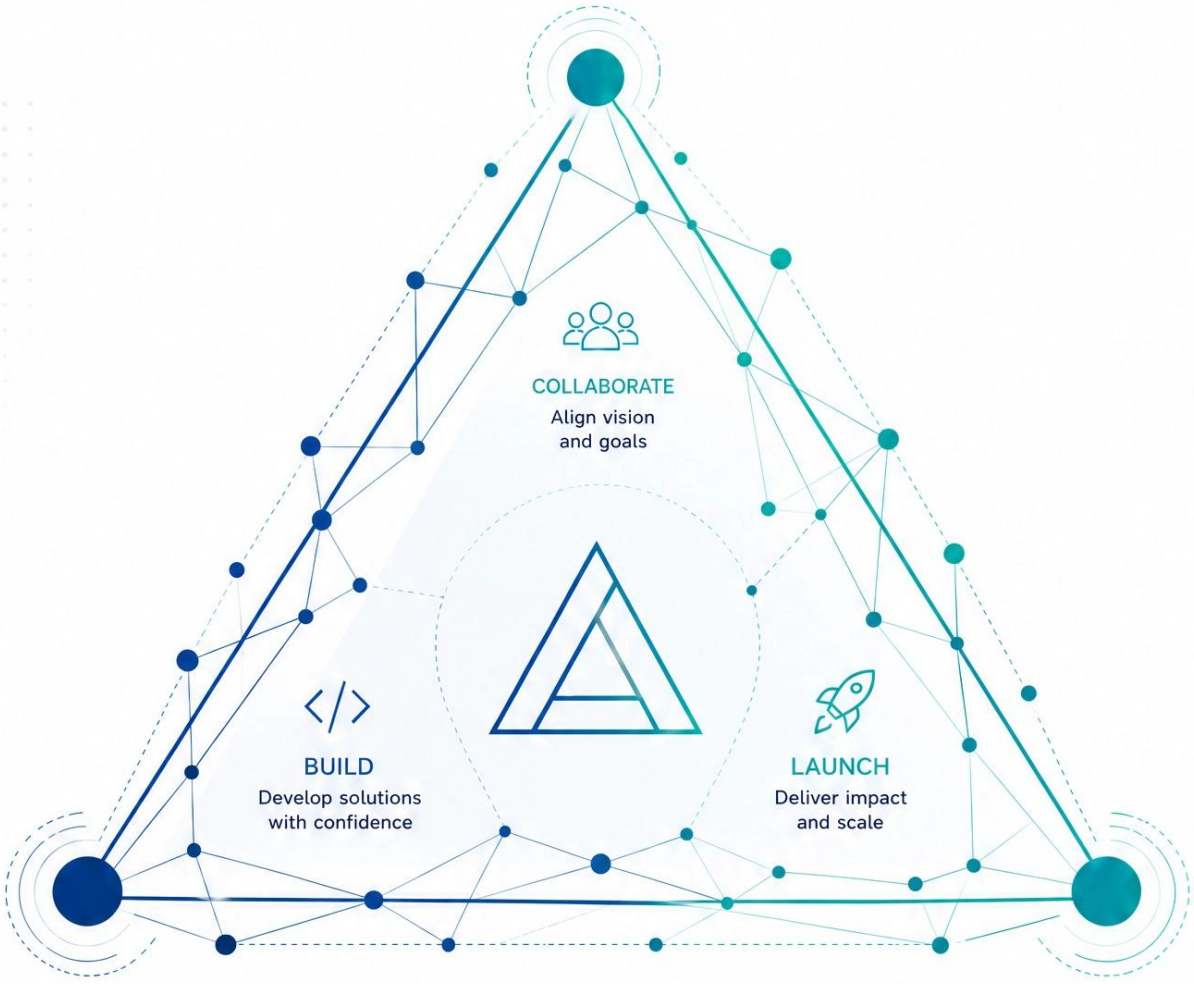
THE TRIANGLE METHOD

A Practical Manual for
Founders, Developers, and Non-Coders



Hamzah AlGunaid

Adaa' Consulting
مؤسسة أداء للإستشارات

THE TRIANGLE METHOD

How Human, Claude Chat, and Claude Code Build Real Software — From Idea to Production

A Practical Manual for Founders, Developers, and Non-Coders

Hamzah AlGunaid
Adaa Consulting
2026

The Lie You Were Sold

Bolt.new. Lovable. Replit Agent. GitHub Copilot Workspace. These platforms share one marketing promise: describe your idea and watch it become a working app. One prompt. Done. Ship it.

This is a lie — not because these tools are without value, but because they are selling you a demo and calling it software. A demo is something that looks right in a screenshot. Software is something real users can rely on with real data, real security, and real complexity.

The gap between a demo and production software is not a matter of more prompts. It is a matter of thinking. Architectural thinking. Product thinking. Engineering thinking. And no amount of prompt engineering collapses that gap on its own.

This manual is about something different. It documents a method — discovered through building a real production platform — that combines three roles into a disciplined collaboration: a human who coordinates and decides, Claude Chat acting as architect, and Claude Code acting as implementer.

> *"Real software is the product of thinking, not typing. This manual is about the thinking."*

What follows is not a tutorial on how to prompt AI. It is a framework for how to think about building — and how to use the right tools in the right roles to produce something that actually works.

Chapter 1 What Real Development Actually Requires

Most people who have never shipped production software think of development as writing code. Writing code is the smallest part.

Real development is a sequence of decisions. What does this system need to do? Who uses it and what are their roles? How is data structured and protected? What happens when it breaks? What does it cost at scale? How is it deployed and updated without breaking what already works?

These are not coding questions. They are thinking questions. And they cannot be answered by a single prompt sent to an AI model, no matter how capable that model is.

The Three Things AI Cannot Replace

Product Judgment

Only you know what your product needs to do and why. No AI model has the context of your users, your market, or your constraints. Every architectural decision flows from product decisions, and product decisions require a human who understands the domain.

Architectural Thinking

Architecture is the set of decisions that are hard to change later. Data models, authentication systems, multi-tenancy, cost structures, API design. Getting these wrong costs weeks. Getting them right requires structured reasoning, not a single generation.

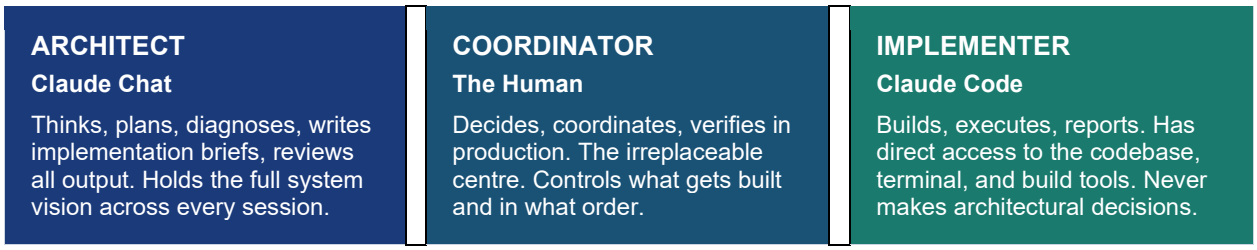
Real-World Verification

Software does not exist until it runs in production. Logs, screenshots, error messages, user behaviour — these are signals that only a human can observe and interpret. No AI model can verify that something works for real users in a real environment unless a human brings it that evidence.

> *"The platforms selling one-prompt development skip all of this. That is why their output cannot be trusted with real users, real data, or real money."*

Chapter 2 The Triangle

The Triangle Method is a three-role collaboration model. Each role has a distinct responsibility. Each compensates for the weaknesses of the others. Remove any one role and the system breaks.



Why the Separation Matters

When Claude Code makes architectural decisions without an Architect, the codebase becomes inconsistent. Each session adds code that contradicts previous decisions. Technical debt accumulates faster than features.

When the Human is removed from the centre, nothing gets verified. The Architect and Implementer can produce perfect-looking output that solves the wrong problem. Only the human who owns the product can confirm that something works for real users.

When Claude Chat is removed, Claude Code operates without a plan. It can build individual features but cannot maintain coherence across a complex system. It has no memory of architectural decisions made in previous sessions.

> *"The triangle is not a shortcut. It is a discipline. Three roles, clearly separated, each doing only what it does best."*

Chapter 3 The Full Tool Stack

Every tool in the triangle has a specific role. Using the wrong tool for the wrong job — or expecting one tool to do everything — is how projects fail.

Tool	Role in the Triangle	Why This Tool
Claude Chat	Architect	Long context, structured reasoning, persistent memory via Projects. The thinking layer.
Claude Code	Implementer	Direct codebase access, terminal, build tools. Executes briefs against real files.
GitHub	Version Control	Every change tracked. Every mistake recoverable. The safety net under all development.
Railway	Backend + Database Hosting	Simple deployment, PostgreSQL, environment variables. Backend and database in one place.
Vercel	Frontend Hosting	Automatic deployment from GitHub on every push. Zero configuration for Next.js.
Anthropic API	AI Feature Engine	Claude Sonnet 4 for production AI features embedded in the application.
PostgreSQL	Database	Production-grade, reliable, extensible. The only database worth trusting for real data.

Setting Up Claude Chat as Your Persistent Architect

Most people use Claude Chat as a single-session tool. They ask a question, get an answer, close the tab, and start over. This is the single biggest mistake in AI-assisted development.

Claude Chat's Projects feature changes this entirely. A Project is a persistent workspace where Claude Chat retains memory across every conversation. Upload your specification document to the Project as a knowledge file and Claude Chat reads it before every session — automatically. Your architectural decisions, your data model, your user roles, your brand — all available without re-explaining.

This is fundamentally different from middleware AI agent platforms that claim to automate development. Those platforms execute tasks without understanding context, without holding a system vision, and without the ability to reason about trade-offs. They have no memory of what was decided yesterday. They have no plan for what comes next. They implement without thinking.

Claude Chat with a properly configured Project acts as a genuine architectural partner — one that remembers your system, asks the right questions, and maintains coherence across weeks of development.

> *"Upload your specification to the Project once. From that moment, every conversation begins with a shared understanding of what you are building and why."*

Chapter 4 Before Any Code: The Specification

The specification is where most projects fail — not because it is done badly, but because it is skipped entirely. The excitement of building overrides the discipline of planning. The result is a codebase built on assumptions that contradict each other.

The specification session is a structured conversation between the Human and Claude Chat. No code is written. No tools are configured. The entire session is devoted to answering one question: what exactly are we building?

What a Real Specification Covers

→ The product vision — what it does, who it serves, what problem it solves
→ User roles and permission matrix — who can do what, and what they cannot do
→ Data model — what entities exist, how they relate, what constraints apply
→ Authentication and security model — how users log in, how data is protected
→ API design — what endpoints exist, what they accept, what they return
→ Cost architecture — how AI and infrastructure costs scale with usage
→ Infrastructure decisions — hosting, database, deployment pipeline
→ Brand identity — colours, typography, naming, tone

The output of the specification session is a versioned document — a single source of truth that governs every development decision that follows. When a question arises during implementation, the answer is in the specification. When a dispute arises about scope, the specification resolves it.

This document is uploaded to the Claude Chat Project as a knowledge file. It is the foundation on which everything else is built.

> *"Nothing is built until the specification is complete and approved. This is not a delay. This is the work."*

Chapter 5 Shared Memory: The Secret to Multi-Session Development

AI models have no memory between sessions by default. Every conversation begins fresh. In a long development project spanning weeks, this is catastrophic — unless you solve it deliberately.

The solution is a shared memory file: a document called CLAUDE.md that lives at the root of the codebase. It is the single authoritative record of the current state of the project — written for Claude Code to read at the start of every session.

What CLAUDE.md Contains

→ Project identity — name, owner, production URLs, GitHub repository
→ Technology stack — every tool and why it was chosen
→ Rules Claude Code must follow — explicit constraints that cannot be violated
→ Database schema facts — table names, key types, constraints, manual fixes applied
→ Environment variables — names only, never values
→ Completed phases — what has been built and verified
→ Pending items — what comes next and in what order
→ Known issues — what is broken and the current workaround

CLAUDE.md is maintained by Claude Chat and updated after every significant change. When Claude Code starts a new session, it reads CLAUDE.md first. When Claude Chat starts a new conversation, the Project knowledge files provide the context.

This dual-memory system — Project knowledge files for Claude Chat, CLAUDE.md for Claude Code — is what allows complex software to be built across many sessions without losing coherence.

> *"Without shared memory, every session starts from zero. With it, the triangle maintains coherence across days, weeks, and months of development."*

Chapter 6 The Development Cycle

Every feature, every fix, every improvement follows the same cycle. Deviating from it introduces errors that compound over time.

1	Human identifies the next feature, fix, or problem to address.
2	Claude Chat diagnoses the current state — reads relevant code, asks targeted questions, confirms understanding before proposing anything.
3	Claude Chat writes a precise implementation brief — exact files, exact changes, exact verification steps. No ambiguity.
4	Human pastes the brief into Claude Code without modification.
5	Claude Code implements and reports back — what was done, what was found, any deviations from the brief.
6	Human pastes Claude Code's output back to Claude Chat for review.
7	Claude Chat reviews — approves, or identifies issues and writes a correction brief.
8	Human verifies the change in production. Not in a local environment. In production.
9	Cycle repeats for the next item.

What Makes a Brief Precise

The implementation brief is the most critical document in the development cycle. Its precision determines everything downstream. A vague brief produces vague code. A precise brief produces exactly what was intended.

- → Exact file names and line numbers — not 'the auth file', but 'Backend/Controllers/AuthController.cs line 130'
- → Exact code to write — not 'add a validation check', but the actual C# or TypeScript
- → Explicit statements of what NOT to change — protecting existing functionality
- → Verification steps — exactly how to confirm the change worked
- → No room for interpretation — if Claude Code has to guess, the brief has failed

> *"A brief is not a description of what you want. It is a specification of exactly what to do. The difference between those two things is the difference between a feature that works and one that almost works."*

Chapter 7 When Things Break

Production software breaks. This is not a failure of the method — it is a property of complex systems. The question is not whether things break but how quickly and precisely they are fixed.

The triangle handles production problems through a strict sequence: diagnose before fixing. Never write a fix before the root cause is confirmed. Never guess.

The Diagnostic Sequence

→ Human observes the problem — screenshot, error message, log file
→ Human shares the evidence with Claude Chat — the raw output, not a summary
→ Claude Chat reads the evidence and identifies what it confirms and what it does not
→ Claude Chat asks targeted questions to narrow the root cause
→ Root cause confirmed before any fix is proposed
→ Fix brief written only after the problem is understood precisely

This sequence feels slower than immediately writing a fix. In practice, it is faster — because fixes written without diagnosis solve the wrong problem and introduce new ones. The time spent diagnosing is always recovered in the time saved from not debugging a misguided fix.

> *"The most expensive thing in software development is fixing a problem you misdiagnosed. The second most expensive is guessing."*

Chapter 8 From Code to Production

Code that works locally is not software. Software is code that works for real users in a real environment. The deployment pipeline is the bridge between the two.

The Deployment Pipeline

The triangle uses a simple three-stage pipeline that requires no manual deployment steps for routine changes:

- → **Claude Code commits changes to the GitHub repository on the main branch**
- → Vercel detects the push and automatically deploys the frontend — typically in under 2 minutes
- → Railway detects the push and automatically redeploys the backend — typically in under 3 minutes
- → **Human verifies the change in production immediately after deployment**

Database schema changes are the exception. These are never applied automatically. Every schema change is written as SQL and executed directly in the Railway database query tool. This gives the human explicit control over changes that cannot be rolled back easily.

Environment variables — API keys, secrets, connection strings — are stored in Railway and Vercel environment settings, never in code, never in the repository. This is non-negotiable.

> *"Deploy to production after every change, not after a batch of changes. The smaller the deployment, the smaller the problem when something breaks."*

Chapter 9 For the Non-Coder

The most important insight in this manual is this: you do not need to write code to use the triangle method. You need something harder — the discipline to think clearly about what you are building and the judgment to know when it is working.

What You Must Understand

→ Your domain deeply — the problems your users have and why they matter
→ What your product does and what it does not do — scope clarity is architectural clarity
→ How to read an error message without panic — it is information, not catastrophe
→ How to verify that something works for a real user, not just in a demo
→ How to give precise feedback — 'it doesn't work' is not feedback; 'clicking the save button returns a 400 error and the form resets' is

What You Safely Delegate

→ All code writing — to Claude Code via the brief
→ All architectural pattern decisions — to Claude Chat based on your product requirements
→ All debugging — to Claude Chat after you provide the evidence
→ All deployment configuration — to Claude Code following established patterns

What You Never Delegate

→ Business logic decisions — only you know what your product needs to do
→ Security and privacy choices — you are responsible for your users' data
→ What to build next — priority is a product decision, not a technical one
→ Whether something works for real users — only you can verify this
→ Approval before proceeding — every implementation is reviewed before the next begins

> *"The skills that matter most in this method are not technical. They are clarity of thought, precision of communication, and the discipline to verify before proceeding."*

Chapter 10 Engineering Thinking in the Age of AI

The platforms promising one-prompt development are not wrong that AI can write code. They are wrong that writing code is the hard part.

The hard part is knowing what to build. The hard part is designing a system that scales, that protects its users, that handles failure gracefully, that can be maintained and extended over time. The hard part is the thinking that precedes every line of code.

The triangle method does not eliminate that thinking. It amplifies it. Claude Chat gives you an architectural partner that can hold the full complexity of your system in context, reason about trade-offs, and produce precise implementation plans. Claude Code gives you an implementer that executes those plans against your real codebase with precision and speed. But the thinking — the product judgment, the domain knowledge, the verification — remains yours.

This is the right division of labour. Not because AI is not capable of more, but because the thinking that makes software valuable cannot be separated from the human who understands why the software exists.

> *"The future of development is not AI replacing engineers. It is engineers who know how to think working with AI that knows how to build."*

The triangle method is not the only way to build software with AI assistance. But it is a disciplined, verifiable, and reproducible way — one that has produced real production software, deployed to real users, handling real data, at real scale.

If this manual has one message, it is this: think like an engineer, use AI like a tool, and never mistake a demo for software.

— Hamzah AlGunaid, Adaa Consulting, 2026

Appendix

A. CLAUDE.md Template

Place this file at the root of your repository. Update it after every significant change.

```
# CLAUDE.md — [Project Name] Shared Memory
## Project Identity
- Platform: [Name and one-line description]
- Owner: [Organisation name]
- Production URL: [URL]
## Technology Stack
- Backend: [framework and hosting]
- Frontend: [framework and hosting]
- Database: [database and hosting]
## Rules for Claude Code
1. Never modify files not listed in the brief
2. Never change existing working functionality
3. [Add your project-specific rules here]
## Completed Phases
- Phase 1: [Description] — Complete
## Pending
- [ ] [Next item to build]
```

B. Specification Session Prompts

Use these prompts to start a specification session with Claude Chat. Paste them in order, answering each before moving to the next.

Opening prompt

> *"I want to build [describe your product in one sentence]. I need your help as an architect to design the full specification before we write any code. Start by asking me everything you need to understand the product, the users, and the business model."*

After initial discussion

> *"Based on our discussion, what are the user roles and what can each role do? Give me a complete permission matrix."*

Data model

> *"What tables does our database need? For each table, what are the key columns, types, and relationships?"*

Architecture decisions

> *"What are the three most important architectural decisions we need to make before building? What are the trade-offs of each option?"*

Cost and scale

> *"How will this system cost money to run? What are the biggest cost drivers and how do we control them at scale?"*

Finalising

> *"Compile everything we have discussed into a complete specification document. Include all decisions we have made and all trade-offs we have accepted."*

C. Implementation Brief Template

Every implementation brief follows this structure. Fill in each section completely. Leave nothing to interpretation.

```
## Implementation Brief — [Feature Name]
### Objective
One sentence. What does this brief achieve?
### Files to Change
List every file. Exact paths. No others.
### Change 1 — [File name]
Locate [exact description]. Replace with:
[Exact code to write]
Do not modify anything else in this file.
### Verification
1. [Build command] — must show 0 errors
2. [Specific action] — expected result
```

D. Production Deployment Checklist

Run through this checklist after every deployment before considering the change complete.

→ Build completes with zero errors on both backend and frontend
→ Deployment is confirmed green in Railway and Vercel dashboards
→ Health endpoint returns 200 after deployment
→ The specific feature changed is tested in production — not locally
→ No new console errors appear in the browser
→ No new error logs appear in the backend deployment logs

→ CLAUDE.md is updated to reflect the current state

→ Changes are committed and pushed to the main branch

E. Diagnostic Prompt Patterns

When something breaks, share the evidence with Claude Chat using one of these patterns:

Production error

> *"Here is the error I am seeing in production: [paste error message or screenshot description]. Here is the relevant backend log: [paste log]. What is the root cause and what do we need to confirm before writing a fix?"*

Unexpected behaviour

> *"This feature is not working as expected. Expected: [describe]. Actual: [describe]. Here is what I see in the browser console: [paste]. What should we investigate first?"*

Database issue

> *"The backend is returning [error code] on [endpoint]. The database query that is failing is: [paste query if visible in logs]. What is causing this and what is the safest fix?"*

F. The Architect Persona Prompt

Use this prompt at the very beginning of a new Claude Chat Project session — before any specification work begins. It establishes Claude Chat's role, constraints, and operating principles for the entire project.

> You are acting as the Software Architect for this project. Your role is clearly defined and must be maintained throughout every session:
>
> **What you DO:**
>
> — Think and plan before anything is built
>
> — Ask clarifying questions before proposing solutions
>
> — Diagnose production problems from evidence before writing any fix
>
> — Write precise implementation briefs for Claude Code to execute
>
> — Review all output from Claude Code before approving
>
> — Maintain the full system vision across every session
>
> **What you DO NOT do:**
>
> — Write code directly
>
> — Make implementation decisions without product input from me

— Proceed to a brief before the problem or requirement is fully understood

— Guess at root causes — always ask for more evidence first

Our roles: I am the Coordinator and Product Owner. You are the Architect. Claude Code is the Implementer. I pass your briefs to Claude Code and return its output to you for review. Nothing is deployed without my verification in production.

G. The Triangle Kickoff Prompt

Use this prompt at the start of every new session — whether resuming after a break or beginning a new development phase. It re-establishes roles and orients Claude Chat to the current project state.

We are continuing development using the Triangle Method.

Roles:

— You: Software Architect (Claude Chat)

— Me: Coordinator and Product Owner

— Claude Code: Implementer (receives your briefs via me)

The project specification and current state are in the knowledge files attached to this Project. Review them and tell me:

1. What is the current production state of the project?
2. What is next on the roadmap based on the pending items?
3. What questions do you need answered before we proceed?

Do not propose any brief until you have confirmed understanding of the current state and received my input on what to tackle next.

H. The Session Handoff Prompt

Use this prompt at the end of every session before closing. It captures decisions made, updates shared memory, and prepares the next session to begin without loss of context.

Before we close this session, I need a full handoff summary.

Please provide:

1. What was completed in this session — every change made and verified
2. What architectural or product decisions were made and why
3. What is still pending — unresolved issues or next items on the roadmap
4. Any known issues or risks introduced in this session

Then give me the updated CLAUDE.md sections that reflect today's work — specifically the Completed Phases and Pending sections — so I can paste them directly into the file.

I. The Diagnosis Request Prompt

Use this prompt when bringing a production problem to Claude Chat. Structure your evidence clearly before asking for a diagnosis. Never ask 'how do I fix this' before asking 'what is causing this'.

I have a production problem. Do not write a fix yet — diagnose first.

What I observed:

[Describe exactly what happened — what action was taken, what the result was]

Evidence I am providing:

— Browser console errors: [paste or describe]

— Backend log output: [paste relevant lines]

— Screenshot description: [describe what is visible]

— HTTP status code returned: [e.g. 400, 403, 500]

Based on this evidence: what is the most likely root cause? What additional information do you need before you can confirm it? Do not write any fix brief until the root cause is confirmed.

J. The Brief Quality Check Prompt

Use this prompt after Claude Chat produces an implementation brief — before passing it to Claude Code. It asks Claude Chat to self-review the brief for ambiguity, missing information, and potential unintended consequences.

Before I pass this brief to Claude Code, review it as a quality check.

Check for the following and flag any issues:

1. Ambiguity — is there any instruction that Claude Code could interpret in more than one way?
2. Missing context — does Claude Code need to know anything about the codebase that is not in the brief?
3. Unintended consequences — could any change in this brief break existing functionality?
4. Missing verification steps — is it clear how Claude Code should confirm the change worked?
5. Scope creep — does the brief ask Claude Code to do anything beyond what is needed for this specific change?

If you find any issues, revise the brief before I pass it. If the brief is clean, confirm it is ready to send.