ally

WELCOME!

Claude Code 102

More Superpowers Coming Your Way

Presented by **Cameron Burrows**, **Blaise Moses** & **Jeremy Proffitt**

You've already got the foundation — now we go deeper. This session is about the **mindset shifts** that separate casual Claude Code users from engineers who multiply their output. Async workflows, context mastery, planning with Claude, and the habits that make you feel like you have a whole team working for you. You're not starting from zero. You're **leveling up**.

"People rarely succeed unless they have fun in what they are doing." — Dale Carnegie

Session Goals

What we'll walk through together today

- **The 10x Engineer** — the move to asynchrony
- **MCP vs CLI** — the world around us, and its token cost
- **Observability & LLM-Friendly Logs** — Dynatrace + GUIDs
- **Skill Building** — boutique to world renowned
- **Demo:** Custom Agent + Release Skill — *live workflow*
- **Ally Marketplace** — internal plugin registry, install & use
- **Cron vs Monitoring** — filling the void, keeping cache warm
- **Prompts 102** — XML tags, precision, pointing at pixels

SECTION 1

The 10x Engineer

Async Workloads & The New Normal

Don't Ask for One Thing — Send a Journey

Ping-pong prompts waste ACUs and time. Batch the whole mission into one ask and walk away.

❌ THE JAB — one step at a time

```
> add a retry to the fetch call
(wait for Claude…)
> now add a test
(wait for Claude…)
> commit it
(wait for Claude…)
> push and open an MR
```

You're the bottleneck. You sit and wait between each turn, context repeats, ACUs multiply, and you ended up sitting there for 40 minutes.

✅ THE JOURNEY — one richly-briefed ask

```
> In src/api/client.ts, add exponential-backoff
  retry (3 attempts, jitter) to every fetch. Then:
  1. add a Vitest for the retry path
  2. run the test suite until green
  3. commit with an OC- ref from the branch
  4. push and open an MR with a Summary
     and a Test Plan. Ping me when it's up.
```

Fire. Walk away. Claude owns the whole arc — you come back to a reviewable MR.

The three ingredients of a good journey prompt:

1

Set the scene

Which file, repo, or system. What the bug/feature/goal is. What "done" looks like for a reviewer.

2

List the stops

Numbered steps: write, test, run, commit, push, MR. Give Claude the whole route so it doesn't stop at the first gas station.

3

Name the finish line

"Ping me when the MR is open," "tell me if the pipeline goes red," "stop and ask before deleting anything." Let Claude know where to hand back.

BUG-FIX JOURNEY

"Find the NPE reported in OC-842, add a failing test that reproduces it, fix it in the smallest possible diff, re-run the full suite, then open an MR against `develop`."

OBSERVABILITY JOURNEY

"Check Dynatrace for the last hour of `c3_logs` 5xx errors on payments-api, group by endpoint, write up the top 3 offenders with root-cause guesses, and drop it in the OC-842 ticket."

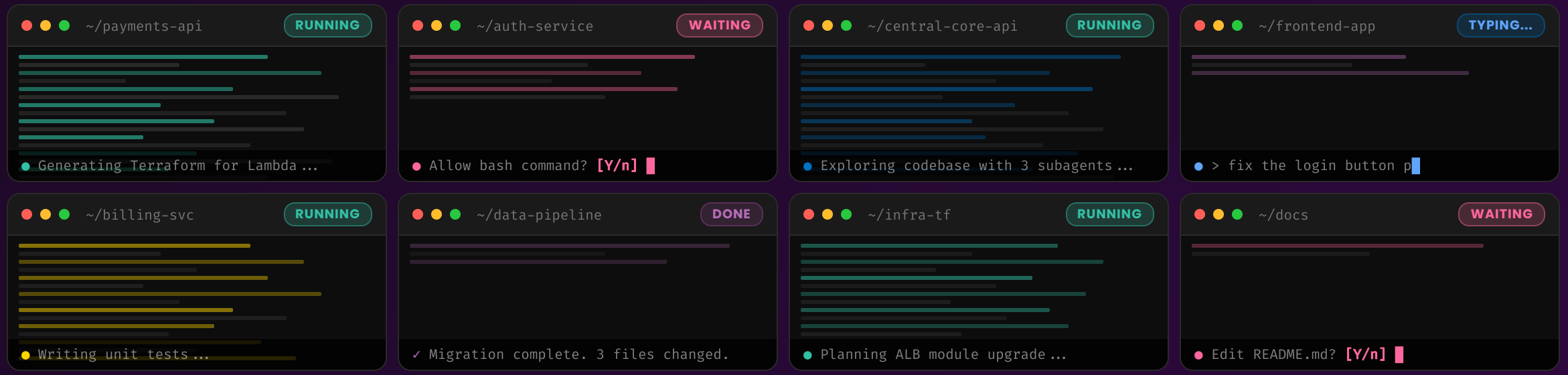
Rule of thumb: if your next three messages to Claude are all predictable, roll them into the first one. You'll finish faster, burn fewer ACUs, and unlock the 10x mindset coming up next.

What 10x Looks Like!

Fire and move on — don't wait for a response, that's the whole mindset

Email

Code Review



The 10x engineer doesn't wait for a response. They fire a session, move on immediately, and come back to results — like a chef with four burners going at once. Staring at the terminal is wasted time. While Claude works, *you work too*.

SECTION 2

MCP Servers

The World Around Us

What Is MCP?

Model Context Protocol — Claude's universal adapter to the rest of your world

Give Claude hands, not just a brain

An open standard from Anthropic that lets Claude securely reach **outside** the terminal — into Jira, GitLab, Dynatrace, Confluence, anywhere an MCP server exists.

Why it matters at Ally

- No more copy-paste tickets into terminal
- No more hunting dashboards for state
- No more SSH-ing to run one query
- Claude just *does it* — it has the tools

```
// The old way
1. Open Jira in browser
2. Find ticket OC-1234
3. Copy description
4. Paste into Claude
5. Ask Claude to build it

// With MCP
> Build OC-1234
Claude: fetching ticket ...
Claude: reading acceptance criteria ...
Claude: starting implementation ...
```

Mental model: MCP servers are apps that give Claude a new set of buttons. Install once, Claude uses them forever.

⚠ **TURN OFF MCPs YOU'RE NOT USING!**

Every enabled MCP ships its tool catalog into Claude's context *every turn*. Six idle servers can add 20–30k tokens per prompt — slower replies, higher ACU cost. Use `/mcp` to toggle servers off when not in use.

Ally's MCP Servers

Hosted and wired up — already installed, updates automatic

Atlassian
Jira, Confluence, ProForma, comments

GitLab
MRs, pipelines, issues, reviews

Dynatrace
DQL, problems, SLOs, traces, CVEs

CloudQuery
SQL over AWS & Ally Cloud inventory

Terraform Docs
Ally module registry, versions

Playwright
Browser automation, screenshots, UI

Yours Next
Build one. Publish. One-line install.

Where they live

`confluence.int.ally.com` — search **"Hosted MCP Servers"**. One-line install, shared config, zero maintenance on your side.

Try this today

`> show me my assigned Jira tickets` — Claude queries Atlassian MCP, returns the list. No browser tabs opened.

MCP vs CLI

Both give Claude superpowers — but one is cheaper to keep loaded

🌐 MCP Server

Structured tool definitions baked into Claude's context at every turn. Discoverable, typed, auto-completing.

- ✓ Self-describing — Claude sees every tool & schema
- ✓ Structured JSON output, easy to reason about
- ✓ Great for rich APIs (Jira, Confluence, Dynatrace)
- ✗ Every tool definition costs tokens *on every turn*
- ✗ A chatty server (50+ tools) can eat 10k+ tokens before you type

💻 CLI via Bash

Claude shells out to `glab`, `dyna`, `aws`, `gh`, `acli`, etc. — zero tool definitions in context.

- ✓ **Near-zero context cost** — just the Bash tool
- ✓ Use any binary: `kubectl`, `terraform`, `jq`, `git`
- ✓ Output is targeted — only what you piped through
- ✗ Claude has to remember flags (or read `--help`)
- ✗ Parsing unstructured text is error-prone without `-o json`

💰 **TOKEN MATH**

Loading 6 MCP servers with 30 tools each ≈ **30,000+ input tokens every turn** just for the menu. A CLI like `glab` or `dyna` adds **zero**. Pick MCP when you need rich structured output — reach for CLI when you just need to run a command.

🎯 Use MCP when…

…you need typed fields (Jira custom fields, Dynatrace DQL verify), interactive browsing (Playwright), or you're asking Claude to reason about rich structured data.

⚡ Use CLI when…

…you know the command, the output is small, and you care about speed or token cost. `glab ci view`, `dyna dql run`, `aws s3 ls` — all faster than their MCP equivalents.

SECTION 3

Observability

Making Dynatrace MCP Work for You

Dynatrace Superpowers

Stop hunting through dashboards — ask Claude in plain English

🚨 Incidents

- `list_problems`
- `investigate_problem`
- `get_incident_brief`

🔎 Query Anything

- `execute_dql`
- `query_metrics`
- `query_spans / logs`

🔍 Find Things

- `find_entity`
- `find_by_tag`
- `list_buckets`

🛡️ Security

- `list_vulnerabilities`
- `get_vulnerability_details`
- `get_security_overview`

🎯 SLOs & Health

- `list_slos`
- `get_slo_status`
- `get_service_health`

📂 Dashboards

- `list_documents`
- `create_document`
- `add_dql_to_notebook`

Prompt it, don't click it: `"what problems hit payments-api in the last hour, and what does the log pattern look like?"` — Claude picks the right tools in the right order.

DQL in the Real World

Claude learned the rules so you don't have to

The rules that save your budget

1. Bucket-first. Every `fetch logs` filters `dt.system.bucket` first — drops query cost from 186 GB to 3 GB.

2. Verify before execute. `verify_dql` is free — catches syntax errors without burning budget.

3. Probe with `maxResults: 5` before running the full query — confirms the data shape is what you expect.

4. Filter before summarize. Cut the data down first, then aggregate — faster and cheaper.

A real investigation flow

```
> why is c3-api slow this morning?

Claude runs:
1. find_entity("c3-api")
2. get_service_health (p99 spike!)
3. list_problems (24h window)
4. execute_dql {
fetch logs
| filter bucket="c3_logs"
| filter loglevel=="ERROR"
| summarize count() by pattern
}

✓ Found: downstream timeout to
salesforce-api causing retries.
```

Takeaway: one sentence in — a root cause out. Claude orchestrates the tools.

LLM-Friendly Logging

Write logs AI can read — and shrink the minutes between bug report and fix

🔍 The loop that changes everything

1. Error fires in the UI → surfaces a **GUID / error ID** in a toast.
2. User pastes the GUID into Claude.
3. Claude greps the logs, finds the matching record, and sees the *function, file, and state* that blew up.
4. Claude proposes the fix — often before you finish your coffee.

IN THE UI

```
⚠️ Something went wrong.
Error ID: 404-RT-9f3a
(click to copy, paste into Claude)
```

📝 What makes a log AI-friendly

- **Unique error ID** — stable across UI, server, logs
- **Function + file breadcrumb** — Claude knows where to open
- **Structured JSON** — greppable, filterable, parseable
- **State snapshot** — inputs, expected, actual

```
# Ugly log — Claude has to guess
ERROR: user not found

# LLM-friendly log
[ERROR]
  id: "404-RT-9f3a"
  function: get_user_data
  file: @api/users.py:142
  user_id: 42
  msg: "user lookup miss"
```

The magic prompt: *"Fix error* `404-RT-9f3a` *in* `@api/users.py`*."* — Claude opens the file, reads the function named in the log, sees the state, and ships the patch. No back-and-forth, no hunting.

SECTION 4

Skill Building

Boutique to World Renowned

What's a Skill?

A markdown file that becomes a slash command — reusable, shareable, unstoppable

The anatomy

A skill is just a markdown file with a YAML frontmatter `description`. Drop it in `~/.claude/skills/` and Claude auto-loads it. Type `/skill-name` to invoke.

```
~/.claude/skills/ship-it.md

---
description: Run tests, push, watch pipeline
---

# Ship It
1. Run the test suite.
2. If green, commit and push.
3. Watch pipeline until pass.
4. Slack the MR link.
```

Already at your fingertips

Some skills installed on your machine today:

`/review-mr` `/pushit` `/investigate` `/build-it` `/audit` `/send-outlook-email` `/whats-next` `/security-review`

💡 Pro move

Ask Claude: *"make this a skill"* after a good workflow. It writes the file, places it correctly, and it's live on the next session.

Boutique → World Renowned

Four stages of skill maturity — your personal hack becomes everyone's superpower

STAGE 1

Boutique

`~/.claude/skills/` — *just you*

- Custom MR review rules
- App-specific observability probes
- "Is my critical path working?"

STAGE 2

Team

`.claude/skills/` in repo

Everyone on the project gets it.

STAGE 3

Ally-wide

Ally plugin marketplace

Any engineer at Ally installs it.

STAGE 4

World Renowned

Anthropic public marketplace

The whole industry uses it.

The secret: most world-renowned skills started as boutique scripts someone wrote for themselves. Build it for YOU first — polish and ship later. Every stage just promotes the same markdown file.

Demo: Confluence Agent + Release Skill

Building a custom agent that uses CLI tools, then invoking a real-world release workflow

Part 1: Build the Agent

- Create `confluence-agent.md` in `~/.claude/agents/`
- Define tools: `Bash` , `Read` , `Write` , `Edit`
- Teach it to use the Confluence REST API via `curl`
- Add known pages, auth config, error handling

Part 2: Run the Release Skill

- Invoke `/release-candidate` on the gen-ai repo
- Watch it gather MRs, map Jira tickets, create the RC branch
- Confluence page generated automatically
- Feature flags analyzed and documented

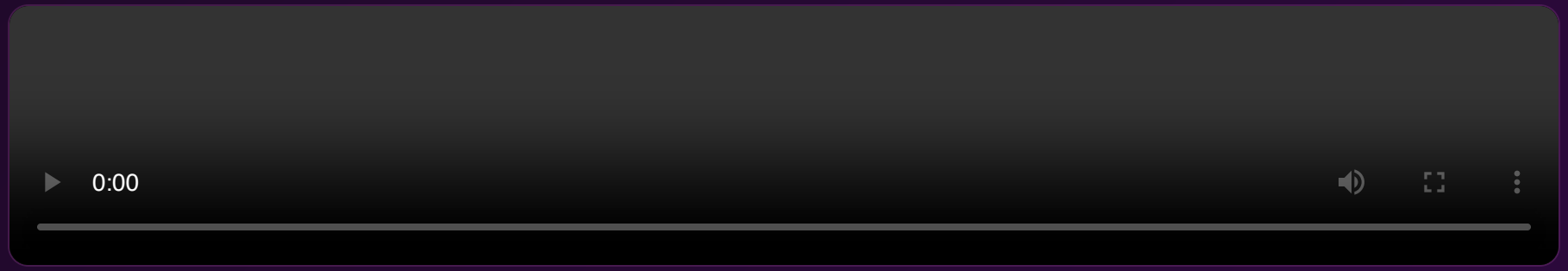
What to watch for

- Agent **delegates** to sub-agents (jira-agent, confluence-agent)
- CLI tools (`glab` , `jira-cli`) used as the integration layer
- Multi-step orchestration from a **single slash command**
- Real data: actual MRs, actual Jira tickets, actual Confluence output

Key insight: The agent file is just markdown instructions. The skill file is just markdown steps. No SDK, no framework — just clear instructions + tool access.

LIVE DEMO

Custom Agent + Release Workflow



Demo: Building the Confluence agent & running /release-candidate

The Ally Marketplace

Think npm for Claude Code skills — an internal registry of vetted, installable plugins backed by a GitLab repo

~/.claude/settings.json

```
"extraKnownMarketplaces": {
  "ally-claude-plugins-official": {
    "source": {
      "source": "git",
      "url": "https://gitlab.com/ ... /ally-claude-plugins-official.git",
      "ref": "develop"
    }
  }
}
```

1 Register
Add the marketplace config to your settings

2 Browse
Type `/install` to see available plugins

3 Enable
Select a plugin — skills, agents, config all install

4 Use
Invoke with `/plugin:skill` immediately

What a plugin installs

- Custom slash commands
- Specialized agents
- Documentation + patterns
- Auto-updates from source

Why an internal marketplace?

- **Discoverability** — no digging through repos or Slack history
- **Consistency** — same scaffold pattern across every team
- **Velocity** — one command, working immediately
- **Governance** — CoE-approved, security-reviewed
- **Evergreen** — plugin updates when the source repo updates

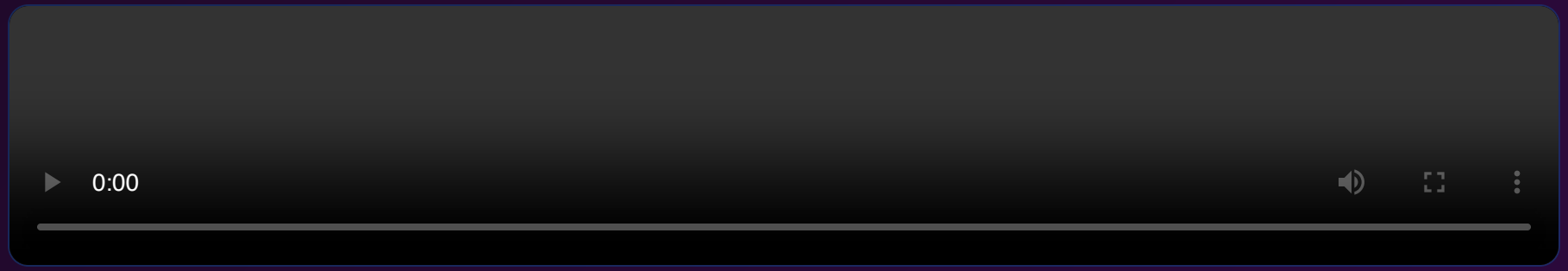
In the demo you'll see

- Adding the Ally marketplace from scratch
- Installing the **AllyAI** plugin (LangGraph toolkit)
- Running `/allyai:scaffold` to generate a new agent

Stage 3 in action: This is the maturity model happening in real time. One engineer builds it, publishes to the marketplace, and now *every* Claude Code user at Ally has it.

LIVE DEMO

Ally Marketplace Plugins



Demo: Registering the Ally marketplace, browsing plugins, and installing the AllyAI toolkit

SECTION 5

Claude's Cron

To Fill the Void

Fill the Void

Schedule Claude to work while you're away — coffee, meetings, sleep

Claude's built-in scheduler

`CronCreate`

Session-local cron that fires a prompt when Claude is idle. Standard 5-field cron in your local time zone — no timezone math, no external job runner, no setup.

```
# Every 5 minutes, idle-triggered
cron: "*/5 * * * *"
prompt: "check the MR pipeline"

# Weekdays 8:47 AM
cron: "47 8 * * 1-5"
prompt: "/whats-next"
```

The async mindset

Old you: stare at the terminal, refresh, wait, context-switch, refresh again.

10x you: "Claude, every 7 minutes check if the pipeline passed. Ping me when it's green or broken." Walk away. *Get coffee.*

Notification flow: pair scheduling with `PushNotification` so Claude taps you on the shoulder — phone or desktop — only when it actually matters.

Real Scheduling Patterns

Steal these. Adapt them. Ship them to your team.

🔍 Pipeline watcher

```
cron: "*/7 * * * *"
prompt: "watch the MR pipeline"
```

Fires every 7 minutes while this session is alive. Claude checks status, pings you when it fails or passes, and auto-diagnoses broken jobs on the spot.

☕ Morning briefing

```
cron: "47 8 * * 1-5"
prompt: "/whats-next"
```

Weekdays at 8:47 AM, Claude scans tickets, recent commits, and open MRs. Delivers your "what to work on" before your second sip of coffee.

Timing tip: pick *off-minute* values like `8:47` or `:57` instead of `:00` / `:30`. When everyone's crons land on the same minute, you're fighting the whole company for API capacity.

Cron vs Monitoring

Two tools, two shapes of time — pick the right one for the job

⏰ `CronCreate` — clock-driven

Fires a prompt **on a schedule** when Claude is idle. Great for things the *calendar* decides.

- ✓ Morning briefing at 8:47 weekdays
- ✓ Hourly summary of open MRs
- ✓ End-of-day recap & memory updates
- ✓ Auto-expires after 7 days

```
cron: "47 8 * * 1-5"
prompt: "/whats-next"
```

👁️ `Monitor` — event-driven

Streams **stdout lines from a long-running script**. Each line becomes a notification. Great for things *the system* decides.

- ✓ Pipeline heartbeats & failure alerts
- ✓ `tail -f` a log for new errors
- ✓ Poll an API until state changes
- ✓ Exits when the watched thing ends

```
~/.claude/scripts/glab-ci-monitor.sh
services develop
```

PRO TIP — KEEP YOUR KV CACHE WARM

🔥 Anthropic's prompt cache has a **5-minute TTL**. Set Monitor heartbeats **no more than 4 minutes apart** — the cache stays warm, each reply is faster, and you pay the discounted cache-hit rate instead of re-reading the whole conversation. Go beyond 5 minutes and you burn the full context again, every time.

Quick rule: is the trigger a *time*? Use Cron. Is it an *event in a stream*? Use Monitor. Do both if the job is "every 4 minutes, check if the deploy log shows errors."

SECTION 6

Prompts 102

More Ingenious Precision

Be a Briefing Officer

You're not asking Google — you're briefing a smart colleague who just walked in

❌ Google-style

```
fix the auth bug
```

Which file? Which bug? What did you try? What's the expected behavior? Claude has to guess — and guesses are expensive.

✅ Briefing-style

```
Login silently fails for SSO users when
their session expires mid-request.
Repro: auth.service.ts:142, look at the
refreshToken branch. I've tried adding
try/catch but the 401 still bubbles up.
Want: retry with refresh, then redirect
to /login on failure. Test with the
existing mock in auth.spec.ts.
```

Claude knows *where*, *what*, what's been tried, the success criterion, and where to test. One shot.

The pattern: *here's what's broken — here's where — here's what I tried — here's what success looks like — here's where to verify.* A good brief saves three follow-ups.

Structured Prompts with XML Tags

Industry-standard scaffolding that makes Claude address every requirement

The building blocks

- `<role>` — persona: *"Senior SRE," "Staff Sec Eng"*
- `<instructions>` — the task list Claude must complete
- `<context>` — versions, goals, prior decisions
- `<code_to_review>` — or `<data>`, the payload
- `<constraints>` — hard "do not" rules
- `<thought>` — chain-of-thought reasoning
- `<examples>` — few-shot good vs. bad pairs

Multi-layer review — one shot

```
<role>Senior SRE</role>
<instructions>
  <security> SQLi, XSS </security>
  <perf> O(n²), leaks </perf>
  <style> PEP8 </style>
</instructions>
<constraints>no new deps; thread-safe</constraints>
<code_to_review>@api/users.py</code_to_review>
```

💰 Token-efficiency moves

- **@filename** beats pasting file content
- **Markdown > raw HTML** — HTML is token-heavy
- **Short IDs** (`OC-842`) > descriptive strings
- **File paths**, not `localhost` URLs

Why tags work: they give Claude unambiguous boundaries between rules, data, and output. Nested tags force the model to address each layer independently instead of surface-skimming.

The Precision Pattern

Four slots. Fill them. Ship it.

1. Context

What's the situation? Which files, which users, which constraint? Set the stage in one line.

2. Task

What do you actually want done? Use verbs: *refactor, add, diagnose, summarize, generate.*

3. Constraints

What's off-limits? "Don't touch the DB schema." "Keep the public API stable." "No new deps."

4. Output

How should the answer land? *Diff, plan, bullet list, commit message, MR description.*

🔃 Iterate to sharpness

- *"Make it half the length."*
- *"Show me another way."*
- *"Argue the opposite case."*
- *"Explain it to a junior."*

✏ The meta move

Ask: *"before you answer, what questions would a senior engineer ask about this?"* — Claude surfaces the ambiguity you missed, you clarify, the answer is surgical the first time.

Point at the UI

Precision when the target is a component, a div, or a single pixel

Six ways to name a component

1. **By text:** *"the 'Submit Order' button"*
2. **By selector:** `.card-purple` , `[data-testid=save]`
3. **By role / ARIA:** *"first primary button in the form"*
4. **By file + line:** `Button.tsx:42` — opens to that line
5. **By component name:** `<UserCard>` , `<CheckoutForm>`
6. **By position:** *"third card in row 2", "top-right nav"*

Vague → surgical

❌ **VAGUE**

"fix the save button styling"

✅ **SURGICAL**

"In `Button.tsx:42` , the primary variant uses `bg-blue-500` — change to `bg-purple-600` and bump padding from `py-2` to `py-3` . Keep the hover state."

📷 **The hidden superpower:** screenshot → circle the problem → paste it. Claude sees what you see. Playwright MCP automates the capture so you can describe pixels with *pixels*, not English.

✏ **Pro move:** ask *"what selector uniquely identifies this?"* — Claude suggests the most stable one (data-testid beats nth-child).