

Coding Context CLI

Dynamically Assemble Context for AI Coding Agents

A command-line tool that collects, filters, and delivers rich context to AI models

The Problem

AI coding agents need comprehensive context to make informed decisions:

-  **Project-specific** coding standards and conventions
-  **Architecture** patterns and structure
-  **Technology stack** and dependencies
-  **Team practices** and guidelines
-  **Task-specific** requirements and constraints

Manual assembly is tedious and error-prone

The Solution: Coding Context CLI

A tool that **automatically assembles** the right context for AI agents:

-  Discover rules from multiple sources
-  Filter based on task requirements
-  Substitute runtime parameters
-  Support remote rule repositories
-  Integrate with any AI agent

One command → Rich, relevant context

Key Features



Dynamic Context Assembly

- Merges context from various source files
- Supports multiple file formats and locations



Rule-Based Context

- Reusable context snippets (rules)
- Frontmatter filtering for precision



Task-Specific Prompts

- Different prompts for different tasks
- Parameter substitution for runtime values

Key Features (continued)



Remote Directories

- Load rules from Git, HTTP, S3
- Share context across teams and projects



Bootstrap Scripts

- Fetch or generate context dynamically
- Execute setup tasks before context assembly



Token Estimation

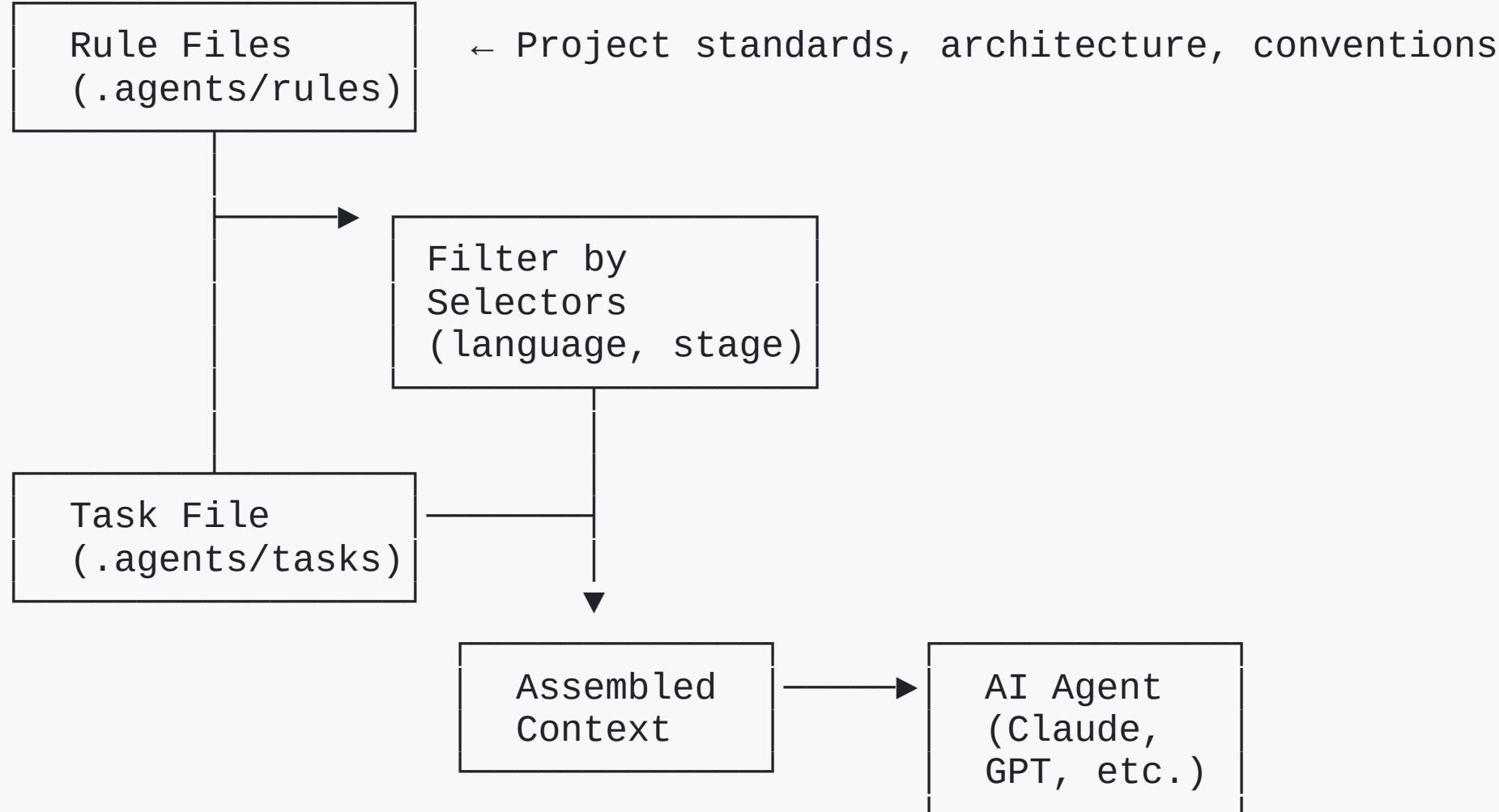
- Monitor context size
- Optimize for model limits

Supported AI Agents

Works with configuration files from major AI coding tools:

Agent	Configuration Files
Anthropic Claude	CLAUDE.md , .claude/
GitHub Copilot	.github/copilot-instructions.md , .github/agents/
Cursor	.cursor/rules , .cursorrules
Google Gemini	GEMINI.md , .gemini/
OpenCode.ai	.opencode/agent , .opencode/rules
Generic	.agents/rules , AGENTS.md

How It Works



Installation

Linux (AMD64):

```
sudo curl -fsL -o /usr/local/bin/coding-context \
https://github.com/kitproj/coding-context-cli/releases/download/v0.0.23/coding-context_v0.0.23_linux_amd64
sudo chmod +x /usr/local/bin/coding-context
```

macOS (Apple Silicon):

```
sudo curl -fsL -o /usr/local/bin/coding-context \
https://github.com/kitproj/coding-context-cli/releases/download/v0.0.23/coding-context_v0.0.23_darwin_arm64
sudo chmod +x /usr/local/bin/coding-context
```

Basic Usage

```
coding-context [options] <task-name>
```

Simple example:

```
coding-context fix-bug | llm -m claude-3-5-sonnet-20241022
```

With parameters:

```
coding-context -p issue_key=BUG-123 fix-bug | llm -m gemini-pro
```

With selectors:

```
coding-context -s languages=go -s stage=implementation implement-feature
```

Command-Line Options

Option	Description
-c <dir>	Change to directory before doing anything
-p key=value	Parameter to substitute in the prompt
-s key=value	Include rules with matching frontmatter
-a <agent>	Target agent (excludes that agent's own rules)
-d <path>	Remote directory with rules (git://, http://, s3://)
-m <url>	URL to manifest file with search paths
-r	Resume mode: skip rules, select resume task

Example: Fix a Bug

Command:

```
coding-context \
  -s languages=go \
  -s priority=high \
  -p issue_number=PROJ-1234 \
  fix-bug | llm -m claude-3-5-sonnet-20241022
```

What happens:

1. Finds task file: .agents/tasks/fix-bug.md
2. Includes Go-specific rules with high priority
3. Substitutes \${issue_number} → PROJ-1234
4. Outputs combined context to AI agent

Rule Files

Rules are reusable context snippets with optional YAML frontmatter:

```
---
```

```
languages:
  - go
stage: implementation
---
```

```
# Backend Coding Standards
```

- All new code must be accompanied by unit tests
- Use the standard logging library
- Follow Go project layout conventions

Selectors match top-level YAML fields only

Task Files

Tasks define what the AI agent should do:

```
---  
selectors:  
  languages: go  
  stage: implementation  
---  
# Task: Fix Bug in ${issue_number}
```

Analyze the following issue and provide a fix:

Issue Number: \${issue_number}
Priority: \${priority}
Description: \${description}

Parameters are substituted at runtime using -p flags

Content Expansion

Task and rule content supports three types of dynamic expansion:

1. Parameter Expansion - `${parameter_name}`

Issue: `${issue_key}`

Description: `${description}`

2. Command Expansion - `!`command``

Current date: `!`date +%Y-%m-%d``

Git branch: `!`git rev-parse --abbrev-ref HEAD``

3. Path Expansion - `@path`

Current configuration:

`@config.yaml`

Remote Directories

Load rules from remote sources for team collaboration:

```
# From a Git repository
coding-context \
  -d git::https://github.com/company/shared-rules.git \
    fix-bug

# From HTTP/HTTPS
coding-context \
  -d https://cdn.company.com/coding-standards \
    implement-feature

# From S3
coding-context \
  -d s3::https://s3.amazonaws.com/my-bucket/rules \
    deploy
```

Supports: git, http/https, s3, file, and more via go-getter

Bootstrap Scripts

Execute scripts before processing rules or tasks:

Rule bootstrap (.agents/rules/jira-bootstrap):

```
#!/bin/bash
# Install jira-cli if not present
if ! command -v jira-cli &> /dev/null; then
    echo "Installing jira-cli..." >&2
    # Installation commands
fi
```

Task bootstrap (.agents/tasks/fix-bug-bootstrap):

```
#!/bin/bash
# Fetch issue details
echo "Fetching issue information..." >&2
jira-cli get-issue ${issue_number}
```

File Search Paths

The tool automatically discovers files in multiple locations:

Tasks:

- `./agents/tasks/*.md`
- `~/.agents/tasks/*.md`

Rules:

- `./agents/rules/`, `./cursor/rules/`, `./github/agents/`
- `CLAUDE.md`, `CLAUDE.local.md`, `AGENTS.md`, `GEMINI.md`
- `~/.agents/rules/`, `~/.claude/`, `~/.gemini/`
- User home and system-wide directories

Precedence: Local → User home → System-wide

Frontmatter Selectors

Filter rules precisely using YAML frontmatter:

Rule file:

```
---
languages: go
stage: implementation
priority: high
---
# Go Implementation Guidelines
...
```

Select it:

```
coding-context \
  -s languages=go \
  -s stage=implementation \
fix-bug
```

Task Frontmatter Selectors

Tasks can automatically apply selectors:

```
---
```

```
selectors:
  languages: go
  stage: implementation
```

```
---
```

```
# Implement Feature
```

Implement following Go best practices...

When you run:

```
coding-context implement-feature
```

It's equivalent to:

```
coding-context -s languages=go -s stage=implementation implement-feature
```

Resume Mode

Continue work without re-sending all rules:

Initial invocation:

```
coding-context fix-bug | ai-agent  
# Includes all rules + initial task
```

Resume invocation:

```
coding-context -r fix-bug | ai-agent  
# Skips rules, uses resume-specific task
```

Task files:

- `fix-bug-initial.md` with `resume: false`
- `fix-bug-resume.md` with `resume: true`

Saves tokens and reduces context size

Targeting Specific Agents

Exclude agent-specific paths (agent reads them itself):

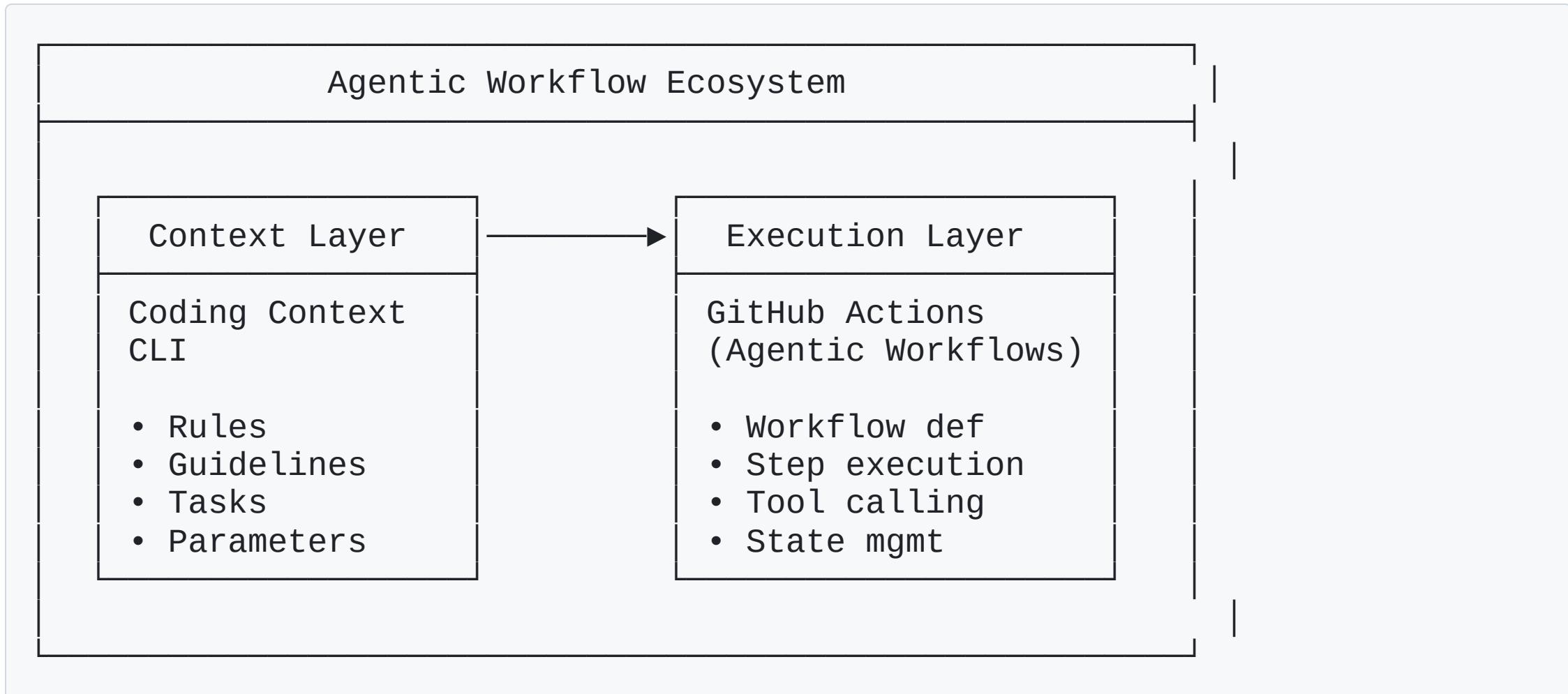
```
# Using with Cursor
coding-context -a cursor fix-bug
# Excludes: .cursor/rules, .cursorrules
# Includes: .github/agents, .agents/rules, etc.

# Using with GitHub Copilot
coding-context -a copilot implement-feature
# Excludes: .github/copilot-instructions.md, .github/agents
# Includes: .cursor/rules, .agents/rules, etc.
```

Avoids duplication while including cross-agent rules

Agentic Workflows Integration

Perfect for autonomous AI workflows:



GitHub Actions Integration

Use in CI/CD workflows:

```
name: Agentic Code Review
on: [pull_request]

jobs:
  review:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - name: Assemble Context
        run: |
          coding-context \
            -s task=code-review \
            -p pr_number=${{ github.event.pull_request.number }} \
            code-review > context.txt

      - name: Execute AI Review
        uses: github/agent-action@v1
        with:
```

Multi-Stage Workflows

Different context for different stages:

```
jobs:  
  plan:  
    steps:  
      - name: Planning Context  
        run: coding-context -s stage=planning plan-feature > plan.txt  
  
  implement:  
    steps:  
      - name: Implementation Context  
        run: coding-context -s stage=implementation implement > impl.txt  
  
  test:  
    steps:  
      - name: Testing Context  
        run: coding-context -s stage=testing test-feature > test.txt
```

Best Practices

1. Version Control Your Rules

- Store `.agents/rules` and `.agents/tasks` in Git
- Track changes to context over time

2. Use Selectors Strategically

- Filter by language, stage, priority
- Keep context relevant and focused

3. Parameterize Task Prompts

- Use `-p` for runtime values
- Make tasks reusable

4. Organize by Concern

Best Practices (continued)

5. Use Bootstrap Scripts

- Fetch real-time data (Jira, GitHub)
- Install required tools
- Prepare environment

6. Monitor Token Count

- Tool reports token estimates to stderr
- Stay within model limits
- Optimize rule selection

7. Share Team Rules

- Use remote directories (-d flag)
- Maintain organization-wide standards

Example: Multi-Language Project

Project structure:

```
.agents/
└── rules/
    ├── go-standards.md      (languages: [go])
    ├── python-standards.md   (languages: [python])
    ├── js-standards.md       (languages: [javascript])
    └── testing.md           (stage: testing)
└── tasks/
    ├── fix-bug.md
    └── implement-feature.md
```

Usage:

```
# Work on Go code
coding-context -s languages=go fix-bug

# Work on Python code
coding-context -s languages=python implement-feature
```

Example: Remote Rules Repository

Scenario: Company maintains shared coding standards

```
# Use company-wide rules
coding-context \
  -d git::https://github.com/company/coding-standards.git \
  -s languages=go \
  implement-feature | ai-agent

# Mix local and remote
coding-context \
  -d git::https://github.com/company/standards.git \
  -d https://team.company.com/guidelines \
  -s priority=high \
  fix-bug | ai-agent
```

Benefits: Centralized, versioned, reusable

Real-World Use Cases



Bug Triage & Fixing

```
coding-context -p issue=BUG-123 -s languages=go fix-bug
```



Feature Implementation

```
coding-context -s stage=implementation implement-feature
```



Code Review

```
coding-context -p pr_number=456 code-review
```



Documentation Updates

```
coding-context -s type=documentation update-docs
```

Real-World Use Cases (continued)



Deployment Tasks

```
coding-context -s environment=production -p version=1.2.3 deploy
```



Refactoring

```
coding-context -s languages=java -p module=auth refactor
```



Test Writing

```
coding-context -s stage=testing -s languages=python write-tests
```



Performance Optimization

```
coding-context -s priority=high optimize-performance
```

Token Estimation

Tool provides real-time token estimates:

```
$ coding-context -s languages=go fix-bug > context.txt
[INFO] Processing rules...
[INFO] Token estimate: ~2,450 tokens
[INFO] Task: fix-bug (~500 tokens)
[INFO] Total estimate: ~2,950 tokens
```

Helps you:

- Stay within model limits (GPT-4: 8K-128K, Claude: 200K)
- Optimize rule selection
- Monitor context growth

Security & Privacy

-  **Single-pass expansion** prevents injection attacks
-  **Bootstrap output** goes to stderr (not AI context)
-  **No secrets** in version-controlled rules
-  **Local execution** - data stays on your machine

Best practices:

- Use environment variables for secrets
- Keep sensitive data in bootstrap scripts
- Review generated context before sending to AI

Language Support

Common languages supported through selectors:

Frontend: javascript , typescript , html , css , dart

Backend: go , java , python , ruby , rust , csharp , php

Mobile: swift , kotlin , objectivec , dart

Other: shell , yaml , markdown , scala , elixir , haskell

Note: Use lowercase in frontmatter and selectors

Project Structure Example

```
my-project/
  └── .agents/
      ├── rules/
      │   ├── go-standards.md
      │   ├── testing.md
      │   └── security.md
      └── tasks/
          ├── fix-bug.md
          ├── implement-feature.md
          └── code-review.md
    └── commands/
        ├── pre-deploy.md
        └── post-deploy.md
  └── .github/
      └── copilot-instructions.md
  └── CLAUDE.local.md
```

Comparison with Alternatives

Feature	Coding Context CLI	Manual Context	Static Prompts
Dynamic Assembly	Automatic	Manual	Static
Filtering	Frontmatter	Copy-paste	None
Parameterization	CLI flags	Text edit	Hardcoded
Reusability	High	Low	Medium
Team Sharing	Git/Remote	Manual	Git
Version Control	Native	Manual	Native
Token Optimization	Automatic	Manual	None

Getting Started (5 Steps)

1. Install the CLI

```
curl -fsL -o /usr/local/bin/coding-context <release-url>
chmod +x /usr/local/bin/coding-context
```

2. Create rule file (.agents/rules/standards.md)

```
# My Coding Standards
- Use meaningful names
- Write tests
```

3. Create task file (.agents/tasks/fix-bug.md)

```
# Fix Bug: ${issue}
```

Getting Started (continued)

4. Run the CLI

```
coding-context -p issue=123 fix-bug | llm -m claude-3-5-sonnet-20241022
```

5. Iterate and refine

- Add more rules
- Use selectors for filtering
- Parameterize tasks
- Share with team

Resources



Documentation

- [Full Documentation](#)
- [GitHub Repository](#)



Guides

- [Getting Started Tutorial](#)
- [How-to Guides](#)



Integration

- [Agentic Workflows Guide](#)
- [GitHub Actions Integration](#)

Community & Support

Get Help

- [GitHub Issues](#)
- [Discussions](#)

Contribute

- [Contributing Guide](#)
- Pull requests welcome!

License

- MIT License
- Free for personal and commercial use

Roadmap



Upcoming Features

- Enhanced token optimization
- Rule validation and linting
- Context caching for faster assembly
- More agent integrations
- AI-powered rule selection
- Workflow context injection
- Agent memory persistence

Follow the project for updates!

Key Takeaways

-  **Automate context assembly** for AI coding agents
-  **Filter and optimize** with frontmatter selectors
-  **Parameterize** task prompts for reusability
-  **Share rules** via Git, HTTP, S3
-  **Integrate** with GitHub Actions and workflows
-  **Support all major** AI coding agents
-  **Open source** and extensible

Thank You!

Coding Context CLI

Give AI agents the context they need to excel

-  kitproj.github.io/coding-context-cli
-  github.com/kitproj/coding-context-cli

Questions?

Appendix: Command Reference

Usage:

```
coding-context [options] <task-name>
```

Options:

- C string
 Change to directory before doing anything. (default ".")
- d value
 Remote directory containing rules and tasks
- m string
 Go Getter URL to a manifest file
- p value
 Parameter to substitute (key=value)
- r
 Resume mode (skip rules)
- s value
 Include rules with matching frontmatter (key=value)
- a string
 Target agent (cursor, opencode, copilot, etc.)

Appendix: Supported go-getter Protocols

Protocol	Example	Description
http://	http://example.com/rules.tar.gz	HTTP download
https://	https://example.com/rules.tar.gz	HTTPS download
git::	git::https://github.com/user/repo.git	Git clone
s3::	s3::https://s3.amazonaws.com/bucket/path	S3 bucket
file://	file:///path/to/local/dir	Local file path

See [go-getter docs](#) for more

Appendix: File Extensions

Extension	Description
.md	Markdown rule or task file
.mdc	Markdown component (alternative extension)
-bootstrap	Executable bootstrap script (no extension)

Examples:

- standards.md - Rule file
- fix-bug.md - Task file
- jira-bootstrap - Bootstrap script