RSLT.DIGITAL

PULSE

Framework for controlled AI-supported development

Fundamentals, methodology, and application in agent-based software development

Author: Manuel Fuß

Position: Head of AI, RSLT.DIGITAL

Version: 1.0

Date: November 2025

RSLT.DIGITAL GmbH
Ulmer Str. 40
73728 Esslingen am Neckar

Geschäftsführer: Tim Resl

Sitz der Gesellschaft
Esslingen am Neckar
HRB 759452, Amt. Stuttgart

St.-Nr. 59340/73888
USt-IdNr. DE 310112889

Kontakt
Telefon +49 711 65 68 87 - 10

hi@rslt.digital
www.rslt.digital

Bankverbindung
Volksbank Plochingen eG
RSLT.DIGITAL GmbH

IBAN DE46 6119 1310 0839 0660 07
BIC GENODES1VBP

About this framework

This document is based on over a year of practical work with AI-assisted development. Hundreds of hours in Cursor Agent Mode, dozens of features delivered productively. The documented patterns were developed from specific errors and their systematic analysis.

The framework shows how ChatGPT, Claude, and Cursor can be professionally into existing development processes – without compromising quality and with a demonstrable ROI.

Characteristics of this framework

Practical: All content comes from productive application.
Honest: It documents what works and what doesn't.
Applicable: All prompts can be copied, all exercises can be performed.
Open: The methodology can be adapted to individual working methods.

Distinction

This framework is not a prompt engineering guide, tool documentation, or manifesto about the future of development.
It is a working methodology.

Target

The framework is aimed at developers with at least two years of professional experience who want to use AI productively. A prerequisite is a willingness to critically question established working methods.

Not suitable for programming beginners, people without a technical background, or readers who expect ready-made copy-paste solutions.

Notes on use

1. Read Part I (Foundations) in its entirety
2. Complete the three exercises from Part III in parallel with Part II
3. Use Part IV (Resources) as a reference work
4. Adapt the .cursorrules for your own projects

Note for teams: The .cursorrules can be shared within the team, ensuring consistent code quality across all participants.

RSLT.DIGITAL GmbH
Ulmer Str. 40
73728 Esslingen am Neckar

Geschäftsführer: Tim Resl

Sitz der Gesellschaft
Esslingen am Neckar
HRB 759452, Amt. Stuttgart

St.-Nr. 59340/73888
USt-IdNr. DE 310112889

Kontakt
Telefon +49 711 65 68 87 - 10

hi@rslt.digital
www.rslt.digital

Bankverbindung
Volksbank Plochingen eG
RSLT.DIGITAL GmbH

IBAN DE46 6119 1310 0839 0660 07
BIC GENODES1VBP

RSLT.DIGITAL

What is PULSE?

PULSE describes a way of working in which AI systems run autonomously in loops—and you only control them at crucial points.

A pulse is always a precise intervention:

- **Start pulse** (prompt or context that starts a loop)
- **Correction pulse** (intervention in case of deviations)
- **Review pulse** (validation, quality gate)

The loop runs. You set the pulses. The system remains under control.

The chapters of this framework show you:

- Where pulses originate (foundations)
- How to formulate pulses (6 elements)
- Which patterns work (workflow patterns)
- When pulses are necessary (Safety & Escalation)
- How to interrupt faulty loops (Debug)
- How to control multiple loops in parallel (Parallel Work)

Why "PULSE"?

Your heartbeat runs continuously—but without the rhythmic peaks, blood would not circulate. The pulse is the moment when energy is pushed into the system.

The same applies to AI-supported development: the loop runs autonomously. But without your targeted interventions, it loses direction and power. Each pulse pushes new energy into it – a context reset, a correction, a quality check.

Not constant monitoring. Rather, rhythmic control.

RSLT.DIGITAL GmbH
Ulmer Str. 40
73728 Esslingen am Neckar

Geschäftsführer: Tim Resl

Sitz der Gesellschaft
Esslingen am Neckar
HRB 759452, Amt. Stuttgart

St.-Nr. 59340/73888
USt-IdNr. DE 310112889

Kontakt
Telefon +49 711 65 68 37 - 10

hi@rslt.digital
www.rslt.digital

Bankverbindung
Volksbank Plochingen eG
RSLT.DIGITAL GmbH

IBAN DE46 6119 1310 0839 0660 07
BIC GENODES1VBP

PART I: FOUNDATIONS

Your technical setup

Before you write a single prompt: Your setup determines success or failure. Most developers fail not because of bad prompts, but because of a lack of system.

1. The 3-level architecture

AI-assisted development is not a single-tool job. You need an architecture in which each tool has its specific role.

Level 1: Concept & Strategy (ChatGPT/Claude)

Purpose: Thinking, not building

What happens here:

- Understanding and structuring requirements
- Discuss architecture decisions
- Create technical concepts as .md files
- Breaking down complex problems into solvable steps

Why not start directly in Cursor?

Cursor is a construction worker. If you tell it to "build me something nice," it will build it. But without a blueprint, it will be a mess.

Rule of thumb: The bigger the feature, the more time you should spend in ChatGPT/Claude BEFORE Cursor.

Practical setup:

ChatGPT Projects / Claude Projects:

- **One main project per work context**—e.g., "RSLT AI Development"
- **Store context files**—brand guidelines, tech stack, coding standards
- **DO NOT create one project per customer** – this will become confusing
- **Restart each chat** – avoids bias from previous conversations

Level 2: Build & Implementation (Cursor)

Purpose: Turn concept into working code

What happens here:

- Load concept file (.md) as a basis
- Composer + Agent Mode for multi-file edits
- Continuous Git commits (every 5-10 minutes)
- Independent testing in the browser (where possible)

Cursor modes in use:

Mode	When to use	Typical use case
Composer + Agent	Default mode for everything	Build feature across multiple files
Chat	Only for questions	Questions about understanding architecture
Cmd+K	For Git only	Git Push after approval
Plan Mode	For complex features	Strategy first, then build

⚠ **CRITICAL RULE:** Never leave Agent Mode running unattended for longer than 30 minutes.

Experience shows: After 30 minutes without supervision, you can usually reject everything and start over. The AI loses context or builds in the wrong direction.

Level 3: Escalation & problem solving

Purpose: If the cursor freezes, you need a second opinion

When to escalate?

- Cursor makes the same mistake 2-3 times (loop detected)
- Cursor says "it's fixed" but obviously isn't
- Cursor does not understand your problem, even after rephrasing
- Solution feels architecturally wrong

The escalation workflow:

Step 1: Ask the cursor

"Explain our current problem to me in such a way that I can explain it to another developer. Describe: current state, target state, what we have tried, why it doesn't work."

Step 2: Export the problem

Either copy Cursor's explanation OR use the project export script:

"Create a script that writes all project-relevant code files to a text file with absolute path information."

Step 3: To ChatGPT/Claude as an "external dev"

"You are a senior developer looking at an existing project. Here is the current status [export], here is the problem [Cursor's explanation]. Give me a solution that I can give to my developer (Cursor) as an instruction."

Step 4: Solution back to Cursor

Give ChatGPT's solution as instructions to Cursor—don't copy and paste code, but rather the idea/strategy.

Why this detour?

Because Cursor is stuck in the project context. A fresh pair of eyes (ChatGPT/Claude without project history) often sees the problem differently. You are the translator between the two.

2. Multi-model strategy

You don't need ONE perfect model. You need the RIGHT model for the task.

The model hierarchy in practice:

Model	Main use	Strengths	When to use
Claude Sonnet 4.5	Master in Cursor	Code quality, comprehension	Standard for everything
GPT-5	Escalation	Different perspective	When Claude hangs
Opus 4.1	Complex architecture	Deep reasoning	Difficult design decisions
ChatGPT	Concept phase	Rapid iteration	Developing ideas
Claude Projects	Context layer	Memory, documents	Long-term contexts

Practical example: Payment integration

Phase 1 – Concept (ChatGPT): "Help me develop the architecture for Stripe integration in our app."

Phase 2 – Build (Cursor + Claude 4.5): Load concept as .md, implement feature

Phase 3 – Blocker (Cursor 3x error in webhook handling): Escalate problem to GPT-5

Phase 4 – Architecture review (Opus 4.1): "Check the entire payment implementation for security and best practices"

3. Git as a safety net & thinking tool

Git is not optional. Without Git, Agent Mode is Russian roulette.

The Git workflow:

1. Project start: Create Git repo immediately

git init
git add .
git commit -m "Initial commit"

2. While Agent Mode is running: Checkpoint every 5-10 minutes

Not manually – Cursor does this automatically if you configure it correctly. But you should keep an eye on Git Log.

3. After each feature: Meaningful commit

feat: add user authentication with JWT

- Added JWT token generation and validation
- Implemented refresh token mechanism
- Added middleware for protected routes
- Tests for all auth flows

Closes #123

Why such detailed commits?

- **Cherry-picking:** You want to transfer individual features to other projects later
- **Debugging:** "When did we change this and why?"
- **Documentation:** Git history is a searchable knowledge base
- **Onboarding:** New team members read the Git log to understand the project

Language in commits:

ALWAYS English. Even if your code is commented in German. Why? Because you will be working internationally, your tools are international, and English commits are more searchable in Stack Overflow, GitHub, etc., even for AI.

4. The documentation system

Code alone is not documentation. You need a system that grows as you work—not something you have to do at the end as a mandatory exercise.

3-level documentation:

Level 1: .cursorrules (project-specific rules)

Location: In the root directory of each project

Contents:

- README content (project overview, setup instructions)
- API documentation (endpoints, parameters, responses)
- Deployment information (how to deploy, where are secrets)
- Recurring problems & solutions ("If X happens, do Y")

Why in .cursorrules? Because you can reference them in prompts using @. Cursor reads them automatically during critical operations.

Level 2: /docs/ (feature documentation)

Location: /docs/ folder in the root

Naming convention:

docs/
2025-11-12_payment-integration-problem.md
2025-11-10_auth-flow-redesign.md
2025-11-08_database-migration-learnings.md

Content per file:

- Problem description (What was broken/unclear)
- Solution approaches (What we tried)
- Final solution (What worked)
- Lessons learned (What did we learn)

Level 3: Global cursor rules

Location: Settings -> User Rules (applies to ALL projects)

Content:

- System Info (Mac M2, Node Version, Python Version)
- General coding standards (formatting, naming)
- Security rules (never commit secrets, etc.)
- Cross-cutting best practices

Who writes the documentation?

The AI. Always.

When a problem has been solved:

"Document this problem and the solution in an .md file. File name: docs/[DATE]_[SHORT DESCRIPTION].md. Structure: Problem, approaches to solving it, final solution, lessons learned."

When a problem occurs repeatedly:

"Add this insight to the .cursorrules so that we automatically take the right action next time."

The principle: Self-learning system

Every problem you solve makes your project smarter. The .cursorrules become the institutional memory. New developers (or AI agents) automatically benefit from everything that has been learned before.

Summary: Your setup checklist

Before you start a new project:

- ✅ ChatGPT/Claude Project created with context files
- ✅ Initialize Git repo
- ✅ .cursorrules file created (copy template from Part IV)
- ✅ /docs/ folder created
- ✅ Cursor Composer + Agent Mode understood
- ✅ Escalation strategy clear (when to use GPT-5/Opus)
- ✅ 30-minute rule internalized

If even ONE of these is missing: Stop. Set it up. It takes 10 minutes now, saves you hours later.

PULSE POINT:

The 3-level architecture

The setup defines WHERE you set pulses:

- Level 1 (concept): Start pulse – you set the direction
- Level 2 (Cursor): The loop runs - you observe
- Level 3 (Escalation): Peak Pulse - you intervene when the loop tips

Without this setup, you don't know when a pulse is needed.

RSLT.DIGITAL GmbH
Ulmer Str. 40
73728 Esslingen am Neckar

Geschäftsführer: Tim Resl

Sitz der Gesellschaft
Esslingen am Neckar
HRB 759452, Amt. Stuttgart

St.-Nr. 59340/73888
USt-IdNr. DE 310112889

Kontakt
Telefon +49 711 65 68 37 - 10

hi@rslt.digital
www.rslt.digital

Bankverbindung
Volksbank Plochingen eG
RSLT.DIGITAL GmbH

IBAN DE46 6119 1310 0839 0660 07
BIC GENODES1VBP

PART II: THE 6-ELEMENT FRAMEWORK

How to write prompts that work

Most developers don't fail because of AI. They fail because of their prompts.

"Build me a login form" → AI builds something
"That's wrong" → AI changes something
"Still wrong" → "AI doesn't work"

Problem: Not the AI, but a lack of context.

The 6 elements of an effective prompt

Every good prompt consists of a maximum of 6 elements. Not all of them are always necessary, but at least 3-4 should be included.

Element 1: ROLE

What is it?

You explicitly tell the AI from which perspective it should work. Not "you are an AI," but a specific professional role.

Examples

❌ *Bad: "Help me with this code."*

✅ *Good: "You are a senior backend developer with 10 years of Node.js experience."*

✅ *Better: "You are a senior backend developer who has been managing this project from the beginning and knows it better than anyone else."*

Why does this work?

LLMs are pattern matchers. "Senior developer" activates different patterns than "AI assistant." The role gives the AI a context from which to make decisions.

Role library for different tasks:

Task	Role
Architecture decision	You are a solutions architect with 15 years of experience in scalable web applications
Bug fixing	You are a debugging specialist who systematically narrows down problems
Code review	You are a tech lead who reviews code for quality, security, and maintainability
Database design	You are a database engineer with expertise in PostgreSQL and performance optimization
Frontend work	You are a senior frontend developer specializing in React/Next.js
API design	You are an API architect who designs RESTful APIs according to best practices

Element 2: CONTEXT

What is it?

Where are you right now? What happened before? What is the goal?

The problem most developers have:

They think AI "knows" what they mean. It doesn't. You have to say EVERYTHING that's in your head.

Examples:

❌ *Bad:*
"The API isn't working."

✅ *Good:*
"We have a Next.js app with API routes. The /api/users route returns 500. It was working yesterday. This morning we updated Prisma to v5."

✅ *Better:*
"Context: Next.js 14 app, API routes in /pages/api, Prisma ORM v5 since this morning. Last working version: yesterday at 6:00 p.m. (Git Hash: abc123). User-facing problem: Login fails. Error log: [screenshot here]."

Context checklist:

ALWAYS specify:

- Tech stack (which frameworks, versions)
- What worked (last known good state)
- What has changed (updates, new features, configs)
- What the goal is (not just the problem, but what you want to achieve)
- Who is affected (only you when testing? All users? Certain browsers?)

Element 3: INPUT

What is it?

The material the AI is supposed to work with. Don't describe it abstractly—show it.

Input types:

Input type	How to provide	Example
Error log	Screenshot or copy-paste	Terminal output with stack trace
Existing code	Upload file or code block	The component to be changed
Requirements	Text or document	Customer requirements specification, user stories
Design	Screenshot or Figma link	UI mockup for new feature
API documentation	URL or PDF	Stripe API Docs for payment
Conversation transcript	Text	Customer conversation with feature requests

Rule of thumb: More input = better results. But structured, not chaotic.

Practical example:

❌ *Vague:*
"Make the table more attractive"

✅ *Specific with input:*
"Here is the current table code [code]. Here is the design mockup [screenshot]. Make the table responsive and add hover states as shown in the mockup."

Element 4: OUTPUT

What is it?

What should the result be? In what format? How detailed?

The most common problem:

Developers don't describe what they want in enough detail.

- "Make a concept" → AI makes some kind of concept.
- "Make a 3-page technical concept as Markdown with an architecture diagram and code examples" → AI knows what to do.

Output specification:

- **Format:** Markdown, JSON, code, table, diagram?
- **Scope:** Short (1 paragraph), Medium (1 page), Detailed (5+ pages)?
- **Structure:** Which sections? In what order?
- **Target audience:** For you, for the team, for the customer, for non-technical people?
- **Level of detail:** High-level overview or implementation-ready?

Output examples for different scenarios:

Scenario 1: Feature concept

Output requirement:
"Create a technical concept as a Markdown file with the following sections:
1. Feature overview (2-3 sentences)
2. User flow (step-by-step)
3. Technical architecture (front end, back end, database)
4. API endpoints (method, path, payload, response)
5. Edge cases (what can go wrong)
6. Implementation roadmap (5-7 milestones)

Scenario 2: Bug analysis

Output requirement:
"Analyze the bug and give me:
1. Root cause (one line)
2. Why it happened (technical explanation)
3. Fix strategy (step-by-step)
4. Code changes (specific files and lines)
5. Test plan (how do we validate the fix)"

Scenario 3: Code implementation

Output requirement:
"Write the code for the login form using:
- TypeScript + React Hooks
- Formik for form handling
- Yup for validation
- Tailwind for styling
- Error messages in German
- Unit tests with Jest

Practical tip: Use Plan Mode with Cursor and get suggestions that match the desired result. Name your plan, your desired result with the appropriate input, and have a goal-oriented conversation with the AI. Here you can make adjustments before implementation and discuss with the AI within the plan context why which solution is the best, from your point of view and from the AI's point of view.

Element 5: ACTION

What is it?

What specifically should the AI DO? Analyze, write, rebuild, test, explain?

Action verbs and their meanings:

Action	Meaning	Example prompt
Analyze	Understand, don't change	"Analyze the code and explain the architecture to me."
Create	Rebuild from scratch	"Create an API for user management"
Expand	Add to existing	"Extend the login form with OAuth"
Refactor	Rewrite without changing functionality	"Refactor the auth code to TypeScript"
Fix	Solve problem	"Fix the memory leak in the WebSocket connection"
Test	Perform validation	"Test all API endpoints with different inputs"
Document	Write explanation	"Document payment integration for new developers"
Optimize	Make it better/faster	"Optimize DB queries for faster loading times"

⚠ **IMPORTANT: One action per prompt**

Beginner's mistake: "Analyze the code, find the bug, fix it, write tests, and document it."

That's 5 actions. The AI will then do them all half-heartedly. Better:

Prompt 1: "Analyze the code and find the bug"
[AI responds]

Prompt 2: "Fix the bug you found"
[AI responds]

Prompt 3: "Write tests for the fix."
[AI responds]

Prompt 4: "Document the problem and the solution"

Element 6: EXAMPLES

What is this?

Show the AI what you want AND what you DON'T want. Positive and negative examples.

Why is this the most important element?

Because "good" and "bad" are subjective. What you consider "clean code" may be something else to the AI. Examples calibrate the AI to YOUR standards.

Practical example: Code style

Without examples:
"Write me an API route for user login."
→ AI does something

With examples:

"Write me an API route for user login.

Do it like this example:
✅ *GOOD:*

```
export async function POST(req: Request) {
  try {
    const { email, password } = await req.json()
    // validation
    // business logic
    return NextResponse.json({ success: true })
  } catch (error) {
    return handleError(error)
  }
}
```

Do NOT do it like this:
❌ *BAD:*

```
app.post('/login', (req, res) => {
  // no try-catch
  // no types
  res.send('ok')
})
```

Follow the GOOD example: TypeScript, try-catch, proper error handling.

Example types for different situations:

Situation	Example strategy
Code style	Show a code block in the desired style + one in the undesired style
Naming	List 5 good variable names + 5 bad ones
Architecture	Show desired folder structure vs. chaotic one
Error messages	User-friendly vs. technical messages
API response	Desired JSON format vs. poor
documentation	Good README vs. incomplete

The complete 6-element framework in action

Now let's put it all together. Here is a real-world prompt with all 6 elements:

Example 1: Feature implementation

Scenario: You are tasked with building a user management system.

Bad prompt:

"Build me a user management system."

Good prompt with 6 elements:

[1. ROLE]
You are a senior full-stack developer with expertise in Next.js and Prisma ORM.

[2. CONTEXT]
We are building a SaaS application for team management. Currently, we have:
- Next.js 14 with App Router
- Prisma with PostgreSQL
- Clerk for authentication (already integrated)
- Tailwind CSS for styling

We now need an admin page where administrators can manage users.

[3. INPUT]
Here is our current Prisma schema:
[Code block]

Here is a similar feature that we already have (team management):
[Code block]

[4. OUTPUT]
Create:
1. A Markdown file with technical concept (feature overview, API endpoints, frontend structure, database changes)
2. Prisma schema extensions (if necessary)
3. API route definitions with types
4. React component structure

[5. ACTION]
First, do NOT create the code, but rather the concept. We will then develop it step by step together.

[6. EXAMPLES]
Use the team management feature [see Input] as a guide. Use the same patterns:
✅ *Server actions for mutations*
✅ *React Query for fetching*
✅ *Shadcn UI Components*
✅ *Optimistic Updates*

Do NOT do it this way:
❌ *Client-side fetch calls*
❌ *Inline styles instead of Tailwind*
❌ *Direct DB calls from the frontend*

Example 2: Bug fixing

Scenario: Your API returns 500, but you don't know why.

Bad prompt:

"The API isn't working, fix it."

Good prompt with 6 elements:

[1st ROLE]
You are a debugging specialist with expertise in Node.js and API error analysis.

[2. CONTEXT]
Situation:
- Next.js API route: /api/orders/[id]
- Yesterday it worked
- This morning: Prisma updated to v5.7
- Since then: 500 Internal Server Error
- Only with specific order IDs, not all of them
- Browser console shows: "Failed to fetch"

[3. INPUT]
Here is the API code:
[Code block]

Here is the server log:
[Error log]

Here is the Prisma client code:
[Code block]

[4. OUTPUT]
Give me:
1. Root cause analysis (one line: what is broken)
2. Why is this happening (technical explanation with reference to Prisma v5 changes)
3. Fix strategy (step-by-step)
4. Code patch (exact lines that need to be changed)

[5. ACTION]
First analyze the problem. Do NOT give me the fix right away, but explain what you see. Then we will develop the fix together.

[6. EXAMPLES]
I had a similar problem with another endpoint:
The problem was: Prisma v5 has a breaking change in relations
The solution was: .include() changed to .select()

Check if this is also the case here.

RSLT.DIGITAL

The 3 biggest beginner mistakes when prompting

Mistake 1: Impatience + "set in stone"

The problem:

Developers write a prompt, AI gives an answer, answer is not perfect → "AI doesn't work for me"

The reality:

No prompt is perfect the first time around. The first result is a SUGGESTION, not the truth. You have to iterate.

Doing it right:

Prompt 1: "Understand this first" → [AI explains]

Prompt 2: "Good, now let's develop the concept step by step"

Prompt 3: "The concept looks good, let's start with the implementation"

Prompt 4: "The login flow is still missing, add that"

Prompt 5: "Now let's add error handling."

→ 5 prompts, but each intermediate result is better

Mistake 2: Too vague + too many assumptions

The problem:

"Build me an API." → AI doesn't know: Which stack? Which authentication? Which database schema? REST or GraphQL?

The reality:

AI can't read your mind. What is "obvious" to you ("of course I mean Next.js API Routes") is not obvious to AI.

Do it right:

Say EVERYTHING explicitly. Even if it seems redundant to you. The AI needs:

- Tech stack ("Next.js 14, TypeScript, Prisma, PostgreSQL")
- Conventions ("We use server actions, not API routes")
- Your setup ("Mac M2, Node 20, Docker for DB")
- Existing code ("Here is our auth implementation for reference")
- Coding standards ("TypeScript strict mode, ESLint, Prettier")

Mistake 3: Sticking to your own standards

The problem:

"I've never done it that way before" → Developer rejects AI solution because it's different from what they're used to.

The reality:

AI thinks differently than humans. Sometimes it finds better solutions that you wouldn't have seen. Sometimes it just does things differently, but not worse.

Doing it right:

- Ask yourself: "Is this objectively worse, or just different?"
- If functionally correct: Accept the different approach.
- If it really is worse: Explain WHY it is worse (don't just say "do it my way").
- Use examples to show your standard, but be open to alternatives.

RSLT.DIGITAL GmbH
Ulmer Str. 40
73728 Esslingen am Neckar

Geschäftsführer: Tim Resl

Sitz der Gesellschaft
Esslingen am Neckar
HRB 759452, Amt. Stuttgart

St.-Nr. 59340/73888
USt-IdNr. DE 310112889

Kontakt
Telefon +49 711 65 68 87 - 10

hi@rslt.digital
www.rslt.digital

Bankverbindung
Volksbank Plochingen eG
RSLT.DIGITAL GmbH

IBAN DE46 6119 1310 0839 0660 07
BIC GENODES1VBP

The iterative principle: Develop together

The most important insight from 1 year of AI-supported development:

AI is not a machine, but a thinking partner.

This means that you don't give a perfect prompt and get a perfect result. You develop the solution TOGETHER with the AI, step by step.

The ideal development flow:

Phase 1: Understanding

You: "Here is the requirement [input]. Understand that first and explain to me how you would implement it."

AI: [Explains understanding and approach]

You: "Understood. But I would do point 3 differently because XY. What do you think?"

AI: [Adjusts approach]

Phase 2: Develop concept

You: "Now we're going to develop the technical concept. Step by step, not all at once. Start with the database structure."

AI: [Shows DB schema]

You: "Almost there. The user table is still missing the 'role' field. Add that."

AI: [Updated schema]

You: "Perfect. Now the API structure."

AI: [Shows API concept]

You: "Good. Now the front-end components."

[...further iterations...]

Phase 3: Implementation

You: "The concept is ready. Save it as concept.md. Then we'll start the implementation in Cursor."

[Switch to Cursor]

You: "Here's the concept [concept.md]. Implement Milestone 1: Database setup."

Cursor: [Builds DB schema, migrations, seed data]

You: "Good. Test it yourself."

Cursor: [Tests, finds issue]

You: "Fix the issue and then move on to Milestone 2."

[...more milestones...]

Why this flow works:

- You remain in control (every step is validated)
- AI stays focused (not overwhelmed with too much at once)
- You understand the code (because you helped develop each step)
- Errors are detected early (not just at the end)
- Quality remains high (iterative improvement)

Summary: 6-element cheat sheet

Save this template and use it for every important prompt:

 # My prompt template

1. ROLE
You are [specific professional role with expertise]

2. CONTEXT
Situation: [What is the current status]
Tech stack: [Frameworks, tools, versions]
Goal: [What needs to be achieved]
Constraints: [What needs to be considered]

3. INPUT
[Code block / Screenshot / Document / Error log]

4. OUTPUT
Give me:
1. [Specific deliverable 1]
2. [Specific deliverable 2]
3. [Specific deliverable 3]

Format: [Markdown / Code / JSON / etc.]
Scope: [Short / Medium / Detailed]

5. ACTION
[Analyze / Create / Expand / Fix / Test / ...]

Important: [Special instructions on how to proceed]

6. EXAMPLES
Do it THIS WAY:
✅ *GOOD: [Positive example]*

DON'T do it like this:
❌ *BAD: [Negative example]*

Got it? Then let's proceed step by step.

💡 **Pro tip:** Save this template as a file in your ChatGPT/Claude Project. Then you can insert it at any time using @ and just adjust the values.

PULSE POINT:

Prompts as Initial Pulses

Every prompt is a pulse. It starts a loop or corrects it.

The 6 elements ensure that your pulse is precise enough

to steer the loop in the right direction.

Bad prompt = weak pulse = loop drifts off course.

RSLT.DIGITAL GmbH
Ulmer Str. 40
73728 Esslingen am Neckar

Geschäftsführer: Tim Resl

Sitz der Gesellschaft
Esslingen am Neckar
HRB 759452, Amt. Stuttgart

St.-Nr. 59340/73888
USt-IdNr. DE 310112889

Kontakt
Telefon +49 711 65 68 37 - 10

hi@rslt.digital
www.rslt.digital

Bankverbindung
Volksbank Plochingen eG
RSLT.DIGITAL GmbH

IBAN DE46 6119 1310 0839 0660 07
BIC GENODES1VBP

PART III: WORKFLOW PATTERNS

Real-world prompts

Theory is nice. Practice is better. Here are real prompts from real projects—unfiltered, just as they were actually written. Plus derived examples in the same style.

1. Feature extension

If you want to extend an existing feature, context is crucial. The AI needs to understand: what is there, what is missing, why is it missing.

Real example: Expanding exercise selection

Context: User has a meditation app. The exercise selection is currently based only on mood. To be expanded to include time budget.

Original prompt:

Create an expanded version with additional logic for the selection that is based not only on mood but also on time budget, so that the user does not get long exercises when they have little time.

What makes this prompt good?

- **Clear goal:** "expanded version"
- **Specific addition:** "also based on time budget"
- **User need:** "so that the user does not get long exercises"
- **Implicit logic:** Short on time = only short exercises

What could be better?

If the prompt were written with 6 elements:

[ROLE] You are a senior front-end developer with UX expertise.

[CONTEXT]
We have a meditation app. Currently: User selects mood → App suggests exercises.
Problem: Users with little time are also suggested 20-minute exercises.
Tech stack: React Native, TypeScript, State for state.

[INPUT]
Here is the current selection algorithm:
[Code block of existing logic]

[OUTPUT]
Extend the logic with:
1. Time budget filter (5 min / 10 min / 15 min / 20 min+)
2. Combined score calculation (mood + time match)
3. Fallback if no suitable exercise (suggest the next shortest)

[ACTION]
Extend the existing code. Do not change the function signature, only the internal logic.

[EXAMPLES]
User input: Mood="anxious," Time="5min"
✅ *Should suggest: Breathing exercise (4 min)*
❌ *Should NOT suggest: Body scan (15 min) even if perfect for anxious*

Derived example: E-commerce filter

Scenario: Online shop has product filters by category. To be expanded to include price range.

Prompt in communication style (iterative toward the goal):

Create an extended version with additional price filter logic that not only filters by category but also by price range, so that the user only sees products within their budget.

With 6 elements (more preparatory work, fewer iterative steps):

[ROLE] You are an e-commerce front-end developer.

[CONTEXT]
Next.js shop with server components. Filter state in URL parameters.
Current: category filter works.
Goal: Add price range filter (0-50€, 50-100€, 100-250€, 250€+)

[INPUT]
Current filter code:
[Code]

[OUTPUT]
Extend by:
1. Price range select component
2. Combined filter query (category AND price)
3. URL param handling for both filters
4. Reset button that clears both filters

[ACTION]
Extend existing code. Use the same patterns as for the category filter.

[EXAMPLES]
URL: /shop?category=electronics&price=50-100
✅ *Should display: Only electronics between $50-100*
❌ *Should not display: All electronics, regardless of price*

2. Analysis & Documentation

If you want AI to understand and document a project, you need to give it a role and specify a clear output structure.

Real example: Project delta analysis

Context: Customer has specifications for phase 2. You want to know: What has been implemented, what is missing.

Original prompt:

Again. Here is the Phase 2 requirements specification.pdf in the root directory. This document contains the customer's plans. Go through the entire project, see what we already have and what we don't have yet, and write this in a new file, with precise details of what is there and what is not yet there based on the requirements specification. Don't include time estimates for now; we'll do that separately. But the document must be as comprehensive and accurate as possible. You do this in your role as the senior developer who has been overseeing the project from the beginning and knows it better than anyone else.

What makes this prompt effective?

- **"Again":** Signals: Previous attempt was not good enough
- **Explicit file reference:** "Phase 2 specifications.pdf in root"
- **Clear scope:** "Go through the entire project"
- **Output specification:** "new file," "precise details of what is there/missing"
- **Scope limitation:** "without time estimates for now"
- **Quality requirement:** "as comprehensive and accurate as possible"
- **Role assignment:** "senior dev who knows the project better than anyone else"

Structured with 6 elements:

[ROLE]
You are the senior developer who has been overseeing this project from the beginning and know it better than anyone else.

[CONTEXT]
The customer has sent the specifications for phase 2. We are in the middle of phase 1.
I need a gap analysis: what do we already have, what is still missing?
I will use the document for the quote and scheduling.

[INPUT]
- Phase 2 specifications: /root/Lastenheft_Phase2.pdf
- Current project status: Everything in the /src/ folder
- Previous commits: Git log for the last 3 months

[OUTPUT]
Create delta_analysis.md with:
1. Executive summary (3-5 bullet points: what is there, what is missing)
2. Feature matrix:
- Feature name | Status | Details
- Status: ✅ Complete / 🔄 In progress / ❌ Missing / ⚠️ Different than desired
3. For each feature: Brief description of the current status
4. Open questions (where specifications are unclear)

IMPORTANT: Initially WITHOUT time estimates. We will do these separately.

[ACTION]
1. Read the requirements specification in full
2. Scan the project code
3. Match features from the requirements specification with the code
4. Document findings

Be as detailed as possible—it's better to have too much information than too little.

[EXAMPLES]
✅ GOOD Format:
Feature: User dashboard
Status: 🔄 In progress (60% complete)
Details:
- Dashboard route exists (/dashboard)
- User stats are loaded
- Still missing: Activity timeline, export function
- Different from specifications: We are using Chart.js instead of D3

✗ *BAD Format:*
Dashboard: partially there

Derived example: Code audit after migration

Scenario: You have migrated from JavaScript to TypeScript. You want to know: What has been converted, what is still missing.

Prompt in manual style:

Go through the entire project, see which files are already TypeScript and which are still JavaScript. Write this in a new file with precise details for each file: what has been converted, what is still missing, which types are any, where is there still unsafe code. The document must be as comprehensive and accurate as possible. You do this in your role as the TypeScript expert who checks code quality.

With 6 elements:

[ROLE]
You are a TypeScript expert who checks code quality and type safety.

[CONTEXT]
We are gradually migrating from JavaScript to TypeScript.
Project: React app with Node backend.
Goal: 100% TypeScript, no 'any', strict mode.

[INPUT]
- Entire project in /src/
- tsconfig.json (strict: true)
- Already converted: /src/components/auth/

[OUTPUT]
Create typescript_migration_audit.md:
1. Summary:
- X% converted
- Y files still JS
- Z 'any' types found
2. File matrix:
- File | Status | Type Coverage | Issues
3. Critical findings (unsafe code, missing types)
4. Migration roadmap (which files next)

[ACTION]
Scan all .js and .ts files, analyze types, find 'any'.

[EXAMPLES]
✅ *GOOD Entry:*
src/utils/api.js
Status: ❌ *Still JavaScript*
Type coverage: 0%
Issues:
- Fetch without response types
- Error handling untyped
Priority: HIGH (used in 15+ components)

❌ *BAD Entry:*
api.js: not yet converted

3. Safety rules & deployment protection

Agent Mode is powerful—but dangerous without safeguards. You have to explicitly tell the AI what it can and cannot do.

Real example: Git deployment rules

Context: Agent deployed parameters directly to production instead of testing them locally. This must not happen again.

Original prompt:

Create a .gitdeploy file in which you write something like this, for example, that the parameters in section or background are not allowed in individual components, and you specifically name the two discussed here as examples. I always include this file during git deployment so that you know what is allowed and what is not.

What makes this prompt important?

- **Reactive:** Arises from a real problem
- **Specific examples:** "section," "background"
- **Persistent rule:** File that is read during every deployment
- **Explicit purpose:** "so you know what is allowed and what is not"

Structured with 6 elements:

[ROLE]
You are a DevOps engineer who defines deployment rules.

[CONTEXT]
Problem: Agent has just distributed parameters from the CSS section to individual components and deployed them directly. Production is now broken.
Root cause: Global parameters (section, background) belong in the layout, not in individual components.

[INPUT]
The two parameters that caused the problem:
- section: { padding, background }
- background: { color, image }

[OUTPUT]
Create a .gitdeploy file (plain text, no code) with:
1. Rule description
2. Specific examples (the two parameters)
3. What is allowed, what is prohibited
4. When this rule applies (to which files/paths)

[ACTION]
Write a clear, machine-readable rule file.

[EXAMPLES]
✅ *GOOD Format:*

RULE: Global Layout Parameters
AFFECTED: section., background.**
ALLOWED: Only in /layouts/.tsx*
FORBIDDEN: In /components/.tsx*
REASON: These are layout-level concerns

❌ *BAD Format:*
Don't put section stuff in components

Derived example: DB migration rules

Scenario: During DB migration, the agent changed the production DB instead of the test DB. You want to prevent this.

Original prompt:

Create a .dbmigration file with rules stating that migrations must always be tested locally first, then on staging, and only then on production. Give specific examples from the last 3 migrations we have done. I will then include this file with every migration so that you know what the process is.

With 6 elements:

[ROLE]
You are a database administrator with production safety expertise.

[CONTEXT]
Problem: Agent executed Prisma migration directly on production.
Result: Schema change was breaking, production down for 2 hours.
We need: Strict migration pipeline Local → Staging → Production.

[INPUT]
Last 3 migrations that were problematic:
1. add_user_role_column (affected existing users)
2. change_order_status_enum (breaking change)
3. create_analytics_table (performance issue)

[OUTPUT]
Create .dbmigration rules file:
1. Migration pipeline (3 stages)
2. Test requirements per stage
3. Rollback strategy
4. Forbidden actions (direct production changes)

[ACTION]
Write clear rules based on the 3 problem migrations.

[EXAMPLES]
✅ *GOOD Rule:*

STAGE: Local
REQUIRED:
- npm run db:migrate:dev
- npm run test:integration

- Manual verification: seed data works
NEXT: Only if all pass → Staging

❌ *BAD Rule:*
Test it first

4. Bug fixing with test instructions

When dealing with bugs, it is important to not only describe the problem, but also how you test and on which setup.

Real example: Mobile viewport fix

Context: Users can scroll left/right on mobile devices. This should not be possible.

Original prompt:

The viewport on mobile must be fixed in width; I must not be able to scroll it to the right or left. Test independently. I am testing on iPhone 12 Pro.

What makes this prompt effective?

- **Clear constraint:** "fixed in width"
- **Negative specification:** "I must not be able to move it"
- **Test autonomy:** "test independently"
- **Test device:** "iPhone 12 Pro" – important for reproduction

Structured with 6 elements:

[ROLE]
You are a mobile-first frontend developer.

[CONTEXT]
Bug report: User can scroll horizontally on mobile.
Should be: Content is always 100vw, no horizontal scrolling possible.
Probable cause: Some element is wider than the viewport.

[INPUT]
- Current CSS files in /styles/
- Problematic page: /dashboard
- Screenshot where the problem is visible: [attach]

[OUTPUT]
1. Root cause (which element/CSS causes overflow)
2. Fix (CSS changes)
3. Test confirmation (test and confirm independently)

[ACTION]
1. Find the problem
2. Fix it
3. Test independently in the browser
4. Confirm that it works

[EXAMPLES]
Test environment: iPhone 12 Pro (390x844px)
✅ *PASS: No horizontal scrollbar*
✅ *PASS: All elements within 390px*
❌ *FAIL: Any element larger than viewport*

Derived example: API performance bug

Scenario: API endpoint is too slow. 5+ seconds response time.

Manual-style prompt:

The /api/products endpoint is much too slow, response time over 5 seconds. Find the problem, fix it, test it yourself. I'm testing with 1000 products in the database.

With 6 elements:

[ROLE]
You are a backend performance engineer.

[CONTEXT]
Problem: GET /api/products only responds after 5+ seconds.
Acceptable: <500ms
Setup: PostgreSQL DB with 1000 products.
User impact: Shop page takes forever to load.

[INPUT]
- API code: /api/products/route.ts
- DB schema: prisma/schema.prisma
- Current query: [code block]
- Server log with timing: [Log]

[OUTPUT]
1. Performance analysis (where is the bottleneck)
2. Fix strategy (DB optimization, caching, pagination)
3. Code changes
4. Test results (before/after timing)

[ACTION]
1. Analyze query performance
2. Optimize (DB index, query optimization, or caching)
3. Test with 1000 products
4. Confirm <500ms response time

[EXAMPLES]
Test setup: 1000 products in DB
✅ *PASS: Response time <500 ms*
✅ *PASS: All products returned*
❌ *FAIL: Still >1s*

RSLT.DIGITAL GmbH
Ulmer Str. 40
73728 Esslingen am Neckar
Geschäftsführer: Tim Resl

Sitz der Gesellschaft
Esslingen am Neckar
HRB 759452, Amt. Stuttgart
St.-Nr. 59340/73888
USt-IdNr. DE 310112889

Kontakt
Telefon +49 711 65 68 37 - 10
hi@rslt.digital
www.rslt.digital

Bankverbindung
Volksbank Plochingen eG
RSLT.DIGITAL GmbH
IBAN DE46 6119 1310 0839 0660 07
BIC GENODES1VBP

5. Git workflow & branching strategies

Agent Mode can perform Git operations—but you need to control when and how. Branch strategies are essential.

Real-world example: Feature branch for PR

Context: You want to develop a feature in isolation and have it reviewed via PR before merging.

Original prompt:

Create an extra branch for the XYZ component and push it there so I can live it via pr.

What makes this prompt good?

- **Casual but clear:** "create" – relaxed tone, clear task
- **Feature isolation:** "separate branch for the XYZ component"
- **Explicit purpose:** "so I can push it live via PR"
- **Implicit permission:** Git push is otherwise prohibited, but explicitly allowed here

Structured with 6 elements:

[ROLE]
You are a Git expert who manages feature branches.

[CONTEXT]
We are developing XYZ integration.
Workflow: Feature branch → PR → Review → Merge to main
Current: All changes on main (not ideal)
Goal: Isolated branch for clean PR.

[INPUT]
- Current branch: main
*- New files: /components/XYZ/**
- Changed files: /api/booking/route.ts

[OUTPUT]
1. New branch created: feature/XYZ-integration
2. All relevant changes committed
3. Branch pushed to origin
4. Ready for PR

[ACTION]
1. git checkout -b feature/XYZ-integration
2. git add [only XYZ-relevant files]
3. git commit with meaningful message
4. git push origin feature/XYZ-integration

[EXAMPLES]
✅ *GOOD commit message:*
feat: add XYZ booking integration

- Added XYZ embed component
- Implemented booking webhook handler
- Updated booking API with XYZ support
- Tests for webhook validation

❌ *BAD Commit message:*
added stuff

Derived example: Hotfix branch for production

Scenario: Critical bug in production. You need to fix it immediately without waiting for feature work.

Original prompt:

Create a hotfix branch from the production tag, fix the payment bug in it, push it so I can deploy it directly without taking the feature work with it.

With 6 elements:

[ROLE]
You are a production support engineer who manages hotfixes.

[CONTEXT]
CRITICAL BUG in Production:
- Payment flow breaks down during Stripe checkout
- Affects: All customers for 2 hours
- Root cause: API key validation is faulty
Main branch: Has unfinished features (not deploy-ready)
Solution: Hotfix branch from last production tag

[INPUT]
- Production tag: v2.3.1
- Bug location: /api/checkout/route.ts line 47
- Fix: API key check must be optional for test mode

[OUTPUT]
1. Hotfix branch from v2.3.1
2. Bug fixed
3. Tested locally
4. Pushed for immediate deployment

[ACTION]
1. git checkout v2.3.1
2. git checkout -b hotfix/payment-api-key-validation
3. Fix bug
4. git commit
5. git push origin hotfix/payment-api-key-validation

[EXAMPLES]
✅ *GOOD Hotfix commit:*
fix(payment): make API key validation optional in test mode

- Bug: Stripe checkout failed for all test transactions
- Cause: API key validation was too strict

- Fix: Skip validation when STRIPE_MODE=test
- Tested: Test and live transactions work

Fixes #567

❌ *BAD Commit:*
Fixed bug

6. Design constraints & responsive fixes

If you want design changes, you need to be very specific. "More attractive" is not a constraint—but "dynamic height based on content" is.

Real example: Dynamic tile height

Context: Tiles have a fixed height. If there is a lot of text, it gets cut off.

Original prompt:

Tiles 1 and 2 need to be taller. Please make the height truly dynamic based on the text content, see screenshots 1 and 2.

What makes this prompt effective?

- **Specific elements:** "tiles 1 and 2" - not all of them, just these
- **Emphasis:** "really" - signals that the previous attempt was wrong
- **Technical solution:** "dynamically based on the text content"
- **Visual examples:** "see screenshots 1 and 2"

Structured with 6 elements:

[ROLE]
You are a CSS specialist with expertise in Flexbox/Grid.

[CONTEXT]
Problem: Tiles 1 and 2 have a fixed height (200px).
With long text: Text is cut off.
User feedback: "I can't see the whole text."
Goal: Height adapts to content.

[Example]

In the attachment, you will find two screenshots: one showing the current display and the other showing the desired display.

[INPUT]
- Current CSS: .tile { height: 200px; }
- Tiles: #tile-1, #tile-2
- Content: Variable text length

[OUTPUT]
CSS changes for:
1. Tile 1: Dynamic height based on text
2. Tile 2: Dynamic height based on text
3. Min-height: 200px (not smaller than now)
4. Max height: none (grows indefinitely)

[ACTION]
Change CSS from fixed height to min-height + auto.

[EXAMPLES]
✅ *GOOD Solution:*

```
.tile {
  min-height: 200px;
  height: auto;
  display: flex;
  flex-direction: column;
}
```

❌ *BAD Solution:*

```
.tile { height: 300px; } // Still fixed
```

RSLT.DIGITAL GmbH
Ulmer Str. 40
73728 Esslingen am Neckar

Geschäftsführer: Tim Resl

Sitz der Gesellschaft
Esslingen am Neckar
HRB 759452, Amt. Stuttgart

St.-Nr. 59340/73888
USt-IdNr. DE 310112889

Kontakt
Telefon +49 711 65 68 87 - 10

hi@rslt.digital
www.rslt.digital

Bankverbindung
Volksbank Plochingen eG
RSLT.DIGITAL GmbH

IBAN DE46 6119 1310 0839 0660 07
BIC GENODES1VBP

7. Content generation with variants

When you need content, never ask for ONE result. Ask for several variants – then you can choose or combine.

Real example: Trust box variants

Context: You need different texts for trust elements on your landing page.

Original prompt:

Create 4 variants of the Trust Copy Box snippet with different texts for each box. Collect the texts of all 4 boxes from the HTML files: Create a file with all 4 HTML texts for Builder.io.

Structure with 6 elements:

[ROLE]
You are a conversion copywriter for B2B landing pages.

[CONTEXT]
Landing page for SaaS product.
Section: Trust indicators (social proof)
Current: 1 version, but we want to do A/B testing.
Target platform: Builder.io (requires HTML snippets)

[INPUT]
- Current trust box text: [HTML code]
- Target audience: B2B decision-makers
- Tone: Professional but approachable

[OUTPUT]
Create trust_variants.html with:
1. Variant A: Stats-Focused ("500+ Companies trust us")
2. Variant B: Testimonial-Style ("Saved us 20 hours/week")
3. Variant C: Security-Focused ("Bank-level encryption")
4. Variant D: Speed-Focused ("Setup in 5 minutes")

Per variant: 4 boxes (2-3 lines of text each)

[ACTION]
Write 4 complete variants as separate HTML sections.

[EXAMPLES]

✅ *GOOD Box (Stats-Focused):*

```
<div class="trust-box">
  <h3>500+</h3>
  <p>Companies using our platform</p>
</div>
```

❌ *BAD Box:*

```
<div>We're trusted</div>
```

Summary: Your prompt patterns

What you should take away from this chapter:

- Feature extension: ALWAYS mention user needs ("so that the user...")
- Analysis & documentation: Role assignment + "as detailed as possible"
- Safety rules: Learn from real problems, incorporate them into persistent rules
- Bug fixing: Describe the problem + how you test + on which setup
- Git workflow: Explicit permission for push, clear branch purpose
- Design constraints: Specify technical solution, not vague "nicer"
- Content: Never generate just one version, always generate multiple variants

Quick reference: Prompt starters according to situation

Situation	Prompt starters
Extend feature	Create an extended version with [X] to meet [user need]
Analyze project	Go through the entire project, look at [what], write it down in [file]
Safety rule	Create a [.file] with rules that [constraint] so that [reason]
Fix bug	The [element] must be [constraint]. Test independently. I will test on [device].
Git branch	Create a [branch-name] for [feature], push it so that I [reason]
Customize design	[Element] must be [constraint], make the [property] really [technical-solution]
Content	Create [N] variants of [content-type] with different [aspect]

PART IV: PEAK DETECTION & ESCALATION

Agent Mode under control

Agent Mode is not autopilot. It is a powerful employee that needs supervision.

In this chapter: How to use Agent Mode productively without it causing chaos. Based on real disasters and how to prevent them.

1. The 30-minute rule

The most important rule from 1 year of Agent Mode experience:

Never leave Agent Mode unattended for more than 30 minutes.

Why exactly 30 minutes?

Experience shows that after 30 minutes without supervision, there is a high probability that you will have to reject everything and start over.

What happens after 30 minutes:

- Agent loses context (makes changes that don't match the original goal)
- Builds in the wrong direction (feature creep without you noticing)
- Makes too many changes at once (debugging becomes impossible)
- Overwrites things that work ("I'll do it better")
- Creates dependencies you don't want (installs random packages)

RSLT.DIGITAL GmbH
Ulmer Str. 40
73728 Esslingen am Neckar
Geschäftsführer: Tim Resl

Sitz der Gesellschaft
Esslingen am Neckar
HRB 759452, Amt. Stuttgart
St.-Nr. 59340/73888
USt-IdNr. DE 310112889

Kontakt
Telefon +49 711 65 68 37 - 10
hi@rslt.digital
www.rslt.digital

Bankverbindung
Volksbank Plochingen eG
RSLT.DIGITAL GmbH
IBAN DE46 6119 1310 0839 0660 07
BIC GENODES1VBP

Practical implementation:

When starting a project with a concept file:

- Start agent mode with concept as input
- Set a 30-minute timer
- Watch what the agent does (don't walk away!)
- Understand: WHAT it does, WHY it does it that way
- After 30 minutes: Checkpoint, review, restart or continue

During feature build:

- Every 5-10 minutes: Git commit (automatic or manual)
- Continuously read what the agent is doing
- At the first sign of a loop or wrong direction: Stop
- Maximum 30 min until next manual review

Note: 30-minute autonomous runs are rather rare; this should really only be understood as the potential maximum autonomy for the agent. Most autonomous agent runs are much shorter.

2. Safeguards: What the agent is NOT allowed to do

Agent Mode needs clear boundaries. These rules are non-negotiable.

The 5 critical safeguards:

1. DELETE only upon request

The agent may NOT decide to delete files on its own. Every DELETE must be explicitly confirmed.

In .cursorrules:

RULE: File Deletion
NEVER delete files without explicit confirmation.
ALWAYS ask: "Should I delete [filename]? This cannot be undone."
WAIT for user confirmation before proceeding.

The "Do not delete automatically" setting is also set as a fixed parameter in the cursor settings; the addition in .cursorrules is only a double safeguard.

2. GIT PUSH only after request

Agent may commit (that's OK), but NOT push without permission. You want the option to review before it goes remote.

In .cursorrules:

RULE: Git Push
NEVER push to remote without explicit permission.
Local commits: OK
Remote push: ALWAYS ask first
Exception: When explicitly told "push this"

The "GIT PUSH" command must not be explicitly stored among the permitted commands. The information in the .cursorrules file is only intended as a double safeguard.

3. Test locally FIRST, THEN deploy

Agents should NEVER deploy directly to production. Always validate locally first.

In .cursorrules:

RULE: Deployment
Testing Pipeline: Local → Test → Staging → Production
NEVER skip testing steps
NEVER deploy directly to production
ALWAYS test locally first using npm run dev or similar

4. NO breaking changes without warning

If the agent notices "this will be breaking," it must warn BEFORE doing so.

In .cursorrules:

RULE: Breaking Changes
Before making changes that break existing functionality:
1. Identify impact (which components/features affected)
2. Warn user: "This will break [X]. Proceed?"
3. Wait for confirmation
Examples: API signature changes, DB schema changes, prop renames

5. NO secrets in the code

Agents must NEVER write API keys, passwords, or tokens directly into code.

In .cursorrules:

RULE: Secrets Management
NEVER hardcode:
- API keys
- Passwords
- Access tokens
- Database URLs

ALWAYS use:
- Environment variables (process.env.XYZ)
- .env files (not committed)
- Secret management services

If you need a secret: Ask user to add to .env

3. Loop Detection: When to escalate?

Loops are the most common problem with Agent Mode. You need to learn to recognize them quickly.

The 4 most common loop types:

Loop type 1: "Fixed" but obviously not

Agent makes a change, says "problem solved," but you see that the problem still exists.

Detection:

- Agent committed "fix: [problem]"
- You test: Problem is still there
- Agent makes another change: "fix: [same problem]"
- You test: Still not fixed

Action after 2-3 iterations:

1. STOP agent mode
2. Reject last 2-3 commits
3. Escalate to ChatGPT/Claude:
"Cursor has tried 3 times to fix [problem], but it didn't work. Here is the code [paste], here is the error [paste]. Give me a solution that I can give to Cursor as an instruction."

Loop type 2: Back and forth between 2 solutions

Agent makes change A, then back to change B, then back to A...

Recognition:

- Commit 1: "use approach X"
- Commit 2: "revert to approach Y"
- Commit 3: "actually X is better"
- Git Diff shows: Always the same lines changed

Immediate action:

1. STOP Agent Mode
2. Git reset to last stable state
3. Make a clear decision (you or with ChatGPT)
4. Restart agent with EXPLICIT instruction:
"Use Approach X. NOT Approach Y. Here's why: [reason]"

Loop type 3: Does not understand the problem

You describe the problem, the agent does something, but it's obvious that it hasn't understood.

Recognition:

- Agent makes changes to the wrong files
- Agent fixes a completely different problem
- You repeat the instruction 2-3 times, but the agent still does something else

Immediate action:

1. STOP agent mode
2. Switch to chat (not Composer)
3. Ask agent: "Explain to me what you understood. What is the problem, what are you trying to solve?"
4. If the explanation is wrong: Correct your understanding BEFORE continuing
5. Only when the agent explains correctly: Return to Composer

Loop type 4: Doing too much at once

Agent makes 20 changes at once, something breaks, you don't know what.

Detection:

- One commit changes 15+ files
- Mix of feature work + refactoring + bug fixes
- You can't figure out what broke

Prevention (better than cure):

Give agents smaller milestones:

❌ *Bad:*
"Build the entire user management feature"

✅ *Good:*
"Milestone 1: Only the user model and Prisma migration"
[Agent does it, committed]
"Good. Milestone 2: Only the API route for user list"
[Agent doing, committed]
"Good. Milestone 3: Only the frontend component"
[Agent doing, committed]

4. Escalation strategies

When Cursor hangs, you need a system. Not panic, but process.

The 3-step escalation:

Step 1: Ask Cursor "Explain the problem"

Before you escalate: Let Cursor explain what the problem is.

Prompt:

"Explain our current problem to me in such a way that I can explain it to another developer. Describe:
1. Current state (what is happening right now)
2. Desired state (what should happen)
3. What we have tried (what solutions we have attempted)
4. Why it doesn't work (your theory)"

Why this is important:

- Cursor must structure the problem (helps him himself)
- You get material for escalation
- Sometimes Cursor finds the solution himself

Step 2: Escalate to ChatGPT/Claude as an "external dev"

Take Cursor's explanation and give it to fresh AI.

Prompt template:

[ROLE]
You are a senior developer looking at an existing project.

[CONTEXT]
My developer (Cursor AI) is stuck on a problem.
Here is his explanation:
[Paste Cursor's explanation]

[INPUT]
Current code status:
[Code block or project export]

Error logs:
[Logs]

[OUTPUT]
Give me:
1. Your analysis (what is the actual problem)
2. Solution formulated as instructions for my developer
3. If necessary: code snippet as a reference

[ACTION]
Analyze and give me a solution that I can pass on as instructions.

[EXAMPLES]
Format of the solution:
"Tell your developer:
1. [Step 1]
2. [Step 2]
3. [Step 3]"

Stage 3: Model switch (GPT-5 or Opus 4.1)

If ChatGPT/Claude doesn't help either: Different AI, different perspective.

When to switch models:

- The problem is very complex (architecture decision)
- ChatGPT/Claude gives an unsatisfactory answer
- You need a "second opinion"
- Problem is domain-specific (e.g., ML, crypto)

Prompt remains the same, only different model.

5. Git as an undo button & experiment safety net

Git is not just version control. It is your most important safety tool.

Checkpoint strategy: Every 5-10 minutes

Why so often?

- An agent can cause a lot of damage in 5 minutes
- You want granular rollback points
- Small commits = easy cherry-picking later
- You understand code better when changes are small

Practical implementation:

Cursor automatically makes commits during Agent Mode. But you should check commit messages.

RSLT.DIGITAL

The secure Git workflow:

Before starting:

```
git status      # Clean state?
git log --oneline # Where are we?
git checkout -b feature/xyz  # New branch
```

During Agent Mode:

```
# Every 5-10 minutes (Agent does this automatically, you just watch):
git log --oneline -5  # What has been committed?
git diff HEAD~1      # What has changed?
```

When Problem:

```
# Option 1: Reject last commit
git reset --soft HEAD~1  # Undo but changes remain
git reset --hard HEAD~1  # Undo and changes gone

# Option 2: Reject multiple commits
git reset --hard HEAD~3  # Go back to 3 commits ago

# Option 3: Go back to a specific commit
git log --oneline        # Find good commit hash
git reset --hard abc123  # Go back to this state
```

After feature done:

```
# Cleanup commits (optional)
git rebase -i HEAD~10    # Squash small commits

# Push to Remote
git push origin feature/xyz

# Create PR (outside of cursor)
```

6. Review checklist after Agent Mode session

After each agent mode session (or every 30 minutes): Review. Don't trust blindly.

The agent mode review checklist:

✅ **Code quality:**

- Do I understand what the code does?
- Are naming conventions being followed?
- Are there any obvious code smells?
- Is error handling in place?
- Have edge cases been considered?

✅ **Functionality:**

- Does the feature work as intended?
- Have I tested it locally?
- What happens in case of invalid input?
- What happens in the event of network errors?
- Are there any performance issues?

✅ **Security:**

- No secrets hardcoded?
- Input validation in place?
- SQL injection safe?
- XSS protection?
- Authentication/authorization correct?

✅ **Git history:**

- Are commit messages meaningful?
- Can I understand the changes?
- Are there useful breakpoints for rollback?
- Is the branch clean (no merge conflicts)?

✅ **Documentation:**

- Are new features documented?
- Are breaking changes documented?
- Is README up to date?
- Are API docs updated?

🚩 Red flags – immediate action required:

- 🚩 Code you don't understand (then you can't debug it either)
- 🚩 Hundreds of lines in one commit (too much at once)
- 🚩 Dependencies you don't know (why was X installed?)
- 🚩 Files deleted without consultation (where did X go?)
- 🚩 Production URLs in the code (should be in .env)
- 🚩 console.log everywhere (debug code not cleaned up)
- 🚩 Commented-out code (why not deleted?)
- 🚩 TODO comments without issue numbers (will be forgotten)

Summary: Safety-first mindset

The most important takeaways:

- **30-minute rule:** Never leave unattended for longer than 30 minutes, review every 30 minutes
- **5 Safeguards:** DELETE, PUSH, DEPLOY, BREAKING, SECRETS – all protected
- **Loop detection:** Know 4 loop types, detect early, escalate quickly
- **3-step escalation:** Ask cursor → ChatGPT/Claude → Model switch
- **Git as a safety net:** Commits every 5-10 minutes, granular rollback points
- **Review checklist:** After each session: code, functionality, security, Git, docs

Final Wisdom:

Agent Mode is not autopilot. It is a co-pilot. You are always the pilot. Always.

The best developers with AI are not those who blindly trust AI. They are those who know when to trust and when to intervene.

PULSE POINT:

Peak Detection

This chapter defines WHEN you need to set a pulse:

- Loop Detection = Peak detected
- 30-minute rule = regular check pulse
- Escalation = correction pulse to another model

Safety is the early warning system for your pulse points.

RSLT.DIGITAL GmbH
Ulmer Str. 40
73728 Esslingen am Neckar

Geschäftsführer: Tim Resl

Sitz der Gesellschaft
Esslingen am Neckar
HRB 759452, Amt. Stuttgart

St.-Nr. 59340/73888
USt-IdNr. DE 310112889

Kontakt
Telefon +49 711 65 68 37 - 10

hi@rslt.digital
www.rslt.digital

Bankverbindung
Volksbank Plochingen eG
RSLT.DIGITAL GmbH

IBAN DE46 6119 1310 0839 0660 07
BIC GENODES1VBP

PART V: CORRECTIVE PULSES & DEBUGGING

When the code doesn't do what it's supposed to

Debugging with AI is different from classic debugging.

You are no longer the only one who has to find the error. But you have to decide: Which error is relevant? When do you escalate? When do you roll back?

This chapter shows you the complete debugging workflow—from the first error message to the solution.

1. The first error: Understand before you act

Most developers make a mistake: they see an error and immediately pass it on to the AI.

The problem: Not every error is responsible for the problem. Sometimes there are side effects, sometimes the actual error is elsewhere.

The 3-step debugging workflow:

Step 1: Read and validate the error yourself

Ask yourself:

- Is this error responsible for my current problem?
- Or is this an old error that was already there before?
- If there are multiple errors, which one is the primary one?
- What exactly happens when the error occurs?

Practical example:
User cannot log in. In the terminal: 5 different warnings + 1 error.
Read all 6 messages. Often only the error is relevant; the warnings are noise.

Step 2: Transfer error + context to AI

Only when you are sure that the error is relevant:

Example prompt:

IS: User clicks on "Login," form is submitted, but nothing happens
ERROR (see screenshot): "Cannot read property 'token' of undefined"
SHOULD: After logging in, the user should be redirected to the dashboard

Context: Login form sends POST to /api/auth/login, backend returns token, frontend should store token in localStorage.

Please fix and explain what the problem was.

Step 3: Validate AI solution

Cursor makes a fix. Now you need to check:

- Is the error gone? (Not just "Cursor says it's gone")
- Does the feature really work now?
- Did the fix break anything else?
- Do you understand what Cursor has changed?

⚠ **Important:** If Cursor says "I fixed it" but the error comes back, that's the first indicator of a loop (see next section).

2. Loop Detection: When the AI runs in circles

The most dangerous situation: Cursor "fixes" the same error over and over again, but nothing gets better.

In practice: If Cursor claims to have fixed something 2-3 times, but the error remains, it's a loop.

The 4 most common types of loops:

Loop type 1: "It's fixed," but obviously not

Symptoms:

- Cursor: "Error is fixed, please test"
- You test: Error is still there
- Cursor: "Ah sorry, now I've fixed it"
- You test: Still there
- Repeat 2-3 times

Action:
After 2-3 attempts: STOP.
1. Reject the last 2-3 edits
2. Ask cursor: "Explain the problem to me in a way that I can explain it to another developer."
3. Escalate this explanation + original error to ChatGPT/Claude

Loop type 2: Always the same file changed

Symptoms:

- Cursor always only edits component.tsx
- But the error is probably in the backend
- Or: Cursor always only changes CSS, but the problem is in the logic

Action:
If the cursor changes the same file three times but the problem remains:
"The problem is not in [file]. Look at [other areas]: [backend/state management/API/etc.]"

Loop type 3: Undoes its own fixes

Symptoms:

- Cursor changes line 45 from A to B
- Next edit: Line 45 back from B to A
- Next edit: Line 45 changes back from A to B
- Cursor is unsure of itself

Action:
Stop immediately. This is uncertainty, not a real debugging strategy.
Escalate to ChatGPT/Claude with the question: "What is the right approach here?"

Loop type 4: Too many changes, nothing works anymore

Symptoms:

- Cursor has changed 10+ files
- Original error is gone
- But now there are 5 new errors
- You lose track of what still works

Action:
ROLLBACK. Return to the last working state (Git).
Then: Start over, but this time step by step with checkpoints.

3. Rollback vs. forward patch: The 50/50 rule

If something breaks, you have 2 options:

Option	When to use it?	Advantage
Rollback (Git Reset / Reject)	Cross-file changes Becomes confusing You no longer understand what is where	Clean restart Maintain an overview
Forward patch (Continue iterating)	Small changes One file affected You understand the problem	Faster No context loss

The 50/50 rule from practice:
50% of cases: Forward patch (minor problems, clearly defined)
50% of cases: Rollback (cross-file, confusing)

Decision criterion: **clarity.** As soon as you can no longer see exactly what has been changed and where = reject.

Practical decision-making aid:

✅ **Forward patch if:**

- Only 1-2 files affected
- You understand every edit
- The problem can be clearly localized
- Changes are minor (<20 lines per file)

🔄 **Rollback if:**

- 5+ files changed
- You scroll through diffs and no longer understand
- New errors appear
- You think, "What did Cursor actually do there?"

4. Escalation: When ChatGPT/Claude is used as a "second opinion"

Cursor is your primary build tool. But sometimes Cursor gets stuck—then you need a second perspective.

The 3-step escalation:

Step 1: Cursor alone (80% of cases)

When:
Clear tasks, standard features, no unexpected problems

Procedure:
• ACTUAL/TARGET prompts
• Agent mode under supervision
• Iterative development
• Git checkpoints every 5-10 minutes

Stage 2: ChatGPT/Claude as advisor (15% of cases)

Trigger:

- Cursor stuck in loop (2-3 attempts)
- Cursor does not understand your problem
- You don't understand Cursor's solution
- Architecture decision required

Procedure:
1. Ask Cursor: "Explain our problem to me in a way that I can explain it to another developer."
2. Send this explanation + code (optional: export script) to ChatGPT/Claude
3. ChatGPT/Claude provides a solution "for your developer."
4. Send this solution back to Cursor with instructions

Example escalation prompt:

My developer (Cursor) is stuck on the following problem:
[Insert Cursor's explanation here]
Code context: [Optional: Export relevant code]
Give me a solution that I can give to my developer as an instruction.

Level 3: Model change (5% of cases)

Trigger:

- ChatGPT/Claude is also stuck
- Extremely complex problem
- Requires specialized model (e.g., GPT-5 for math, Opus 4.1 for reasoning)

Procedure:
Switch between Claude Sonnet 4.5 → GPT-5 → Opus 4.1
Depending on the strength of the model for your specific task

5. Max Mode: Security & Quality Review

After major features: Have the AI review everything again.

Max Mode = Large context window, project-wide analysis

When to use Max Mode?

- After completion of a major feature
- Before deployment to production
- After many rapid iterations
- When you have made many changes "blindly"

The Max Mode review prompt:

Scan the entire project and check for:

1. Security issues:
- *Unprotected API endpoints*
- *Missing input validation*
- *SQL injection risks*
- *XSS vulnerabilities*

2. Data protection issues:
- *Sensitive data in logs*
- *Credentials in code*
- *Lack of encryption*

3. Code quality:
- *Duplicates*
- *Dead code paths*
- *Performance bottlenecks*

Give me a prioritized list: Critical, High, Medium, Low

Best practice: Max Mode Review every 2-3 days or after every major feature. Don't wait until production—then it's too late for major fixes.

6. Reject strategy: Avoid trash code

Not every edit made by Cursor is beneficial. Sometimes Cursor creates workarounds that generate technical debt.

The reject function in Composer is your friend.

When to reject?

Code works, but is "hacky"
Cursor has built a workaround that works, but you can already see that it will cause problems later.

Too many changes at once
Cursor has changed 8 files. You no longer understand what is connected.

Code is unreadable
Works, but no one will be able to understand it in two weeks.

Violates best practices
e.g., all logic in a 500-line function, no separation of concerns

The correct reject workflow:

1. Reject in Composer
2. Prompt: "This works, but [describe problem]. Make it cleaner: [describe expectation]"
3. Cursor makes a new attempt
4. If still not good: Escalate to ChatGPT/Claude for architecture input

⚠ **Important:** Reject is not "Undo." It only discards Cursor's last suggestion. For a real undo, you need Git rollback.

7. Error log documentation: Learn from mistakes

Every bug is a learning opportunity. But only if you document it.

From practice: AI can document errors; you just have to set up the system.

The debug log system:

Structure:
/docs/ folder in the project root
File names: debug_YYYY-MM-DD_short-description.md
AI writes the logs, you review them

What belongs in a debug log?

- Problem description (What should work but doesn't?)
- Error message (complete, with stack trace)
- What was tried (Which fixes did not work?)
- What worked (the final solution)
- Root cause (Why did the problem occur?)
- Prevention (How can we prevent this in the future?)

Example prompt for debug log:

Document this bug in /docs/debug_2025-11-12_login-token.md:

· Problem: User could not log in
· Error: "Cannot read property 'token' of undefined"
· What didn't work: [list]
· What worked: [Solution]
· Root cause: [Why?]
· Prevention: [How to prevent?]

From debug log to .cursorrules:

The self-learning system:
1. Bug occurs → Debug log is written
2. Does the same type of bug occur again? → Pattern recognized
3. Pattern recognition goes into .cursorrules
4. Cursor no longer makes the same mistake

Example:
Bug: Cursor deploys to remote instead of testing locally
→ Debug log documents the problem
→ Happens twice
→ In .cursorrules: "Never deploy to remote with cp. Always test locally first."

Summary: Debug workflow cheat sheet

The complete debug workflow at a glance:

Situation	Action
Error appears	Read the error itself → Is it relevant? → Then move to cursor with ACTUAL/TARGET
Cursor says "fixed" but not	Loop detector: After 2-3 times → Escalation
Cursor always changes the same file	Redirect: "Problem is elsewhere, look in [X]"
Too many changes, confusing	Rollback (Git) → New with checkpoints
Small change, clearly defined	Forward patch → Continue iterating
Cursor is stuck	Escalation: Cursor explains → ChatGPT/Claude advises → Back to Cursor
After major feature	Max Mode Review: Security, data protection, quality

🎯 **Core principle:** Debugging with AI does not mean "AI does everything." It means: You control, AI executes. You decide when to escalate, when to roll back, when to continue iterating.

AI is your tool. You are the developer.

PULSE POINT:

Correction Pulse

Debugging is the most intense pulse moment. The loop is disrupted, and you need to stabilize it again. The methods here are your tools for precise correction pulses.

RSLT.DIGITAL GmbH
Ulmer Str. 40
73728 Esslingen am Neckar

Geschäftsführer: Tim Resl

Sitz der Gesellschaft
Esslingen am Neckar
HRB 759452, Amt. Stuttgart

St.-Nr. 59340/73888
USt-IdNr. DE 310112889

Kontakt
Telefon +49 711 65 68 87 - 10

hi@rslt.digital
www.rslt.digital

Bankverbindung
Volksbank Plochingen eG
RSLT.DIGITAL GmbH

IBAN DE46 6119 1310 0839 0660 07
BIC GENODES1VBP

PART VI: PARALLEL WORK & ENERGY MANAGEMENT

The art of multi-project management

With Agent Mode, you can work on multiple projects in parallel.

That sounds good. But: It's also the main source of exhaustion.

In this chapter: How to manage 2-3 projects at the same time without burning out. What really tires you out (spoiler: it's not the coding). And how to structure your workflow so that you stay productive.

1. The 3-project rotation: How it works

The setup:
You have 3 projects open at the same time. Each in its own cursor window. While Agent Mode works on Project 1, you check Project 2, start Agent on Project 3, then go back to Project 1.

That's 3x output in the same amount of time. But also 3x cognitive load.

The rotation workflow:

Step by step:

1. Project 1: Start Agent
   Activate Agent Mode in Composer, give the feature request, and Agent starts running. Expected duration: 15-30 minutes.
2. Switch to Project 2
   While Project 1 agent is running: Open Project 2, check what the agent has built, test, prompt for the next step, start agent in Project 2.
3. Switch to Project 3
   Project 2 agent is now running. Switch to Project 3, check status, review problems, trigger agent.
4. Back to Project 1
   Project 3 agent is running. Back to Project 1: Agent should be finished. Review, next step, restart agent.
5. Repeat
   1 → 2 → 3 → 1 → 2 → 3 → ...

Timing example (realistic):

Time	Action	What is happening in parallel?
09:00	P1: Start agent (login feature)	-
09:05	P2: Review, test, start new agent	P1 Agent running
09:15	P3: Review, fix problems, start agent	P1+P2 agents running
09:25	P1: Agent finished, review, next step	P2+P3 agents running
09:35	P2: Agent finished, review, next step	P1+P3 Agents running
09:45	P3: Agent finished, review, next step	P1+P2 agents running

⚡ **The reality:** In 45 minutes, you have made progress on 3 projects. Each project received 15-20 minutes of agent time, but you were never idle.

The secret: While one agent is working, you work on the next one.

2. What really tires you out: Context switching

The biggest misconception: "I'm tired because I've been coding so much."

Wrong. You're tired because you're constantly switching between projects.

The context switching problem:

What happens every time you switch projects:

- Reload mental model ("Where were we? What was the problem?")
- Understand the stack (backend? frontend? Which APIs? Which state?)
- Reconstruct context from Git history/docs
- Assess agent status ("What did the agent build? Is it good?")
- Plan next steps ("What comes next?")

The hidden bill:

• Per project change: ~3-5 minutes of context loading
• 3 projects = 6 switches per hour (1→2→3→1→2→3)
• 6 × 4 minutes = 24 minutes/hour just for context switching

40% of your time is spent on "Where was I again?"

Comparison: 1 project vs. 3 projects

Factor	1 project (8-hour day)	3 projects (8-hour day)
Pure working time	~7 hours (1 hour breaks)	~5 hours (3 hours context switching)
Output	1 feature complete	3 features 60-70% complete
Mental load	Low (one context)	High (three contexts)
Exhaustion	Normal	Significantly higher

The productivity paradox:

• 3 projects = 3x output (correct)
• But: 3x exhaustion (also true)
• It's not the coding that's tiring, it's the switching

You're not tired from working. You're tired from switching.

RSLT.DIGITAL

RSLT.DIGITAL GmbH
Ulmer Str. 40
73728 Esslingen am Neckar

Geschäftsführer: Tim Resl

Sitz der Gesellschaft
Esslingen am Neckar
HRB 759452, Amt. Stuttgart

St.-Nr. 59340/73888
USt-IdNr. DE 310112889

Kontakt
Telefon +49 711 65 68 87 - 10

hi@rslt.digital
www.rslt.digital

Bankverbindung
Volksbank Plochingen eG
RSLT.DIGITAL GmbH

IBAN DE46 6119 1310 0839 0660 07
BIC GENODES1VBP

3. The 2-project maximum rule

After 1 year of experience with parallel work: Most people should stick to a maximum of 2 projects.

Recommendation from practice:

- Standard: 1 project (deep work)
- During agent waiting times: 2 projects (productive)
- Only in exceptional cases: 3 projects (deadline pressure)

2 projects = the sweet spot between productivity and sustainability.

Summary: Parallel Work Cheat Sheet

The complete parallel work framework:

- Rotation: 1→2→3→1 - While the agent is running in project 1, you work on project 2
- What causes fatigue: Context switching—not the coding, but the mental shift
- Sweet spot: 2 projects - Productive, but sustainable. 3 only when under deadline pressure.
- Energy: Limited resource - After a 3-project day, you need recovery

🎯 **Key insight:** Parallel work with Agent Mode makes you 2-3x more productive. But it's not a license for endless work. Energy management is just as important as code management.

PART VII: WORKSHOP EXERCISES

From knowledge to practice

Theory is important. Practice is essential.

In this chapter: 3 concrete exercises you can work through right away. Each exercise builds on the 6-element framework and trains specific skills: From concept to code, loop detection, and agent mode under control.

Total time required: ~2-3 hours. Learning outcome: You will understand how vibe coding really works.

Exercise 1: From concept to code

Duration: 45 minutes | Difficulty: Beginner

Learning objective

What you will learn:

- Practical application of the 6-element framework
- ChatGPT/Claude → Experience the cursor workflow
- Understanding that 1st result ≠ final result
- Iterative development with "Understand that first"

Task

Build a contact form component with validation

Requirements
• Fields: Name, Email, Message
• Validation: Email format, name min. 2 characters, message max. 500 characters
• Error messages in German
• Submit button disabled as long as validation fails
• Success message after submit

Phase 1: Create concept (10 min)

Tool: ChatGPT or Claude

Your prompt (use the 6-element framework):

ROLE: You are a senior front-end developer with React expertise

CONTEXT: I want to build a contact form for an agency website. Users should be able to enter their name, email, and message.

INPUT: No existing codebase, new build

OUTPUT: A technical concept as a Markdown file with:
• Component structure
• State management approach
• Validation logic
• Tech stack (React, which libraries?)

ACTION: Create the concept. Briefly explain each point. Don't code yet, just the concept.

EXAMPLES:
✅ *How it should be: Clear component hierarchy, modular validation*
❌ *How it shouldn't be: Everything in one huge component, no separation of logic and UI*

Important: Before you continue: Have ChatGPT/Claude confirm that it has understood. Ask: "Do you understand the concept? Summarize it in 2-3 sentences."

Phase 2: Implement in Cursor (25 min)

Tool: Cursor Composer + Agent Mode

Procedure:

1. Open Cursor, new project
2. Save the concept as concept.md in the project
3. In Composer: "Read concept.md and understand the concept. Confirm briefly."
4. After confirmation: "Build the component step by step. Start with the basic structure."
5. Activate Agent Mode, let it work
6. Observe: What is the agent building? In what order?
7. After completion: Test in the browser
8. Not working perfectly? → Iterate: "Field X should..."

Phase 3: Reflection (10 min)

Questions to ask yourself:

- How often did you have to make adjustments?
- Did Cursor understand the concept correctly?
- Where was the result different than expected?
- Would you have coded it differently yourself?
- What was better than expected? What was worse?

Success criteria

You have understood the exercise if:
• The form works (even if not perfectly)
• You have iterated at least twice ("Not like this, but like this...")
• You understand what Cursor built (even if you would have done it differently)

Exercise 2: Debug loop & escalation

Duration: 40 minutes | Difficulty: Medium

Learning objective

What you will learn:

- Recognizing loops (2-3 attempts = loop)
- Understanding the escalation workflow
- Treating the cursor as a "developer" and ChatGPT/Claude as a "consultant"
- Letting go of "not my way"

Task

Fix intentionally broken code with 3 different bug types

You are given a project with 3 bugs:
1. Obvious error (typo)
2. Loop candidate (race condition)
3. Architecture problem (state management)

Setup: Code repository with bugs

Create a simple React project with these bugs:

Bug 1 - Typo (minor):
```javascript
const [userName, setUserName] = useState('');
// Later in the code:
console.log(usrName); // Typo: usrName instead of userName
```

Bug 2 - Race Condition (Loop Candidate):
```javascript
useEffect(() => {
  fetchData();
  setData(response); // setData directly, no async/await
}, []);
```

Bug 3 - State problem (difficult):
```javascript
// Parent passes state to child, but child changes state directly
<ChildComponent data={parentData} onChange={setParentData} />
// Child does: props.data.value = newValue (instead of onChange)
```

Phase 1: Fix bug 1 (5 min)

Cursor prompt:

ACTUAL: Console displays "usrName is not defined"
TARGET: userName should be logged
Please fix

Expectation: Cursor fixes this immediately (1 attempt). Easy win.

Phase 2: Address bug 2 (15 min)

Cursor prompt:

ACTUAL: fetchData is called, but data remains undefined
ERROR: Cannot read property of undefined
SHOULD: Data should be set after fetch
Please fix.

Observe: Cursor will probably need 2-3 attempts. It will say "fixed," but it won't work. That's your loop experience.

After 3 attempts: ESCALATE

1. Ask cursor: "Explain our problem to me in a way that I can explain it to another dev."
2. This explanation + code to ChatGPT/Claude: "My developer is stuck here..."
3. Solution from ChatGPT/Claude back to Cursor

Phase 3: Bug 3 - The difficult problem (15 min)

Cursor prompt:

ACTUAL: Child component changes data directly, parent does not see changes
TARGET: Changes should arrive in parent
Please fix.

Note: Cursor will probably suggest a different approach than you expected. Maybe Context API instead of props, or useReducer.

Your task: Accept it**.** Even if it's not your way—if it works, it's good.

Phase 4: Documentation (5 min)

Cursor prompt:

Document all 3 bugs in /docs/debug_2025-11-12_exercise.md:
· What was the problem?
· What did we try?
· What worked?
· Lessons learned for .cursorrules

Success criteria

You have understood the exercise if:
• You recognized a loop (bug 2)
• You escalated to ChatGPT/Claude
• You accepted an unexpected solution (bug 3)
• All 3 bugs are fixed and documented

Exercise 3: Multi-file feature with safety

Duration: 50 minutes | Difficulty: Advanced

Learning objective

What you will learn:

- Using Agent Mode productively, but in a controlled manner
- Understanding Git as an "undo button"
- Experience multi-file orchestration
- Understand code, don't blindly trust it
- Define and enforce safeguards

Task

Build a user dashboard with backend, frontend, and database

Features
• Backend: API endpoint /api/users (GET)
• Frontend: Dashboard component displays user list
• Database: User model with name, email, created_at

Twist: With safeguards

Setup: Define safeguards (10 min)

BEFORE you start: Create .cursorrules with safeguards

 # Safety rules for this project

1. NEVER DELETE without explicit confirmation
2. NEVER GIT PUSH without confirmation
3. ALWAYS test locally first, then deploy
4. For database operations: Dry run first
5. No credentials in the code, only env variables

Create Git repo:

```bash
git init
git add .
git commit -m "Initial commit - before Agent Mode"
```

Phase 1: Concept in ChatGPT/Claude (10 min)

Prompt (6 elements):

ROLE: Senior Full-Stack Developer

CONTEXT: User dashboard for agency, shows all users from DB

INPUT: New project, stack: React + Node.js + SQLite

OUTPUT: Technical concept with:
· File structure (frontend/backend/database)
· API design (/api/users)
· Component structure (Dashboard.jsx)
· Database schema (users table)

ACTION: Create concept as Markdown

EXAMPLES:
✅ Clean separation: frontend/ backend/ db/
❌ Everything in one file

Phase 2: Agent mode in Cursor (30 min)

Procedure:

1. Load concept file into Cursor
2. Transfer .cursorrules via @
3. Composer: "Read concept.md and .cursorrules. Build the feature completely."
4. Activate Agent Mode
5. OBSERVE what happens:
   - In what order does the agent build?
   - Does it test itself?
   - Does it adhere to .cursorrules?
6. Git checkpoints: After each part (backend finished → commit)
7. Intervene in case of safeguard violation

What to look out for:

- Agent wants to deploy directly → STOP, test locally first
- Agent wants to delete files → Ask why
- Agent makes too many changes at once → Checkpoint missing
- Agent commits without you → Enforce rule

RSLT.DIGITAL GmbH
Ulmer Str. 40
73728 Esslingen am Neckar
Geschäftsführer: Tim Resl

Sitz der Gesellschaft
Esslingen am Neckar
HRB 759452, Amt. Stuttgart
St.-Nr. 59340/73888
USt-IdNr. DE 310112889

Kontakt
Telefon +49 711 65 68 87 - 10
hi@rslt.digital
www.rslt.digital

Bankverbindung
Volksbank Plochingen eG
RSLT.DIGITAL GmbH
IBAN DE46 6119 1310 0839 0660 07
BIC GENODES1VBP

Phase 3: Code review (10 min)

Questions for review:

- Do you understand what the agent has built?
- Can you explain the purpose of each file?
- Where would you do things differently?
- What was done well?
- Is there any code that looks "hacky"?

Action: If you don't understand the code: Ask the cursor, "Explain to me why you built [part] this way."

Success criteria

You have understood the exercise if:

- The dashboard works (backend + frontend + DB)
- You had to intervene at least once (safeguard)
- You have 3+ Git commits (checkpoints)
- You understand what happens in each file
- You can explain why the agent proceeded in this way

RSLT.DIGITAL

After the exercises: Export your framework

You have completed the 3 exercises. Now: consolidate your knowledge.

Task: Create your personal cheat sheet

Cursor prompt:

Create a personal vibe coding cheat sheet for me in /framework_export.md:

- *My 6-element prompt template*
- *Top 3 learnings from the exercises*
- *.cursorrules template (my most important rules)*
- *Workflow notes (What works for me?)*

Format it so that I can print it out and place it next to my monitor.

What should be included in your cheat sheet?

6-element template
Your personal template for prompts

Loop detection
After how many attempts do you escalate?

Git workflow
How often do you commit? How do you describe commits?

Safeguards
What should Agent never do without asking?

Tools
ChatGPT for what? Cursor for what?

Learnings
What was surprising? What works well?

Peer learning: Share your cheat sheet

If you work in a team:

- Show each team member one learning from your exercises
- Discuss: Where do you have different approaches?
- Create common team cursorrules

Different approaches are OK—there is no "one" right way.

Summary: Workshop flow

Complete workshop schedule (2-3 hours):

Exercise 1 (45 min): From concept to code
→ Build a contact form
→ Apply the 6-element framework
→ ChatGPT/Claude → Cursor workflow

Exercise 2 (40 min): Debugging & escalation
→ Fix 3 bugs
→ Identify loop (bug 2)
→ Escalate to ChatGPT/Claude
→ Accept unexpected solution (bug 3)

Exercise 3 (50 min): Multi-file with safety
→ User dashboard (backend/frontend/DB)
→ Agent mode under control
→ Enforce safeguards
→ Git checkpoints

Conclusion (15 min): Export framework
→ Create personal cheat sheet
→ Document learnings
→ Share with team

🎯 **After this workshop:** You will not only have learned the theory, but also experienced how vibe coding works in practice. You will know your own patterns, know when to escalate, and have a framework that works for you.

That's the difference between "I've read about it" and "I can really do it."

PART VIII: RESOURCES & TEMPLATES

Your vibe coding toolkit

Everything you need to get started right away.

In this chapter: Copyable templates, tool setup, .cursorrules examples, success metrics, and further resources.

Use this chapter as a reference—not to read through, but to pick out what you need right now.

1. The recommended tool stack

Core tools (essential):

ChatGPT/Claude
Concept phase, strategy, "second opinion"
ChatGPT: Quick concepts, creative approaches
Claude: In-depth analysis, code review, long contexts

Cursor
Build phase, main work tool
Composer + Agent Mode for multi-file edits
Git integration, context search, model switching

Git
Version control, checkpoints, undo button
Branches for experiments, detailed commits

Model hierarchy in cursor:

Model	When to use?	Strengths
Claude Sonnet 4.5	Standard (80%)	Fast, reliable, good understanding of code
GPT-5	Escalation (15%)	Complex logic, math, special algorithms
Opus 4.1	Deep reasoning (5%)	Architectural decisions, complex problems

Supplementary tools (optional but helpful):

- Notion: Project management, documentation, lead pipeline
- Loom: Screen recordings for handovers
- Docker: Deployment, environments
- Postman/Insomnia: API testing

2. Copy-paste templates

Template 1: 6-element prompt

ROLE: [Who should the AI be? Senior dev, architect, code reviewer?]

CONTEXT: [Where are you? What happened? Which project?]

INPUT: [What is available? Code, screenshot, error, concept?]

OUTPUT: [What should the result be? Code, documentation, explanation, file?]

ACTION: [What should the AI do? Build, analyze, fix, explain?]

EXAMPLES:
✅ *How it should be: [Positive examples]*
❌ *How it should NOT be: [Negative examples]*

Template 2: ACTUAL/TARGET prompt (cursor)

ACTUAL: [What is the current state? What is not working?]

SHOULD: [What should happen? What behavior do you expect?]

[Optional: ERROR log, screenshot, code snippet]

[Optional: CONTEXT - Stack, dependencies, special features]

Template 3: Escalation prompt

My developer (Cursor) is stuck on the following problem:

[Insert Cursor's explanation of the problem here]

Code context:
[Optional: Relevant code or export script output]

What Cursor has tried:
- *[Attempt 1]*
- *[Attempt 2]*
- *[Attempt 3]*

Give me a solution that I can give to my developer as instructions.

Template 4: Max Mode Security Review

Scan the entire project and check for:

1. Security issues:
- *Unprotected API endpoints*
- *Missing input validation*
- *SQL injection risks*
- *XSS vulnerabilities*
- *Credentials in code*

2. Data protection issues:
- *Sensitive data in logs*
- *Lack of encryption*
- *GDPR violations*

3. Code quality:
- *Code duplicates*
- *Dead code paths*
- *Performance bottlenecks*
- *Missing error handling*

Give me a prioritized list: Critical → High → Medium → Low

3. .cursorrules templates

Basic template (for every project):

 # Project rules for [project name]

Safety First
- NEVER DELETE without explicit request
- NEVER GIT PUSH without confirmation
- ALWAYS test locally first, then deploy

Code Style
- Language: [German/English] for variables & comments
- Comments: Only when necessary, code should be self-explanatory
- Naming: camelCase for JS/TS, snake_case for Python

Testing
- Every new function: Test with edge cases
- Before each commit: Run tests

Git workflow
- Commits: Detailed in English
- Format: "type: description" (feat, fix, docs, refactor)
- Branches: feature/xyz, bugfix/xyz, hotfix/xyz

Documentation
- Keep README up to date
- Document API changes
- Complex logic: inline comments

Advanced: Privacy-first project

 # Privacy rules (in addition to basic rules)

Data handling
- NO sensitive data in logs
- NO credentials in code (only environment variables)
- All user data: Store encrypted

GDPR compliance
- User consent before data collection
- Implement delete function for user data
- Provide data export function

Testing with sensitive data
- ONLY mock data in tests
- Real user data: Anonymize locally

Advanced: Team project

 # Team Rules (in addition to Basic Rules)

Code review
- No direct push to main
- Pull requests: At least 1 reviewer
- PR description: What, why, test instructions

Communication
- Breaking changes: Slack message to team
- Major refactors: Discuss beforehand
- New dependencies: Discuss first

Onboarding
- Keep README up to date for new developers
- Provide setup script
- Maintain architecture diagram

4. Success metrics: Are you on the right track?

How can you tell that vibe coding is working for you?

Technical metrics:

Features per week
Establish a baseline: How many features are you currently able to complete?
Goal: 2-3x more than before (after 1-2 months of practice)

Code quality
Can you understand the code that AI writes?
Goal: 90%+ of the code is comprehensible, 0% "magic"

Bug rate
How many bugs per feature?
Goal: Equal to or lower than with manual coding

Refactoring requirement
How often do you have to rewrite AI code later?
Target: <20% of the code needs refactoring later

Process metrics:

Loop frequency
How often does the cursor get stuck in loops?
Target: <1 loop per day (after 2 months)

Escalation rate
How often do you escalate to ChatGPT/Claude?
Target: 10-20% of tasks (more is OK for complex problems)

Time to first code
From prompt to first working code?
Target: <15 min for simple features, <60 min for complex ones

Git commit frequency
How often do you commit?
Goal: Every 5-15 min (checkpoints work)

Subjective metrics (important!):

Do you understand your code?
Can you explain what each file does?
Red flag: "No idea, AI built it."

Do you feel productive?
Or do you feel like a copy-paste bot?
Goal: You orchestrate, AI executes

Are you still learning?
Do you see new patterns, approaches?
Red flag: "I only do what AI tells me to do."

Can you do without AI?
If the cursor fails, can you continue working?
Goal: Yes, but more slowly

RSLT.DIGITAL GmbH
Ulmer Str. 40
73728 Esslingen am Neckar

Geschäftsführer: Tim Resl

Sitz der Gesellschaft
Esslingen am Neckar
HRB 759452, Amt. Stuttgart

St.-Nr. 59340/73888
USt-IdNr. DE 310112889

Kontakt
Telefon +49 711 65 68 87 - 10

hi@rslt.digital
www.rslt.digital

Bankverbindung
Volksbank Plochingen eG
RSLT.DIGITAL GmbH

IBAN DE46 6119 1310 0839 0660 07
BIC GENODES1VBP

5. Red flags: When does it go wrong?

Warning signs that you're straying from the path:

🚩 **"I don't understand the code anymore."**
Problem: You leave Agent alone for too long without thinking
Solution: Smaller agent tasks, more checkpoints, understand every edit

🚩 **"I just copy what AI says"**
Problem: You have become passive
Solution: Invest more in the concept phase, challenge AI

🚩 **"Everything is getting slower and slower"**
Problem: Tech debt is piling up, code is getting worse
Solution: Max Mode reviews, more rejections for poor code

🚩 **"I can't do anything without AI anymore"**
Problem: Skills are atrophying
Solution: 1 day/week without agent mode, only manual work

🚩 **"Agent keeps doing the same thing wrong"**
Problem: .cursorrules not up to date/not used
Solution: Document learnings in .cursorrules

🚩 **"I am completely burned out"**
Problem: Too much parallel work, no breaks
Solution: Go back to 1-2 projects, take energy management seriously

6. Further resources

Official documentation:

- Cursor Docs: https://docs.cursor.com
- Claude API Docs: https://docs.anthropic.com
- OpenAI API Docs: https://platform.openai.com/docs

Community & Learning:

- Cursor Forum: Cursor Community
- r/ClaudeAI: Reddit community for Claude users
- AI Dev Tools Discord: Discussion of various tools

Recommended reading:

- Prompt Engineering Guide (Anthropic): Best practices for Claude
- Cursor Handbook: Unofficial guide by power users
- The Pragmatic Programmer: Classic, many principles still apply

7. Quick Start: Your first week

From reading to doing in 7 days:

Day 1: Setup
- Install Cursor & create an account
- Create ChatGPT/Claude Project
- Configure Git
- First test project: Hello World

Day 2: Practice 6 elements
- Write 10 prompts (without code)
- Role, context, input, output, action, examples
- Customize the template to suit your needs

Day 3: Exercise 1
Build a contact form (45 min)
Concept → Cursor → Iteration

Day 4: Exercise 2
Debug loop & escalation (40 min)
Recognizing loops, practicing escalation

Day 5: Exercise 3
Multi-file feature (50 min)
Agent mode under control

Day 6: First real project
Small real feature from your work
Using everything you have learned

Day 7: Review & Framework Export
What worked? What didn't?
Create a personal cheat sheet

RSLT.DIGITAL

Support, workshops, and lectures

I don't train developers—I provide support, give workshops, and hold lectures.

What I offer:

Support & mentoring (ongoing)

For: Teams that work productively with AI but need input

- Regular sparring over weeks/months
- Escalation support when you get stuck
- Workflow reviews: What works, what doesn't
- Architecture discussions on equal footing
- 2-4 hours/month as needed

Open workshops

For: Individuals or small groups

- Regular public workshop dates
- Hands-on: setup, first feature, best practices
- Small groups (max. 8-10 people)
- Dates at rslt.digital/workshops

Lectures & keynotes

For: Events, company meetings, conferences

- Practical: What really works
- 30-90 min, depending on format
- Topics: AI development, vibe coding, AI in e-commerce

Setup sprint (1 day)

For: Teams that want to get started right away

- Tool stack decisions for your team
- .cursorrules for your projects
- Define Git & safety workflows
- Implementing a real feature together

→ At the end: The system is up and running, and you can continue on your own

RSLT.DIGITAL GmbH
Ulmer Str. 40
73728 Esslingen am Neckar

Geschäftsführer: Tim Resl

Sitz der Gesellschaft
Esslingen am Neckar
HRB 759452, Amt. Stuttgart

St.-Nr. 59340/73888
USt-IdNr. DE 310112889

Kontakt
Telefon +49 711 65 68 87 - 10

hi@rslt.digital
www.rslt.digital

Bankverbindung
Volksbank Plochingen eG
RSLT.DIGITAL GmbH

IBAN DE46 6119 1310 0839 0660 07
BIC GENODES1VBP

Final words

Productive vibe coding doesn't replace skill. It multiplies it.

Architectural decisions, debugging, understanding your own code—that's still up to you. What's eliminated: hours spent on boilerplate, battles with syntax errors, manually typing everything.

AI is a tool. A powerful one. But you remain the developer.

PULSE gives you the structure for it. The loop runs, you set the impulses in the right places. Controlled development instead of blind trust.

A note on the models: Claude Sonnet 4.5, GPT-5, and Opus 4.1 are the most powerful as of November 2025. That will change. The principles in this framework remain the same—the tools adapt.

Manuel Fuß
Head of AI, RSLT.DIGITAL
November 2025

Contact & questions:

LinkedIn: Manuel Fuß
Website: RSLT.DIGITAL