Member-only story

From Prototype Chaos to Production-Ready: The Framework AI Desperately Needs

17 min read · Oct 7, 2025

Reza Rezvani Following

Listen Share More

15-minute technical read with working code examples

Hours into debugging, I watched user sessions bleed across accounts in production.

The AI-generated authentication code looked flawless. Clean syntax. Thoughtful comments. Proper error handling. It passed every test I threw at it locally.

But under concurrent load, everything fell apart. The AI had confidently generated a singleton pattern where none should exist — a mistake no senior developer would make, hidden inside code that looked professional enough to ship.

That's when I stopped accepting AI output at face value and started demanding structure.

I didn't believe Spec-Driven Development would fix anything. It sounded like another layer of documentation theater — more process, same problems. But after three months of using it on real production code, I've found it's the only approach that makes AI-generated code survive contact with users.

Here's what actually works when you strip away the hype.

The Problem: AI Generates Code That Works in Isolation

You know this pattern. You describe what you want, the model spits out code, it compiles. Looks good. Ships to staging. Maybe even makes it to production.

Then reality hits.

The code works perfectly in isolation but fractures when integrated. Why? Because AI doesn't understand your architecture, your team's patterns, or the decisions you made three sprints ago when you refactored the data layer.

Claude Code 2.0 Prototype chaos to production-ready framework

GitClear analyzed 211 million lines of AI-assisted code across repos from Google, Microsoft, Meta, and enterprise companies.

What they found confirms what I've been seeing in production:

- Code duplication exploded — blocks with 5+ duplicated lines increased 8x during 2024.
- Refactoring collapsed from 25% of changed lines down to under 10%.
- Copy/paste code now exceeds *"moved"* code for the first time in history.

One engineer on Reddit captured it perfectly:

> "Three months in, any change means editing dozens of files. The design hardened around early mistakes, and every fix brings three new bugs."

But here's what made me question everything: The METR study from mid-2025 ran a randomized trial with experienced open-source developers. Half used AI tools — primarily Cursor Pro with Claude Sonnet — half didn't.

Developers using AI were 19% slower. Not faster. Slower.

The kicker? They believed they were faster. Before the study, they predicted AI would make them 24% faster. After finishing slower, they still reported feeling 20% faster. The AI had hijacked their perception of productivity.

That's when I realized we weren't optimizing the right thing.

What I Tried Before SDD (And Why It All Failed)

My first attempt was obvious: better prompts. I spent weeks crafting detailed prompts with architecture context, coding standards, even examples of our best components.

It helped. Marginally.

The AI still generated 500-line God classes when I asked for authentication. Still introduced *async/await* patterns where we used observables. Still created new service classes when it should have extended existing ones.

Second attempt: Custom rules files. I built a `.cursorrules` file with our entire tech stack, ESLint config, snippets of our best patterns.

Better. But not enough.

The AI would follow the rules for the first few generations, then drift. Context window limitations meant it forgot the earlier constraints. Each new file was a fresh start, and the architectural coherence fell apart.

Third attempt: Aggressive code review. Every AI-generated line got scrutinized. Test coverage requirements. Mandatory pair reviews. The works.

This actually worked. But it destroyed the productivity gains. We were spending more time reviewing and fixing AI code than we would have just writing it ourselves.

That's when I found Spec-Driven Development. Not because I wanted to try another methodology. Because nothing else was working, and I was running out of options.

The Core Concept: Specifications as Source of Truth

SDD inverts the entire relationship between specs and code.

Traditionally, we write code first, then document it. The code is the source of truth. Documentation is scaffolding we build and usually let rot.

SDD flips this: The specification becomes executable. It generates the code. The spec isn't a suggestion — it's a contract. Update the spec, regenerate the code. The implementation becomes disposable. The intent remains stable.

GitHub released Spec Kit as an open-source CLI to formalize this process. *I was skeptical. Another tool promising to fix AI chaos?* But I was also desperate enough to give it a real try.

Here's what actually changed over three months of production use.

Phase 1: Specify — Writing Intent That AI Can Execute

Forcing Clarity Before Code

The first phase forces you to articulate what you're building without mentioning how. No tech stack. No frameworks. Just user flows, edge cases, and success criteria.

I started with a Pokemon search feature for an Angular app. Here's what I wrote:

```
/specify Build a Pokemon search feature. Users type keywords, see
real-time filtered results showing name, image, and type. Click any
result to view full details including stats, abilities, and evolution
chain. Must handle:
```

```
- No matches: display helpful message with search suggestions
- API failures: show retry option with cached results if available
- Slow connections: debounce search, show loading states
- Concurrent searches: cancel in-flight requests
```

```
Accessible (WCAG 2.1 AA), mobile-first, supports keyboard
navigation.
```

The AI generated a complete `spec.md`:

```
## User Stories
**US-001: Basic Search**
As a Pokemon fan, I want to search by name so I can quickly find specific Pokem
Acceptance Criteria:
- Search input updates results in real-time
- Results show name, image, type with visual indicators
- Minimum 2 characters required to trigger search
- Search is case-insensitive
**US-002: Type Filtering**
As a user, I want to filter by type so I can explore Pokemon categories.
Acceptance Criteria:
- Type selector shows all 18 Pokemon types
- Multiple type selection supported
- Results update immediately on selection change
- Active filters visually indicated
## Entities
**Pokemon**
- id: number (unique identifier)
- name: string (display name)
- types: Type[] (primary/secondary types)
- sprites: { front_default: string }
- stats: Stat[]
**SearchState**
- query: string
- selectedTypes: Type[]
- results: Pokemon[]
- loading: boolean
- error: Error | null
## Edge Cases
- No results: Display "No Pokemon match '{query}'" with suggestion to try diffe
- API timeout: Show cached results with warning banner
- Malformed API response: Log error, show generic failure message
```

What hit me immediately: **The AI forced me to think through edge cases I would have skipped.** The spec became a checklist of everything that could go wrong before I wrote a single line of code.

The Validation Gate That Changes Everything

But here's what made it stick — validation checkboxes the AI generates:

```
### Content Quality
- [x] No implementation details (languages, frameworks, APIs)
- [x] Focused on user value and technical requirements
- [x] All edge cases identified with handling strategies
- [x] Success criteria are testable
```

Every unchecked box was a gap in my thinking. I had to go back and fix it before moving forward. No shortcuts. No "I'll handle that later."

> *The spec became the contract, and I couldn't proceed until it was complete.*

Phase 2: Plan — Locking Down Architecture

Injecting Technical Constraints

Phase 2 is where you inject all the technical constraints the spec deliberately avoided. Stack choices. Architectural patterns. Deployment targets. Security requirements.

For my Pokemon app:

```
/plan Angular 20, standalone components, signals for reactivity.
TypeScript strict mode, no implicit any. Angular Material following
Material Design v3 guidelines.
```

```
Data from PokeAPI V2 (https://pokeapi.co/api/v2/). Implement
PokemonService using HttpClient with retry logic (exponential backoff,
max 3 attempts). Cache responses in-memory using Map<string, Pokemon>.
State management: Angular signals, no NgRx. Search debounce: 300ms.
Favorites in localStorage (JSON serialization).
Testing: Jasmine/Karma, minimum 80% coverage. Component tests must
mock HttpClient. E2E with Cypress for critical paths.
Performance: Lazy load detail components. Virtual scrolling for
```

```
result lists >100 items. Image lazy loading with placeholder.
Build: Angular, ESLint + Prettier, strict CSP headers.
```

The AI generated a detailed `plan.md`:

Architecture Overview

```
src/
├── app/
│   ├── core/
│   │   ├── services/
│   │   │   ├── pokemon.service.ts
│   │   │   └── cache.service.ts
│   │   ├── interceptors/
│   │   │   └── retry.interceptor.ts
│   │   └── models/
│   │       ├── pokemon.model.ts
│   │       └── search-state.model.ts
│   ├── features/
│   │   ├── search/
│   │   │   ├── search.component.ts
│   │   │   └── search.component.spec.ts
│   │   └── detail/
│   │       ├── detail.component.ts
│   │       └── detail.component.spec.ts
│   └── shared/
│       └── components/
│           └── pokemon-card/
│
```

Service Layer Design

PokemonService

- Singleton service using providedIn: 'root'
- HttpClient with retry interceptor
- In-memory caching with TTL (5 minutes)
- Debounced search using RxJS operators

```
interface PokemonService {
  searchPokemon(query: string): Observable<Pokemon[]>;
  getPokemonDetail(id: number): Observable<PokemonDetail>;
  getByType(type: string): Observable<Pokemon[]>;
  clearCache(): void;
}
```

State Management Strategy

Using Angular signals for reactive state:

```
// search.component.ts
export class SearchComponent {
  searchQuery = signal('');
  selectedTypes = signal<string[]>([]);
  results = signal<Pokemon[]>([]);
  loading = signal(false);
  error = signal<Error | null>(null);

  // Computed signals
  filteredResults = computed(() => {
    const query = this.searchQuery();
    const types = this.selectedTypes();
    return this.filterPokemon(this.results(), query, types);
  });
}
```

Error Handling Patterns

1. Network failures: Retry with exponential backoff (100ms, 200ms, 400ms)
2. API errors: Display user-friendly message, log to console
3. Parsing errors: Fallback to cached data if available
4. Timeout: 10s max, then show cached results with warning

This plan became the contract. The AI couldn't deviate.

When I later tried to add a feature that required a different state management approach, the AI rejected it based on the plan. I had to consciously update the plan to allow the change.

That's when I understood the power. **The plan wasn't documentation — it was enforcement.**

Phase 3: Tasks — Granular, Testable Units

Breaking Down to Prevent God Objects

Phase 3 breaks everything into independent, verifiable chunks. This is where SDD saved me from the biggest AI pitfall: massive, tightly coupled implementations that look fine until you try to test them.

Running `/tasks` generated:

~~~
## Phase 1: Foundation (Parallel Eligible)

### T001: Core Models [P]
File: src/app/core/models/pokemon.model.ts
```typescript
export interface Pokemon {
  id: number;
  name: string;
  types: PokemonType[];
  sprites: PokemonSprites;
  height: number;
  weight: number;
}
export interface PokemonType {
  slot: number;
  type: {
    name: string;
    url: string;
  };
}
export interface PokemonSprites {
  front_default: string;
  front_shiny?: string;
}
~~~

Tests: Verify type definitions compile without errors

T002: Cache Service [P]

File: src/app/core/services/cache.service.ts Implementation:

- Generic Cache<K, V> class
- TTL-based expiration
- LRU eviction (max 100 entries)

- localStorage persistence for favorites only

Tests:

- Set/get operations
- TTL expiration after timeout
- LRU eviction when max size exceeded
- localStorage sync for favorites

T003: HTTP Retry Interceptor [P]

File: src/app/core/interceptors/retry.interceptor.ts

```
export class RetryInterceptor implements HttpInterceptor {
  intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any
    return next.handle(req).pipe(
      retry({
        count: 3,
        delay: (error, retryCount) => {
          if (error.status >= 500) {
            return timer(Math.pow(2, retryCount) * 100);
          }
          throw error;
        }
      })
    );
  }
}
```

Tests:

- Retries 3x on 500 errors
- Exponential backoff timing
- No retry on 4xx errors
- Throws after max retries

```
Each task was completely isolated. I could build T001, T002, T003 in parallel.
```

~~~
### Catching Bugs Before They Compound

When I implemented T003, I found a problem. The exponential backoff
wasn't resetting between different requests. The test caught it
immediately:

```typescript
it('should reset retry count for new requests', () => {
  // First request fails 3 times
  service.search('fail').subscribe({
    error: (e) => expect(e).toBeTruthy()
  });

  // Second request should start at attempt 1, not 4
  service.search('succeed').subscribe({
    next: (result) => expect(result).toBeDefined()
  });
});
~~~

The AI had generated the interceptor as a singleton with shared state.

I updated the spec to clarify:

> **"Retry state must be per-request, not shared."** *Regenerated the plan. The AI fixed it.*

This is what vibe coding never gave me — **checkpoint validation before the code pile got too big to debug.**

Phase 4: Implementation — Where the Spec Saves You

When Perfect-Looking Code Breaks

By the time I hit **Phase 4**, most hard decisions were locked in. The AI just executed. But here's where personal experience became critical.

The AI generated this search component:

```
export class SearchComponent implements OnInit {
  searchQuery = signal('');
  results = signal<Pokemon[]>([]);
```

```
  constructor(private pokemonService: PokemonService) {}

  ngOnInit() {
    this.pokemonService.searchPokemon(this.searchQuery())
      .subscribe(results => this.results.set(results));
  }
}
```

It compiled. It worked. But it was wrong.

The search wasn't debounced, and it only fired on init. I had specified *"real-time search with 300ms debounce"* in the spec.

AngularJS — Prototyping Framework

Refining Intent, Not Code

I went back to the spec, added explicit acceptance criteria:

```
**US-001: Real-time Search**
Acceptance Criteria:
- Search triggers on input change, not on submit
- 300ms debounce from last keystroke
- Cancel in-flight requests when new search starts
- Display loading state during search
```

Regenerated the tasks. This time:

```
export class SearchComponent {
  private searchSubject = new Subject<string>();
  searchQuery = signal('');
  results = signal<Pokemon[]>([]);
  loading = signal(false);

  constructor(private pokemonService: PokemonService) {
    this.searchSubject.pipe(
      debounceTime(300),
      distinctUntilChanged(),
      tap(() => this.loading.set(true)),
      switchMap(query =>
        this.pokemonService.searchPokemon(query).pipe(
          finalize(() => this.loading.set(false))
        )
      )
    ).subscribe(results => this.results.set(results));
  }

  onSearchInput(query: string) {
    this.searchQuery.set(query);
    this.searchSubject.next(query);
  }
}
```

Perfect. The spec forced clarity, and the AI followed it precisely.

Catching Edge Cases That Matter

But then I hit an edge case I hadn't specified: what happens when the user searches, then immediately navigates away? The subscription leaked.

Added to spec:

```
## Edge Cases
- Component destruction during in-flight request:
  Must unsubscribe to prevent memory leaks
```

The AI regenerated with proper cleanup:

```
export class SearchComponent implements OnDestroy {
  private destroy$ = new Subject<void>();
```

```
    constructor(private pokemonService: PokemonService) {
      this.searchSubject.pipe(
        takeUntil(this.destroy$),
        debounceTime(300),
        switchMap(query => this.pokemonService.searchPokemon(query))
      ).subscribe(results => this.results.set(results));
    }

    ngOnDestroy() {
      this.destroy$.next();
      this.destroy$.complete();
    }
  }
```

This became the pattern that changed everything:

Spec → Plan → Tasks → Implement → Find Gap → Update Spec → Regenerate

The cycle was fast because I wasn't refactoring code — I was refining intent.

What Actually Changed in My Workflow

The Paradigm Shift

The biggest change wasn't the tool. It was how I thought about AI.

Before SDD, I treated AI like a faster keyboard. Type a description, get code, fix what broke. The AI was a means to an end — the code.

With SDD, the AI became a compiler for intent. I write specs, the AI translates them into implementation.

When the implementation is wrong, I don't fix the code — I fix the spec.

This sounds like a small distinction. It's not.

When you fix code, you're patching symptoms. When you fix specs, you're correcting root cause. The next regeneration doesn't make the same mistake.

A Concrete Example: Adding Favorites

Without SDD, I would have:

1. Asked AI to add a favorites button

2. Got some localStorage code
3. Fixed bugs where it didn't sync across tabs
4. Fixed the memory leak where old favorites weren't garbage collected
5. Fixed the race condition where rapid clicks duplicated entries

With SDD, I updated the spec:

```
**US-005: Favorites Management**
As a user, I want to save favorite Pokemon so I can quickly access them later.

Acceptance Criteria:
- Click star icon to toggle favorite status
- Favorites persist across browser sessions
- Favorites sync across tabs in real-time
- Duplicate favorites prevented
- Maximum 50 favorites (oldest removed on overflow)
- Clear all favorites option with confirmation
Technical Requirements:
- localStorage with BroadcastChannel for tab sync
- Debounce save operations (1s) to prevent excessive writes
- Handle localStorage quota errors gracefully
```

Updated the plan with implementation details:

```
export class FavoritesService {
  private readonly STORAGE_KEY = 'pokemon_favorites';
  private readonly MAX_FAVORITES = 50;
  private channel = new BroadcastChannel('favorites_sync');
  private saveSubject = new Subject<Pokemon[]>();

  favorites = signal<Pokemon[]>([]);

  constructor() {
    this.loadFromStorage();
    this.setupSync();
    this.setupDebouncedSave();
  }

  toggle(pokemon: Pokemon): void {
    const current = this.favorites();
    const exists = current.some(p => p.id === pokemon.id);

    if (exists) {
      this.favorites.set(current.filter(p => p.id !== pokemon.id));
    } else {
      const updated = [pokemon, ...current].slice(0, this.MAX_FAVORITES);
      this.favorites.set(updated);
    }

    this.saveSubject.next(this.favorites());
  }

  private setupDebouncedSave(): void {
    this.saveSubject.pipe(
      debounceTime(1000),
      tap(favorites => this.saveToStorage(favorites)),
      tap(favorites => this.channel.postMessage(favorites))
    ).subscribe();
  }

  private setupSync(): void {
    this.channel.onmessage = (event) => {
      this.favorites.set(event.data);
    };
  }
}
```

The AI generated it correctly on the first try. No bugs. No memory leaks. No race conditions.

Why? Because the spec forced me to think through the edge cases before any code existed. The debouncing, the sync, the quota handling — all specified upfront.

The Technical Debt Problem SDD Actually Solves

Stopping Code Duplication at the Source

GitClear's research showed code duplication increased 8x with AI. I saw this firsthand before SDD.

The AI would generate a data fetching function. Then another feature needed similar fetching. Instead of reusing, the AI generated a new function with slight variations.

By the third feature, I had four versions of essentially the same HTTP call:

- `searchPokemon()` in SearchComponent (inline HttpClient)

- `loadPokemon()` in DetailComponent (inline with different error handling)
- `fetchPokemonData()` in a helper file (duplicated retry logic)
- `getPokemon()` in another service (same structure, different cache)

Each had variations that made them "unique" to the AI, so it never suggested reuse.

Architecture as Enforcement

With SDD, the plan explicitly defined service boundaries:

~~~
## Service Architecture
**Data Fetching: Single Responsibility**
All HTTP calls go through PokemonService. No component-level HTTP.
**PokemonService Interface**
```typescript
interface IPokemonService {
  // Search operations
  search(query: string): Observable<Pokemon[]>;
  searchByType(type: PokemonType): Observable<Pokemon[]>;

  // Detail operations
  getById(id: number): Observable<PokemonDetail>;
  getEvolutionChain(id: number): Observable<EvolutionChain>;

  // Batch operations
  getMultiple(ids: number[]): Observable<Pokemon[]>;
}
~~~

All API calls must use this interface. No direct HttpClient injection in components.

```
When I asked for a new feature needing Pokemon data, the AI didn't create a new

The duplication stopped because the architecture was specified, not implied.
## Where SDD Still Falls Short
I've been using Spec Kit for three months on production code. It's not perfect,
### The Upfront Cost Is Real
**Issue #1: Initial spec overhead.**
Writing a complete spec for a feature takes time. I measured it-about 40% longe
But here's what I also measured: The second version took 70% less time. Because
The break-even point is around iteration three. If you're building throwaway pr
### Learning to Think Like a Compiler
**Issue #2: Spec precision is hard.**
You have to think like a compiler. Ambiguous specs generate ambiguous code.
```

```
"Handle errors gracefully" means nothing.
"On 4xx errors, display error.message to user. On 5xx errors, retry 3x with exp
Learning to write executable specs is a skill. I'm still learning it. The AI he
### Legacy Code Pain
**Issue #3: Legacy integration is brutal.**
Spec Kit is designed for greenfield projects. Trying to retrofit it onto an exi
I tried it on a 50,000-line Angular app. The reverse-engineering took two weeks
For legacy modernization, this can be powerful. Extract the "what" from ancient
## Your Implementation Checklist
If I could go back three months and talk to my skeptical past self, here's the
### Getting Started (Week 1)
**□ Pick ONE complex feature to start**
- Not the entire app, not a simple CRUD
- Something with edge cases, async operations, state management
- Your first spec will be messy-that's expected
**□ Focus on edge cases in the spec**
- Happy path is easy, AI handles it fine
- Bugs come from boundaries: What happens on network failure?
- What happens when the user does two things simultaneously?
- What happens with malformed data, empty lists, null values?
**□ Use the plan for architectural constraints**
- Spec = product intent (what users need)
- Plan = technical implementation (how you'll build it)
- Put patterns in the plan: service boundaries, state management, error handlin
### Validation Gates (Throughout)
**□ Don't skip validation checklists**
- Every unchecked box is a gap that will bite you in production
- If the spec says a checkbox should be checked but you can't check it, fix the
- The AI won't let you proceed-this is a feature, not a bug
**□ Pair SDD with code review tools**
- Spec catches logical errors
- You still need linters for style
- Security scanners for vulnerabilities
- Test coverage tools for completeness
### Workflow Integration (Week 2+)
**□ Establish the regeneration cycle**
- Find bug → Update spec → Regenerate → Validate
- Don't patch code directly anymore
- Let the spec be your source of truth
**□ Build your constitution template**
- Document your architectural decisions once
- Let it enforce consistency across all AI generations
- Update it as you learn what works
## The Setup I Actually Use
Here's my current production workflow as of this writing:
### Tools
- **Cursor** with Claude Sonnet (latest version)
- **GitHub Spec Kit** (actively maintained, check releases for latest)
- **ESLint + Prettier** with strict configs
- **Vitest** for unit tests
- **Playwright** for E2E
- **SonarQube** for code quality gates
### Directory Structure
### Constitution Template
This gets injected into every AI conversation:
# Project Constitution
## Architecture Principles
- Domain-Driven Design with clear bounded contexts
- Dependency injection via framework (Angular/NestJS)
- Repository pattern for data access
- CQRS for complex state mutations
## Code Standards
- TypeScript strict mode, no implicit any
- Functional programming preferred (pure functions, immutability)
- Error handling: Never catch without rethrowing or handling
- Async: Observables for streams, Promises for single values
## Testing Requirements
- TDD: Tests before implementation
- Unit tests: 80% coverage minimum
- Integration tests: All API endpoints
- E2E tests: Critical user paths
## Performance Budgets
- Lighthouse: 90+ on mobile
- First Contentful Paint: <1.5s
- Time to Interactive: <3s
- Bundle size: <200kb gzipped
## Security Requirements
- All API calls authenticated via JWT
- XSS protection: Sanitize all user input
- CSRF protection: Token-based
- Content Security Policy: Strict
## Deployment Constraints
- Zero-downtime deploys
- Database migrations before code deploy
- Feature flags for risky changes
- Rollback plan for every deploy
```

The AI can't violate these rules without explicitly getting permission. It's architecture as code.

The Actual Bottom Line

SDD isn't about AI generating better code. **It's about forcing yourself to think before the AI executes.**

The spec is a forcing function. You can't hand-wave edge cases. You can't skip error handling. You can't ignore performance implications. The checkboxes won't let you.

The plan is a forcing function. You can't defer architectural decisions. You can't leave security as an afterthought. You can't ship without test coverage. The validation won't let you.

The tasks are a forcing function. You can't build God objects. You can't skip TDD. You can't create tightly coupled code. The isolation requirements won't let you.

Is it more work upfront? *Yes.*

Is it worth it? Only if you care about code that survives contact with production.

What This Really Means

I'm not saying SDD is the universal answer. I'm saying it's the only approach I've found that makes AI-generated code production-ready without requiring more debugging time than the AI saved in the first place.

Your mileage will vary. Your team dynamics are different. Your tech stack has different constraints. Your tolerance for process varies.

But if you're tired of debugging AI-generated code that looked perfect until it wasn't — code that passed tests but broke under load, code that compiled but leaked memory, code that worked in isolation but fractured when integrated — **give SDD a real try.**

Not a weekend experiment. A full feature, from spec to deploy. Measure the time investment. Count the bugs. Compare the refactoring cycles.

Then decide if the structure is worth the overhead.

. . .

I'm collecting war stories about AI-generated bugs that made it to production. *What's the worst one you've shipped?* Drop it in the comments — let's figure out together if SDD would have caught it, or if we need a different approach entirely.

If this changed how you think about AI-assisted development, hit that clap button and follow for more deep dives into what actually works when the hype dies down.

. . .

Resources to Get Started

Claude Code 2.0:

- **Documentation:** https://docs.anthropic.com/claude/docs/claude-code
- **Installation:** `npm install -g @anthropic-ai/claude-code`
- **Pricing:** https://www.anthropic.com/pricing

My templates (open source):

- CLAUDE.md template
- Subagent configurations

Community:

- Join the discussion in the comments below
- Share your results with #ClaudeChallenge on Twitter/X
- Connect with other participants to share learnings

Have questions before you start? Drop them in the comments — I'll answer as many as I can.

Have you tried Claude Code 2.0 for legacy refactoring? What were your results? Share your experiences in the comments — especially the challenges and limitations. The more honest feedback we share, the better this technology becomes.

What's your experience with AI consistency in development workflows? Share your team's approach to reliable AI assistance — I'm always interested in how different teams solve these challenges.

Or check out my new Master Guide for Spec-Driven Development:

👉 **Step 1:** Read this Master Guide — your evergreen hub for Spec-Driven Development resources.
👉 **Step 2:** Read Part 1: The Foundation to set up your memory system, constitution, and specification workflow.
👉 **Step 3:** Continue with Part 2: Execution and Scaling to turn specs into plans, tasks, and tested features.

About the Author

Alireza Rezvani is a Chief Technology Officer, Senior Full-stack Architect, and Software Engineer, as well as an AI Technology Specialist, with expertise in modern development frameworks, cloud-native applications, and agent-based AI systems. With a focus on *ReactJS*, *NextJS*, *Node.js*, *AngularJS* and cutting-edge AI technologies and concepts of AI engineering, Alireza helps engineering teams leverage tools like *Gemini CLI*, and *Claude Code* or *Codex from OpenAI* to transform their development workflows.

Connect with Alireza at *alirezarezvani.com* for more insights on AI-powered development, architectural patterns, and the future of software engineering.

Looking forward to connecting and seeing your contributions — check out my *open source projects on GitHub*!

✨ Thanks for reading! If you'd like more practical insights on AI and tech, hit **subscribe** to stay updated.

I'd also love to hear your thoughts — drop a comment with your ideas, questions, or even the kind of topics you'd enjoy seeing here next. Your input really helps shape the direction of this channel.

Angularjs   Claude Code   Software Development   Learning To Code

Following

Written by Reza Rezvani

882 followers · 68 following

As CTO of a Berlin AI MedTech startup, I tackle daily challenges in healthcare tech. With 2 decades in tech, I drive innovations in human motion analysis.

Responses (1)

Bgerby

What are your thoughts?

Pierre-Emmanuel Larrouturou
Oct 7

@Reza Rezvani: FYI the end section "Where SDD Still Falls Short" is in a Markdown file, which makes it hard to read.

Reply

More from Reza Rezvani

In nginity by Reza Rezvani

The Flutter Architecture That Saved Our Team 6 Months of Rework

Sep 14 391 14

Reza Rezvani

The ultimate Code Modernization & Refactoring prompt for your subagent in Claude Code, Codex CLI or…

Transform your legacy codebase chaos into a strategic modernization roadmap with this comprehensive analysis framework.

Oct 4 130 2

Reza Rezvani

I Discovered Claude Code’s Secret: You Don’t Have to Build Alone

I’ve been coding long enough to know that the late-night debugging sessions aren’t glamorous. They’re just necessary.

Sep 19 151 2

In nginity by Reza Rezvani

I Let Claude Sonnet 4.5

IMAGINE this: It’s 6 a.m., the kind of quiet dawn where the world’s still wrapped in that soft, hazy light filtering through your blinds...

Sep 29 101 1

See all from Reza Rezvani

Recommended from Medium

In Dare To Be Better by Max Petrusenko

Claude Skills: The $3 Automation Secret That’s Making Enterprise Teams Look Like Wizards

How a simple folder is replacing $50K consultants and saving companies literal days of work

Oct 17 283 4

In ITNEXT by Mario Bittencourt

Up your AI Development Game with Spec-Driven Development

Spec Driven Development is new promising way to adopt AI and keep the developer in the loop. I will cover all aspects of SpecKit here.

5d ago 310 2

Reza Rezvani

Context Engineering: The Complete Guide to Building Production-Ready AI Coding Agents

Why Your AI Agent Fails (And How Context Engineering Fixes It)

Jul 11 13

In AI Software Engineer by Joe Njenga

Why Claude Weekly Limits Are Making Everyone Angry (And $100/Month Plan Will Not Save You)

Yesterday, I finally hit my weekly Claude limit, and I wasn't surprised, since I see dozens of other users online going crazy over these...

5d ago 119 20

The CS Engineer

Forget JSON — These 4 Data Formats Made My APIs 5× Faster

A 1.2 KB JSON payload transformed a smooth request into a 120 ms wait. Switching formats cut that to under 20 ms for some endpoints.

Sep 29 1.2K 35

In Realworld AI Use Cases by Chris Dunlop

The complete guide to Claude Code’s newest feature “skills”

Claude Code released a new feature called Skills and spent hours testing them so you don’t have to. Here’s why they are helpful

4d ago 61 4

See more recommendations