Sync Agents 102

Commands, Skills & Tools

What We'll Cover

1. **Foundation:** Testing infrastructure, AGENTS.md, and file organization
2. **Planning & Context:** How to work effectively with agents
3. **Codification:** Custom commands, skills, and toolkits
4. **Tools & Verification:** External tools and advanced verification patterns

What You Learned in Sync 101

The fundamentals of synchronous agent workflows:

- Capturing meeting notes (Granola, Wispr Flow)
- Feeding notes to agents to extract context
- Asking clarifying questions to refine understanding
- Turning results into structured plan.md files

Today: We codify these patterns into reusable commands and skills.

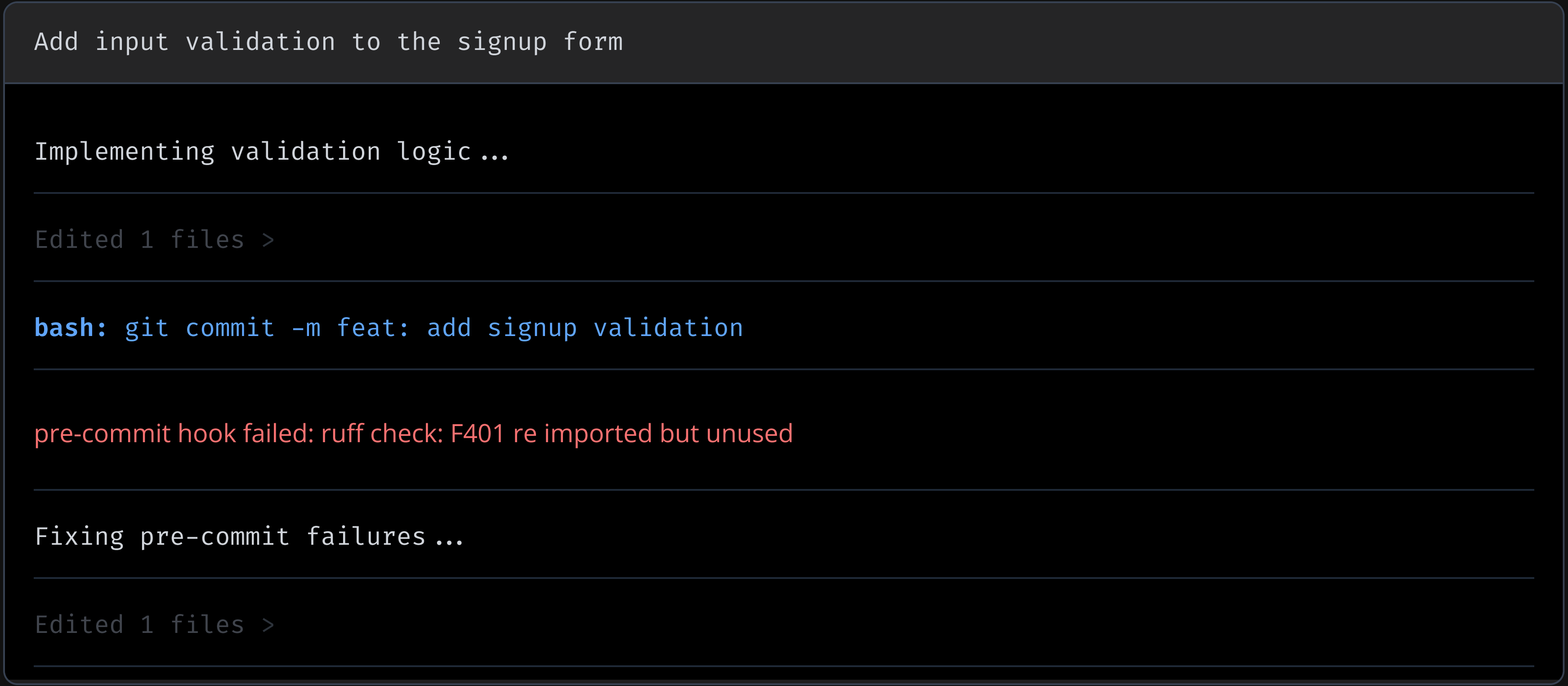
What stuck out most from 101? (one phrase is fine)

Part 1: Foundation

Infrastructure, AGENTS.md, and File Organization

Self-Correction Through Infrastructure

When agents can verify their work, they can run longer and more reliably.



```
Add input validation to the signup form

Implementing validation logic...

Edited 1 files >

bash: git commit -m feat: add signup validation

pre-commit hook failed: ruff check: F401 re imported but unused

Fixing pre-commit failures...

Edited 1 files >
```

Tools That Enable Self-Healing

- **Linting:** ESLint, Ruff, Prettier catch errors and style issues
- **Type systems:** TypeScript, mypy catch type errors
- **Formatters:** Consistent code style agents can follow
- **Tests:** Verify behavior matches expectations
- **Pre-commit hooks:** Block commits until checks pass

All the things engineers have complained about are now essential for agent self-healing.

Git Hooks: Pre-commit vs Pre-push

Two checkpoints for catching issues:

Hook	When	What to Run
Pre-commit	Before each commit	Fast checks: formatting, linting
Pre-push	Before pushing	Slower checks: type checking, fast tests

Agents can't push unless all checks pass. Humans can bypass with `--no-verify`.

Not sure how to get started? Ask your coding agent: "Audit what pre-commit setup can be done and install it for me" - it will set everything up.

Fast CI: Split Fast vs Slow Tests

Code is cheap now. Slow feedback loops kill velocity.

Split your tests:

Test Type	Duration	When to Run
Fast	< 2 min	Every push (pre-push hook)
Slow	> 2 min	PR creation, nightly, or manually

Rule: Pre-push hooks run fast tests only. Save slow tests for CI.

> Ask your agent: "Audit our tests and mark them as fast or slow, then set up pre-commit to run fast tests and CI to run slow tests"

If your test suite takes 40 minutes, that’s a separate problem to solve.

Test-Driven Development with Agents

Tests give agents a clear target to iterate against.

1. Write tests based on expected input/output pairs
   - Be explicit that you're doing TDD (avoid mock implementations)
2. Run tests and confirm they fail
   - Explicitly say not to write implementation code yet
3. Commit the tests when satisfied
4. Write code that passes the tests
   - Tell the agent not to modify the tests
   - Let it iterate until all tests pass
5. Commit the implementation

Why this works: Agents perform best with verifiable goals. Tests provide the feedback loop.

Audit Your Setup: Four Prompts

Run these prompts one at a time. Copy-paste into Cursor:

1. **Pre-commit hooks:**

   ```
   Check if we have pre-commit hooks. If not, set up pre-commit with ruff format and ruff check.
   ```

2. **Testing infrastructure/preferences:**

   ```
   Audit our testing infrastructure. What percentage of our codebase has test coverage? Which areas are untested? What
   ```

3. **Test speed:**

   ```
   Look at our CI pipeline. How long does it take? Which tests are slowest? How could we split fast vs slow tests?
   ```

4. **AGENTS.md:**

   ```
   Review our AGENTS.md file. What's missing? What patterns do we follow that aren't documented?
   ```

Browser Control for Visual Debugging

Cursor’s browser control enables visual debugging:

- Embedded browser for visual debugging and design iteration
- Screenshot capture to verify UI changes
- DOM interaction - click buttons, fill forms
- Console access for reading logs and errors
- Visual verification - compare before/after screenshots

Workflow: Code changes, agent opens browser, takes screenshot, verifies fix matches requirements.

AGENTS.md Files: Always-On Context

Configuration files that provide always-on context to AI agents.

- **Always-on context:** Rules loaded into every agent chat automatically
- **Minimal by design:** Start with nothing, add only when needed
- **Living documentation:** Updated when agents make mistakes
- **Tool-agnostic:** Most tools support `agents.md` or `AGENTS.md`

Start Minimal: Add Rules When Needed

Don’t create AGENTS.md upfront. Add instructions only when needed.

When to add rules:

- Model fails 2-3 times on the same issue
- Team identifies repeated patterns that need standardization
- Tool preferences need documenting (pnpm vs npm, etc.)

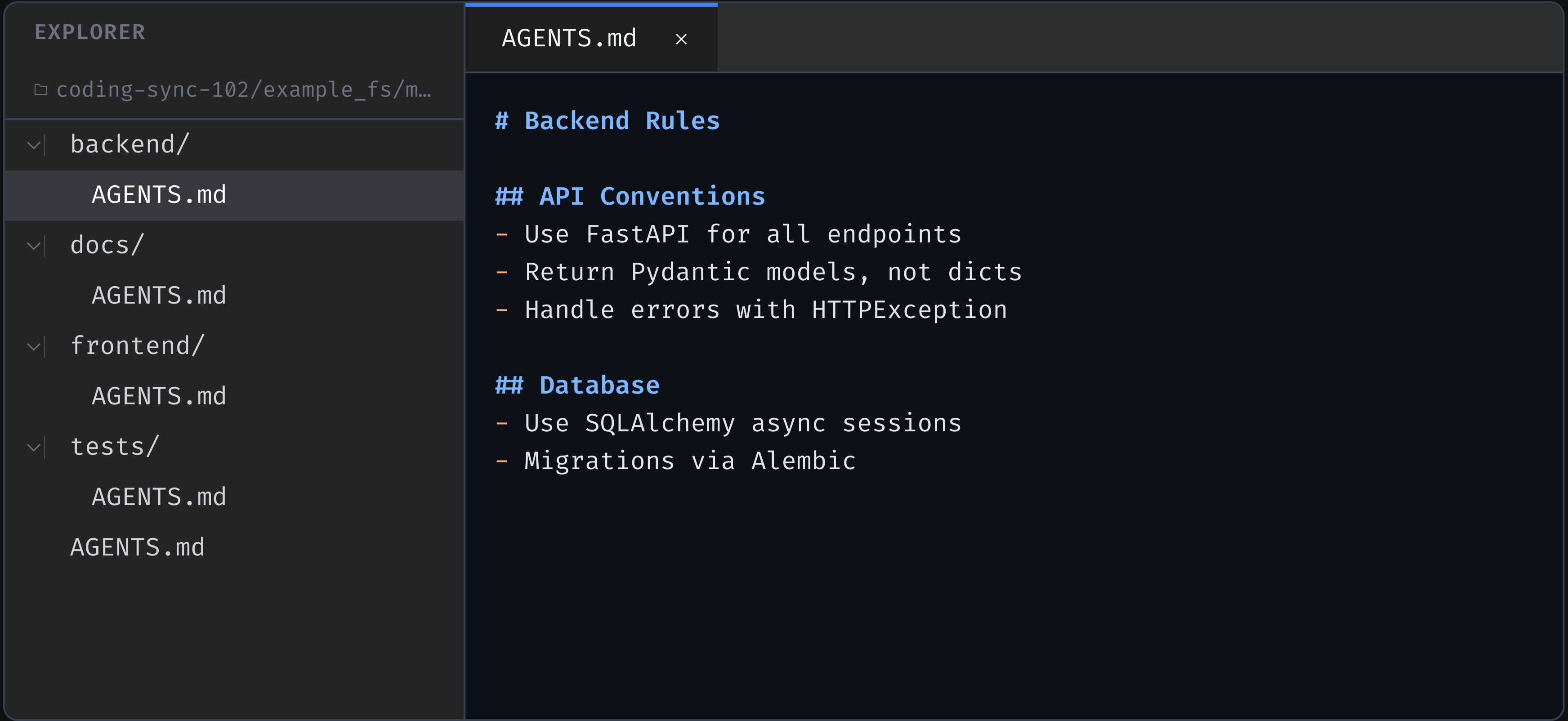
What to document:

- Tool preferences (pnpm vs npm, uv vs pip)
- Test behavior (watch vs non-watch mode)
- Style guidelines specific to your codebase
- Common patterns the model gets wrong

Tell it why, not just what: "Use TypeScript strict mode because we’ve had production bugs" is better than just "Use TypeScript strict mode."

Directory-Specific Rules for Monorepos

For monorepos, use directory-specific files. Agents check the current directory first, then parents up to root.



```
backend/
  AGENTS.md
docs/
  AGENTS.md
frontend/
  AGENTS.md
tests/
  AGENTS.md
AGENTS.md
```

```
# Backend Rules

## API Conventions
- Use FastAPI for all endpoints
- Return Pydantic models, not dicts
- Handle errors with HTTPException

## Database
- Use SQLAlchemy async sessions
- Migrations via Alembic
```

Are you in a monorepo? If yes, what’s the hardest area to work in?

Update AGENTS.md During PR Reviews

Critical Practice: Update AGENTS.md during code reviews.

1. Review PR and notice agent made incorrect choice
2. Fix the code in the PR
3. `@AGENTS.md` and tell the agent to add a rule preventing the mistake

This is "Compounding Engineering" - each PR review teaches the agent, and those lessons compound over time.

The AI Champion Role

Your team needs an AI champion. This is going to be a lot of information - someone needs to own it.

The AI champion is responsible for:

- Setting up repos with AGENTS.md, pre-commit hooks, and infrastructure
- Creating new commands and managing the command library
- Ensuring consistency and maintaining quality standards
- Codifying repeated patterns into reusable commands

If that role doesn’t exist, push your leadership to define it. As the team gets more skills, everyone can contribute - but someone needs to drive it.

Who usually catches quality issues on PRs? (everyone / TLs / a few reviewers / unclear)

File Organization for AI Navigation

Pro tip: Create two side-by-side directories:

```
repo/
  modular/              # Services in subdirectories (easier for AI)
    services/
      auth/
        service.ts
        types.ts
      payment/
        service.ts
        types.ts

  non-modular/          # Different organization
    auth-service.ts
    payment-service.ts
    auth-types.ts
    payment-types.ts
```

Code duplication is more acceptable when AI manages the overhead.

Part 2: Planning & Context

How to Work Effectively with Agents

Plan Mode Basics

Press Shift+Tab to toggle Plan Mode. Instead of immediately writing code, the agent will:

1. **Research** your codebase to find relevant files
2. **Ask clarifying questions** about your requirements
3. **Create a detailed plan** with file paths and code references
4. **Wait for approval** before building

Plans are editable Markdown. Remove unnecessary steps, adjust the approach, or add context the agent missed.

Click "Save to workspace" to store plans in `.cursor/plans/`. This creates documentation, makes it easy to resume interrupted work, and provides context for future agents.

Not every task needs a plan. For quick changes or tasks you've done many times, jumping straight to the agent is fine.

When to start over: Sometimes fixing an in-progress agent is slower than starting fresh.

Planning Becomes Primary Work

As coding automates, planning becomes the bottleneck.

The recommended cycle:

1. **Synchronous planning:** Use IDE tools to understand codebase, ask questions, create specs
2. **Asynchronous coding:** Delegate implementation to agents
3. **Synchronous testing:** Test and refine agent’s changes
4. **Continuous cycle:** While one agent codes, you’re planning next task

File-Based Planning for Large Features

For larger features, use the file system:

```
Make plans in `.plans/{feature-name}/plan.md` with:
- Phases with clear feature branch names
- Files to reference
- Steps and decision log
- Components and interfaces

As you work, add notes to `.plans/{feature-name}/learnings.md` documenting:
- Progress and decisions
- Issues encountered
- Context to pick up where you left off
```

Plans can be 500-1000 lines. If implementation is wrong, delete code, iterate on plan, rebuild.

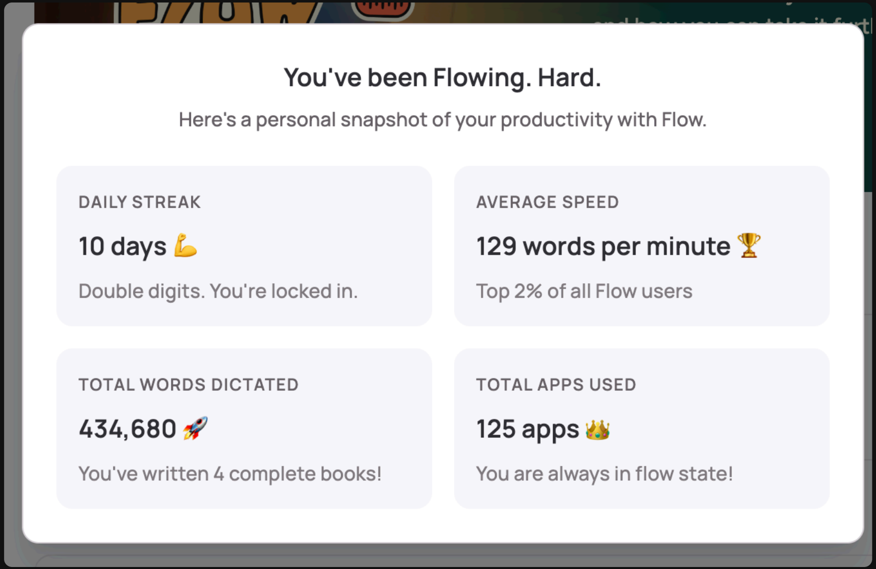
This snippet is in some of my AGENTS.md files. You can add `.plans/` *to* `.gitignore` *if you prefer to keep plans local - but you may need to update agent configs to read ignored files.*

Voice-Driven Planning: 3x Faster

Talking is 3x faster than typing and gives richer context.

Instead of typing terse prompts, just talk through your thinking in detail while exploring the codebase.

Use Wispr Flow: wisprflow.ai/r?JASON50 (free month for you, and me)



You've been Flowing. Hard.

Here's a personal snapshot of your productivity with Flow.

DAILY STREAK

10 days 💪

Double digits. You're locked in.

AVERAGE SPEED

129 words per minute 🏆

Top 2% of all Flow users

TOTAL WORDS DICTATED

434,680 🚀

You've written 4 complete books!

TOTAL APPS USED

125 apps 👑

You are always in flow state!

Prompt Evolution: Shorter + Visual

As models improve and you build trust, prompts get shorter.

Stage	Prompt Style
Early	Long, elaborate prompts explaining everything
Later	Short prompts with images - "fix padding", "redesign this"

The pattern: Show the model what’s wrong instead of explaining it.

Manage Context Windows

Context degrades before you hit the limit. At 20-40% usage, quality starts to drop.

Signs you need to clear:

- Agent keeps making the same mistake despite corrections
- Conversation has irrelevant context from earlier tasks
- Agent seems confused or contradictory

Don’t copy-paste context back in. Instead:

- Tell the agent: "Read the plan file and tell me what’s done and what’s next"
- Tell the agent: "Look at git history to understand the current state"

When to Start a New Conversation

One of the most common questions: continue or start fresh?

Start a new conversation when:

- You’re moving to a different task or feature
- The agent seems confused or keeps making the same mistakes
- You’ve finished one logical unit of work

Continue the conversation when:

- You’re iterating on the same feature
- The agent needs context from earlier in the discussion
- You’re debugging something it just built

> Long conversations cause agents to lose focus. After many turns and summarizations, context accumulates noise. If effectiveness is decreasing, start fresh.

Then load context fast using built-in features:

Load Context Through Questions

Ask agents to explain systems to you, rather than explaining to them.

Many coding agents prune content to avoid limits. Questions ensure the right context gets loaded.

After clearing context:

- Don’t dump knowledge back in
- Use git/plan files:
  - "Look at what’s staged"
  - "Recent commits on this branch"
  - "Read plan file - what’s done and next"

Verify understanding:

- Correct = right context loaded
- Wrong = catch misunderstanding early
- Wrong answers = gaps in AGENTS.md

Understanding Large Codebases

Use agents to understand before making changes:

- Ask for feature outlines to map system dependencies
- Add log statements at key logic points to follow execution
- Ask "What is this feature? What systems is it using?"
- Use new engineer onboarding questions - agents give immediate answers

Spend time understanding features before writing code.

Cross-Project References

Reuse solutions you've already solved elsewhere.

- "look at …/other-project and do the same for feature"
- "check how …/vibetunnel handles changelogs and implement that here"

Agent infers context from project structure and adapts patterns automatically.

This is a benefit of monorepos. If you don't have monorepos, you can mount additional workspaces in Cursor.

Architecture Diagrams for Code Review

Ask agents to visualize significant changes.

For complex changes, ask the agent to generate architecture diagrams:

```
Create a Mermaid diagram showing the data flow for our
authentication system, including OAuth providers, session
management, and token refresh.
```

Why this helps:

- Reveals architectural issues before code review
- Creates documentation as a byproduct
- Forces the agent to understand system relationships
- Easier to spot missing pieces or wrong dependencies

Good for: API changes, new services, refactors that touch multiple systems.

Part 3: Codification

Custom Commands, Skills & Tools

Custom Slash Commands

Commands represent codified experience.

Every time you give AI the same instructions, turn it into a command.

After three months, a well-built toolkit handles 90% of repetitive cleanup.

The goal: Never have the same conversation with AI twice.

Core Git Commands

Commands that automate Git-related tasks:

- `/de-slop` - Remove AI artifacts (redundant comments, mock-heavy tests)
- `/gh-commit` - Group changes into logical commits (feat, test, docs, refactor, chore)
- `/gh-fix-ci` - Download CI logs and fix failures automatically
- `/gh-review-pr` - Review PRs with structured feedback
- `/gh-address-pr-comments` - Address PR feedback systematically

Available at: github.com/jxnl/dots

Example: /gh-commit Command

```
# Commit

Create small, logical commits with conventional commit messages.

## Steps

1. Safety
   git branch --show-current
   - If on `main`/`master`: stop and create a branch

2. Inspect changes
   git status --porcelain
   git diff --stat

3. Batch changes (feat|fix, then test, then docs, then refactor/chore)
   - Keep commits atomic
   - Never `git add .`

4. Stage + commit each batch
   git add path/to/file1 path/to/file2
   git commit -m "type(scope): short description"
```

Async PR Review Workflow

Combining commands for async code review:

1. **Agent creates PR:** Tell your agent to create a PR at the end of its task
2. **Review remotely:** Get GitHub notifications on phone, review and leave comments
3. **Address feedback:** Run `/gh-address-pr-comments` - fetches line-by-line comments
4. **Fix CI:** Run `/gh-fix-ci` to download logs and fix failures

Review PRs remotely, address everything when you're back at your desk.

Team-Specific Commands

Beyond Git commands, create usage-specific commands for your team:

- **Deploy patterns:** Complex deployment workflows, multi-step processes
- **Testing patterns:** Specific test suites, environment setup
- **Domain-specific workflows:** Database migrations, API validation, infrastructure

Question: What workflows do you have that could become commands?

Think about repetitive tasks, repeated AI conversations, cleanup steps before shipping.

Case Study: PlanetScale’s Changelog

PlanetScale uses Cursor commands to automate changelog entries.

Their approach:

1. Complete a task with Cursor normally
2. At the end, ask Cursor to create a command for the process
3. Run the command a few times, tweak when needed
4. After iterations, the workflow becomes reliable

Prompt: "Create a new Cursor command for the process we just went through, call it /changelog"

From Commands to Skills

Commands codify single workflows. Skills package entire capabilities.

When a command grows complex or needs supporting resources, it becomes a skill.

The progression:

1. **Commands** - Simple, repeatable workflows
2. **Skills** - Structured capabilities with resources and scripts
3. **Toolkits** - Collections of related skills and commands

What Are Skills?

Skills are structured packages of instructions, resources, and scripts.

Unlike commands which are single markdown files, skills are folders containing:

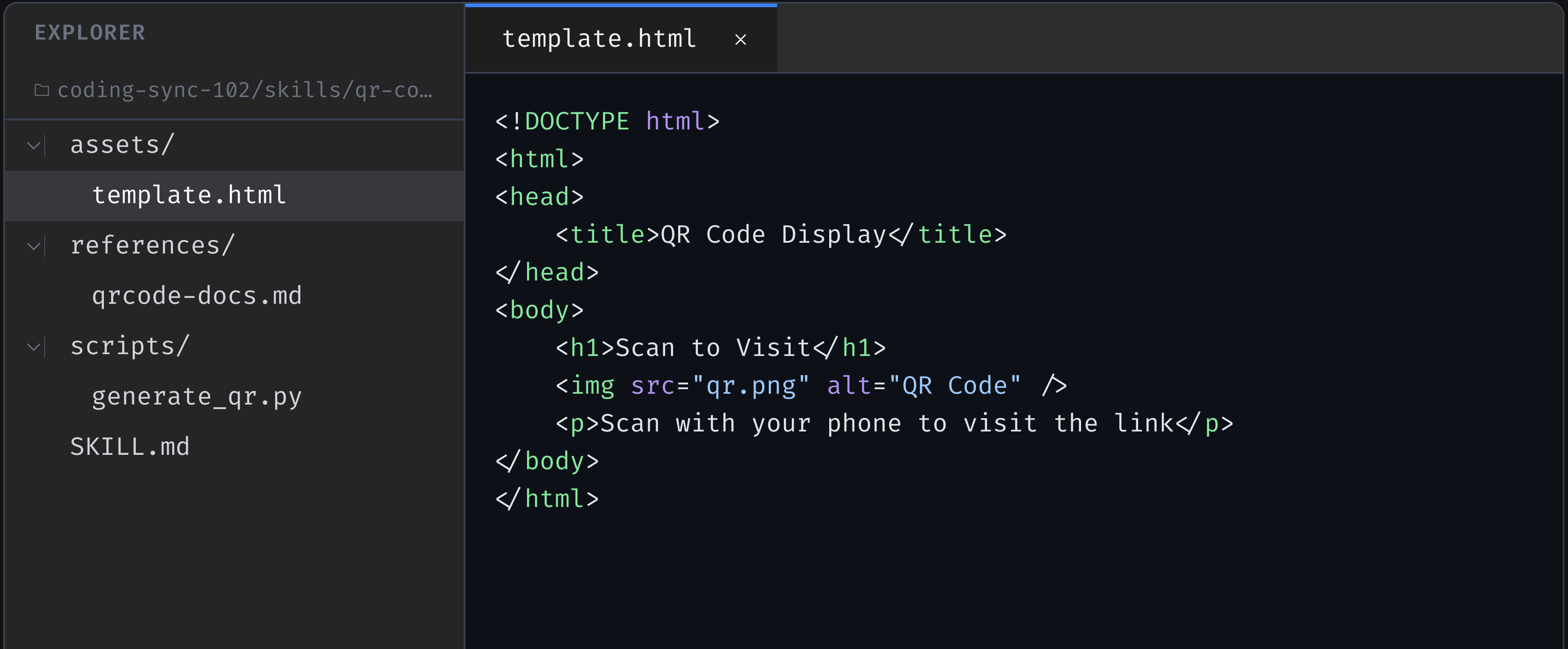
- **SKILL.md** - Main instructions and workflow
- **Scripts** - Executable helpers (Python, bash, etc.)
- **References** - Documentation, examples, templates
- **Assets** - Images, configs, or other files

Progressive disclosure: Skills load by name/description first, then activate fully when invoked.

Skills and Toolkits

Skills are like commands, but more structured.

A skill packages instructions, resources, and scripts. Skills use progressive disclosure - loaded by name/description, activated when needed.



```
coding-sync-102/skills/qr-co…
assets/
    template.html
references/
    qrcode-docs.md
scripts/
    generate_qr.py
SKILL.md
```

```html
<!DOCTYPE html>
<html>
<head>
    <title>QR Code Display</title>
</head>
<body>
    <h1>Scan to Visit</h1>
    <img src="qr.png" alt="QR Code" />
    <p>Scan with your phone to visit the link</p>
</body>
</html>
```

Skill Examples

Skills package your preferences, not just tool knowledge:

- **Slidev slide generation:** Create presentation slides using Slidev with your preferred layouts, components, and styling conventions
- **Deploy system:** Standardize deployment workflows, environment configs, and rollback procedures
- **Systems access:** Access systems via SOPs and MCP servers - tools to get A/B testing results, run analytics queries, or interact with internal APIs

The agent might know these tools, but skills capture how YOU want to use them.

Code You’d Never Write (Part 1)

Development tools for rapid iteration

Test data generation:

- "Use psql, look at the database, see how threads work, make some test threads"
- Agent examines schema, creates valid data, fixes its own mistakes
- Never worth the time to write manually, but agents do it in minutes

Throwaway utilities:

- UI state toggles to switch between component variations
- Debug panels, feature flag toggles, configuration editors

Admin backdoors:

- Temporary admin endpoints to test edge cases
- Bypass authentication for local testing

Code You’d Never Write (Part 2)

CLI tools agents can use programmatically

Analysis scripts as reusable tools:

- "Find all users who signed up in the last week but haven’t logged in"
- "Show me the distribution of API response times by endpoint"

Why CLI tools matter:

- Agents can invoke them with different parameters
- Reusable across multiple debugging sessions

Example workflow:

1. Agent creates `analyze-user-activity.py --days 7 --min-logins 0`
2. Uses it to verify feature impact
3. Reuses it later as a new tool in analytics skill

Scaffold Commands and Skills

The `jxnl/dots` repo includes commands to scaffold new commands and skills:

- `/new-cmd` - Create a new command with proper structure
- `/new-skill` - Create a new skill with SKILL.md template and folder structure

Use these to quickly bootstrap your own commands and skills.

When to Use Sub-Agents

Sub-agents are useful when you want to contain how much context you burn.

Use sub-agents for tasks that would bloat your main agent’s context:

- Log data analysis: identify slow functions, report findings
- Complex UI debugging: process verbose debug output
- Metrics analysis: process logs to identify bottlenecks
- Verbose test logs: extract failures and errors from noisy test output
- Codebase research: explore codebase to find answers or source of something

Sub-agent runs separately, summarizes findings, then hands concise results back to main agent.

Part 4: Tools & Verification

External Tools & Advanced Verification Patterns

MCP Servers

MCP servers connect AI agents to external tools.

The promise: eliminate context switching by letting agents read from Linear, Notion, Figma, etc.

The reality: for most workflows, copy-pasting into Cursor works fine.

What I’ve actually used:

- **Notion** - Read and write to Notion pages
- **Linear** - Create tickets, read issues (the must-have for ticketing)
- **Browsers** - Built-in browser control in Cursor
- **Context7 Docs** - Pull up-to-date library docs into the agent

For everything else: Build a CLI version and move it into a skill.

For finding more: smithery.ai has 3,400+ integrations.

Case Study: Agent-Driven Prompt Optimization

Building CLI tools agents can use to verify and iterate:

1. **Eval runner CLI** - Runs evals given a prompt
2. **SQLite storage** - Results write to local SQLite database

The agent becomes the optimization loop:

- Run eval against current prompt
- Query results from SQLite
- Analyze what’s working and failing
- Iterate on the prompt
- Run eval again to verify

Real API Testing Over Fixtures

Use real-time API calls in tests, not static JSON fixtures.

- Static fixtures become outdated
- Real API calls catch bugs mock data never exposes
- Agents can create, test, and tear down real resources

Example: Fly.io API client tests actually created servers, waited for boot, then tore them down.

More expensive in API costs, but catches actual integration issues.

Debug Mode for Tricky Bugs

When standard prompting struggles, Debug Mode provides a different approach.

Instead of guessing at fixes, Debug Mode:

1. **Generates hypotheses** about what could be wrong
2. **Instruments your code** with logging statements
3. **Asks you to reproduce** the bug while collecting runtime data
4. **Analyzes actual behavior** to pinpoint the root cause
5. **Makes targeted fixes** based on evidence

Best for:

- Bugs you can reproduce but can’t figure out
- Race conditions and timing issues
- Performance problems and memory leaks
- Regressions where something used to work

Key Takeaways

The Core Message

These capabilities exist for when you need them.

Don’t overthink it upfront. As you use AI more:

- You’ll make mistakes - add rules to AGENTS.md
- You’ll repeat yourself - turn it into a command
- You’ll see patterns - codify them into skills

The real benefits come from good tests and infrastructure.

Everything else builds on that foundation.

Key Traits of Successful Agent Users

Developers who get the most from agents share these traits:

1. **Write specific prompts**
   - "Add tests for auth.ts covering the logout edge case, using patterns in `__tests__/` "
   - Not: "add tests for auth"
2. **Iterate on setup**
   - Add rules only after repeated mistakes
   - Add commands only after figuring out workflows
3. **Review carefully**
   - AI code can look right while being subtly wrong
   - The faster agents work, the more important review becomes
4. **Provide verifiable goals**

Quick Wins to Start

Today:

- Check if your repo has pre-commit hooks
- How complete are your tests?

This Week:

- Start an AGENTS.md with one rule when model fails
- Create one custom command for something you repeat

This Month:

- Update AGENTS.md during PR reviews
- Build a small toolkit of 3-5 commands

Self-Assessment Questions

Ask yourself:

1. How complete are your tests?
2. Do you have pre-commit hooks?
3. What conversations do you keep having with AI?
4. What cleanup steps happen before every PR?
5. Who is your team’s AI champion?

What’s your biggest blocker to getting started? (tests / hooks / permissions / something else)

What’s Next

This Afternoon: Async Agents 101

- Kicking off background agents
- Linear, Slack, and GitHub integrations
- Parallel execution patterns
- Delegation philosophy

Topics covered today are prerequisites for async workflows.

Questions?

Resources:

Commands: github.com/jxnl/dots