SPECSTORY

Patterns of Prompting: From Real-World Practice

Distilled from 200+ conversations building production micro-SaaS with AI agents

> **Note**: While most patterns work across AI coding assistants, some techniques (marked as "Claude Code Specific") leverage unique capabilities of Claude Code and Anthropic models.

🚀 The 5-Minute Quick Start

Start with these four high-impact patterns:

1. Context-Intent-Constraint (CIC) Opening

```
❌ "Fix the login bug"
✅ "We have a login flow at @/app/auth [CONTEXT], fix the 'invalid token'
error users get after entering correct password [INTENT], we're prototyping so
quick fix is fine [CONSTRAINT]"
```

Formula: Context + Intent + Constraint = Clear Direction

2. The Instant Pivot (2-3 Attempts Max)

```
After 2 failed attempts:
❌ "Let's try another way to fix this..."
✅ "looks like we need to take a step back. let's switch to [completely
different approach]"
```

Saves: 3-5x time vs incremental debugging

3. Quality Language That Ships Features

```
❌ "Make the UI good with proper styling and nice colors"
✅ "make it feel like Stripe's checkout - clean, minimal, trustworthy"
```

Result: AI instantly understands the quality bar and aesthetic

4. Show, Don't Tell (Data & Examples)

```
❌ "The performance is slow, please optimize"
✅ "Here's what I'm seeing: [paste actual metrics/errors]
    The page takes 8s to load. Target: <2s"
```

Impact: Eliminates guesswork and back-and-forth

📐 Core Patterns for Effective Prompting

Pattern 1: The Context-Intent-Constraint (CIC) Formula

- **When to use**: Starting any new task or feature
- **Frequency in practice**: 80% of successful sessions

The Structure:

1. **Context**: "we have an API in @apps/api/"
2. **Intent**: "now we need to add authentication"
3. **Constraint**: "we're prototyping so keep it simple"

Pattern 2: Progressive Disclosure

- **When to use**: Complex features requiring multiple steps
- **Why it works**: Manages cognitive load, maintains focus

The Technique:

```
Initial: "we have opportunities to improve the auth flow"
↓ (After initial response)
Then: "specifically these UX issues: [numbered list]"
↓ (After understanding shown)
Finally: "make it feel like butter - smooth and instant"
```

Red Flags to Avoid:

- Starting with 15+ requirements
- Mixing abstraction levels
- No clear success criteria

Pattern 3: Read-First Discovery

- **When to use**: Before modifying existing code
- **Impact**: 50% reduction in rework

The Approach:

```
"read @README.md @apps/api/README.md and @apps/docs/README.md
then let's plan how to handle git integration"
```

Note: Claude Code has a built-in Read tool. Other assistants may need file contents pasted directly.

Why This Matters:

- Prevents misaligned assumptions
- Grounds solutions in actual code
- Builds shared context efficiently

Pattern 4: The Pivot Protocol

- **When to use**: After 2-3 failed attempts at same approach
- **Saves**: 3-5x time vs incremental fixes

Recognition Signals:

- Same error appearing repeatedly
- Increasing complexity without progress
- "This is getting complicated" feeling

The Response:

```
"looks like we need to take a step back.
let's try a different approach using [alternative]"
```

Pattern 5: Quality Language Analogies

- **When to use**: When specs can't capture what you want
- **How it works**: Aesthetic language guides technical decisions

Examples That Work:

- "make it feel like butter" → smooth, instant UX
- "keep it clean and simple" → minimal dependencies
- "make it feel solid" → comprehensive error handling
- "like Stripe's documentation" → quality benchmark

Impact: 40% fewer revision cycles

Pattern 6: Opportunity Framing

- **When to use**: When addressing problems or improvements
- **Creates**: Collaborative energy vs corrective dynamic

The Approach:

```
Instead of: "The auth flow is broken in these ways..."
Use: "We have opportunities to improve the auth flow..."

Instead of: "Fix these security vulnerabilities..."
Use: "Let's strengthen our security posture by addressing..."
```

Why This Works:

- Focuses on future state rather than current problems
- Maintains positive momentum
- Reduces defensive responses from AI

Pattern 7: Data-Driven Context

- **When to use**: Debugging, optimization, decision-making
- **Impact**: Eliminates guesswork and multiple rounds

The Method:

```
"We're seeing these performance issues:
[paste actual metrics/logs/data]
Based on this data, let's optimize for..."
```

What to Include:

- Error logs for debugging
- Performance metrics for optimization
- User analytics for product decisions
- Actual data directly from your backend
- Actual output when something's wrong

Pattern 8: Step-by-Step Progression

- **When to use**: Complex multi-part features
- **Prevents**: Scope creep and overwhelming responses

The Structure:

```
"Let's break this into steps:
Step 1: [specific task]
Step 2: [specific task]
Step 3: [specific task]

Start with Step 1 only."
```

Key: Complete and validate each step before moving to the next

---

🎯 Additional Techniques

The ULTRATHINK Directive (Claude Code Specific)

- **When to use**: Security analysis, architecture decisions, complex integrations
- **Works with**: Claude Code and other Anthropic models that support deep reasoning

```
"Let's ULTRATHINK about the performance implications of this approach,
considering database load, caching strategies, and user experience"
```

Note: This pattern leverages Claude's extended thinking capabilities. May not work with other AI coding assistants. Reserve for truly complex problems - overuse dilutes effectiveness

Numbered Constraint Lists

When to use: Security reviews, quality audits, comprehensive checks

```
"evaluate @/app for security concerns:
1. XSS vulnerabilities
2. SQL injection vectors
3. CSRF protection
4. Rate limiting
5. Input validation"
```

Limit: Keep the list to a reasonable maximum

Context Continuation Architecture

- **When to use**: Multi-session projects, context limit reached

```
"This session continues from our previous conversation...
Summary:
1. Primary goal: [what we're building]
2. Decisions made: [key technical choices]
3. Current state: [what's working]
4. Next steps: [what we're doing now]"
```

Pro Tip: Save Your Context with SpecStory

Install SpecStory's free extensions (docs.specstory.com) for your AI coding assistant:

- Works with: Cursor, Claude Code, VSCode + Copilot
- Auto-saves all your chat history and conversations
- Makes context continuation seamless between sessions

Benefits:

- Never lose context between coding sessions
- Build on past work with full understanding
- Transform conversations into reusable specifications
- Learn from your own prompt evolution

Immediate Testing Requirements

- **When to use**: After every implementation
- **Catches issues**: In 5 minutes vs 50 minutes later

```
"tell me what I should be able to do once I run the server to test"
"create a test page to verify this works"
"show me the exact commands to validate this"
```

Why critical: Validates shared understanding immediately

Reference Architecture Anchoring

- **When to use**: When describing complex systems
- **10x faster**: Than writing specifications from scratch

```
"build this like GitHub's PR review interface"
"use a pattern similar to Vercel's deployment flow"
"look at https://github.com/[example] for the approach"
```

Include: Screenshots, URLs, or specific product references

🚫 Anti-Patterns to Avoid

1. Specification Sprawl

- ❌ Writing 500-line detailed specs
- ✅ Short intent + external reference + constraints

2. Perfectionism in Prototypes

- ❌ "Add comprehensive error handling, logging, monitoring..."
- ✅ "We're prototyping - basic error messages are fine"

3. Context Assumption

- ❌ "Continue where we left off" [without context]
- ✅ [Provide explicit summary of previous work]

4. Vague Quality Requests

- ❌ "Make it good"
- ✅ "Make it like Vercel's deployment flow - simple and clear"

📊 Pattern Combinations That Amplify Results

The "Discovery-Plan-Execute" Combo

1. **Read-First**: Understand existing code
2. **Reference Architecture**: "like this example: [link]"
3. **Progressive Disclosure**: Reveal requirements gradually
4. **Immediate Testing**: "tell me how to test this"

The "Constraint Liberation" Stack

1. **CIC Opening**: Set context, intent, and constraints
2. **Declare Prototyping**: Removes complexity
3. **Quality Language**: "clean and simple"

   Result: 90% reduction in solution space, paradoxically increasing creativity

The "Error-to-Excellence" Flow

1. **Try approach** (2-3 attempts max)
2. **Hit errors** → "I'm seeing these errors: [paste]"
3. **Pivot decisively** → "let's take a step back and try [alternative]"
4. **Test immediately** → Validate new approach

   Result: 3-5x faster than incremental debugging

🎮 Practice Scenarios

Scenario 1: Building a New Feature

You need to add real-time collaboration to a document editor.

1. **CIC Opening**: "We have a working editor at @/app/editor, now we need real-time collaboration, we're prototyping so WebSockets is fine"
2. **Progressive Disclosure**: Start with "sync cursor positions" then add "conflict resolution"
3. **Reference Architecture**: "like Google Docs but simpler"
4. **Immediate Testing**: "create a test page with two windows"

Scenario 2: Debugging Production Issue

Users report slow page loads on dashboard.

1. **Read-First**: "read @/app/dashboard and check for performance issues"
2. **Data Context**: "here are the performance metrics: [data]"
3. **Numbered Constraints**: List specific areas to investigate
4. **Pivot Ready**: If first approach fails, try different angle

---

📈 Measuring Your Progress

Track these metrics over your first 10 sessions:

Metric	Starting	Target	Why It Matters
Clarification requests from AI	5-7	1-2	Clear initial prompts
Iterations to working code	4-5	1-2	Better framing
Time to first success	45min	15min	Efficient patterns
Pivot recognition time	Never	2-3 attempts	Avoid rabbit holes

🔑 Key Takeaways

1. **Constraints paradoxically increase creativity** - "we're prototyping" unlocks speed
2. **Quality language beats technical specs** - "make it feel like butter" > 10 requirements
3. **Spatial awareness via @** - Always use @ for file references
4. **Errors are navigation** - Pivot quickly when stuck
5. **Progressive disclosure manages complexity** - Don't dump everything at once
6. **Assume competence** - Never ask "are you familiar with..." - just dive in
7. **Test immediately** - Validate every step before moving forward
8. **External examples > internal specs** - Reference existing products saves time

---

💡 Your Next Steps

This Week:

1. Practice the CIC opening formula on every request
2. Use @ notation for all file references
3. Declare "prototyping" to reduce complexity

Next Week:

1. Add progressive disclosure for complex tasks
2. Practice pivot recognition (2-3 attempts max)
3. Experiment with quality language

Ongoing:

1. Build your personal pattern variations
2. Track which patterns work best for your domain
3. Share discoveries with your team

🧠 The Psychology Behind the Patterns

Why These Patterns Work

- **Cognitive Load Management**: Breaking complex tasks into patterns reduces mental overhead for both human and AI.
- **Shared Mental Models**: The @ system and reference architectures create common understanding instantly.
- **Positive Reinforcement Loop**: Quality language and opportunity framing maintain collaborative energy.
- **Constraint as Freedom**: Removing options ("we're prototyping") paradoxically increases creative solutions.

Building Pattern Intuition

After ~10 sessions, you'll notice:

- Patterns become automatic (unconscious competence)
- You'll combine patterns without thinking
- You'll develop personal variations
- Your velocity will dramatically increase

Remember: These patterns emerged from hundreds of real conversations building production software agentically. They're not rules - they're tools. Adapt them to your context, and create your own.

Questions about patterns? Try them first, then reach out:

- greg@specstory.com
- jake@specstory.com