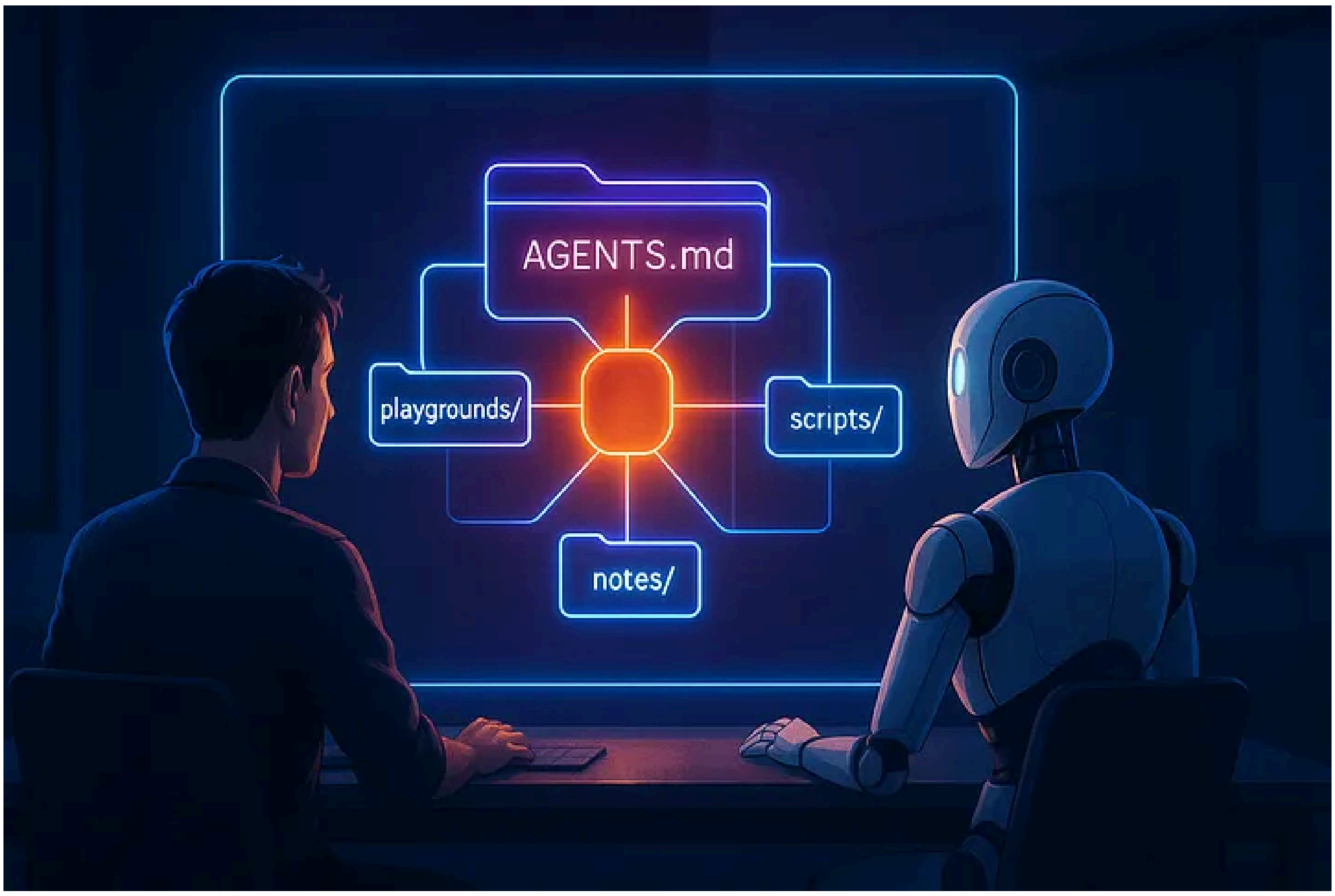
How to Work with Claude Code Effectively

19 min read · 3 days ago

Andi Ashari Follow



I have been working with Claude Code since February 2025 now, their engineering ability is just amazing, it has become my sort of pair engineer now. I've learned a lot since I started using Claude Code, and I would like to share everything here, mainly how to set up the AI Workspace/Playground to work with Claude Code to ensure best performance and results.

> *this setup works with other AI agents like Cursor, Gemini, Codex, and more, but in my experience, none match Claude Code's intelligence and adaptive behavior*

. . .

Core Principles

1. Full Context: One workspace for all related repos (frontend, backend, infrastructure) with centralized credentials

2. Organic Growth: Start minimal (just `.env`, `AGENTS.md`, `playgrounds/`), let `scripts/` and `notes/` grow as needed

3. Principles Over Rules: Teach Claude HOW to discover and think, not WHAT to do step-by-step

4. Dynamic Tools: Claude creates scripts/notes when needed, checks for existing tools before building new ones

5. Trust Intelligence: Claude is smart — avoid micromanaging, give it room to adapt and experiment

Think of it like onboarding a engineer: give them access to everything, teach them your principles, then let them figure out the details.

. . .

Why This Matters

Think about when engineer joins your team. You don’t have them work from a single repository. You give them:

- Access to all related codebases (frontend, backend, infrastructure)

- Credentials to databases and services

- Documentation and architecture notes

- Space to experiment and run scripts

- Context about how everything connects

Claude Code works best with the same setup. Instead of running `claude` inside a specific project directory, we create a workspace that contains everything Claude needs to understand and work on your entire system.

. . .

The Dynamic Workspace Philosophy

Before diving into the structure, understand this core concept: your workspace should grow and adapt dynamically, not be rigidly defined upfront.

Start Minimal

```
my-workspace/
├── .env           # Basic credentials
├── AGENTS.md      # Minimal principles (50-100 lines)
├── playgrounds/   # Clone your repos
└── (that's it - notes/, scripts/, guides/ don't exist yet)
```

Grow Organically (illustrative)

- Day 1: Claude creates `scripts/database-backup.sh` because you asked for a backup
- Day 2: Claude creates `notes/2025-10-20-auth-flow.md` after researching authentication
- Week 1: Claude creates `guides/production-deployment.md` after you deploy a few times
- Week 2: `scripts/README.md` exists with 10 tools, `notes/` has 30 docs

The workspace evolves based on actual needs, not imagined ones.

Claude should:

- Create tools when needed (scripts/, notes/)
- Update existing resources before creating new ones
- Organize things logically (scripts/database/, scripts/git/)
- Document what it creates (scripts/README.md, update AGENTS.md if patterns change)
- Use what exists before building new

You should:

- Let workspace grow naturally
- Review and refine AGENTS.md quarterly (not daily)
- Archive old projects/notes when no longer relevant
- Trust Claude to organize itself

Don’t micromanage. Give Claude principles and let it figure out the details.

. . .

The Workspace Structure

Let me show you how I structure my AI workspaces. Here’s a typical setup:

```
my-ai-workspace/
├── .env                    # All credentials in one place
├── AGENTS.md               # Instructions for Claude
├── CLAUDE.md -> AGENTS.md  # Symlink for compatibility
├── guides/                 # Operation-specific rules (optional)
│   ├── database-migration.md
│   ├── production-deployment.md
│   └── security-review.md
├── notes/                  # Claude's knowledge base
│   ├── 2025-09-02-woodpecker-ci-setup.md
│   ├── 2025-10-15-authentication-flow-analysis.md
│   └── 2025-10-18-project-architecture.md
├── playgrounds/            # All your git repositories
│   ├── my-project-api/
│   ├── my-project-frontend/
│   ├── my-project-infrastructure/
│   └── my-project-mobile/
├── scripts/                # Automation scripts
│   ├── pull-latest-base-branch.sh
│   ├── deploy-to-staging.sh
│   └── database/
│       └── backup-db.sh
└── tmp/                    # Additional context
    ├── Screenshot 2025-10-19 at 01.09.36.png
    ├── slack-conversation.txt
    ├── api-response.json
    └── user-requirements.md
```

Running Claude from the workspace root:

```
cd /home/andi/my-ai-workspace
claude
```

Why This Structure Works

This structure is effective because it provides complete context for Claude to work on your entire system:

1. Multi-repo awareness: Claude can see and work across frontend, backend, infrastructure, and mobile just like a full-stack engineer would
2. End-to-end understanding: Claude gets the full picture from UI → backend → database → infrastructure
3. Persistent knowledge: Notes accumulate over time, building institutional knowledge
4. Safe experimentation: Separate workspace means Claude won't accidentally clutter your actual project repos
5. Centralized credentials: One `.env` file for all projects in the workspace

Instead of running Claude inside a specific repository where it only sees that one project, we give it the entire ecosystem.

. . .

Breaking Down Each Component

guides/ — Operation-Specific Rules (Optional)

This directory contains mandatory procedures for complex operations. Think of these as step-by-step checklists that Claude must follow exactly.

When to use guides:

- Complex multi-step operations (database migrations, production deployments)
- Operations requiring safety checks and verification steps

- Workflows with strict compliance requirements
- Procedures that must be done the same way every time

Example guides:

```
guides/
├── database-migration.md       # Step-by-step DB migration procedure
├── production-deployment.md    # Production release checklist
├── security-review.md          # Security audit process
└── incident-response.md        # Emergency response procedures
```

Important:

Guides are RULES, not suggestions. When Claude is doing work that matches a guide's purpose, it must:

1. Read the ENTIRE specific guide first (not just skim)
2. Follow every step in order
3. Complete all verification checks
4. Document deviations (if any emergency requires it)

In AGENTS.md, reference guides like this:

```
## Specialized Guides

If `guides/` directory exists, check for operation-specific rules:

- `database-migration.md` Database schema changes
- `production-deployment.md` Deploying to production
- `security-review.md` Security audit checklist

**CRITICAL:** Must read ENTIRE content of specific guide before
starting work.

Guides override general instructions when applicable.
```

.env — Your Credential Hub

This is where you centralize all access credentials that Claude can use. Think of it as giving Claude the keys to the kingdom (in a secure way):

```
# Database connections
PGHOST=localhost
PGPORT=5432
PGDATABASE=your_database_name
PGUSER=your_username
PGPASSWORD=your_password

# Cloud services
SUPABASE_URL="https://[YOUR_PROJECT_REF].supabase.co"
SUPABASE_SERVICE_ROLE_KEY="[YOUR_SERVICE_ROLE_KEY]"

# Monitoring & Logging
DD_API_KEY="<YOUR_DATADOG_API_KEY>"
DD_SITE="datadoghq.com"

# CI/CD
WOODPECKER_SERVER="<https://your-ci-server.com>"
WOODPECKER_TOKEN="<YOUR_TOKEN>"

# Cloud providers
AWS_PROFILE="your-profile"
AWS_REGION="your-region"
```

Why this matters? With these credentials, Claude can:

- Connect to your databases to inspect schema and query data
- Access cloud services to understand infrastructure
- Check logs in monitoring platforms (Datadog, New Relic, CloudWatch, etc.)
- Interact with your CI/CD pipelines
- Manage cloud resources

Security note:

If you're not comfortable giving write access to Claude, control permissions at the platform level (e.g., use read-only database users, or read-only API keys).

notes/ — Claude's Knowledge Base

This is where Claude stores and retrieves institutional knowledge. Every time Claude researches something, solves a problem, or understands your architecture, it documents it here.

File naming convention: `YYYY-MM-DD-slug.md`

Examples from my workspaces:

- `2025-09-02-woodpecker-ci-complete-guide.md`
- `2025-10-15-authentication-flow-analysis.md`
- `2025-10-18-dev-environment-architecture.md`

Why this is powerful:

- Claude builds up knowledge over time instead of starting from scratch every session
- You can read these notes to understand what Claude learned
- Notes serve as living documentation that stays up-to-date
- When you switch context, Claude can quickly refresh its memory by reading relevant notes

How it works:

1. Before starting work, Claude reads relevant notes for context
2. After completing work, Claude updates existing notes or creates new ones
3. Over weeks and months, you build a comprehensive knowledge base

Important pattern:

Claude should update existing notes rather than creating duplicates. If there's already `authentication-flow.md`, don't create `authentication-flow-2.md` - update the original.

playgrounds/ — Where the Real Work Happens

This is where all your actual project repositories live. Think of it as your engineering playground. Structure example:

```
playgrounds/
├── my-product-api/          # Backend API
├── my-product-frontend/     # Web application
├── my-product-mobile/       # Mobile app
├── my-product-infrastructure/ # Terraform/IaC
├── my-product-workers/      # Background jobs
└── some-other-project-api/  # Unrelated project
```

Key principle:

The workspace root is NOT a git repository. Each subdirectory in `playgrounds/` is its own git repository with its own dependencies, `.env` files, and CI/CD configuration.

Why this works:

- Claude can work across multiple repos in the same session
- You can ask Claude to "check how authentication works in the frontend and backend" and it sees both
- Infrastructure changes can be coordinated with application changes
- Context flows naturally from UI → API → Database → Infrastructure

Real-world example:

I have three different workspace setups:

1. Enterprise infrastructure workspace (15+ repositories): Terraform repositories for different environments and services
2. Product workspace (5–7 repositories): Tightly integrated services for a single product (frontend, backend, workers, bot, ML service)

scripts/ — Tools That Grow Over Time

This directory starts empty and grows organically as Claude builds reusable tools. Think of it as Claude's toolbox that gets better over time.

The Growth Pattern:

Session 1:

```
You: "Find documentation in Confluence about authentication"
Claude: [manually calls Confluence API using credentials from .env]
Claude: [creates scripts/confluence-search.py to wrap the API call]
Claude: [updates scripts/README.md documenting the new tool]
```

Session 2 (weeks later):

```
You: "Check Confluence for deployment procedures"
Claude: [checks scripts/README.md first]
Claude: "Found existing scripts/confluence-search.py, using that..."
Claude: [runs the script instead of manually calling API]
```

After months, your scripts/ grows naturally:

```
scripts/
├── README.md              # Index of all scripts (Claude maintains this!)
├── confluence-search.py   # Search Confluence docs
├── git/
│   ├── pull-all-repos.sh
│   └── create-branches.sh
├── database/
│   ├── backup-db.sh
│   └── common-queries.sql
└── api-wrappers/
    ├── confluence-search.py
    └── ticket-lookup.py
```

In AGENTS.md, teach this pattern:

```
## Scripts Discovery & Growth

**ALWAYS check scripts/ before doing something manually:**

1. Check if scripts/README.md exists - read it for available tools
2. Check `ls scripts/` for relevant scripts
3. If script exists: use it
4. If script doesn't exist but task is repetitive: create script +
update README.md
```

```
**API Wrappers Pattern:**
When accessing external systems (Confluence, Jira, monitoring
tools):

- Check .env for credentials
- Create reusable script in scripts/api-wrappers/
- Document in scripts/README.md with usage examples
- Use the script in future sessions instead of manual API calls
```

Why this works:

- Scripts accumulate as you work, workspace gets more powerful
- Claude learns your patterns and automates repetitive tasks
- Next engineer (or future you) benefits from past automation
- Less manual work over time, more autonomous operation

tmp/ — Additional Context

This is where you (or Claude) put any temporary context needed for the current work.

What goes here:

- Screenshots of bugs or designs
- Slack conversation exports
- WhatsApp message conversations
- CSV data files
- API response samples
- Meeting notes
- User requirements documents

Example scenario:

```
tmp/
├── Screenshot 2025-10-19 at 01.09.36.png  # Bug screenshot
├── slack-product-requirements.txt         # Product discussion
```

```
    ├── api-response-sample.json                # Actual API response
    └── customer-data.csv                       # Data to analyze
```

You might say: "Claude, look at the screenshot in `tmp/`, read the Slack conversation about requirements, and implement the feature in the API following the response format in `api-response-sample.json`"

Key insight: Giving Claude visual context (screenshots), conversation context (Slack/WhatsApp), and data samples (JSON/CSV) dramatically improves its understanding and output quality.

. . .

Workspace Rule (AGENTS.md / CLAUDE.md)

This is the core of the implementation. This file tells Claude about your workspace structure, conventions, and how to work with your codebase.

Think of `AGENTS.md` as Claude's employee handbook - it explains:

- How your workspace is organized
- What each directory is for
- When to create notes vs update existing ones
- How to handle credentials
- Your development workflows
- Your tech stack and deployment process
- Critical rules and conventions

Why `AGENTS.md` AND `CLAUDE.md`? Create `AGENTS.md` as the main file, then create `CLAUDE.md` as a symlink:

```
ln -s AGENTS.md CLAUDE.md
```

Reason: Some AI tools recognize `CLAUDE.md`, others recognize `AGENTS.md`. The symlink ensures compatibility with both. There's even a movement around agents.md for standardizing AI agent instructions.

What Goes in AGENTS.md?

Critical Philosophy: Be Less Prescriptive, Not More

The biggest mistake is writing AGENTS.md like a rigid manual. Instead, teach principles and discovery patterns, not step-by-step instructions. You want Claude to think, not blindly follow rules.

Too Prescriptive:

“Always run backend on port 8881, frontend on port 3000, start backend first, wait 10 seconds, then start frontend”

Better:

“Projects may have service dependencies. Check project README for startup order. If unclear, look for inter-service API calls to understand dependencies.”

Why less is more:

- Claude is smart — over-specifying narrows its behavior and makes it less adaptable
- Projects change — prescriptive rules become outdated and misleading
- Discovery beats instruction — teaching how to find answers > giving answers
- Flexibility matters — different projects need different approaches

Workspace Structure & Organization

Don’t hardcode project lists — teach Claude how to discover them:

```
## Workspace Structure

**Directory Layout:**

- `playgrounds/` - All project repositories (each subdirectory is a
separate git repo)
- `notes/` - Documentation and knowledge base (YYYY-MM-DD-slug.md
format)
- `guides/` - Operation-specific rules and procedures (optional, but
mandatory when present)
- `scripts/` - Automation scripts and utilities
- `tmp/` - Temporary context files (screenshots, data samples,
conversations)
- `.env` - Workspace-level credentials

**Discovery Commands:**
When you need to understand the workspace, use:

<bash>
# See all projects
ls -la playgrounds/

# Check what each project is (look for package.json, go.mod,
requirements.txt, etc)

find playgrounds/ -name "package.json" -o -name "go.mod" -o -name
"requirements.txt"

# Find recent notes for context

ls -lt notes/ | head -20

# Check available scripts

ls -la scripts/
</bash>

**Key Principles:**

- Workspace root is NOT a git repository
- Each project in playgrounds/ is its own git repo with own
dependencies
- Notes use YYYY-MM-DD-slug.md convention (get date: `date +%Y-%m-
%d`)
- ALWAYS update existing notes rather than creating duplicates
- Project-specific documentation lives in the project's README, not
here
```

Core Principles (Not Rules)

Teach principles, not commands. Let Claude figure out HOW:

```
## Core Principles

**Source of Truth Hierarchy:**

1. Running code and actual behavior (what the system does)
2. Actual configuration files (.env, configs as they exist)
3. Recent git commits (what changed recently)
4. Documentation and notes (intended design, but verify!)

When docs/notes and code conflict → trust code, update docs.

**Discovery Before Action:**
Before doing anything:
```

```
- Check if tools already exist (scripts/, notes/, project docs)
- Search for similar implementations (how was this solved before?)
- Understand context (what depends on this? what does this depend
on?)
- Read project-specific docs (README, CONTRIBUTING, etc.)

Don't assume. Don't guess. Discover, then act.

**Process Management Principles:**

- Check if something is already running (avoid conflicts)
- Understand dependencies (what needs to start first?)
- Use bounded commands (don't hang with infinite streams)
- Verify completion (did it actually work?)

Figure out the HOW based on the project's tech stack.
```

Tech Stack & Patterns

Teach patterns, not specific technologies:

```
## Tech Stack Discovery

**Don't assume tech stack - discover it:**

- Check package.json for Node.js projects
- Look for go.mod for Go projects
- Find requirements.txt for Python projects
- Read project README for specifics

**Common Patterns in This Workspace:**

- Hot reload: Changes auto-restart (PM2, nodemon, Next.js dev mode)
- Containerization: Most projects use Docker
- CI/CD: Check for .github/workflows/, .woodpecker.yml, or similar
- Package managers: Check what project uses (npm, bun, pip, go mod)

**Process Management:**

- Development: Usually hot reload (don't restart unnecessarily)
- Production: Typically Docker containers
- Always check if service is already running before starting
```

General Workflow Principles

Keep it generic — projects have their own workflows:

```
## General Workflow Principles

**Before Starting Work:**

1. Check if services are already running (avoid conflicts)
2. Read project's README for specific startup instructions
```

```
3. Check notes/ for any known issues or dependencies
4. Look for project-specific docs (CONTRIBUTING.md, etc.)

**Making Changes:**

1. Research existing implementation (Grep for similar features)
2. Follow established patterns in that project
3. Test locally first (check project's test commands)
4. Update project docs if you introduce new patterns

**General Practices:**

- Never edit code directly on servers (use version control)
- Test in isolated environment first
- Check project's CI/CD setup (look for .github/, .woodpecker.yml,
etc.)
- Coordinate with deployment process (documented in project or
notes/)
```

Credential Management

Explain the credential hierarchy:

```
## Credentials

**Check in order:**

1. Workspace `.env` (root) - Shared credentials
2. Project `.env` (playgrounds/project/) - Project-specific
3. Global (~/.config, ~/.ssh) - System-wide

**Usage:**
<bash>

# Source workspace .env first

source /path/to/workspace/.env && {command}

# Example: Database access

source .env && psql -h $DB_HOST -U $DB_USER -d $DB_NAME
</bash>
```

Workspace-Specific Conventions

Only document workspace-level rules, not project specifics:

```
## Workspace Conventions

**Research Before Implementing:**
Before adding any feature:

1. Search for similar implementations: `grep -r "feature_name"
playgrounds/`
```

```
2. Read relevant notes for context: `ls notes/ | grep -i "topic"`
3. Check project's own README and docs
4. Follow established patterns in that project

**Multi-Project Coordination:**
When changes affect multiple projects:

- Identify all affected repos first
- Update related projects together
- Test integration points
- Document cross-repo dependencies in notes/

**Service Dependencies:**
If projects depend on each other (check notes/ or project docs):

- Start services in correct order
- Verify dependencies are running before starting dependents
- Check ports and connections

**Specialized Guides:**
If `guides/` directory exists:

- These are RULES for specific complex operations
- Check `ls guides/` for available operation guides
- Read relevant guide COMPLETELY before starting work
- Follow guide procedures exactly (they contain safety checks and
required steps)
- Guides override general instructions when applicable
```

. . .

Real-World Workspace Examples

Let me show you how I've set up workspaces for different scenarios:

Scenario 1: Enterprise Infrastructure Operations

Context: Managing cloud infrastructure across multiple AWS accounts

Structure:

```
@enterprise-infra-workspace/
├── .env                         # Project tracking, monitoring credentials
├── AGENTS.md                    # 365 lines
├── guides/                      # Specialized operation guides
│   ├── ticket-review.md
│   ├── iac-workflow.md
│   └── infrastructure-migration.md
├── notes/                       # 138 technical documents
├── playgrounds/
│   ├── terraform-prod-account/
│   ├── terraform-qa-account/
│   ├── terraform-staging/
│   └── lambda-extensions/
└── scripts/
    ├── ticket-cli.py
    └── deployment-scripts/
```

Key patterns in AGENTS.md:

- Mandatory guide reading before complex operations
- Strict authentication workflow (no direct credentials, use SSO/federated auth)
- Ticket-based workflow for tracking changes (TICKET-XXXX format)
- Infrastructure as Code only (no manual console changes)

Why this works: Enterprise work requires strict compliance and procedures. The `guides/` contain 500-1500 line step-by-step workflows for complex operations. Claude must read the full guide before starting work.

Scenario 2: Product Development

Context: Single product with tightly integrated microservices

Structure:

```
@product-workspace/
├── .env                    # Database, backend services, message queue
├── AGENTS.md               # 646 lines
├── notes/                  # 77 architecture documents
├── logs/                   # Service logs (frontend.log, backend.log)
├── playgrounds/
│   ├── product-frontend/   # Next.js (e.g. port 3000/3330)
│   ├── product-backend/    # Express (e.g. port 8080/8881)
│   ├── product-workers/    # Background processing
│   ├── product-ml/         # Python ML service (e.g. port 5000)
│   └── product-bot/        # Bot integration
└── scripts/
    ├── git/
    │   └── create-feature-branches.sh
    └── database/
        └── backup-db.sh
```

Key patterns in AGENTS.md:

- Critical service startup order (backend before frontend)
- Service dependencies and shared database
- Development vs production environment differences
- Hot reload configuration (PM2 for backend, Next.js for frontend)

Why this works: Product development requires understanding how all services connect and depend on each other. The `logs/` directory helps debug service interactions, and the startup order documentation prevents common issues.

. . .

How Claude Builds Knowledge: The Note-Taking Pattern

One of the most powerful aspects of this workspace approach is how Claude builds and maintains knowledge over time.

The Note System Workflow

Before starting work:

1. Claude searches existing notes: `ls notes/ | grep -i "authentication"`
2. Reads relevant notes for context
3. Starts with existing knowledge instead of from scratch

During work:

- Claude discovers how things actually work (by reading code)
- Finds discrepancies between notes and reality
- Mentally tracks what needs updating

After completing work:

- Claude updates existing notes with new findings
- Creates new notes for entirely new topics

- Marks outdated information with dates

Example workflow:

You ask: “How does our authentication system work?”

Claude:

1. Finds `notes/2025-09-15-authentication-flow.md`
2. Reads it for context
3. Verifies against actual code in `playgrounds/api/auth/`
4. Discovers note says “JWT expires in 24 hours” but code says “7 days”
5. Updates the note with correction and date: `[Updated 2025-10-20: JWT actually expires in 7 days]`

Over time: You build comprehensive, accurate documentation that stays up-to-date with your codebase.

Real Note Examples

Technical deep-dives:

- `2025-09-11-lambda-extensions-deep-dive.md` (How AWS Lambda extensions work)
- `2025-10-18-dev-environment-architecture.md` (Complete infrastructure documentation)
- `2025-09-27-project-relationships.md` (How services connect and depend on each other)

Implementation tracking:

- `2025-01-22-demo-booking-implementation-live-tracking.md` (Feature implementation progress)
- `2025-09-12-lambda-extension-deployment-learnings.md` (What we learned during deployment)

Problem solving:

- `2025-09-02-container-exit-code-1-troubleshooting.md` (Debugging guide for specific error)
- `2025-09-22-emergency-rollback-guide.md` (Emergency procedures)

Patterns and conventions:

- `2025-09-12-logging-standards.md` (How we handle logging)
- `2025-09-03-agents-md-update-guide.md` (How to maintain workspace instructions)

Note Organization Best Practices

DO:

- Use YYYY-MM-DD prefix for chronological sorting
- Use descriptive slugs: `authentication-flow` not `auth`
- Update existing notes rather than creating duplicates
- Mark outdated info with dates: `[Updated 2025-10-20]`
- Reference code files and line numbers: `auth.ts:234`

DON'T:

- Create `feature-v2.md`, `feature-final.md`, `feature-final-final.md`
- Leave contradictory information across multiple notes
- Forget to update notes after code changes
- Create notes for trivial one-time tasks

. . .

Best Practices: Common Patterns Across All Workspace Types

After working with Claude across these different setups, I've identified universal patterns that make any workspace effective:

Principles Over Prescriptions

Start AGENTS.md with core principles, not rigid rules:

```
## Working Principles

**Think, Don't Just Follow:**
You're a capable AI agent. These are principles to guide your
thinking, not rules to constrain you. Adapt to what you discover.

**Code is Reality, Docs are Intentions:**

- Verify everything against actual code/config/behavior
- When docs conflict with reality, trust reality and update docs
- Recent git commits show what actually changed

**Discovery Pattern:**

1. What already exists? (scripts/, notes/, project docs)
2. How was this solved before? (search codebase)
3. What's the current state? (git log, running processes)
4. What does the project say? (README, configs)
5. Now act based on what you learned

**Autonomy with Safety:**

- Experiment and figure things out, but don't break production
- Test locally before touching anything shared
- Understand impact before making changes
- When uncertain about safety, ask
```

Source of Truth Hierarchy

Tell Claude explicitly what to trust:

```
## Source of Truth Hierarchy

1. **Actual code** - What the system actually does
2. **Live environment** - Production behavior
3. **Recent git commits** - What changed recently
4. **Environment config** - .env files
5. **This AGENTS.md and notes** - Intended design

**Workflow:**

- Read notes for context
- Verify against actual code
- When conflict, trust code
- Update notes to match reality
```

Investigation Protocol

Teach Claude how to debug systematically:

```
## Investigation Protocol
```

```
**Phase 1: Understand**

1. What fails? (collect exact error messages)
2. Which components/services affected?
3. What changed recently? (`git log --oneline -10`)
4. Is it isolated to one area or systemic?

**Phase 2: Investigate**
Discover the tech stack first, then investigate appropriate layer:

- Web apps: Browser DevTools, network tab, console logs
- APIs/Services: Application logs, test with curl/Postman
- Databases: Connection status, query logs, schema inspection
- Infrastructure: Service status, resource usage, container logs

**Phase 3: Fix**

1. Fix in local codebase (never directly on servers)
2. Test fix thoroughly in local environment
3. Verify complete fix (not just symptoms)
4. Update relevant documentation (notes/ or project docs)
```

. . .

Advanced Techniques and Patterns

Once you have the basics working, here are advanced patterns and power-user techniques:

Specialized Guides for Complex Operations

For enterprise or complex workflows, create operation-specific guides in `guides/`:

```
guides/
├── ticket-review.md          # Ticket/issue review process
├── iac-workflow.md           # Infrastructure implementation
├── database-migration.md     # Database changes
└── production-deployment.md  # Production deployments
```

In your `AGENTS.md`, tell Claude when to use these:

```
## Specialized Guides

**MANDATORY Guide Reading:**

If `guides/` directory exists, check for operation-specific rules.
```

```
| Task                    | Guide                    | Trigger
Words                              |
| ----------------------- | ------------------------ | ------------
-------------------------------- |
| Review ticket           | ticket-review.md         | "review
TICKET-XXX", "analyze issue"        |
| Implement infrastructure | iac-workflow.md          | "work on
TICKET-XXX", "implement", "deploy" |
| Database changes        | database-migration.md    | "migration",
"schema change"                  |
| Deploy to production    | production-deployment.md | "deploy to
prod", "release"                  |

**CRITICAL:** Must read ENTIRE guide before starting work.
Guides are RULES, not suggestions. Follow every step exactly.
```

Why this matters: For complex enterprise workflows with 10+ steps, checklists, and safety checks, you don't want to repeat the same instructions every time. Document once in `guides/`, reference always.

Context Gathering Shortcuts

Add quick reference commands to `AGENTS.md`:

```
## Quick Context Gathering

**Workspace level:**
<bash>
ls -la playgrounds/ # All projects
ls -lt notes/ | head -10 # Recent notes
cat AGENTS.md | head -100 # Workspace docs
</bash>

**Project level:**
<bash>
cat README.md | head -50 # Project overview
git log --oneline -10 # Recent changes
cat package.json | grep -A 15 '"scripts"' # Available commands
</bash>

**Infrastructure:**
<bash>
find playgrounds/ -name "Dockerfile" # All Docker configs
find playgrounds/ -name "\\*.tf" # All Terraform files
</bash>
```

This helps Claude quickly orient itself when starting a new session.

. . .

Real-World Use Cases

Let me show you real scenarios where this workspace approach shines:

Use Case 1: "Debug why users can't login"

Without workspace:

```
You: Debug the login issue
Claude: Can you show me the login code?
You: *pastes authentication code*
Claude: Can you show me the frontend login form?
You: *pastes frontend code*
Claude: What database are you using?
You: *explains database setup*
Claude: Can you show me the error logs?
You: *pastes logs*
```

With workspace:

```
You: Debug why users can't login. Check the logs in tmp/
Claude:
1. Reading error logs from tmp/frontend-errors.log
2. Checking authentication code in playgrounds/api/auth/
3. Verifying frontend form in playgrounds/frontend/components/Login.tsx
4. Testing database connection using credentials from .env
5. Found the issue: JWT secret mismatch between frontend and backend .env files
6. Updating notes/2025-10-20-login-debug-session.md with findings
```

Result: Claude has full context immediately and can investigate end-to-end without you manually providing each piece.

Use Case 2: "Implement a new analytics feature"

Without workspace:

```
You: Add weekly analytics reports
Claude: What's your current analytics setup?
You: *explains*
Claude: What database structure do you have?
You: *explains*
```

```
Claude: Where should I add this code?
You: *explains file structure*
Claude: Here's the code
You: Wait, this doesn't match our pattern for handling dates
Claude: *adjusts*
You: And we need to handle multiple workspaces
Claude: *adjusts again*
```

With workspace:

```
You: Add weekly analytics reports for email campaigns
Claude:
1. Reading notes/2025-09-27-analytics-architecture.md for current patterns
2. Checking existing analytics in playgrounds/backend/routes/analytics.ts
3. Reviewing database schema in playgrounds/backend/migrations/
4. Found existing daily reports - will follow same pattern
5. Implementing weekly aggregation following workspace-aware pattern
6. Adding to frontend in playgrounds/frontend/components/Analytics/
7. Testing with actual database using .env credentials
8. Creating migration in migrations/2025-10-20-add-weekly-analytics.sql
9. Updating notes/2025-09-27-analytics-architecture.md with new feature
```

Result: Claude researches existing patterns, follows your conventions, and implements features that integrate seamlessly with your existing code.

Use Case 3: "We need to migrate to a new database"

Without workspace: You spend hours explaining current database structure, where it's used, environment configurations, deployment process...

With workspace:

```
You: We're migrating from MongoDB to PostgreSQL
Claude:
1. Reading notes/ to understand current database usage
2. Searching all repos: grep -r "mongoose\\|mongodb" playgrounds/
3. Found database usage in:
   playgrounds/api/models/ (8 models)
   playgrounds/api/services/ (12 services)
   playgrounds/workers/ (5 background jobs)
4. Checking current connection in .env
5. Creating migration plan in notes/2025-10-20-mongodb-to-postgresql-migration.
```

```
6. Estimating: ~50 files need changes across 3 repos
7. Recommending parallel database approach during transition
```

Result: Claude understands the full scope across your entire system and creates a comprehensive migration plan.

. . .

Measuring Success

How do you know if your workspace is working well?

Good signs:

- Claude rarely asks you for basic information (it finds it in workspace)
- Claude follows your patterns and conventions consistently
- Notes directory growing with useful documentation
- You can resume complex tasks after days/weeks because context is preserved
- Claude catches mistakes by cross-referencing multiple repos

Bad signs:

- Claude constantly asking "where is X?" or "how does Y work?"
- Claude suggesting solutions that don't match your patterns
- You're manually pasting code frequently instead of referencing files
- Notes are outdated and contradictory
- Claude makes changes in one repo that break another

Real metrics from my experience:

Before workspace approach:

- 70% of time explaining context
- 30% of time actual implementation

- Frequent back-and-forth for clarifications

After workspace approach:

- 10% of time explaining high-level goals
- 90% of time Claude working autonomously
- Much fewer clarification questions, more one-shot implementation

. . .

Conclusion

The key insights: Treat Claude like am engineer, and give it room to think.

Give Claude:

- Complete codebase access (all repos in playgrounds/)
- System credentials (centralized in .env)
- Space to build tools and knowledge (scripts/, notes/ grow over time)
- Principles and patterns (AGENTS.md), not rigid step-by-step rules
- Freedom to discover, experiment, and adapt

Don’t:

- Run Claude inside a single repository
- Make Claude ask for every piece of context
- Write 500-line prescriptive AGENTS.md that constrains Claude’s thinking
- Micromanage how Claude organizes scripts/ and notes/
- Expect to define everything upfront

Claude Code Agentic Ai Ai Agent Artificial Intelligence

Follow

Written by Andi Ashari

191 followers · 36 following

Tech Wanderer, Passionate about Innovation and Deeply Expertised in Technology. Actively Redefining the Digital Landscape Through Cutting-Edge Solutions.

No responses yet

Bgerby

What are your thoughts?

More from Andi Ashari

Andi Ashari

Getting Better Results from Cursor AI with Simple Rules

Hey everyone! I've been playing around with Cursor AI lately, and it's been super helpful for coding. But sometimes, it needs a little...

Feb 28 161 8

Andi Ashari

Easy-to-Follow Guide of How to Install PyENV on Ubuntu

Originally posted at: A Beginner-Friendly Guide on How to Install PyENV on Ubuntu

Jan 27, 2024 186 2

Andi Ashari

Installing Jupyter Notebook on Ubuntu 22.04: A Step-by-Step Guide

This page is originally posted on: https://ashari.me/posts/installing-jupyter-notebook-on-ubuntu-22-04-a-comprehensive-guide

Sep 30, 2023 32 4

Andi Ashari

Tracing Bun and ElysiaJS with OpenTelemetry and Datadog

Bun's speed and ElysiaJS's elegance make them a powerful combination for building performant web APIs. But as your application grows...

Aug 14, 2024 6

See all from Andi Ashari

Recommended from Medium

Hashbyt | AI-First Frontend & UI/UX SaaS Partner

11 Best AI Tools for Developers to Use in Your Next Coding Project

The right AI tools for developers can transform your coding process, from writing and testing to debugging and deployment.

Sep 9 135 4

Yosif qasim

Weaponizing Claude Code Skills: From 5*5 to Remote Shell

Let’s be clear : - What runs on your machine is ultimately *your* responsibility - The scenario described is targeting lazy vibe coders …

6d ago 11

In UX Planet by Nick Babich

Claude For Code: How to use Claude to Streamline Product Design Process

Anthropic Claude is a primary competitor of OpenAI's ChatGPT. Just like ChatGPT this is versatile tool that can be use used in many...

Oct 16 372 3

Joe Njenga

8 Little-Known Books Every AI Founder Should Read First (Before Building a Unicorn)

We all have dreams, but few reach the finish line. Why?

3d ago 86

In Coding Beauty by Tari Ibaba

Google Just Made Gemini CLI Even More Insane 🤯

Massive new upgrade to this powerful CLI

5d ago 126 2

Pawel

Claude Skills: The AI Feature That Actually Solves a Real Problem

Yesterday, Anthropic quietly released what might be the most practical AI feature of 2025. It's not flashier models or better benchmarks...

6d ago 142 8

See more recommendations