5, "window": "5min"},
  "action": {"lock_duration": "30min"},
  "response": {"code": "ACCOUNT_LOCKED", "message": "..."},
  "acceptance": [
    "Given 4 failures When 5th attempt Then lock account",
    "Given locked account When login Then return ACCOUNT_LOCKED"
  ]
}
```

Advantages: Precise, testable, AI-understandable, auto-verifiable

6.3 Contract Structure

```
{
  "contract_id": "LOGIN-001",
  "version": "1.0.0",
  "name": "User Login",
  "description": "Verify user credentials and return authentication token",

  "input": {
    "username": {
```



```
    "type": "string",
    "constraints": ["non-empty", "length 3-20", "alphanumeric and
underscore only"]
  },
  "password": {
    "type": "string",
    "constraints": ["non-empty", "length 8-128"]
  }
},

"output": {
  "success_case": {
    "token": "JWT token, valid for 3600 seconds",
    "expires_in": "number, seconds"
  },
  "failure_cases": [
    {"code": "INVALID_CREDENTIALS", "message": "Invalid username or
password"},
    {"code": "ACCOUNT_LOCKED", "message": "Account locked, try again in
30 minutes"},
    {"code": "ACCOUNT_DISABLED", "message": "Account disabled, contact
administrator"}
  ]
},

"acceptance_criteria": [
  "Given valid credentials When login Then return valid JWT token,
expires in 3600s",
  "Given invalid password When login Then return INVALID_CREDENTIALS,
don't reveal specifics",
  "Given 5 consecutive failures (within 5 min) When 6th attempt Then
return ACCOUNT_LOCKED",
  "Given disabled account When login Then return ACCOUNT_DISABLED"
],

"boundary_conditions": [
  "Empty username → Reject immediately, no DB query, return 400",
  "Empty password → Reject immediately, no DB query, return 400",
  "Overlong password (>128 chars) → Reject immediately, return 400",
  "Username with special chars → Reject immediately, return 400"
],

"non_functional": {
  "performance": "Response time < 200ms (p99)",
  "security": "Password not logged, use bcrypt(cost=12) hashing",
  "availability": "99.9% uptime"
},

"metadata": {
  "author": "contract-architect@example.com",
```

```
"created": "2026-01-10",
"status": "approved"
}
}
```

6.4 Relationship Between Contracts and Tasks

After contract confirmation, the system automatically decomposes it into specific tasks. Each task produces one concrete artifact.

Contract and Task Interface Display:

Contract: Order Creation Feature

Belongs to: [Order Module ▼] / [- ▼]

Do: [Create order, check inventory, reject if insufficient]

Don't do: [Don't process payment; don't send notifications]

Files: [order.py] [order_api.py] [+]

Dependencies:[User Account ▼] [+]

Precondition:[User table exists; Product table exists]

Postcondition:[Order table has record; Inventory deducted]

Task List [+ Add] [Merge Selected] [AI Re-decompose]

Task 1: Order Table Model

Approved

Key Elements Explained:

| Element | Description |
|---------------------|--|
| Do/Don't do | Define contract boundaries, prevent AI "overreach" |
| Dependencies | Declare inter-contract dependencies, system auto-orders |
| Pre/Post conditions | Define execution conditions before and after |
| Input/Output | Precisely define task I/O specifications |
| Verification method | Specify how to verify task completion (SQL/pytest/curl/manual) |
| Approved/Pending | Human confirms if task decomposition is reasonable |

7. Clarity Assessment Mechanism

7.1 Why Clarity Assessment?

Human natural language is full of ambiguity. ODD uses clarity assessment to identify and resolve ambiguity before code generation.

7.2 Red-Yellow-Green Clarity Scoring

Clarity Assessment: Red-Yellow-Green System

● Green (Clarity 80-100%)

- Contract is precisely defined, no ambiguity
- Can proceed directly to code generation
- Example: "When order amount \geq \$10,000, notify finance via SMS"

● Yellow (Clarity 50-79%)

- Contract mostly clear, but has minor ambiguities
- Need human confirmation on few questions before generation
- Example: "When order amount is large, notify manager"
→ Need clarification: How large? Which manager?

● Red (Clarity 0-49%)

- Contract has major ambiguity or contradictions

- Cannot generate code, must clarify first
- Example: "System should be user-friendly"
 - What's "user-friendly"? No testable definition

7.3 Clarity Assessment Example

Original contract:

"When processing large orders, the system should notify the manager."

Clarity assessment result: ● Yellow (Clarity 45%)

Found 4 ambiguities, please clarify:

Q1: What is the threshold for "large order"? [Impact: 25%]

[A] \$1,000+

[B] \$5,000+

[C] \$10,000+ ← AI recommended (industry standard)

[D] Other: [_____]

Q2: What notification method? [Impact: 20%]

[A] In-system message only

[B] Email

[C] SMS ← AI recommended (large orders need immediate attention)

[D] Multiple methods combined

Q3: Who is "the manager"? [Impact: 20%]

[A] All system administrators

[B] Order department manager ← AI recommended

[C] Designated on-duty manager

Q4: When does "processing" occur? [Impact: 15%]

[A] Order creation ← AI recommended

[B] Order payment

[C] Order shipment

Current clarity: 45% ● → After answering all: Expected 95% ●

Clarified contract (Clarity 95% ●):

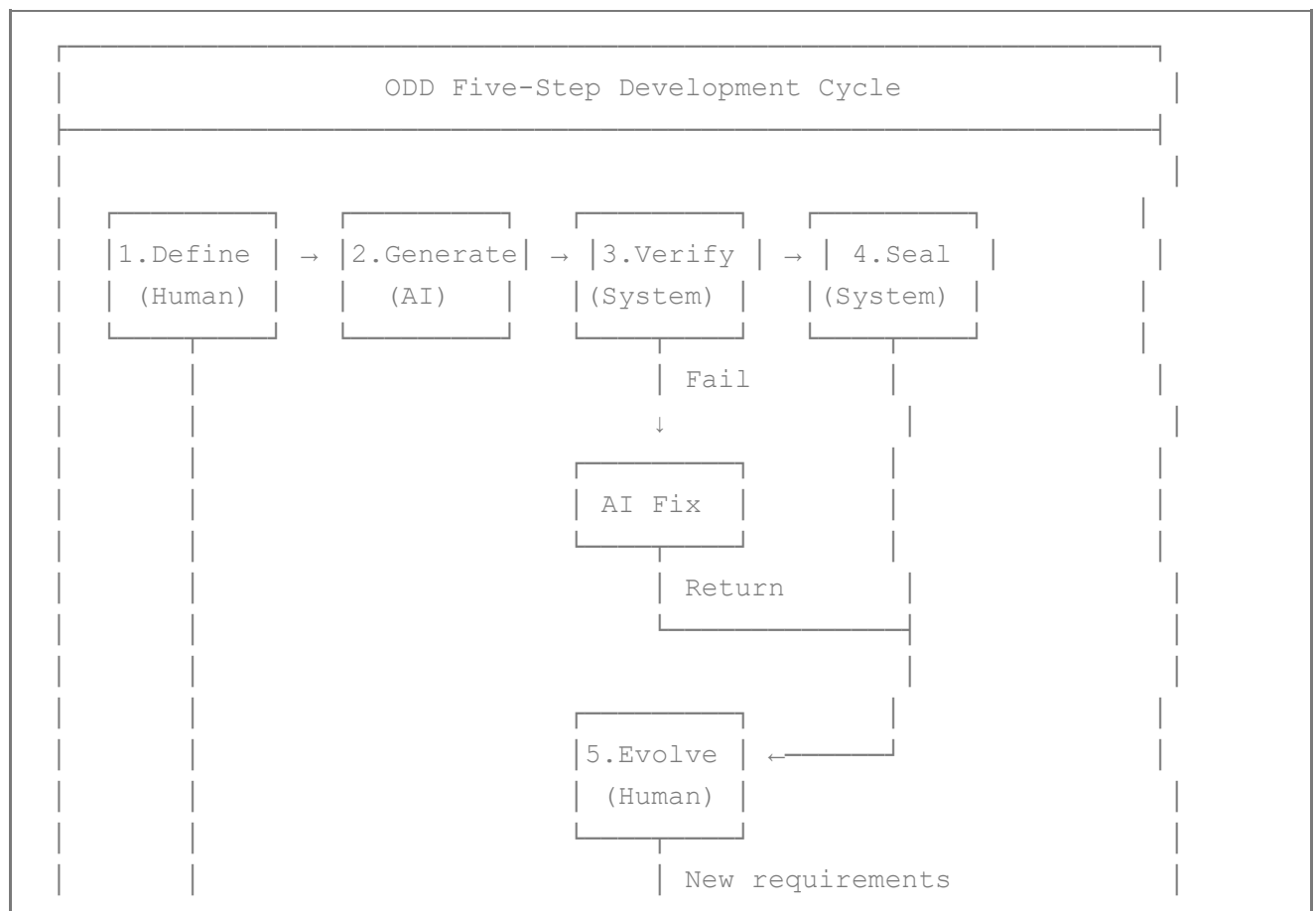
```
{  
  "feature": "large_order_notification",  
  "trigger": {
```

```
    "event": "order_created",
    "condition": "amount >= 10000"
  },
  "action": {
    "method": "sms",
    "recipient": "department_manager_of_order"
  },
  "acceptance_criteria": [
    "Given order amount = $10,000 When created Then SMS notify dept manager",
    "Given order amount = $9,999 When created Then no notification",
    "Given order amount = $50,000 When created Then SMS notify dept manager"
  ]
}
```

Part V: ODD Methodology Framework

8. ODD Five-Step Development Cycle

8.1 Cycle Overview

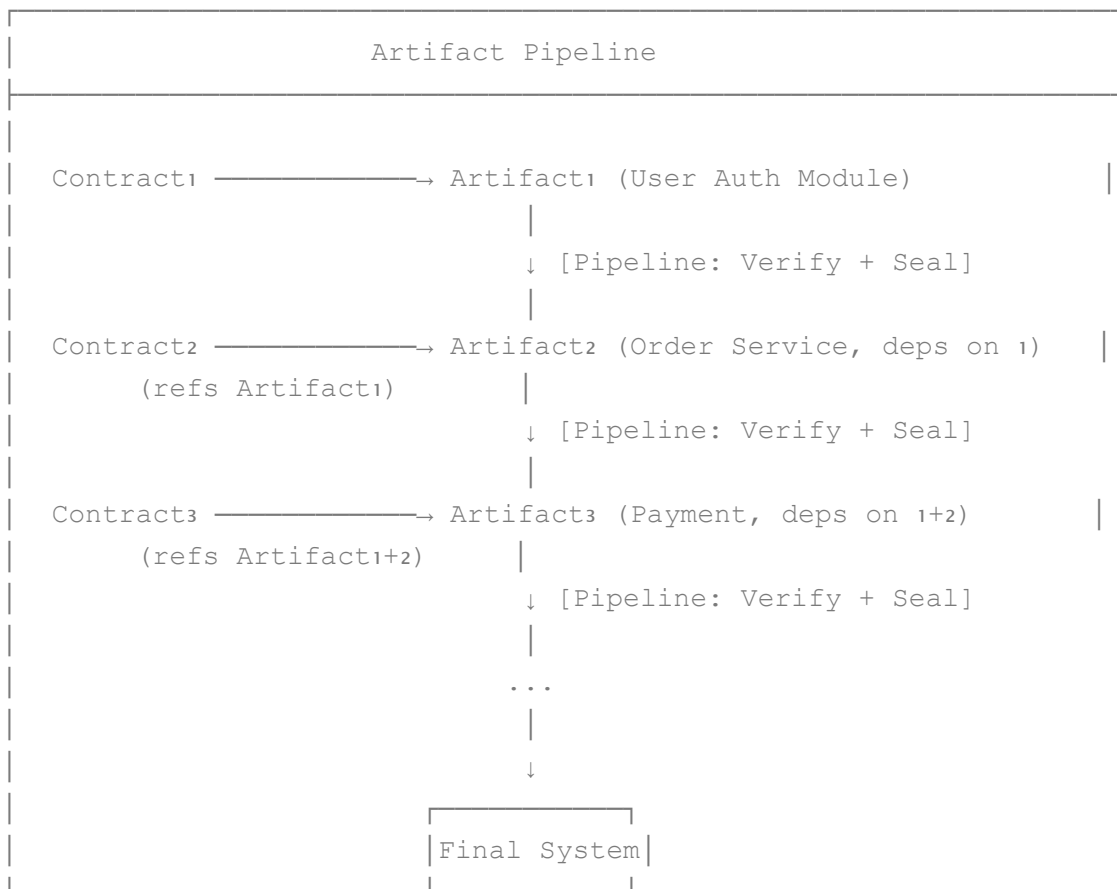


Step descriptions:

1. Define: Human writes contract, defines artifact specs
2. Generate: AI generates code and tests from contract
3. Verify: System runs mutation testing, verifies correctness
4. Seal: After verification, code is sealed and protected
5. Evolve: Based on new needs, human updates contract

8.2 Artifact Pipeline: Artifact → Pipeline → New Artifact

ODD's core insight: **Every artifact is input for the next contract.**



Key insights:

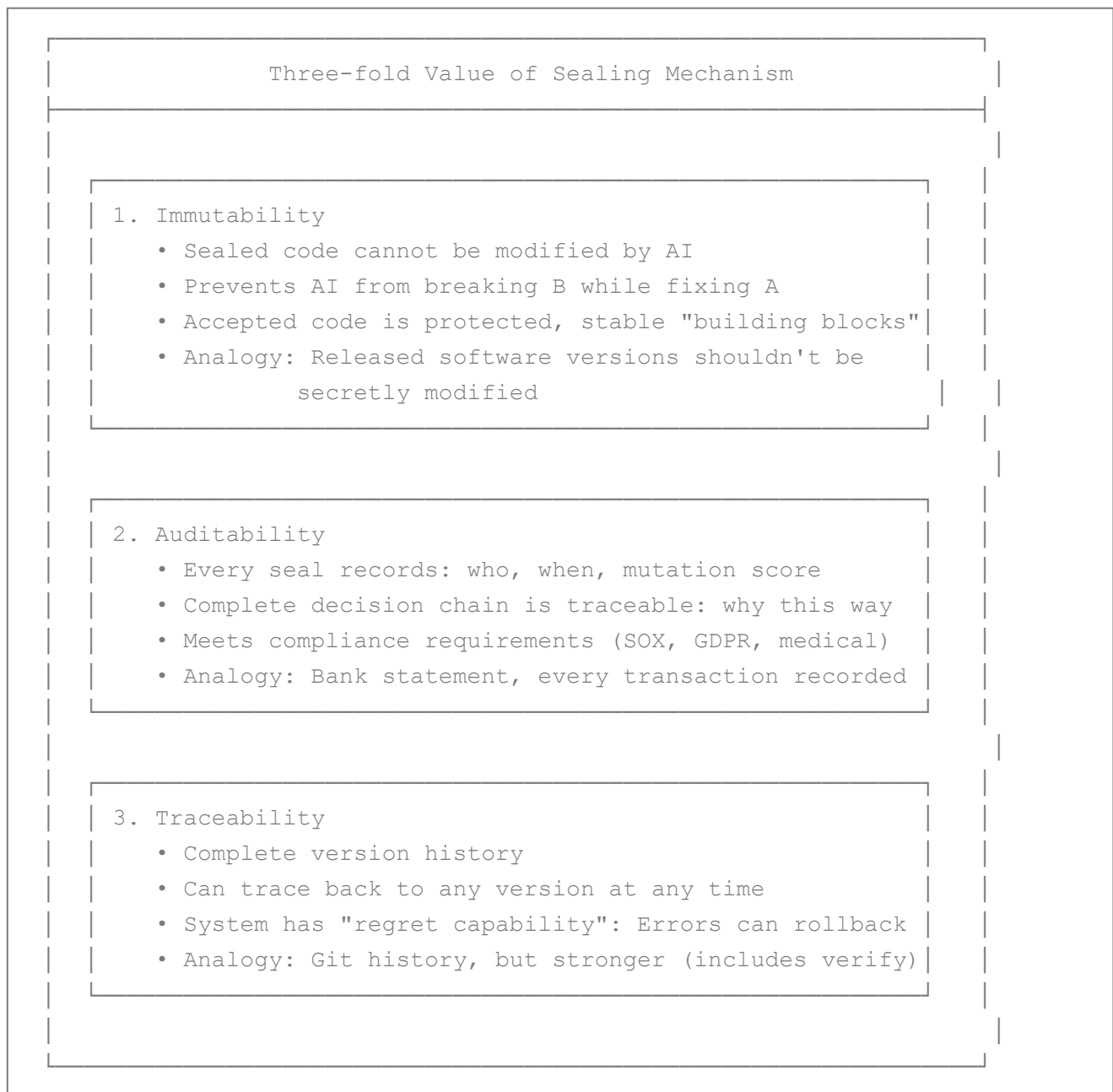
- System builds through layers of sealed artifacts
- Like assembly line parts becoming a complete vehicle
- Each artifact is a stable "building block"
- Dependencies between artifacts are explicit, traceable

Difference from traditional development:

- Traditional: Code depends on code, dependencies unclear
- ODD: Artifacts depend on artifacts, explicit & verifiable

9. Sealing Mechanism: Materialization of Trust

9.1 Three-fold Value of Sealing



9.2 Seal Record Structure

Seal records can be stored in database + code management systems:

```
{
  "seal_id": "SEAL-2026-01-10-001",
  "artifact_id": "USER-AUTH-001",
  "version": "1.0.0",
  "sealed_at": "2026-01-10T14:32:00Z",
  "sealed_by": "system",

  "verification_results": {
    "mutation_score": 92.5,
```



```

    "total_mutants": 156,
    "killed_mutants": 144,
    "survived_mutants": 12,
    "test_count": 48,
    "test_pass_rate": 100
  },

  "hashes": {
    "contract_hash": "sha256:a1b2c3d4e5f6...",
    "code_hash": "sha256:f6e5d4c3b2a1...",
    "test_hash": "sha256:1a2b3c4d5e6f..."
  },

  "dependencies": [
    {"artifact_id": "COMMON-UTILS-001", "version": "2.1.0"},
    {"artifact_id": "DATABASE-001", "version": "1.5.0"}
  ],

  "rollback_info": {
    "previous_version": "0.9.0",
    "can_rollback": true,
    "rollback_command": "odd rollback USER-AUTH-001 --to=0.9.0"
  },

  "audit_trail": [
    {"action": "contract_created", "by": "architect@example.com", "at": "2026-01-05"},
    {"action": "contract_approved", "by": "tech-lead@example.com", "at": "2026-01-08"},
    {"action": "code_generated", "by": "ai-worker-3", "at": "2026-01-09T10:15:00Z"},
    {"action": "mutation_test_started", "by": "system", "at": "2026-01-10T13:00:00Z"},
    {"action": "mutation_test_passed", "by": "system", "at": "2026-01-10T14:30:00Z"},
    {"action": "sealed", "by": "system", "at": "2026-01-10T14:32:00Z"}
  ]
}

```

9.3 Seal History Example

Seal History:

```

└─ v1.0.0 (2026-01-10 14:32, mutation 92%, sealed by: system) ← Current
  production
  │   └─ Contract: LOGIN-001 v3
  │   └─ Audit: Complete records available
└─ v0.9.0 (2026-01-08 10:15, mutation 88%, sealed by: system) ←
  Rollback available
  │   └─ Contract: LOGIN-001 v2

```

```
|   └─ Change reason: Added account locking
└─ v0.8.0 (2026-01-05 09:00, mutation 85%, sealed by: system)  ←
Rollback available
|   └─ Contract: LOGIN-001 v1
|   └─ Change reason: Initial version
└─ v0.1.0 (2026-01-01 08:00, mutation 70%, sealed by: human)  ← Manual
    seal (prototype)
    └─ Note: Prototype validation, mutation below threshold, manually
        approved

If v1.0.0 has problems:
  → Execute: odd rollback USER-AUTH-001 --to=0.9.0
  → System auto-rolls back to v0.9.0 (system has "regret" capability)
  → Records rollback reason and operator
  → New contract fixes issue → Generate v1.1.0 → Verify → Seal
```

Part VI: Trust System

10. Mutation Testing: Mathematical Foundation of Trust

10.1 Why Can Mutation Testing Replace Human Review?

Core question: How do you know your tests are effective?

Traditional approach: Code Coverage. But coverage has a fatal flaw:

The Lie of Code Coverage

```
function divide(a, b) {
  return a / b;  // No check for b == 0
}
```

Test: `divide(10, 2)` → Result: 5 ✓

Code Coverage: 100% ✓

Question: Are the tests actually effective?

If code changes to: `return a * b;`

Test `divide(10, 2)` expects 5, gets 20, test fails ✓

```
But if code changes to: return a / b + 0;
Test divide(10, 2) expects 5, gets 5, test still passes X
This mutant "survives"—tests aren't strict enough

Key: Coverage only shows "code was executed,"
    not "tests can detect errors"
```

Core idea of mutation testing:

*Good tests should detect any subtle error in the code.
If we deliberately introduce errors (mutants), good tests should "kill" these mutants.*

10.2 How Mutation Testing Works

Mutation Testing Workflow

Original code:

```
function isAdult(age) {
  return age >= 18;
}
```

System generates mutants:

```
Mutant 1: return age > 18;      // >= changed to >
Mutant 2: return age >= 17;     // 18 changed to 17
Mutant 3: return age >= 19;     // 18 changed to 19
Mutant 4: return age <= 18;     // >= changed to <=
Mutant 5: return true;         // Logic replaced
```

Run tests against each mutant:

```
Mutant 1: Test isAdult(18) expects true, gets false → Killed
Mutant 2: Test isAdult(17) expects false, gets true → Killed
Mutant 3: Test isAdult(18) expects true, gets false → Killed
Mutant 4: Test isAdult(17) expects false, gets true → Killed
Mutant 5: Test isAdult(10) expects false, gets true → Killed
```

Mutation Score = Killed Mutants / Total Mutants = 5/5 = 100%

Why mutation testing provides trust:

- If a mutant survives, tests have a blind spot
- If all mutants are killed, tests are comprehensive

- This is mathematical proof, not human intuition

10.3 Coverage vs Mutation Score

Code Coverage vs Mutation Score Comparison

| Metric | Code Coverage | Mutation Score |
|--------------|-------------------|-------------------------|
| Measures | How much code ran | Can tests detect errors |
| Tells you | Execution paths | Test effectiveness |
| Fakeability | Easy to fake | Hard to fake |
| Compute cost | Low | High |
| Trust level | Low | High |

Analogy:

- Coverage = Number of pages you've turned in the textbook
- Mutation = Score on the exam

You can flip through all pages (100% coverage) without learning anything (0% mutation score).

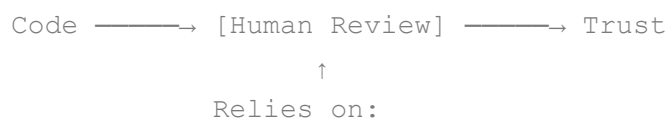
Why mutation testing can replace human review:

- Human review relies on intuition and experience
- Mutation testing relies on mathematical exhaustiveness
- Humans can miss errors; mutation testing finds all testable ones
- Human review doesn't scale; mutation testing scales with compute

10.4 Trust Transfer: From Human Review to System Verification

Trust Transfer: From Human Review to System Verification

Traditional model:



- Experience
- Intuition
- Available time
- Cognitive load

Problems: Doesn't scale, varies by person, can miss errors

ODD model:

Code → [Mutation Testing] → Trust

↑

Relies on:

- Mathematical exhaustiveness
- Automated execution
- Quantifiable results
- Infinite scalability

Advantages: Scales, consistent, finds all testable errors

Key insight:

Trust doesn't disappear—it transfers from human to system.

The source of trust changes, not its existence.

Part VII: Paradigm Evolution and Production Relations Restructuring

11. Software Development Paradigm Evolution Roadmap

Software Development Paradigm Evolution

| | | |
|-------|-----------|---|
| 1960s | Waterfall | Document-driven, sequential Problem: Assumes stable requirements |
| | ↓ | |
| 1990s | Agile | Iterative, user-story driven Problem: Still human-writes-code |

| | | |
|-------|------------------|--|
| 2000s | ↓ TDD | Test-first, red-green-refactor Problem: Self-grading is untrusted |
| 2020s | ↓ AI-assisted | Copilot-style code completion Problem: AI generates, human reviews |
| 2025+ | ↓ ODD | Contract-driven, mutation-verified Solution: System verifies, human defines value |

Each paradigm solves previous paradigm's core contradiction
ODD solves AI era's core contradiction:
"AI generates faster than humans can review"

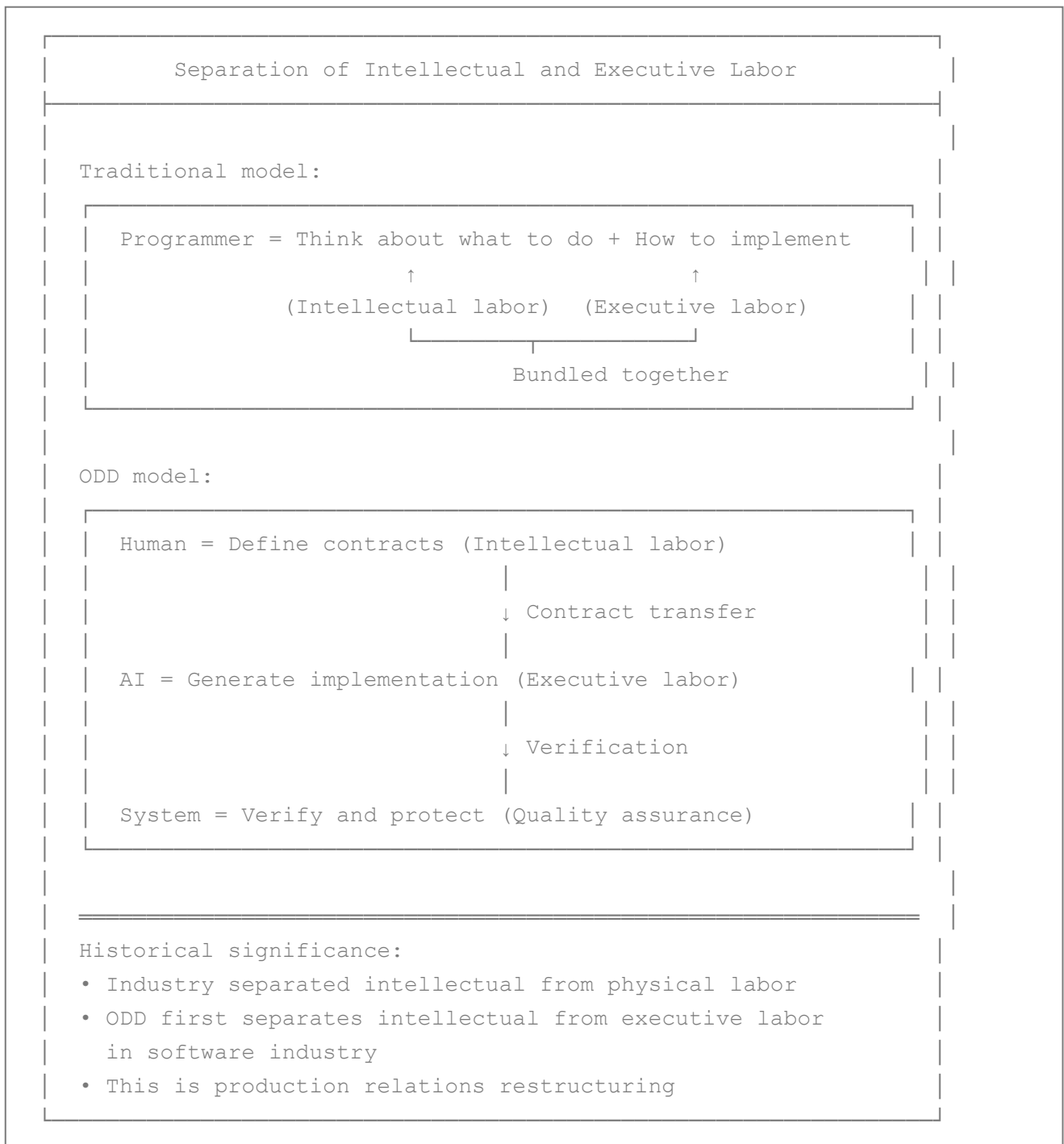
12. Why is ODD a Paradigm Innovation?

According to Thomas Kuhn's definition, a paradigm shift requires:

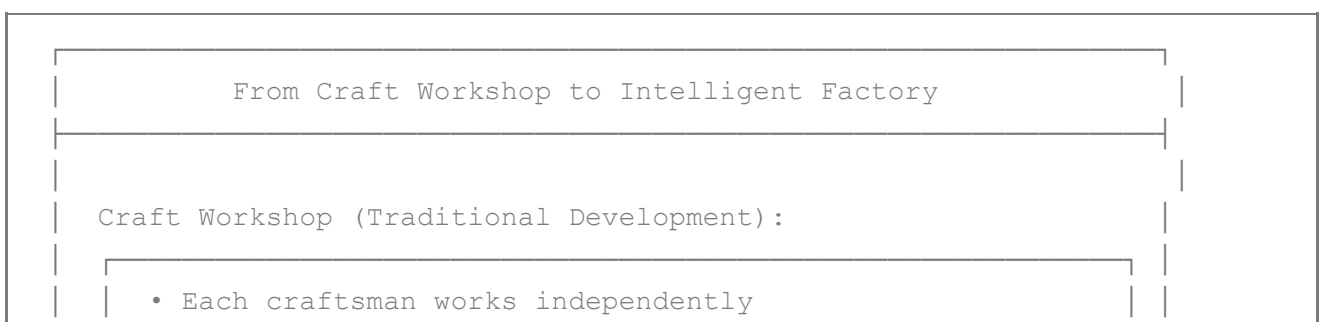
| ODD as Paradigm Shift (Kuhn) | |
|--|---|
| Kuhn's Criteria | ODD Fulfillment |
| 1. Solves problems old paradigm cannot solve | Old paradigm can't handle AI code volume; ODD solves this |
| 2. Redefines core concepts | Code → Artifact (intermediate) Human → Contract definer Review → Mutation testing |
| 3. Changes fundamental assumptions | From "human writes & reviews" to "human defines, AI executes, system verifies" |
| 4. Creates new vocabulary | Artifact, Contract, Sealing, Clarity Assessment, Mutation Score, Trust Transfer |
| 5. Opens new research directions | Contract language design, mutation testing optimization, multi-agent collaboration |

Conclusion: ODD meets all criteria for paradigm shift

13. Separation of Intellectual and Executive Labor



14. From Craft Workshop to Intelligent Factory



- Quality depends on individual skill
- Production doesn't scale
- Knowledge exists in artisan's head
- Losing an artisan means losing knowledge

Intelligent Factory (ODD Development):

- Standardized contracts as "blueprints"
- Quality guaranteed by system verification
- Production scales with compute
- Knowledge exists in contracts (explicit)
- Contracts are organizational assets

Transformation comparison:

| Dimension | Workshop | Factory |
|--------------|------------------|-----------------------|
| Productivity | Person-dependent | Process-dependent |
| Quality | Person-dependent | System-guaranteed |
| Scalability | Add people | Add compute |
| Knowledge | In heads | In contracts |
| Onboarding | Months | Days (read contracts) |

15. ODD Empowers All Groups

15.1 Independent Developer + ODD = Small Team

Independent Developer + ODD = Small Team

Traditional independent developer:

- Must handle: Requirements, design, coding, testing, ops
- Productivity limited by individual capacity
- Can only complete small projects
- Exhausted, hard to scale

Independent developer + ODD:

- Focus only on: Requirements definition, contract writing
- AI handles: Design, coding, testing
- System handles: Verification, sealing, protection
- Productivity equivalent to traditional 5-8 person team

Quantified effect:

| Metric | Traditional | With ODD | Improvement |
|-------------------|--------------|-----------|--------------|
| Features/month | 2-3 | 15-20 | 5-8x |
| Code review time | 40% | 5% | 8x↓ |
| Bug rate | Industry avg | Below avg | Significant↓ |
| Working hours/day | 10-12 | 6-8 | Healthier |

One person with ODD has the firepower of a startup team

15.2 IT Department + ODD = Professional Software Factory

IT Department + ODD = Professional Software Factory

Traditional IT department pain points:

- Business requests pile up, development backlogged
- Legacy system maintenance consumes most resources
- Talent recruitment/retention is difficult
- Internal systems low quality but "good enough to use"

IT department + ODD transformation:

| | |
|----------------|--|
| Business staff | → Contract definition (after training) |
| IT engineers | → Contract review + system maintenance |
| AI workers | → Code generation + test generation |
| ODD system | → Quality assurance + version control |

Transformation effects:

- Development capacity increases 3-5x with same headcount
- Business staff participate directly, shorter communication
- Internal system quality rises to professional level
- IT staff upgrade to "software architects," higher value

IT department transforms from "support unit" to "professional software factory"

15.3 Software Company + ODD = Productivity Revolution

Software Company + ODD = Productivity Revolution

Traditional software company model:

- Revenue \propto Developer headcount
- Gross margin limited by labor costs
- Scaling requires hiring, training, management
- Talent is bottleneck

Software company + ODD model:

- Revenue \propto Contract definition capacity
- Gross margin significantly improved (AI replaces labor)
- Scaling requires compute, not hiring
- Contract quality is bottleneck

Business model transformation:

Before: Sell developer time \rightarrow Labor-intensive

After: Sell artifact output \rightarrow Knowledge-intensive

Before: Linear scaling (add people = add capacity)

After: Exponential scaling (add compute = multiply cap)

Competitive advantage restructuring:

- Contract library becomes core IP
- Domain knowledge encapsulated in reusable contracts
- Delivery speed becomes order-of-magnitude advantage
- Quality consistency becomes trust foundation

Software companies transform from "body shop" to "intelligent manufacturing enterprise"

15.4 Non-Technical Users + ODD = Ideas Realized

Non-Technical Users + ODD = Ideas Realized

Traditional barriers for non-technical users:

- Have ideas but can't implement
- Hiring developers is expensive and hard to communicate
- Low-code/no-code platforms have limited functionality

- Technical debt piles up, maintenance becomes nightmare

Non-technical users + ODD:

1. Describe what you want in natural language
2. System guides clarity assessment, resolves ambiguity
3. Generate structured contract (human readable)
4. AI generates implementation, system verifies
5. Verified artifact delivered for use

Empowerment effects:

- Business experts directly produce business software
- Teachers directly create teaching tools
- Researchers directly build research aids
- Entrepreneurs directly implement MVP

Core principle:

"Know what you want" is the only required skill

"Know how to implement" is no longer necessary

Non-technical users gain the ability to turn ideas into software—the biggest leap in democratizing software creation

15.5 ODD Empowerment Summary

ODD Empowerment Summary

| Group | + ODD = | Key Transformation |
|-----------------------|------------------|---------------------|
| Independent Developer | Small Team | 5-8x productivity |
| IT Department | Software Factory | Professional output |
| Software Company | Productivity Rev | Exponential scaling |
| Non-Technical User | Ideas Realized | Zero coding barrier |

Common pattern:

- Before: Execution labor is bottleneck
- After: Definition capability is bottleneck
- Shift: From "how to do" to "what to do"

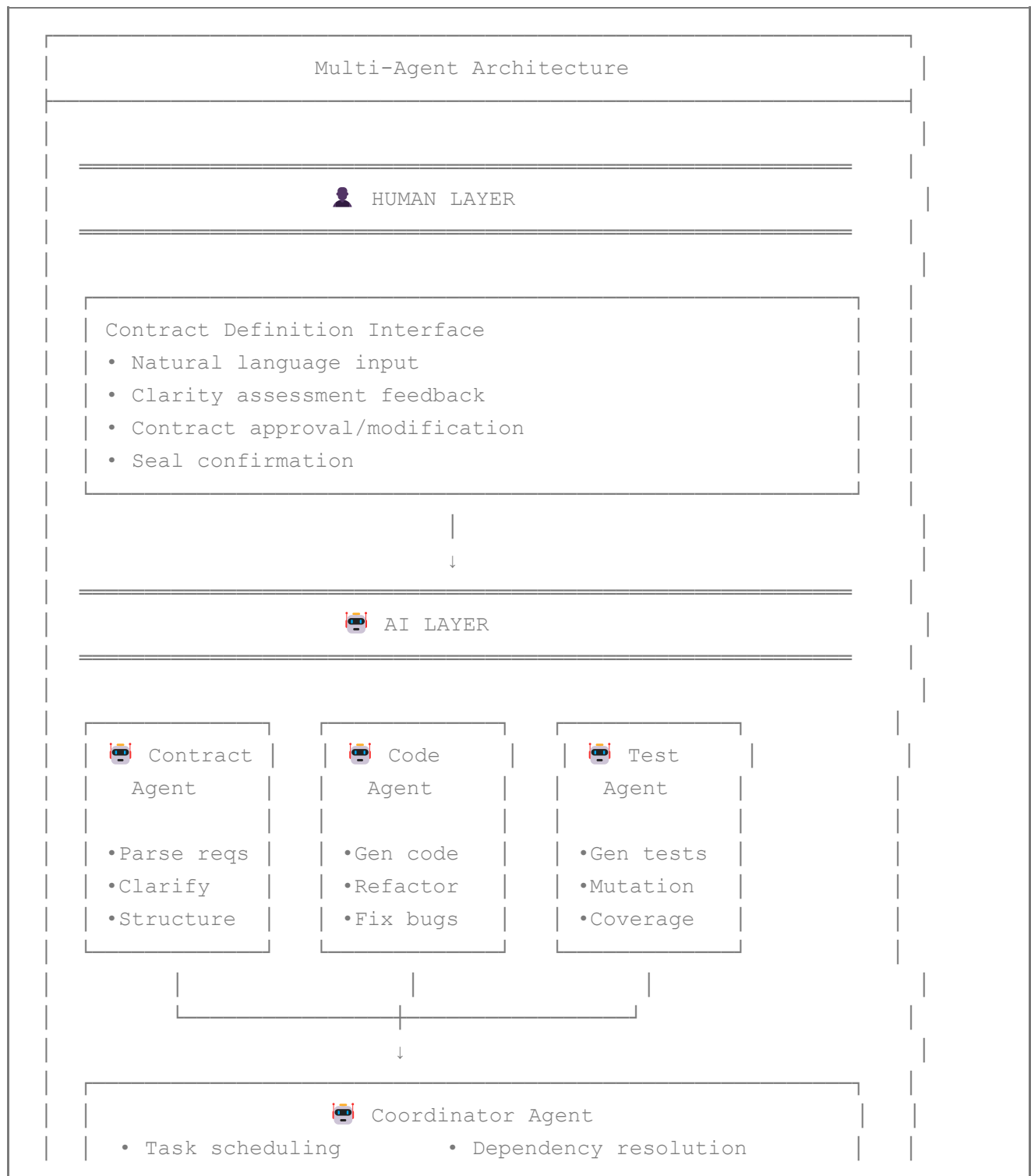
Ultimate vision:

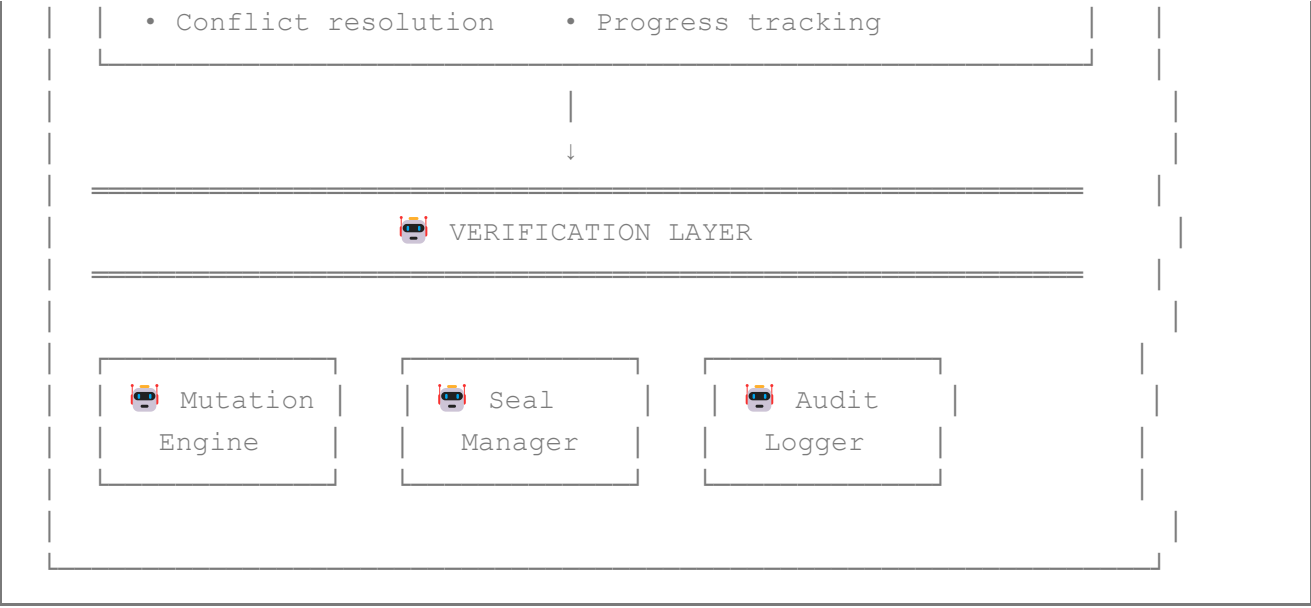
Anyone with a clear vision of what they want can create

software, regardless of technical background.

Part VIII: Engineering Implementation

16. Multi-Agent Architecture





17. Implementation Technology Stack

| Implementation Technology Stack | |
|---|-------------------------------------|
| Layer | Technology Options |
| LLM Engine | Claude / GPT-4 / Gemini / Local LLM |
| Agent Framework | LangChain / AutoGen / Custom |
| Mutation Testing | Stryker / Pitest / mutmut / Custom |
| Version/Sealing | Database |
| Contract Storage | Database |
| Monitoring | Prometheus / Grafana / Custom Code |
| Execution Env | Docker / K8s / Serverless |
| Note: ODD is tool-agnostic; any combination works | |

Part IX: Evaluation and Discussion

18. ODD Effectiveness Evaluation

| ODD Effectiveness Evaluation | | | |
|------------------------------|-------------|-----|-------------|
| Metric | Traditional | ODD | Improvement |

| | | | |
|------------------------|--------------|--------------|-------------|
| Development speed | 1x | 5-10x | 5-10x |
| Human review time | 40% of cycle | 5% | 8x↓ |
| Code quality (defects) | Industry avg | Below avg | Significant |
| Onboarding time | 2-3 months | 1-2 weeks | 4-6x↓ |
| Knowledge retention | In heads | In contracts | Permanent |
| Scalability | Add people | Add compute | Unlimited |

Note: Actual improvements depend on project type and team

19. Limitations and Countermeasures

Limitations and Countermeasures

Limitation 1: Mutation testing compute cost is high

Countermeasures:

- Incremental mutation (only test changed parts)
- Intelligent mutant sampling (statistical coverage)
- Parallel execution (distribute across compute nodes)
- Cache mutation results (skip unchanged code)

Limitation 2: Contract writing has learning curve

Countermeasures:

- Natural language to contract AI assistance
- Contract templates and examples library
- Clarity assessment guides users to improve
- Gradual adoption (start with simple contracts)

Limitation 3: Not all domains are easily contractifiable

Countermeasures:

- Start with well-defined domains (CRUD, APIs)
- Develop domain-specific contract languages
- Hybrid approach (ODD for testable parts, traditional for rest)
- Research into creative/exploratory domain contracts

Limitation 4: Organizational change resistance

Countermeasures:

- Start with pilot projects, demonstrate value
- Training and education programs
- Gradual transition, not big bang

- Highlight career evolution (coder → architect)

Limitation 5: LLM capability boundaries

Countermeasures:

- Contract decomposition (smaller, simpler tasks)
- Human-in-the-loop for complex decisions
- Multiple LLM ensemble for verification
- Continuous improvement as LLMs advance

Part X: Conclusion

20. Summary

This paper proposes **Output-Driven Development (ODD)**, a completely new software development paradigm designed for the AI era.

Core contributions:

1. **Established the central role of artifacts:** The goal of software development is not generating code, but generating artifacts that satisfy human needs. Code is merely an intermediate product.
2. **Redefined contracts:** Contracts are precise agreements defining artifacts—specifications that transform requirements into utility. Contracts are quantifiable, testable, verifiable—more suitable for the AI era than Markdown documents.
3. **Solved the AI review paradox:** Using mutation testing to replace human review achieves "humans don't write code and don't need to review code."
4. **Achieved production relations restructuring:** First-ever separation of intellectual from executive labor in the software industry, moving from "craft workshop" to "intelligent factory."
5. **Empowers all groups:** Independent developer + ODD = Small team; IT department + ODD = Professional software factory; Software company + ODD = Productivity revolution; Non-technical users + ODD = Ideas realized.

The essence of ODD: Let humans return to the role of "defining value," and delegate "implementing value" to AI.

The future software engineer: Not someone who writes code, but someone who **defines artifact specifications**—like a sausage factory's product manager who defines what sausage should be, not someone who personally grinds meat.

Appendices

Appendix A: Complete Contract Example

```
{
  "contract_id": "USER-AUTH-001",
  "version": "1.0.0",
  "name": "User Authentication Module",
  "description": "Handle user login, registration, password reset",

  "interfaces": [
    {
      "name": "login",
      "input": {
        "username": {"type": "string", "min_length": 3, "max_length":
20},
        "password": {"type": "string", "min_length": 8, "max_length":
128}
      },
      "output": {
        "success": {"token": "string", "expires_in": "number"},
        "errors": ["INVALID_CREDENTIALS", "ACCOUNT_LOCKED",
"ACCOUNT_DISABLED"]
      },
      "acceptance_criteria": [
        "Given valid credentials When login Then return JWT token, valid
3600s",
        "Given invalid password When login Then return
INVALID_CREDENTIALS",
        "Given 5 failures in 5min When 6th attempt Then return
ACCOUNT_LOCKED"
      ]
    },
    {
      "name": "register",
      "input": {
        "username": {"type": "string", "min_length": 3, "max_length":
20},
        "email": {"type": "email"},

```



```

        "password": {"type": "string", "min_length": 8, "pattern": "(?=.*[A-Z])(?=.*[0-9])"},
    },
    "output": {
        "success": {"user_id": "string", "verification_sent": "boolean"},
        "errors": ["USERNAME_EXISTS", "EMAIL_EXISTS", "WEAK_PASSWORD"]
    }
},
{
    "name": "resetPassword",
    "input": {
        "email": {"type": "email"}
    },
    "output": {
        "success": {"message": "string"},
        "errors": ["EMAIL_NOT_FOUND", "RATE_LIMITED"]
    }
}
],

"non_functional": {
    "performance": {
        "login_response_time": "<200ms p99",
        "max_concurrent_users": 10000
    },
    "security": {
        "password_hashing": "bcrypt with cost 12",
        "rate_limiting": "5 attempts per minute per IP"
    }
},

"metadata": {
    "author": "contract-architect@example.com",
    "created": "2026-01-10",
    "status": "approved"
}
}

```

Appendix B: Mutation Testing Configuration Example

Stryker Configuration (JavaScript/TypeScript)

```

{
    "$schema": "https://raw.githubusercontent.com/stryker-mutator/stryker/master/packages/core/schema/stryker-schema.json",
    "packageManager": "npm",
    "reporters": ["html", "progress", "dashboard"],

```

```
"testRunner": "jest",
"coverageAnalysis": "perTest",
"thresholds": {
  "high": 90,
  "low": 80,
  "break": 75
},
"mutate": [
  "src/**/*.ts",
  "!src/**/*.spec.ts",
  "!src/**/*.test.ts"
],
"mutator": {
  "excludedMutations": ["StringLiteral"]
}
}
```

Appendix C: Seal Record Structure

```
{
  "seal_id": "SEAL-2026-01-10-001",
  "artifact_id": "USER-AUTH-001",
  "version": "1.0.0",
  "sealed_at": "2026-01-10T14:32:00Z",
  "sealed_by": "system",

  "verification_results": {
    "mutation_score": 92.5,
    "total_mutants": 156,
    "killed_mutants": 144,
    "survived_mutants": 12,
    "test_count": 48,
    "test_pass_rate": 100
  },

  "hashes": {
    "contract_hash": "sha256:a1b2c3d4e5f6...",
    "code_hash": "sha256:f6e5d4c3b2a1...",
    "test_hash": "sha256:1a2b3c4d5e6f..."
  },

  "dependencies": [
    {"artifact_id": "COMMON-UTILS-001", "version": "2.1.0"},
    {"artifact_id": "DATABASE-001", "version": "1.5.0"}
  ],

  "rollback_info": {
    "previous_version": "0.9.0",
```

```
    "can_rollback": true,
    "rollback_command": "odd rollback USER-AUTH-001 --to=0.9.0"
  },

  "audit_trail": [
    {"action": "contract_created", "by": "architect@example.com", "at": "2026-01-05"},
    {"action": "contract_approved", "by": "tech-lead@example.com", "at": "2026-01-08"},
    {"action": "code_generated", "by": "ai-worker-3", "at": "2026-01-09T10:15:00Z"},
    {"action": "mutation_test_started", "by": "system", "at": "2026-01-10T13:00:00Z"},
    {"action": "mutation_test_passed", "by": "system", "at": "2026-01-10T14:30:00Z"},
    {"action": "sealed", "by": "system", "at": "2026-01-10T14:32:00Z"}
  ]
}
```

Appendix D: Glossary

| Term | Definition |
|--------------------|---|
| ODD | Output-Driven Development, a development paradigm centered on artifact correctness |
| Artifact | Verifiable output of software development that satisfies specific human needs and has use-value |
| Contract | Precise agreement defining artifacts—specifications transforming requirements into utility |
| Mutation Testing | Method to evaluate test quality by introducing code mutations |
| Mutation Score | Percentage of mutants killed by tests, measuring test effectiveness |
| Sealing | Locking verified code to prevent modification, with complete audit information |
| Clarity Assessment | Process of identifying ambiguity in contracts, shown as red/yellow/green |
| Trust Transfer | Shift of trust source from human review to system verification |

| Term | Definition |
|-------------------|--|
| Artifact Pipeline | Process of building artifacts layer by layer, each artifact input for the next |

References

1. Kuhn, T. (1962). *The Structure of Scientific Revolutions*. University of Chicago Press.
2. Meyer, B. (1992). "Design by Contract". *IEEE Computer*, 25(10), 40–51.
3. Beck, K. (2002). *Test-Driven Development: By Example*. Addison-Wesley.
4. Jia, Y., & Harman, M. (2011). "An Analysis and Survey of the Development of Mutation Testing". *IEEE Transactions on Software Engineering*, 37(5), 649–678.
5. DeMillo, R. A., Lipton, R. J., & Sayward, F. G. (1978). "Hints on Test Data Selection: Help for the Practicing Programmer". *IEEE Computer*, 11(4), 34–41.
6. Bubeck, S., et al. (2023). "Sparks of Artificial General Intelligence: Early experiments with GPT-4". *arXiv preprint arXiv:2303.12712*.
7. Chen, M., et al. (2021). "Evaluating Large Language Models Trained on Code". *arXiv preprint arXiv:2107.03374*.

End of Document