INTERNAL PUBLICATION

Deterministic Delivery

A point of view on AI-assisted software engineering.

Working Draft
Internal (B3)
Prototype edition

01

Purpose

This document explains how we think about AI-assisted software development.

It is the foundation for the Deterministic Delivery offering.

It is not intended to sell AI or introduce a new methodology.

It answers a simpler question:

> *After months of building production software with AI, what have we actually learned?*

Some conclusions will change as the technology evolves.

Others feel more fundamental.

This document tries to separate the two.

02

Scope

This document is about engineering practice.

It says very little about models, prompting techniques, AI products or IDEs.

Those change quickly.

Good engineering changes much more slowly.

If this document still feels relevant after today's models have been replaced, we probably focused on the right things.

03

Observation

For most of our careers, writing software was one of the expensive parts of software development.

Now it often isn't.

AI can produce working code remarkably quickly.

The work hasn't disappeared.

It has moved.

Someone still has to decide what should be built.

Someone still has to review it.

Someone still has to understand it six months later.

Someone still has to debug it when production behaves differently from everyone's expectations.

Writing code has become cheaper.

Understanding it has not.

> **Software generation has become dramatically cheaper.**
> **Human understanding has not.**

Everything else follows from that observation.

04

Interpretation

Our central thesis is simple.

> **AI changes the economics of software development, not the fundamentals.**

Software still needs architecture.

It still needs tests.

It still needs review.

It still needs someone prepared to say:

> *No. This doesn't belong in the system.*

The fundamentals haven't changed.

The balance of effort has.

As generating software becomes easier, understanding it becomes a larger part of the job.

05

Consequences

Accept the observation, and the rest follows.

5.1 The Bottleneck Has Shifted

The slow part of many projects is no longer writing code.

It's deciding whether the code is right.

We've all reviewed pull requests that looked perfectly reasonable until one hidden assumption changed everything.

AI makes that more common.

The question becomes less:

> *Can we build this?*

and more:

> *Do we understand it well enough to change it safely?*

5.2 Good Engineering Matters More

AI doesn't rescue poor engineering.

If anything, it exposes it faster.

Architecture matters.

Boundaries matter.

Tests matter.

Review matters.

They always did.

Now they're harder to ignore.

5.3 Engineering Judgement Matters More

One pattern keeps repeating.

AI struggles for many of the same reasons junior engineers struggle.

Not because it can't write code.

Because it doesn't always know which assumptions matter.

The typing becomes easier.

The thinking doesn't.

5.4 Confidence Matters More Than Throughput

Generating software quickly feels productive.

Confidence is slower to earn.

Production incidents rarely come from typing too slowly.

They usually come from assumptions that survived design, implementation and review before colliding with reality.

Tests settle arguments.

Review catches context.

Understanding earns confidence.

5.5 Engineering Systems Matter More

One observation has become difficult to ignore.

AI performs noticeably better inside well-engineered systems.

Not because the model changes.

Because the problem does.

Clear architecture gives it fewer opportunities to drift.

Small changes are easier to review.

Well-written tickets leave less room for invention.

Tests answer questions that discussion cannot.

None of this makes AI more capable.

It makes the engineering system more reliable.

06

Engineering Practices

These practices are not a methodology.

They're simply what has repeatedly worked.

CLARIFY INTENT BEFORE IMPLEMENTATION

PRINCIPLE	Understanding is becoming the scarce resource.
PRACTICE	Make the problem clear before implementation begins.
EXAMPLE	A well-understood problem usually beats a perfectly written prompt.

TICKET HYDRATION

PRINCIPLE	AI rarely invents missing requirements. It invents reasonable assumptions.
PRACTICE	Expand incomplete tickets before implementation.
EXAMPLE	The most valuable design discussion often happens before anyone writes code.

PLANNER → BUILDER → VERIFIER

PRINCIPLE	Planning, implementation and verification are different activities.
PRACTICE	Separate them when practical.
EXAMPLE	We consistently produce better outcomes when implementation follows an agreed approach and verification happens independently.

BOUNDED CHANGES

PRINCIPLE	Understanding does not scale with the size of the change.
PRACTICE	Prefer small pull requests with one clear purpose.
EXAMPLE	Every experienced engineer has seen a review where one apparently minor detail changed the entire discussion. Small changes make those moments easier to catch.

EXECUTABLE VERIFICATION

PRINCIPLE	Confidence comes from evidence.
PRACTICE	Use tests to answer questions that discussion cannot.
EXAMPLE	We've repeatedly watched automated tests settle debates that review alone never resolved.

HUMAN OWNERSHIP

PRINCIPLE	Responsibility cannot be delegated.
PRACTICE	Treat generated code as a proposal.
EXAMPLE	The engineer decides whether it belongs in the system.

07

Evidence

This document comes from engineering work.

Production systems.

Internal tools.

Consultancy projects.

Patterns that kept repeating.

Examples include:

- planner → builder → verifier
- ticket hydration
- executable verification
- production debugging
- bounded pull requests
- architecture reviews
- AI-assisted implementation
- cases where engineering judgement rejected plausible AI-generated solutions

As our experience grows, this document should change with it.

08

Things We Deliberately Do Not Claim

We do not claim that:

- AI replaces software engineers.
- AI removes uncertainty.
- AI always improves productivity.
- Prompting replaces engineering.
- Existing engineering practice should be abandoned.
- Junior engineers become unnecessary.
- Deterministic Delivery guarantees successful projects.

Engineering has always required judgement.

Nothing we've seen suggests otherwise.

09

Implications

If our reasoning is right, architecture matters more.

Review matters more.

Testing matters more.

Engineering leadership matters more.

Judgement matters more.

The work shifts away from producing software and towards building systems that teams understand well enough to change with confidence.

10

Open Questions

This document is unfinished by design.

Questions remain.

- Which claims need stronger evidence?
- Which practices survive another generation of models?
- How should organisations measure success?
- Which parts of review can safely become more automated?
- Is *Deterministic Delivery* still the right name?

Good engineering changes when experience demands it.

This document should do the same.

11

Conclusion

Everything in this document begins with one observation.

> **Software generation has become dramatically cheaper. Human understanding has not.**

If that's true, software engineering doesn't become less important.

It becomes more important.

The difficult part is no longer producing software.

It's producing software that people understand well enough to change with confidence.