The Agentic Engineering Journey

A practical guide to creating a software Dark Factory

The frameworks that turn AI chaos into autonomous engineering

v1.0 · avons.github.io

Introduction

Something remarkable is happening in the world of software development. Artificial intelligence tools have moved far beyond the "helpful spellchecker" stage — they are now writing real code, making architectural decisions, running tests, and in some cases, delivering entire features without a human touching a keyboard.

But here is the catch: most people are not getting nearly as much from this revolution as they could be. Not because the technology is not good enough, but because they have not changed how they think about their own role. They are still trying to drive a car that has learned to drive itself.

This document is inspired by **Dan Shapiro's** January 2026 blog post *"The Five Levels: from Spicy Autocomplete to the Dark Factory"*. Dan's framework gave us the map — this document adds the three navigational tools that make the journey actionable: **Domain-Driven Design (DDD)**, **BMAD**, and **Attractor**. Each one is, at its heart, a set of ideas about how to think more clearly and work more effectively as AI takes on more of the engineering workload.

> **Insight:** We do not need to be developers to benefit from this. If we work with software teams, commission software, or are curious about where AI is taking us — this is for you.

PART ONE: The Five Levels of AI-Assisted Development

The Five Levels

How the software industry is being transformed — one gear at a time

In January 2026, **Dan Shapiro** — CEO of Glowforge and a Wharton Research Fellow — published a framework drawing on the US government's five levels of self-driving car automation and applying it to software development. His central observation: *every level feels like we are done. But we are not done.* There is always a higher gear available — if we are willing to change how we work.

Level 0

Manual Driving

"*Your parents' Volvo*"

Our role: Full author of every line**AI's role:**None — maybe a search tool**Bottleneck:**Your typing speed

Level 1

Lane Assist

"*AI intern for tasks*"

Our role: Senior developer**AI's role:**Executes discrete tasks**Bottleneck:**Rate at which you assign tasks

Level 2

Highway Autopilot

"*Paired with a colleague*"

Our role: Senior developer + pair programmer**AI's role:**Fluent collaborator**Bottleneck:**Rate at which you give context

Level 3

Waymo with a Safety Driver

"*We are now a manager*"

Our role: Engineering manager**AI's role:**Senior developer**Bottleneck:**Quality of specs you hand over

Level 4

Robotaxi

"*We are a PM now*"

Our role: Spec author and planner**AI's role:**Full development team**Bottleneck:**Completeness of your specification

Level 5

The Dark Factory

"Specs in, software out"

Our role: Domain expert only**AI's role:**Entire software lifecycle**Bottleneck:**Organisational trust

Level 0 — Manual Driving: You Write Everything

At this level, every line of code that exists is one we typed yourself. We might occasionally copy a snippet from ChatGPT or let our editor auto-complete a variable name — but fundamentally, we are the sole author. In a world where AI can generate entire features in minutes, writing everything manually is like insisting on hand-drawing maps when GPS exists. It is not wrong, but it is increasingly costly.

Insight: The invisible cost here is opportunity cost. Every hour spent on boilerplate code, routine tests, or repetitive fixes is an hour not spent on the creative and strategic work only we can do.

Level 1 — Lane Assist: AI as Your Intern

Here, we start handing off the boring stuff. "Write me a unit test for this function." "Add comments to this code." "Convert this JSON to CSV." The AI is fast, it does not complain, and it frees us up for more interesting work. Tools like GitHub Copilot or a chat interface with Claude or ChatGPT live here.

Insight: The limitation is that we are still fundamentally a developer doing developer work — just faster. The AI is a tool we pick up and put down. We have not yet changed the nature of our role.

Level 2 — Highway Autopilot: True Partnership

This is where most ambitious developers are living right now, and it genuinely feels incredible. We are not just assigning tasks — we are having a conversation. We describe what we want, the AI drafts it, we refine it together, it notices problems we missed, we challenge its assumptions. It is the closest thing to having an expert collaborator available 24 hours a day. Tools like Cursor, Claude in your IDE, or similar AI-native editors make this feel natural.

Watch out: The danger is that Level 2 feels so good that people stop here permanently. Productivity is up, morale is high, and the work is satisfying. But we are still the rate-limiting factor. If we are not at the keyboard, nothing happens.

Level 3 — The Safety Driver: We Become Managers

At Level 3, we stop writing code and start reviewing it — at scale. Our AI agents are running multiple tasks simultaneously. We spend our days looking at diffs (the record of what changed), approving or rejecting suggestions, and redirecting when things go off track. For many people, this feels like a step backwards. We traded the pleasure of building for the cognitive exhaustion of managing.

Watch out: The hidden truth of Level 3 is that almost everyone gets stuck here — not because of the

tools, but because of habits and process. To go further, we need something to change about how we communicate requirements and expectations. That is where the frameworks we discuss later come in.

Level 4 — The Robotaxi: We Write the Brief, Not the Code

At Level 4, we write a spec — a clear description of what we want built. We debate it with the AI, refine the plan, set constraints and schedules. Then we step away. Hours later, we come back and check whether the tests passed. This is not hypothetical: a growing number of practitioners, including Dan Shapiro himself, are operating at this level today.

> **Insight:** The mental shift required to reach Level 4 is significant. We stop thinking like a developer and start thinking like a product owner. Our most valuable skill becomes the ability to describe, with precision and without ambiguity, what we want and why.

Level 5 — The Dark Factory: Autonomous Software Production

Level 5 is named after the Fanuc Dark Factory — a real manufacturing plant in Japan staffed entirely by robots, running in darkness because humans are neither needed nor present. In software terms, a small team (sometimes as few as two or three people) authors specifications, and an autonomous pipeline takes those specs and produces tested, deployable software — without ongoing human involvement.

> **Insight:** This is not science fiction. It exists today, at small scale, in a handful of pioneering teams. The technology is ready. What most organisations lack is the accumulated discipline — the shared language, the structured documentation habits, the trust — to make it work reliably.

PART TWO: Why You Cannot Skip Levels

The Temptation to Jump Ahead

When people first hear about Level 4 or Level 5, a natural reaction is: why not just start there? The answer is that each level is not just about different tools — it requires a different kind of organisational readiness that can only be built progressively.

Readiness Type	What It Means	Built at Level
Conceptual	Everyone speaks the same language about the domain	0 to 2
Linguistic	Requirements are precise enough for AI to act on	2 to 3
Organisational	Teams document decisions and create reusable artifacts	3 to 4
Technical	Pipelines exist that can be trusted to run autonomously	4 to 5

If we ask an autonomous AI to build a payment system, but our team has never agreed on what "payment" means, who owns the transaction, or what happens on failure — the AI will make those decisions for you. Confidently. Incorrectly. And at high speed.

Insight: The frameworks below are not shortcuts to the Dark Factory. They are the path. Each one addresses a different layer of readiness that must be built, step by step.

PART THREE: Domain-Driven Design (DDD)

Domain-Driven Design

The art of speaking the same language as your business

What Problem Does It Solve?

Imagine we work at a bank. The loans department talks about "customers", "accounts", and "risk profiles". The engineers who built the loans software also have "customers", "accounts", and "risk profiles" — but they mean subtly different things, modelled in slightly different ways. Over years these mismatches compound. Features take longer to build. Bugs appear at boundaries. Nobody can fully explain what the system does.

DDD is the antidote. Its core proposal: the language our business uses to describe itself should be the language our code uses too — and all ambiguity should be resolved in conversation, not discovered in production.

Who Created It?

Eric Evans, a software consultant, formalised DDD in his 2003 book *Domain-Driven Design: Tackling Complexity in the Heart of Software*. It drew on decades of field experience with large, complex systems and remains essential reading for software architects more than twenty years later.

Industry Acceptance

DDD's concepts — particularly Bounded Contexts — map directly to the microservices architecture used by Amazon, Netflix, Uber, and Spotify to build highly scalable systems. It is taught in senior engineering programmes and referenced in AWS and Azure architecture guidance.

The Core Ideas

1. Ubiquitous Language

A shared vocabulary for everyone — business people and engineers alike. If the business calls something an Authorisation Request, the code contains a class called `AuthorisationRequest`. Same word, same meaning, everywhere. Ambiguity is eliminated at the source.

2. Bounded Context

A clear boundary within which a model is valid. "Customer" in Billing means a paying account. "Customer" in Support means a person with a problem. Different models, clear boundaries — and a documented map of how they relate.

3. Entities

Things with a persistent identity that change state over time. An Order placed today is the same Order when fulfilled tomorrow. Its identity (order number) is constant; its state (pending, fulfilled, cancelled) changes.

4. Value Objects

Things defined entirely by what they are, not who they are. A 50-rupee note is a 50-rupee note — we do not care which physical note. Money, addresses, and coordinates are classic value objects. Immutable — you replace them, not change them.

5. Aggregates

A cluster of related things treated as one unit. An Order and its OrderLines are an aggregate — we never change an OrderLine without going through the Order. The Order is the Aggregate Root: the single front door to the whole cluster.

6. Domain Events

Significant things that have happened, recorded as facts: `OrderPlaced`, `PaymentFailed`, `CustomerRegistered`. First-class citizens — they drive workflows, trigger notifications, and serve as an audit trail.

7. Repositories

The storage layer designed from the domain outward. We ask a Repository for an Order by ID and get a fully-formed object back. We never write raw database queries in our business logic. The database is an implementation detail.

DDD's Role in the Journey

DDD is the foundation beneath all levels. Before an AI agent can act reliably on our domain, it must understand our domain. Teams that arrive at Level 3 with a well-developed vocabulary find their agents produce far more accurate results. Teams that skip this find their agents confidently building the wrong thing — because nobody told them what the right thing actually means.

> **Insight:** DDD was invented two decades before LLMs mattered. But all the clarity it demands for human collaboration is exactly the clarity AI agents need to operate autonomously. It is retroactively the perfect preparation for everything that comes next.

PART FOUR: BMAD — Breakthrough Method for Agile AI-Driven Development

BMAD

Structured collaboration between humans and AI

What Problem Does It Solve?

Most people's experience of AI-assisted development goes like this: we have an idea, start a conversation, build something, it mostly works, we add to it, context gets messy, we start a new conversation, and slowly coherence dissolves. Two weeks later we cannot explain to the AI what we built

or why certain decisions were made.

BMAD is a remedy for this. It insists on doing things in order: understand the problem first, define requirements second, design the architecture third, break it into stories fourth — and only then write a single line of code. Sounds obvious. Almost nobody does it.

Who Created It?

Brian Grew (known as bmadcode) created and open-sourced BMAD as a practical response to unstructured AI development chaos. It has been iterated rapidly since 2024 based on real-world usage.
Install with: `npx bmad-method install` · stable v4 (recommended)
`npx bmad-method@alpha install` · v6 alpha (cutting edge)
After install, run `/bmad-help` in your IDE — it tells you exactly what to do next.

Industry Acceptance

BMAD has gained significant traction with **39,000+ GitHub stars**, 120+ contributors, and active community discussions. It has become the go-to structure for AI-native application development where reproducibility and coherence matter, with extensive documentation and tutorials available.

The BMAD Personas

BMAD defines a cast of AI personas — each an expert in a specific discipline. Rather than asking one AI to do everything, we work with focused specialists in sequence. When you run `npx bmad-method install`, the installer generates slash commands inside your IDE — for example `.claude/commands/` in Claude Code, or equivalent files for Cursor, Codex, and Windsurf. We never type trigger codes by hand; we run a slash command and the LLM loads the right persona automatically. **Two types of commands are generated:**
• **Load an agent** — puts that persona in charge of the conversation: `/bmad-agent-bmm-pm`, `/bmad-agent-bmm-architect`
• **Run a workflow** — the agent follows a structured multi-step process: `/bmad-bmm-create-prd`, `/bmad-bmm-create-architecture`

Not sure what to do next? Just run `/bmad-help` at any point — it inspects your project state and recommends the next step.

🔍 **Arjun (Analyst)**	Digs into the problem space. What are we solving? Who is affected? What does success look like? Output: a Project Brief. `/bmad-agent-bmm-analyst` → then `/bmad-bmm-create-brief`
📋 **Priya (Product Manager)**	Takes the Brief and produces a PRD — Product Requirements Document. Defines features, constraints, and acceptance criteria. This becomes the contract for everything that follows. `/bmad-agent-bmm-pm` → then `/bmad-bmm-create-prd`
🏗️ **Vikram (Architect)**	Takes the PRD and designs the system. What components exist? How do they communicate? Where are the boundaries? Output: an Architecture Document. `/bmad-agent-bmm-architect` → then `/bmad-bmm-create-architecture`
🎯 **Raj (Scrum Master)**	Takes the Architecture and breaks it into granular Story files — individual units of work a Developer agent can execute without needing extra context. `/bmad-agent-bmm-sm` → then `/bmad-bmm-create-epics-and-stories`

💻 **Arnav (Developer)**	Executes stories. Writes the actual code guided entirely by the Story file. Asks no ambiguous questions because the story already answers them. `/bmad-agent-bmm-dev` → then `/bmad-bmm-implement-story`
🎨 **Saumya (UX Designer)**	Defines how users interact with the product — screens, flows, interactions. Specifications the Developer can implement without guessing. `/bmad-agent-bmm-ux-designer` → then `/bmad-bmm-create-ux-design`
🧪 **Kiran (QA Engineer)**	Defines test contracts and acceptance criteria. What must be true for this feature to be considered complete? Automates test generation for existing features. `/bmad-agent-bmm-qa`
⚡ **Dev (Quick Flow Solo Dev)**	For rapid prototyping — small tasks that don't need full process overhead. `/bmad-bmm-quick-dev`
📝 **Meera (Technical Writer)**	*Triggers: DP, WD, US, MG, VD, EC* Creates documentation, writes specifications, updates standards, generates diagrams, and explains concepts clearly.

Note: These personas come built-in with BMAD. But the idea is not locked to BMAD. BMAD includes a Builder agent specifically for creating your own domain-specific personas — we describe the role we need ("Security Auditor", "Data Engineer", "Compliance Reviewer") and it scaffolds the full agent definition for us. The custom agents live alongside the built-in ones and the orchestrator routes to them automatically. The underlying principle works in any agentic tool — Claude Code, Cursor, Codex, or even a plain chat interface: a specialist who knows their domain deeply and follows a structured process will outperform a generalist every time.

Why Documentation-First Works

BMAD's insistence on a full chain of artifacts before writing code is not bureaucracy. It is recognition that the quality of what an AI builds is limited by the quality of what we tell it to build. Vague input produces vague output. Precise, agreed-upon input produces reliable, coherent output.

Party Mode

Party Mode is BMAD's most creative feature — but it is important to understand exactly what it is and what it is not.

What it is: A single conversation, in your IDE, where one LLM simultaneously holds multiple agent personas — PM, Architect, Developer, QA — and switches between them per response. We play the human. The LLM plays everyone else. There is no external server, no multi-user session, no real-time team chat. The "team" exists entirely inside the LLM's context window.

What it is not: A collaboration tool for your actual human team. It does not replace Slack, GitHub discussions, or your planning meetings. What it produces — a resolved decision, a flagged contradiction, a revised PRD section — is what we then take back to our real team.

Invoke with `/bmad-agent-bmm-orchestrator` → then type `party-mode`. The orchestrator loads the relevant personas for our question and manages who responds when.

> **The real workflow around Party Mode:**
> **Before:** Your human team does event storming together (DDD) — you map the domain, agree on bounded contexts, identify domain events.
> **During:** One team member runs Party Mode to stress-test the build plan against that domain map — catching contradictions between PRD, Architecture, and Stories before a line of code is written.

After: The output (a resolved decision, a corrected story, a flagged risk) is shared with the real team via your normal channels.

Good for:
- ***Stress-testing a plan before implementation starts***
- ***Big architectural decisions with real tradeoffs***
- ***Catching mismatches between PRD, Architecture, and Stories***
- ***Brainstorming when we need multiple expert lenses at once***

Example: Pre-Build Alignment (where DDD meets BMAD)

Our event storming session identified `PaymentSettled` as a domain event. Now we are about to build the payment feature. We run Party Mode to stress-test the story before writing code.

You: "We are building the bill-split feature. Here is the PRD and the story. Does anything not add up?"

Vikram (Architect): "The story assumes synchronous settlement but our Architecture Document specifies an event-driven payment service. These are incompatible — `PaymentSettled` will never arrive synchronously."

Arnav (Developer): "Confirmed. I cannot implement this story as written. I would need to poll for the event or the story needs a different acceptance criterion."

Priya (PM): "The PRD also does not define what happens when a split does not add up to 100%. Kiran cannot write tests for an undefined failure state."

Kiran (QA): "Correct — I have no acceptance criterion for partial failure or rounding errors. This story is not ready."

We fix the PRD and the story. No code was written. No production bug was created.

Example: Brainstorming

You: "How do we make onboarding feel natural instead of like filling a form?"

Saumya (UX Designer): "Progressive disclosure — reveal features as the user needs them, not all at once in a tutorial."

Priya (PM): "What if the first action they take is actually solving a real problem? They learn by doing something valuable, not by reading instructions."

Arnav (Developer): "That changes the data model for onboarding state — I would need to track completion by task completion, not by screen visits."

Insight: BMAD does for AI collaboration what agile did for human team collaboration: it provides a shared language, a repeatable process, and a way to preserve decisions so context is never lost, even as people and tools change.

BMAD's Role in the Journey

BMAD is most valuable climbing from Level 1 to Level 4. At Levels 1 and 2 it structures our prompts and thinking. At Level 3 it provides the management framework for parallel agent work. At Level 4 its artifacts — PRD, Architecture, Stories — become the precise specifications that autonomous pipelines

can execute without hallucinating requirements that were never written down.

PART FIVE: Attractor — Autonomous Pipeline Orchestration

Attractor

The engine that runs the Dark Factory — turning specifications into autonomous pipelines with checkpoints, human approval gates, and visual flow diagrams

What Problem Does It Solve?

By Level 4 we have good specifications and a capable AI. But we still have a coordination problem: how do we orchestrate a multi-step workflow — plan, implement, test, review, fix, re-test — reliably, automatically, and in a way that can be observed, paused, resumed, and debugged?

Without a framework, we end up writing complex orchestration scripts by hand — full of nested loops, error handling, retry logic, and state management that is hard to read, harder to modify, and nearly impossible to visualise. Attractor solves this by turning the workflow itself into a diagram.

Who Created It?

StrongDM — a Zero Trust infrastructure access company — built Attractor internally and published it as an open nlspec (natural language specification) on GitHub. Community implementations already exist in TypeScript and Python.

Industry Acceptance

Attractor occupies a genuinely new space — Level 4/5 AI-native development tooling — where few standards exist. With **500+ GitHub stars** and active community implementations in TypeScript and Python, it is the leading open specification for autonomous coding pipelines.

The Three Layers

Layer 1 `unified-llm-spec`

Talk to any AI model through a single consistent interface — Claude, GPT-4, or Gemini. Switching models is a one-line change. This layer handles the tool loop: when the AI needs to run a command, read a file, or check a result before continuing.

Layer 2 `coding-agent-loop-spec`

Turn an AI model into a stateful code-editing agent. Manages full conversation history, tools available (read file, edit file, run shell), and the ability to inject instructions mid-task. Configurable reasoning effort: fast/cheap for simple tasks, deep/careful for critical ones.

Layer 3 `attractor-spec`

Orchestrate multi-step pipelines as a visual graph. Nodes are tasks, arrows are transitions. The engine walks the graph automatically. Every step creates a checkpoint — if anything fails it resumes from where it stopped. Human approval gates can be placed at any node.

What a Pipeline Looks Like

Here is a real Attractor pipeline for shipping a feature. Even without coding experience, we can read this and understand exactly what it does:

```
digraph ShipFeature {
  node [shape=box, style="rounded"]

  start [label="Start"]
  plan [label="Plan"]
  implement [label="Implement"]
  test [label="Test"]
  review [label="Review", shape=diamond]
  exit [label="Done"]

  start -> plan -> implement -> test
  test -> review [label="pass"]
  test -> implement [label="fail", color=red]
  review -> exit [label="Approve"]
  review -> implement [label="Changes", color=orange]
}
```

Plan, implement, test. Tests pass → review. Tests fail → fix and retry. Review: approve to finish, or request changes to loop back. Visible. Auditable. Resumable.

Attractor's Role in the Journey

Attractor is the execution layer for Levels 4 and 5. At Level 4 it takes the specifications produced by BMAD and executes them as automated pipelines — removing humans from the implementation loop while preserving review and approval gates. At Level 5, entire Bounded Contexts become self-contained pipelines running end-to-end without human involvement. The Dark Factory is Attractor at scale.

PART SIX: Putting It All Together

The Unified Journey

How DDD, BMAD, and Attractor work together

Each of the three frameworks addresses a different layer of the same problem: how do we hand over increasingly complex, valuable work to an AI system — and trust it to get it right?

Framework	What It Is	What It Builds	Best Used At
DDD	Software design philosophy (2003)	Shared language and clear domain model	All levels — foundational

BMAD	Agile AI workflow framework (2024)	Structured artifacts and process discipline	Levels 1 through 4
Attractor	Pipeline orchestration spec (2025)	Autonomous execution and observable pipelines	Levels 4 and 5

The Critical Handoff: BMAD to Attractor

The most important transition happens at Level 4 — when human-guided collaboration gives way to autonomous execution. This is the handoff from BMAD to Attractor.

BMAD's job is to produce artifacts of sufficient quality that an AI can act on them without asking clarifying questions. The PRD, Architecture Document, and Story files are not just good practice for humans — they are the fuel for Attractor's pipelines. A vague story produces vague code. A precise, well-bounded story produces code that can be accepted without a second look.

In DDD terms, the Bounded Contexts defined during domain modelling, articulated in the PRD during BMAD, become the scope boundaries of individual Attractor pipelines. Each pipeline knows its domain because that domain was carefully defined long before the pipeline ran.

> ***"A .dot pipeline file is a Bounded Execution Context — it carries semantic intent in its structure and language, has a clear boundary, and hides implementation details behind a consistent interface."***

A Practical Progression

Stage	What We Are Doing	Primary Tool	Output
Early exploration (L0–L1)	Learning the domain, building shared vocabulary	DDD concepts	Ubiquitous Language glossary
Structured collaboration (L2–L3)	Working with AI personas to plan and specify	BMAD	PRD, Architecture, Stories
Managed automation (L3–L4)	Delegating implementation, reviewing outputs	BMAD + Attractor	Working software with human review
Autonomous execution (L4–L5)	Writing specs, running pipelines, approving results	Attractor	Tested, deployed software

How Existing Roles Evolve

As teams move up the levels, traditional roles tend to shift their focus rather than disappear entirely — at least in the early levels. At Levels 1 and 2, most roles look familiar: developers write code with AI assistance, QA engineers write tests, architects design systems, project managers track progress. The tools change; the job descriptions do not change much.

By Level 3 and 4 the shift becomes more noticeable. Developers spend less time writing and more time reviewing. QA starts expressing acceptance criteria in natural language rather than test scripts. Architects spend more time on domain boundaries and less on implementation details. Project managers find that much of what they coordinated is now coordinated automatically.

What happens beyond that is genuinely uncertain — and we explore that in the Final Thoughts below.

Final Thoughts

Nobody knows exactly how this plays out. Anyone who claims otherwise is guessing — and so are we. But it is hard to look at where things are heading and not wonder.

The task-worker versions of many familiar roles — the developer writing boilerplate, the QA engineer writing test scripts, the DevOps engineer maintaining pipelines, the project manager tracking tickets — these may look very different in ten years. Or they may evolve in ways none of us can predict. What seems likely is that the boundary between "business person" and "technical person" will keep blurring. The most effective people we can imagine at Level 4 or 5 are those who understand both the domain and the systems deeply enough to bridge them.

There is also something quietly interesting happening with language. English, Hindi, and every other human language are becoming a more precise interface with machines than they have ever been. We wonder whether the ability to express intent clearly — to describe what we want, why we want it, and what good looks like — will matter as much as any technical skill. Perhaps more. Perhaps not. It is worth thinking about.

What we can say with more confidence: the climb through the levels is real, the frameworks are useful, and the direction of travel is clear — even if the destination is still coming into focus.

DDD gives us the language. BMAD gives us the process. Attractor gives us the engine. The journey is already underway.

Quick Reference Summary

Framework	Origin	Author	Year	Levels
Five Levels	Practitioner essay	Dan Shapiro	2026	Framework
DDD	Software design book	Eric Evans	2003	0 to 5
BMAD	Open-source project	Brian Grew (bmadcode)	2024	1 to 4
Attractor	Open nlspec by StrongDM	StrongDM team	2025	4 to 5

REFERENCES

Sources & Further Reading

Topic	Source	URL
The Five Levels	Dan Shapiro's blog post (Jan 2026)	danshapiro.com/blog
Domain-Driven Design	Eric Evans' book (2003)	domainlanguage.com/ddd
DDD & Microservices	Software System Design guide	softwaresystemdesign.com

BMAD Method	GitHub Repository (39k+ stars)	github.com/bmad-code-org/BMAD-METHOD
BMAD Documentation	Official docs site	docs.bmad-method.org
Attractor	StrongDM GitHub (500+ stars)	github.com/strongdm/attractor
Attractor Spec	StrongDM Factory	factory.strongdm.ai

💬 Share Your Feedback

This is a living document hosted at **github.com/avonS/agentic-engineering-journey**. We are actively refining these ideas and would love to hear your perspective — whether you are a developer, a product owner, a founder, or simply someone curious about where AI is taking software.

Open an issue to suggest improvements, start a discussion to share your experience at any of the five levels, or watch the repository to be notified of updates. If you find this useful, starring the repo helps others discover it.

☆ View on GitHub & Star | Open an Issue | 💬 Start a Discussion

Prefer email? Reach out at · Follow github.com/avonS for future updates and related work in agentic engineering.

Created with Level 2 collaboration · Cover: Gemini · Draft: Claude Web · Editing: OpenCode + big-pickle