SPEC-DOC-006: Configuration & Customization Guide

Status: Pending **Priority**: P1 (Medium) **Estimated Effort**: 8-12 hours **Target Audience**: Power users, advanced users **Created**: 2025-11-17

Objectives

Comprehensive guide to configuring and customizing the codex CLI: 1. Configuration file structure (config.toml complete schema) 2. 5-tier precedence (CLI, shell, profile, TOML, defaults) 3. Model configuration (providers, reasoning, profiles) 4. Agent configuration (5 agents, subagent commands, quality gates) 5. Quality gate customization (per-checkpoint agent selection) 6. Hot-reload configuration (config_reload.rs, 300ms debounce) 7. MCP server configuration (server definitions, lifecycle) 8. Environment variables

(CODEX_HOME, API keys, overrides) 9. Templates (installation, customization, versioning) 10. Theme system (TUI themes, accessibility)

Scope

In Scope

- Complete config.toml reference (all sections)
- 5-tier precedence system (with examples)
- Model provider configuration (OpenAI, Anthropic, Google, Ollama)
- Agent configuration (gemini, claude, code, gpt_pro, gpt_codex)
- Quality gate customization (per-checkpoint overrides)
- Hot-reload mechanism (300ms debounce, watch system)
- MCP server definitions (local-memory, git-status, hal, custom)
- Environment variables (CODEX_HOME, *API_KEY, SPEC_OPS*)
- Template customization (installing, modifying, versioning)
- TUI theme customization (colors, accessibility)

Out of Scope

- Architecture of config system (see SPEC-DOC-002)
- Installation and setup (see SPEC-DOC-001)
- Security of secrets (see SPEC-DOC-007)

Deliverables

1. **content/config-reference.md** - Complete config.toml schema
2. **content/precedence-system.md** - 5-tier precedence with examples
3. **content/model-configuration.md** - Provider setup, reasoning effort
4. **content/agent-configuration.md** - 5 agents, subagent commands
5. **content/quality-gate-customization.md** - Per-checkpoint overrides
6. **content/hot-reload.md** - Config reload mechanism, debouncing
7. **content/mcp-servers.md** - MCP server definitions, custom servers
8. **content/environment-variables.md** - All env vars, overrides
9. **content/template-customization.md** - Installing, modifying templates
10. **content/theme-system.md** - TUI themes, accessibility options

---

Success Criteria

- ☐ Complete config.toml schema documented
- ☐ 5-tier precedence clearly explained with examples
- ☐ All agent configurations documented
- ☐ Quality gate customization guide complete
- ☐ Environment variables comprehensive list
- ☐ Template customization tutorial complete

Related SPECs

- SPEC-DOC-000 (Master)
- SPEC-DOC-001 (User Onboarding - basic config)
- SPEC-DOC-002 (Core Architecture - config system internals)
- SPEC-DOC-003 (Spec-Kit - quality gate config)

Status: Structure defined, content pending

Agent Configuration

Multi-agent setup, subagent commands, and agent profiles.

Overview

The **multi-agent system** enables consensus-driven decision-making through parallel execution of multiple AI agents.

Use Cases: - **Consensus Planning** - 3+ agents agree on architecture decisions - **Quality Gates** - Multiple agents validate test strategies - **Diverse Perspectives** - Combine strengths of different models

Configuration: `[[agents]]` array in `config.toml`

Agent Configuration Schema

Agent Fields

```
        [[agents]]
        name = "gemini"                    # Display name
        canonical_name = "gemini"          # Canonical identifier (for
quality gates)
        command = "gemini"                 # Executable command
        args = []                          # Command arguments
        read_only = false                  # Force read-only mode
        enabled = true                     # Enable/disable agent
        description = "Google Gemini"      # Human-readable description
        env = {}                           # Environment variables
        args_read_only = []                # Args for read-only mode
(optional)
        args_write = []                    # Args for write mode (optional)
        instructions = ""                  # Per-agent instructions
(optional)
```

Default Agent Configuration

5-Agent Setup

```toml
# ~/.code/config.toml

#
============================================================================

# Agent 1: Gemini (Fast, Cheap Consensus)
#
============================================================================


[[agents]]
name = "gemini"
canonical_name = "gemini"
command = "gemini"
args = []
read_only = false
enabled = true
description = "Google Gemini Flash - Fast consensus agent (12.5x
cheaper than GPT-5)"

#
============================================================================

# Agent 2: Claude (Balanced Reasoning)
#
============================================================================


[[agents]]
name = "claude"
canonical_name = "claude"
command = "claude"
args = []
read_only = false
enabled = true
description = "Anthropic Claude Haiku - Balanced reasoning (12x
cheaper than GPT-5)"

#
============================================================================

# Agent 3: Code (Strategic Planning)
#
============================================================================


[[agents]]
name = "code"
canonical_name = "gpt_pro"
command = "code"
args = ["--model", "gpt-5"]
read_only = false
enabled = true
description = "OpenAI GPT-5 - Strategic planning and complex
reasoning"

#
============================================================================

# Agent 4: GPT-Codex (Code Generation)
```

```
# ============================================================================

[[agents]]
name = "gpt_codex"
canonical_name = "gpt_codex"
command = "code"
args = ["--model", "gpt-5-codex"]
read_only = false
enabled = true
description = "OpenAI GPT-5-Codex - Specialized code generation"

# ============================================================================

# Agent 5: GPT-Pro (Premium Reasoning)
# ============================================================================

[[agents]]
name = "gpt_pro"
canonical_name = "gpt_pro"
command = "code"
args = ["--model", "o3", "--config", "model_reasoning_effort=high"]
read_only = false
enabled = false  # Disabled by default (premium cost)
description = "OpenAI o3 - Premium reasoning for critical decisions"
```

Agent Properties

name vs canonical_name

`name`: Display name, can change

`canonical_name`: Stable identifier used in quality gates

Example:

```
[[agents]]
name = "claude-sonnet"          # Display name (can evolve)
canonical_name = "claude"       # Canonical name (stable)
command = "anthropic"
```

Quality gate reference:

```
[quality_gates]
plan = ["claude"]  # Uses canonical_name, not name
```

Benefit: Can rename display names without breaking quality gate configs

---

read_only Flag

Purpose: Force agent to run in read-only mode

Use Case: Agents that should never write files

Example:

```toml
[[agents]]
name = "readonly-advisor"
canonical_name = "advisor"
command = "gemini"
read_only = true  # Never allow writes
enabled = true
```

---

enabled Flag

Purpose: Temporarily disable agent without removing config

Use Case: Testing, cost control, debugging

Example:

```toml
[[agents]]
name = "gpt_pro"
canonical_name = "gpt_pro"
command = "code"
args = ["--model", "o3"]
enabled = false  # Disable premium agent to save cost
```

Advanced Agent Configuration

args_read_only vs args_write

Purpose: Different arguments for read vs write modes

Example:

```toml
[[agents]]
name = "claude"
canonical_name = "claude"
command = "anthropic"
args = []  # Default args

# Read-only mode: Use faster, cheaper model
args_read_only = ["--model", "claude-3-haiku"]

# Write mode: Use more capable model
args_write = ["--model", "claude-3-5-sonnet"]
```

Behavior: Automatically selects appropriate args based on operation mode

---

Environment Variables

Purpose: Pass environment variables to agent process

Example:

```toml
[[agents]]
```

```
name = "custom-agent"
canonical_name = "custom"
command = "/path/to/agent"
args = []
env = {
    LOG_LEVEL = "debug",
    CUSTOM_CONFIG = "/path/to/config.json",
    FEATURE_FLAGS = "experimental"
}
```

Use Case: Custom agents, debugging, feature flags

Per-Agent Instructions

Purpose: Prepend instructions to every prompt sent to this agent

Example:

```
[[agents]]
name = "security-focused"
canonical_name = "security"
command = "claude"
args = []
instructions = """
You are a security-focused code reviewer. Always prioritize:
1. Input validation and sanitization
2. Authentication and authorization checks
3. Secure cryptographic practices
4. Protection against OWASP Top 10 vulnerabilities

Flag any potential security issues with HIGH severity.
"""
```

Subagent Commands

Default Commands

The spec-kit framework provides **13 slash commands** that use agents:

Native (Tier 0 - Zero agents, FREE): - `/speckit.new` - SPEC creation (template-based, no agents) - `/speckit.clarify` - Ambiguity detection (heuristics) - `/speckit.analyze` - Consistency checking (structural diff) - `/speckit.checklist` - Quality scoring (rubric) - `/speckit.status` - Status dashboard (native)

Single-Agent (Tier 1 - 1 agent, ~$0.10): - `/speckit.specify` - PRD drafting (gpt5-low) - `/speckit.tasks` - Task decomposition (gpt5-low)

Multi-Agent (Tier 2 - 2-3 agents, ~$0.35): - `/speckit.plan` - Architectural planning (gemini-flash, claude-haiku, gpt5-medium) - `/speckit.validate` - Test strategy (gemini-flash, claude-haiku, gpt5-medium) - `/speckit.implement` - Code generation (gpt_codex HIGH, claude-haiku validator)

Premium (Tier 3 - 3 premium agents, ~$0.80): - `/speckit.audit` - Compliance/security (gemini-pro, claude-sonnet, gpt5-high) - `/speckit.unlock` - Ship decision (gemini-pro, claude-sonnet, gpt5-high)

Full Pipeline (Tier 4 - Strategic routing, ~$2.70): - `/speckit.auto` - Full 6-stage pipeline

---

Subagent Command Configuration

Table Format: `[[subagents.commands]]`

```
        [[subagents.commands]]
        name = "plan"                # Command name (/speckit.plan)
        read_only = true             # Force read-only mode
        agents = ["gemini", "claude", "code"]  # Agents to use
        orchestrator_instructions = "Focus on architectural decisions and
trade-offs."
        agent_instructions = "Provide detailed reasoning for all
recommendations."
```

Fields: - `name` (string): Command name (matches `/speckit.<name>`) - `read_only` (boolean): Force read-only mode (default: command-specific) - `agents` (array): Agent names to enable (default: all enabled agents) - `orchestrator_instructions` (string): Extra instructions for orchestrator - `agent_instructions` (string): Instructions appended to each agent prompt

---

Custom Subagent Command

Example: Add custom consensus command

```
        # ~/.code/config.toml

        [[subagents.commands]]
        name = "review"  # Creates /speckit.review command
        read_only = true
        agents = ["claude", "gpt_pro"]
        orchestrator_instructions = """
        Focus on:
        1. Code quality and maintainability
        2. Performance implications
        3. Security concerns
        4. Test coverage adequacy
        """
        agent_instructions = """
        Provide specific, actionable feedback with code examples.
        """
```

Usage:

```
        /speckit.review SPEC-KIT-065
```

Quality Gate Integration

Agent Selection for Quality Gates

Quality gates reference agents by **canonical_name**:

```toml
# Agents configuration
[[agents]]
name = "gemini-flash"
canonical_name = "gemini"  # ← Used in quality gates
# ...

[[agents]]
name = "claude-haiku"
canonical_name = "claude"  # ← Used in quality gates
# ...

# Quality gates configuration
[quality_gates]
plan = ["gemini", "claude", "code"]  # Uses canonical_name
tasks = ["gemini"]
validate = ["gemini", "claude", "code"]
```

Validation: Config loader checks that all quality gate agents exist

---

Agent Cost Tiers

Cost Comparison

Based on OpenAI GPT-5 baseline (1.0x):

Agent	Model	Cost per 1M tokens	Relative Cost
`gemini`	Gemini Flash	$0.40	12.5x cheaper
`claude`	Claude Haiku	$0.40	12x cheaper
`code`	GPT-5	$5.00	1.0x (baseline)
`gpt_codex`	GPT-5-Codex	$5.00	1.0x
`gpt_pro`	o3 (high effort)	$20.00	4x more expensive

Strategic Agent Routing

SPEC-KIT-070: Cost optimization via tiered agent selection

Principle: "Agents for reasoning, NOT transactions"

Tier 0 (Native): Pattern matching → FREE

```
# No agents needed for:
- /speckit.new (template-based SPEC-ID generation)
- /speckit.clarify (regex-based ambiguity detection)
- /speckit.analyze (structural consistency checking)
```

Tier 1 (Single Agent): Simple reasoning → $0.10

```toml
[[subagents.commands]]
name = "specify"
agents = ["gpt5-low"]  # Single cheap agent
```

Tier 2 (Multi-Agent): Complex decisions → $0.35

```toml
[quality_gates]
plan = ["gemini", "claude", "gpt5-medium"]  # 3 agents, diverse perspectives
```

Tier 3 (Premium): Critical decisions → $0.80

```toml
[quality_gates]
unlock = ["gemini-pro", "claude-sonnet", "gpt5-high"]  # Quality over cost
```

Example Configurations

Minimal (Single Agent)

```toml
[[agents]]
name = "gemini"
canonical_name = "gemini"
command = "gemini"
args = []
enabled = true
```

Use Case: Cost-conscious setup, simple tasks

Balanced (3 Agents)

```toml
# Cheap consensus
[[agents]]
name = "gemini"
canonical_name = "gemini"
command = "gemini"

# Balanced reasoning
[[agents]]
name = "claude"
canonical_name = "claude"
command = "claude"

# Strategic planning
[[agents]]
name = "code"
canonical_name = "gpt_pro"
command = "code"
args = ["--model", "gpt-5"]

[quality_gates]
plan = ["gemini", "claude", "gpt_pro"]
tasks = ["gemini"]
validate = ["gemini", "claude", "gpt_pro"]
```

Use Case: Most production workloads

Premium (5 Agents + Specialist)

```toml
# Full 5-agent setup with premium reasoning
[[agents]]
name = "gemini"
canonical_name = "gemini"
command = "gemini"
enabled = true

[[agents]]
name = "claude"
canonical_name = "claude"
command = "claude"
enabled = true

[[agents]]
name = "code"
canonical_name = "gpt_pro"
command = "code"
args = ["--model", "gpt-5"]
enabled = true

[[agents]]
name = "gpt_codex"
canonical_name = "gpt_codex"
command = "code"
args = ["--model", "gpt-5-codex"]
enabled = true

[[agents]]
name = "gpt_pro"
canonical_name = "gpt_pro"
command = "code"
args = ["--model", "o3", "--config", "model_reasoning_effort=high"]
enabled = true  # Enable for critical decisions

[quality_gates]
plan = ["gemini", "claude", "gpt_pro"]
tasks = ["gemini"]
validate = ["gemini", "claude", "gpt_pro"]
audit = ["gemini", "claude", "gpt_codex", "gpt_pro"]  # 4 agents for
security
unlock = ["gemini", "claude", "gpt_pro"]
```

Use Case: Critical projects, maximum quality

Debugging Agent Configuration

List Configured Agents

```bash
code --agents-list
```

Output:

```
Configured Agents (5):
  [✓] gemini       - Google Gemini Flash (enabled)
  [✓] claude       - Anthropic Claude Haiku (enabled)
  [✓] code         - OpenAI GPT-5 (enabled)
  [✓] gpt_codex    - OpenAI GPT-5-Codex (enabled)
  [✗] gpt_pro      - OpenAI o3 (disabled)
```

Validate Agent Commands

```
code --check-agents
```

Output:

```
Checking agent commands...
[✓] gemini: command 'gemini' found
[✓] claude: command 'claude' found
[✓] code: command 'code' found
[✓] gpt_codex: command 'code' found
[✗] gpt_pro: command 'code' found, but agent disabled

All enabled agents have valid commands.
```

Best Practices

1. Use Canonical Names Consistently

Good:

```toml
[[agents]]
canonical_name = "gemini"  # Stable

[quality_gates]
plan = ["gemini"]  # Matches canonical_name
```

Bad:

```toml
[[agents]]
name = "gemini-flash-2024"  # Display name

[quality_gates]
plan = ["gemini-flash-2024"]  # ✗ Breaks if name changes
```

---

2. Enable Minimum Required Agents

Good:

```toml
# Enable only what you need
[[agents]]
canonical_name = "gemini"
enabled = true

[[agents]]
canonical_name = "claude"
enabled = true

[[agents]]
canonical_name = "gpt_pro"
enabled = false  # Disable premium agent unless needed
```

---

3. Use args_read_only for Cost Savings

Example:

```
        [[agents]]
        name = "claude"
        canonical_name = "claude"
        command = "anthropic"
        args_read_only = ["--model", "claude-3-haiku"]  # Cheap for read-
only
        args_write = ["--model", "claude-3-5-sonnet"]   # Capable for writes
```

---

4. Leverage Per-Agent Instructions

Example:

```
        [[agents]]
        name = "security-agent"
        canonical_name = "security"
        command = "claude"
        instructions = "Focus on security. Flag OWASP Top 10
vulnerabilities."
```

Summary

Agent Configuration covers: - 5-agent default setup (gemini, claude, code, gpt_codex, gpt_pro) - Agent properties (name, canonical_name, command, args, enabled) - Advanced features (args_read_only, env, instructions) - Subagent commands (13 built-in commands) - Quality gate integration - Cost tiers (Tier 0-4, $0 to $0.80 per stage)

Best Practices: - Use canonical_name for stability - Enable minimum required agents - Leverage args_read_only for cost savings - Use per-agent instructions for specialization

Next: Quality Gate Customization

Configuration Reference

Complete `config.toml` schema reference.

Overview

Location: `~/.code/config.toml`

Alternative: `~/.codex/config.toml` (legacy, read-only)

Format: TOML (Tom's Obvious, Minimal Language)

Validation: Schema validation on load, old config preserved on error

---

File Structure

Minimal Example

```toml
# ~/.code/config.toml (minimal)

model = "gpt-5"
model_provider = "openai"
approval_policy = "on-request"
```

Complete Example

```toml
# ~/.code/config.toml (comprehensive)

#
============================================================================

# Model Configuration
#
============================================================================


model = "gpt-5"
model_provider = "openai"
model_reasoning_effort = "medium"  # minimal, low, medium, high
model_reasoning_summary = "auto"   # auto, concise, detailed, none
model_verbosity = "medium"         # low, medium, high (GPT-5 only)
model_context_window = 128000      # Override context window size
model_max_output_tokens = 16384    # Override max output tokens
model_supports_reasoning_summaries = false

#
============================================================================

# Model Providers
#
============================================================================


[model_providers.openai]
name = "OpenAI"
base_url = "https://api.openai.com/v1"
env_key = "OPENAI_API_KEY"
wire_api = "responses"  # or "chat"
request_max_retries = 4
stream_max_retries = 10
stream_idle_timeout_ms = 300000  # 5 minutes

[model_providers.anthropic]
name = "Anthropic"
base_url = "https://api.anthropic.com"
env_key = "ANTHROPIC_API_KEY"
wire_api = "chat"

[model_providers.google]
name = "Google"
base_url = "https://generativelanguage.googleapis.com/v1beta"
env_key = "GOOGLE_API_KEY"
wire_api = "chat"

[model_providers.ollama]
```

```toml
name = "Ollama"
base_url = "http://localhost:11434/v1"
# No env_key needed for local Ollama

#
==============================================================================

# Agents (Multi-Agent Configuration)
#
==============================================================================


[[agents]]
name = "gemini"
canonical_name = "gemini"
command = "gemini"
args = []
read_only = false
enabled = true
description = "Google Gemini Flash (fast, cheap consensus)"

[[agents]]
name = "claude"
canonical_name = "claude"
command = "claude"
args = []
read_only = false
enabled = true
description = "Anthropic Claude Haiku (balanced reasoning)"

[[agents]]
name = "code"
canonical_name = "gpt_pro"
command = "code"
args = ["--model", "gpt-5"]
read_only = false
enabled = true
description = "OpenAI GPT-5 (strategic planning)"

[[agents]]
name = "gpt_codex"
canonical_name = "gpt_codex"
command = "code"
args = ["--model", "gpt-5-codex"]
read_only = false
enabled = true
description = "OpenAI GPT-5-Codex (code generation)"

#
==============================================================================

# Quality Gates (Spec-Kit Framework)
#
==============================================================================


[quality_gates]
plan = ["gemini", "claude", "code"]        # Multi-agent planning
tasks = ["gemini"]                          # Single-agent task
breakdown
```

```
        validate = ["gemini", "claude", "code"]    # Multi-agent test
validation
        audit = ["gemini", "claude", "gpt_codex"]  # Security/compliance
review
        unlock = ["gemini", "claude", "gpt_codex"] # Ship decision

        #
==============================================================================

        # Hot-Reload Configuration
        #
==============================================================================


        [hot_reload]
        enabled = true
        debounce_ms = 2000  # Wait 2s after last change before reloading
        watch_paths = ["config.toml"]  # Additional paths to watch

        #
==============================================================================

        # Validation Configuration
        #
==============================================================================


        [validation]
        check_api_keys = true      # Validate API keys on startup
        check_commands = true      # Validate agent commands exist
        strict_schema = true       # Enforce strict TOML schema
        patch_harness = false      # Run patch validation harness

        [validation.groups]
        functional = true   # Functional checks (cargo, tsc, etc.)
        stylistic = false   # Stylistic checks (prettier, shfmt)

        [validation.tools]
        shellcheck = true
        cargo-check = true
        # ... other tools (see Validation section below)

        #
==============================================================================

        # Approval Policy
        #
==============================================================================


        approval_policy = "on-request"  # untrusted, on-failure, on-request,
never

        #
==============================================================================

        # Confirm Guard (Destructive Commands)
        #
==============================================================================
```

```
        [[confirm_guard.patterns]]
        regex = "(?i)^\\s*git\\s+reset\\b"
        message = "Blocked git reset. Reset rewrites the working
tree/index."

        [[confirm_guard.patterns]]
        regex = "(?i)^\\s*(?:sudo\\s+)?rm\\s+-[a-z-]*rf[a-z-]*\\s+"
        message = "Blocked rm -rf. Force-recursive delete requires
confirmation."

        #
================================================================================

        # Sandbox Configuration
        #
================================================================================


        sandbox_mode = "workspace-write"  # read-only, workspace-write,
danger-full-access

        [sandbox_workspace_write]
        exclude_tmpdir_env_var = false
        exclude_slash_tmp = false
        writable_roots = []  # Additional writable paths
        network_access = false
        allow_git_writes = true  # Allow .git/ folder writes

        #
================================================================================

        # Shell Environment Policy
        #
================================================================================


        [shell_environment_policy]
        inherit = "all"  # all, core, none
        ignore_default_excludes = false  # If true, include *KEY*, *TOKEN*
vars
        exclude = ["AWS_*", "AZURE_*"]   # Additional exclusion patterns
        set = { CI = "1" }                # Force-set environment variables
        include_only = []                 # If non-empty, only these
patterns survive

        #
================================================================================

        # MCP Servers
        #
================================================================================


        [mcp_servers.local-memory]
        command = "npx"
        args = ["-y", "@modelcontextprotocol/server-memory"]
        startup_timeout_ms = 10000  # 10 seconds

        [mcp_servers.git-status]
```

```
        command = "npx"
        args = ["-y", "@just-every/mcp-server-git"]
        env = { LOG_LEVEL = "info" }

        #
================================================================================

        # ACE (Agentic Context Engine)
        #
================================================================================


        [ace]
        enabled = true
        mode = "auto"  # auto, always, never
        slice_size = 8  # Max 8 playbook bullets
        db_path = "~/.code/ace/playbooks_normalized.sqlite3"
        use_for = ["speckit.constitution", "speckit.specify",
"speckit.tasks"]
        complex_task_files_threshold = 4
        rerun_window_minutes = 30

        #
================================================================================

        # TUI Configuration
        #
================================================================================


        [tui]
        alternate_screen = true        # Use alternate screen mode
        show_reasoning = false         # Show reasoning content by default

        [tui.theme]
        name = "dark-carbon-night"     # See Theme section for all themes
        # Optional custom color overrides
        colors = {}

        [tui.highlight]
        theme = "auto"  # auto, or specific syntect theme

        [tui.stream]
        answer_header_immediate = false
        show_answer_ellipsis = true
        commit_tick_ms = 50
        soft_commit_timeout_ms = 400
        soft_commit_chars = 160
        relax_list_holdback = false
        relax_code_holdback = false
        responsive = false  # Enable snappier preset

        [tui.spinner]
        name = "diamond"  # Spinner style from cli-spinners

        [tui.notifications]
        # false (disabled), true (all), or array of specific notifications
        notifications = false

        #
```

```toml
# ============================================================================
# History Configuration
# ============================================================================

[history]
persistence = "save-all"  # save-all, none
max_bytes = 10485760      # 10 MB (not currently enforced)

# ============================================================================
# Browser Configuration (Screenshot Tool)
# ============================================================================

[browser]
enabled = false
fullpage = true
segments_max = 10
idle_timeout_ms = 30000
format = "png"  # png, webp

[browser.viewport]
width = 1280
height = 720
device_scale_factor = 2.0
mobile = false

[browser.wait]
delay_ms = 1000  # Wait 1s before screenshot

# ============================================================================
# GitHub Integration
# ============================================================================

[github]
check_workflows_on_push = true
actionlint_on_patch = false
actionlint_strict = false

# ============================================================================
# Project Hooks
# ============================================================================

[[project_hooks]]
event = "session.start"
name = "install-deps"
command = ["npm", "install"]
```

```toml
timeout_ms = 60000

[[project_hooks]]
event = "file.after_write"
command = ["cargo", "fmt", "--all"]

#
================================================================================

# Profiles
#
================================================================================


profile = "default"  # Active profile

[profiles.premium]
model = "o3"
model_reasoning_effort = "high"
model_reasoning_summary = "detailed"
approval_policy = "never"

[profiles.fast]
model = "gpt-4o-mini"
model_reasoning_effort = "low"
approval_policy = "never"

[profiles.ci]
model = "gpt-4o"
approval_policy = "never"
sandbox_mode = "read-only"
disable_response_storage = false

#
================================================================================

# Miscellaneous
#
================================================================================


disable_response_storage = false  # Required for ZDR accounts
file_opener = "vscode"  # vscode, vscode-insiders, cursor, windsurf,
none
hide_agent_reasoning = false
show_raw_agent_reasoning = false
project_doc_max_bytes = 32768  # 32 KiB
notify = []  # Command to execute for notifications
```

Configuration Sections

Model Configuration

Field	Type	Default	Descriptio
`model`	string	`"gpt-5"`	Model name use
			Provider ID

`model_provider`	string	`"openai"`	from `model_provid`
`model_reasoning_effort`	string	`"medium"`	Reasoning effort: minim low, medium high
`model_reasoning_summary`	string	`"auto"`	Summary mode: auto, concise, detailed, nor
`model_verbosity`	string	`"medium"`	Verbosity lev low, medium high (GPT-5 only)
`model_context_window`	integer	`128000`	Context window size tokens
`model_max_output_tokens`	integer	`16384`	Max output tokens
`model_supports_reasoning_summaries`	boolean	`false`	Force reasoning support

Model Providers

Table Format: `[model_providers.<id>]`

Required Fields: - `name` (string): Display name - `base_url` (string): API base URL - `env_key` (string, optional): Environment variable for API key

Optional Fields: - `wire_api` (string): `"chat"` or `"responses"` (default: `"chat"`) - `query_params` (table): Additional query parameters (e.g., Azure `api-version`) - `http_headers` (table): Static HTTP headers - `env_http_headers` (table): HTTP headers from environment variables - `request_max_retries` (integer): HTTP request retries (default: 4) - `stream_max_retries` (integer): SSE stream retries (default: 10) - `stream_idle_timeout_ms` (integer): Idle timeout in ms (default: 300000)

Example:

```toml
[model_providers.azure]
name = "Azure OpenAI"
base_url = "https://YOUR_PROJECT.openai.azure.com/openai"
env_key = "AZURE_OPENAI_API_KEY"
query_params = { api-version = "2025-04-01-preview" }
```

Agents

Array Format: `[[agents]]`

Field	Type	Required	Description
`name`	string	Yes	Agent name (display)
			Canonical identifier

`canonical_name`	string	No	(default: same as `name`)
`command`	string	Yes	Command to execute
`args`	array	No	Command arguments
`read_only`	boolean	No	Force read-only mode (default: false)
`enabled`	boolean	No	Enable agent (default: true)
`description`	string	No	Agent description
`env`	table	No	Environment variables
`args_read_only`	array	No	Args for read-only mode
`args_write`	array	No	Args for write mode
`instructions`	string	No	Per-agent instructions

Quality Gates

Table Format: `[quality_gates]`

Field	Type	Default	Description
`plan`	array	`[]`	Agent names for plan stage
`tasks`	array	`[]`	Agent names for tasks stage
`validate`	array	`[]`	Agent names for validate stage
`audit`	array	`[]`	Agent names for audit stage
`unlock`	array	`[]`	Agent names for unlock stage

Agent names must match `canonical_name` from `[[agents]]`.

Hot-Reload

Table Format: `[hot_reload]`

Field	Type	Default	Description
`enabled`	boolean	`true`	Enable hot-reload
`debounce_ms`	integer	`2000`	Debounce window in ms (default: 2s)
`watch_paths`	array	`[]`	Additional paths to watch

Validation

Table Format: `[validation]`

Field	Type	Default	Description
`check_api_keys`	boolean	`true`	Validate API keys on startup
`check_commands`	boolean	`true`	Validate agent commands exist
`strict_schema`	boolean	`true`	Enforce strict TOML schema
`patch_harness`	boolean	`false`	Run patch validation harness

`tools_allowlist`	array	`null`	Restrict allowed tools
`timeout_seconds`	integer	`null`	Tool execution timeout

Groups (`[validation.groups]`): - `functional` (boolean): Functional checks (cargo, tsc, eslint) - `stylistic` (boolean): Stylistic checks (prettier, shfmt)

Tools (`[validation.tools]`): - `shellcheck`, `markdownlint`, `hadolint`, `yamllint` (stylistic) - `cargo-check`, `tsc`, `eslint`, `mypy`, `pyright`, `golangci-lint` (functional) - `shfmt`, `prettier` (stylistic)

Sandbox Configuration

Field	Type	Default	Description
`sandbox_mode`	string	`"read-only"`	Sandbox mode: read-only, workspace-write, danger-full-access

`[sandbox_workspace_write]` (only applies when `sandbox_mode = "workspace-write"`):

Field	Type	Default	Description
`exclude_tmpdir_env_var`	boolean	`false`	Exclude `$TMPDIR` from writable roots
`exclude_slash_tmp`	boolean	`false`	Exclude `/tmp` from writable roots
`writable_roots`	array	`[]`	Additional writable paths
`network_access`	boolean	`false`	Allow network access
`allow_git_writes`	boolean	`true`	Allow `.git/` folder writes

MCP Servers

Table Format: `[mcp_servers.<name>]`

Field	Type	Required	Description
`command`	string	Yes	Command to execute
`args`	array	No	Command arguments
`env`	table	No	Environment variables
`startup_timeout_ms`	integer	No	Startup timeout in ms (default: 10000)

Example:

```toml
[mcp_servers.custom-tool]
command = "/path/to/mcp-server"
args = ["--port", "8080"]
```

```
env = { API_KEY = "secret" }
startup_timeout_ms = 15000
```

TUI Configuration

Theme (`[tui.theme]`): - `name` (string): Theme name (see Theme System guide) - `colors` (table): Custom color overrides - `label` (string, optional): Custom theme label - `is_dark` (boolean, optional): Dark theme hint

Highlight (`[tui.highlight]`): - `theme` (string): Syntax highlighting theme (default: “auto”)

Stream (`[tui.stream]`): - `answer_header_immediate` (boolean): Show header immediately - `show_answer_ellipsis` (boolean): Show ellipsis while waiting - `commit_tick_ms` (integer): Animation commit rate (default: 50ms) - `soft_commit_timeout_ms` (integer): Soft-commit timeout - `soft_commit_chars` (integer): Soft-commit character threshold - `relax_list_holdback` (boolean): Relax list marker hold-back - `relax_code_holdback` (boolean): Relax code block hold-back - `responsive` (boolean): Enable snappier preset

---

Profiles

Table Format: `[profiles.<name>]`

Profiles can override any top-level config field. See Precedence System for details.

Example:

```
[profiles.premium]
model = "o3"
model_reasoning_effort = "high"
approval_policy = "never"
```

Activation: Set `profile = "premium"` or use `--profile premium` flag.

Validation Rules

Required Fields

None - All fields have defaults

Type Validation

- Strings: Non-empty (whitespace trimmed)
- Integers: Must be positive (where applicable)
- Booleans: `true` or `false`
- Arrays: Can be empty unless semantically invalid

Semantic Validation

1. **Model provider must exist**: `model_provider` must be a key in

`model_providers`
2. **Quality gate agents must exist**: Agent names in `quality_gates.*` must match `canonical_name` in `[[agents]]`
3. **Evidence size must be reasonable**: `evidence.max_size_mb` ≤ 1000
4. **Debounce must be reasonable**: `hot_reload.debounce_ms` ≥ 100

Error Handling

On validation failure: 1. Old config is **preserved** (no reload) 2. `ReloadFailed` event emitted with error message 3. TUI shows notification with error details

Example error:

```
Config validation failed: Agent 'unknown-agent' not found in
quality_gates.plan
Old config preserved.
```

Summary

Config File: `~/.code/config.toml`

Sections: 20+ configuration sections covering: - Model/provider configuration - Multi-agent setup - Quality gates - Hot-reload settings - Validation rules - Sandbox policy - MCP servers - TUI customization - Profiles

Validation: Schema validation with old config preservation on error

Next: Precedence System

Environment Variables

Complete reference for all environment variables and override behavior.

Overview

Environment variables provide **Tier 2 precedence** (higher than config.toml, lower than CLI flags).

Use Cases: - API keys and secrets - Environment-specific overrides (dev, staging, production) - CI/CD configuration - Temporary configuration changes

---

Core Environment Variables

CODEX_HOME / CODE_HOME

Purpose: Installation directory

Default: `~/.code`

Legacy: `~/.codex` (read-only, deprecated)

Usage:

```
export CODEX_HOME="/custom/path"
# or
export CODE_HOME="/custom/path"
```

Precedence: `CODE_HOME` > `CODEX_HOME` > `~/.code`

Files Stored:

```
$CODEX_HOME/
├── config.toml          # Configuration file
├── history.jsonl        # Session history
├── debug.log            # Debug logs
├── mcp-memory/          # MCP memory database
├── mcp-cache/           # MCP tool cache
└── ace/                 # ACE playbook database
    └── playbooks_normalized.sqlite3
```

API Keys

OPENAI_API_KEY

Purpose: OpenAI API authentication

Required: When using `model_provider = "openai"`

Usage:

```
export OPENAI_API_KEY="sk-proj-..."
```

Security: Never commit to git, never store in config.toml

ANTHROPIC_API_KEY

Purpose: Anthropic API authentication

Required: When using `model_provider = "anthropic"`

Usage:

```
export ANTHROPIC_API_KEY="sk-ant-..."
```

GOOGLE_API_KEY

Purpose: Google Gemini API authentication

Required: When using `model_provider = "google"`

Usage:

```
export GOOGLE_API_KEY="..."
```

AZURE_OPENAI_API_KEY

Purpose: Azure OpenAI API authentication

Required: When using Azure model provider

Usage:

```
export AZURE_OPENAI_API_KEY="..."
```

Alternative: `OPENAI_API_KEY` also works for Azure

Custom Provider API Keys

Pattern: `<PROVIDER_NAME>_API_KEY`

Example:

```
[model_providers.custom]
env_key = "CUSTOM_API_KEY"

export CUSTOM_API_KEY="..."
```

Model Configuration Overrides

CODEX_MODEL

Purpose: Override default model

Precedence: Env var > config.toml

Usage:

```
export CODEX_MODEL="o3"
code "task"
```

Equivalent:

```
code --model o3 "task"
```

CODEX_PROVIDER

Purpose: Override model provider

Usage:

```
export CODEX_PROVIDER="anthropic"
code "task"
```

Equivalent:

```
code --config model_provider=anthropic "task"
```

OPENAI_BASE_URL

Purpose: Override OpenAI base URL

Use Case: Custom proxy, Azure, local endpoint

Usage:

```
export OPENAI_BASE_URL="https://custom.openai.com/v1"
```

Overrides: `model_providers.openai.base_url`

OPENAI_WIRE_API

Purpose: Force OpenAI wire protocol

Options: `"responses"` or `"chat"`

Usage:

```
export OPENAI_WIRE_API="chat"  # Force chat completions
```

Overrides: `model_providers.openai.wire_api`

Spec-Kit Environment Variables

SPEC_OPS_CARGO_MANIFEST

Purpose: Override cargo manifest path for workspace commands

Default: Auto-detected (`codex-rs/Cargo.toml`)

Usage:

```
export SPEC_OPS_CARGO_MANIFEST="/path/to/Cargo.toml"
```

SPEC_OPS_ALLOW_DIRTY

Purpose: Allow guardrail commands with dirty git tree

Default: `0` (require clean tree)

Usage:

```
export SPEC_OPS_ALLOW_DIRTY=1
/guardrail.auto SPEC-KIT-065
```

Use Case: Testing, development iteration

SPEC_OPS_TELEMETRY_HAL

Purpose: Enable HAL telemetry collection

Default: `0` (disabled)

Usage:

```
export SPEC_OPS_TELEMETRY_HAL=1
/guardrail.plan SPEC-KIT-065
```

Output: Captures `hal.summary.{status,failed_checks,artifacts}` in telemetry

SPEC_OPS_HAL_SKIP

Purpose: Skip HAL validation (when secrets unavailable)

Default: `0` (run HAL validation)

Usage:

```
export SPEC_OPS_HAL_SKIP=1
/guardrail.audit SPEC-KIT-065
```

Use Case: Development without HAL secrets

SPECKIT_QUALITY_GATES_*

Purpose: Override quality gate agent selection

Pattern: `SPECKIT_QUALITY_GATES_<STAGE>=agent1,agent2,agent3`

Usage:

```
export SPECKIT_QUALITY_GATES_PLAN="gemini,claude,code,gpt_pro"
export SPECKIT_QUALITY_GATES_TASKS="code"
export SPECKIT_QUALITY_GATES_VALIDATE="gemini,claude,code"
export SPECKIT_QUALITY_GATES_AUDIT="gemini,claude,gpt_codex,gpt_pro"
export SPECKIT_QUALITY_GATES_UNLOCK="gemini,claude,gpt_pro"
```

Precedence: Env var > config.toml

Logging and Debugging

RUST_LOG

Purpose: Rust logging level

Options: `error`, `warn`, `info`, `debug`, `trace`

Usage:

```
export RUST_LOG=debug
code
```

Module-Specific:

```
export RUST_LOG=codex_tui::chatwidget::spec_kit=debug
code
```

Multiple Modules:

```
export RUST_LOG=codex_mcp_client=debug,codex_spec_kit=trace
code
```

RUST_BACKTRACE

Purpose: Enable backtraces on panic

Usage:

```
export RUST_BACKTRACE=1  # Short backtrace
export RUST_BACKTRACE=full  # Full backtrace
code
```

Use Case: Debugging crashes

Sandbox and Security

CODEX_SANDBOX_NETWORK_DISABLED

Purpose: Disable network access in sandbox

Auto-Set: When `sandbox_mode = "read-only"` or `sandbox_mode = "workspace-write"` with `network_access = false`

Usage (manual override):

```
export CODEX_SANDBOX_NETWORK_DISABLED=1
```

CI/CD Environment Variables

CI

Purpose: Detect CI environment

Auto-Set: By most CI systems (GitHub Actions, GitLab CI, CircleCI, etc.)

Usage:

```
[shell_environment_policy]
set = { CI = "1" }
```

Effect: Triggers CI-specific behavior (non-interactive mode, strict validation)

GITHUB_ACTIONS

Purpose: Detect GitHub Actions environment

Auto-Set: By GitHub Actions

Usage:

```
if [ "$GITHUB_ACTIONS" = "true" ]; then
  export CODEX_MODEL="gpt-4o"  # Use cheaper model in CI
fi
```

CODEX_AUTO_UPGRADE

Purpose: Enable/disable auto-upgrade

Options: `true`/`false`, `1`/`0`, `yes`/`no`, `on`/`off`

Usage:

```
export CODEX_AUTO_UPGRADE=false  # Disable auto-upgrade in CI
```

Overrides: `auto_upgrade_enabled` in config.toml

Shell Environment Policy

Shell Environment Inheritance

Configuration:

```
[shell_environment_policy]
inherit = "all"  # all, core, none
ignore_default_excludes = false
exclude = ["AWS_*", "AZURE_*"]
set = { CI = "1" }
include_only = []
```

Default Excludes (when `ignore_default_excludes = false`): - `*KEY*` (case-insensitive) - `*TOKEN*` (case-insensitive) - `*SECRET*` (case-insensitive)

Example:

```
# These are excluded by default:
export AWS_ACCESS_KEY="..."       # Excluded (*KEY*)
export GITHUB_TOKEN="..."         # Excluded (*TOKEN*)
export DB_SECRET="..."            # Excluded (*SECRET*)

# These are included (no KEY/TOKEN/SECRET):
export PATH="/usr/bin"            # Included
export HOME="/home/user"          # Included
```

Override Shell Environment Policy

Usage:

```
export SHELL_ENV_INHERIT="core"  # Override inherit mode
export SHELL_ENV_IGNORE_DEFAULT_EXCLUDES="1"  # Include KEY/TOKEN
vars
```

MCP Server Environment Variables

MCP-Specific Variables

Pattern: Set in `env` field of `[mcp_servers.<name>]`

Example:

```
[mcp_servers.database]
command = "/path/to/db-server"
env = {
    DB_HOST = "localhost",
    DB_PORT = "5432",
    DB_NAME = "mydb"
}
```

Scope: Only available to that specific MCP server

Global MCP Environment

Pattern: `MCP_*` prefix

Usage:

```
export MCP_LOG_LEVEL="debug"
export MCP_TIMEOUT="30000"
```

Scope: Available to all MCP servers

HAL Secret Environment Variables

HAL_SECRET_KAVEDARR_API_KEY

Purpose: Kavedarr API key for HAL validation

Required: When running HAL smoke tests or policy validation

Usage:

```
export HAL_SECRET_KAVEDARR_API_KEY="..."
```

Security: Never commit, never store in config

Testing Environment Variables

PRECOMMIT_FAST_TEST

Purpose: Skip test compilation in pre-commit hook

Default: `1` (skip test compilation)

Usage:

```
export PRECOMMIT_FAST_TEST=0  # Run test compilation
git commit
```

PREPUSH_FAST

Purpose: Skip pre-push hooks

Default: `1` (run hooks)

Usage:

```
export PREPUSH_FAST=0  # Skip pre-push hooks
git push
```

Warning: Only use for emergencies

Complete Environment Variable Reference

Core Variables

Variable	Purpose	Default	Example
`CODEX_HOME`	Installation directory	`~/.code`	`/custom/path`
`CODE_HOME`	Alt. installation directory	(uses CODEX_HOME)	`/custom/path`
`RUST_LOG`	Logging level	`info`	`debug`
`RUST_BACKTRACE`	Backtrace on panic	`0`	`1`, `full`

API Keys

Variable	Purpose	Required For
`OPENAI_API_KEY`	OpenAI authentication	`model_provider = "openai"`
`ANTHROPIC_API_KEY`	Anthropic authentication	`model_provider = "anthropic"`
`GOOGLE_API_KEY`	Google Gemini authentication	`model_provider = "google"`
`AZURE_OPENAI_API_KEY`	Azure OpenAI authentication	Azure model provider
`<PROVIDER>_API_KEY`	Custom provider authentication	Custom providers

Model Overrides

Variable	Overrides	Exampl
`CODEX_MODEL`	`model`	`o3`
`CODEX_PROVIDER`	`model_provider`	`anthropic`
`OPENAI_BASE_URL`	`model_providers.openai.base_url`	`https://custom.`
`OPENAI_WIRE_API`	`model_providers.openai.wire_api`	`chat`, `responses`
`CODEX_AUTO_UPGRADE`	`auto_upgrade_enabled`	`true`, `false`

Spec-Kit Variables

Variable	Purpose	Default	Example
`SPEC_OPS_CARGO_MANIFEST`	Cargo manifest path	Auto-detected	`/path/to/Cargo.toml`
`SPEC_OPS_ALLOW_DIRTY`	Allow dirty git tree	`0`	`1`
`SPEC_OPS_TELEMETRY_HAL`	Enable HAL telemetry	`0`	`1`
`SPEC_OPS_HAL_SKIP`	Skip HAL validation	`0`	`1`
`SPECKIT_QUALITY_GATES_*`	Override quality gate agents	(from config)	`gemini,claude,code`

Sandbox and Security

Variable	Purpose	Auto-Set	Manual Override
`CODEX_SANDBOX_NETWORK_DISABLED`	Disable network in sandbox	Yes (when `network_access = false`)	`1`

CI/CD Variables

Variable	Purpose	Auto-Set By	Example
`CI`	CI environment detection	Most CI systems	`1`, `true`
`GITHUB_ACTIONS`	GitHub Actions detection	GitHub Actions	`true`

Testing Variables

Variable	Purpose	Default	Example
`PRECOMMIT_FAST_TEST`	Skip test compilation in pre-commit	`1`	`0`
`PREPUSH_FAST`	Skip pre-push hooks	`1`	`0`

Best Practices

1. Store Secrets in Environment Variables

Good:

```
export OPENAI_API_KEY="sk-proj-..."
export ANTHROPIC_API_KEY="sk-ant-..."
```

Bad:

```
# DON'T: Never store secrets in config.toml
[model_providers.openai]
api_key = "sk-proj-..."  # ✖ Security risk!
```

---

2. Use .env Files (Local Development)

`.env` file (git-ignored):

```
# .env
OPENAI_API_KEY=sk-proj-...
ANTHROPIC_API_KEY=sk-ant-...
GOOGLE_API_KEY=...
```

Load with direnv:

```
# Install direnv
brew install direnv  # macOS
apt install direnv   # Linux

# Enable for shell
echo 'eval "$(direnv hook bash)"' >> ~/.bashrc

# Allow .envrc
echo 'dotenv' > .envrc
direnv allow
```

---

3. Use Profiles for Environment-Specific Config

config.toml:

```
[profiles.dev]
model = "gpt-4o-mini"
approval_policy = "never"

[profiles.staging]
model = "gpt-5"
approval_policy = "on-request"

[profiles.production]
model = "o3"
approval_policy = "on-failure"
model_reasoning_effort = "high"
```

Usage:

```
# Development
code --profile dev "task"

# Staging
code --profile staging "task"

# Production
code --profile production "task"
```

---

4. Document Required Environment Variables

README.md:

```
## Required Environment Variables

- `OPENAI_API_KEY` - OpenAI API key
- `ANTHROPIC_API_KEY` - Anthropic API key (optional)
- `CODEX_HOME` - Installation directory (optional, default: ~/.code)
```

Debugging Environment Variables

List Active Environment Variables

```
# All CODEX_* and *_API_KEY variables
env | grep -E 'CODEX|API_KEY'
```

Output:

```
CODEX_HOME=/home/user/.code
CODEX_MODEL=gpt-5
OPENAI_API_KEY=sk-proj-***
ANTHROPIC_API_KEY=sk-ant-***
```

Check Effective Configuration

```
code --config-dump | grep -A 5 "# Source:"
```

Output:

```
model = "o3"  # Source: Environment variable (CODEX_MODEL)
model_provider = "openai"  # Source: config.toml
approval_policy = "never"  # Source: Profile 'premium'
```

Summary

Environment Variables provide: - Tier 2 precedence (env var > config.toml) - API key storage (secure, never in config) - Environment-specific overrides (dev, staging, production) - CI/CD configuration - Temporary configuration changes

Categories: - Core (CODEX_HOME, RUST_LOG) - API Keys (OPENAI_API_KEY, ANTHROPIC_API_KEY, GOOGLE_API_KEY) - Model Overrides (CODEX_MODEL, CODEX_PROVIDER) - Spec-Kit

(SPEC_OPS_, *SPECKIT_*) - Sandbox (CODEX_SANDBOX_NETWORK_DISABLED) - CI/CD (CI, GITHUB_ACTIONS) - Testing (PRECOMMIT_FAST_TEST, PREPUSH_FAST)

Best Practices: - Store secrets in environment variables - Use .env files (git-ignored) for local development - Use profiles for environment-specific config - Document required variables in README

Next: Template Customization

Hot-Reload

Config reload mechanism with 300ms debouncing.

Overview

Hot-reload enables configuration changes to apply **without restarting** the application.

Benefits: - Instant config updates (<100ms latency) - No session interruption - Safe validation (old config preserved on error)

Performance: <0.5% CPU overhead, <336ms reload latency (p50)

Architecture

Reload Flow

```
File Change → notify crate → Debouncer (300ms) → Validate → Lock →
Replace → Event
                                                        ↓ Fail
                                                Preserve Old Config
```

Components: 1. **File Watcher** (`notify` crate) - Detects filesystem changes 2. **Debouncer** - Buffers events for 300ms to prevent storms 3. **Validator** - Validates new config (schema, semantic) 4. **Lock** - Atomic config replacement (RwLock) 5. **Event** - Notification to TUI/app

---

Configuration

Enable Hot-Reload

Default: Enabled

```toml
# ~/.code/config.toml

[hot_reload]
```

```toml
enabled = true
debounce_ms = 2000  # Wait 2s after last change
watch_paths = ["config.toml"]  # Additional files to watch
```

Configuration Fields

Field	Type	Default	Description
`enabled`	boolean	`true`	Enable/disable hot-reload
`debounce_ms`	integer	`2000`	Debounce window in milliseconds
`watch_paths`	array	`[]`	Additional paths to watch (relative to `~/.code/`)

Debouncing

Purpose: Prevent reload storms from multiple filesystem events

Example Scenario:

```
t=0ms:    File save event 1 (vim writes temp file)
t=50ms:   File save event 2 (vim renames temp file)
t=100ms:  File save event 3 (vim updates mtime)
t=2100ms: No events for 2000ms → Trigger reload
```

Result: Only **one reload** despite 3 filesystem events

Debounce Tuning

Fast Debounce (impatient users):

```toml
[hot_reload]
debounce_ms = 500  # 500ms (more responsive, more reloads)
```

Slow Debounce (complex editors):

```toml
[hot_reload]
debounce_ms = 5000  # 5s (less responsive, fewer reloads)
```

Recommended: 2000ms (2 seconds)

Watch Additional Files

Example: Watch model provider configs

```toml
[hot_reload]
watch_paths = [
    "config.toml",           # Default
    "models/openai.toml",    # Custom model config
    "models/anthropic.toml", # Custom model config
]
```

Use Case: Split configuration across multiple files

Reload Events

Event Types

```rust
pub enum ConfigReloadEvent {
    /// File change detected (before reload attempt)
    FileChanged(PathBuf),

    /// Config successfully reloaded
    ReloadSuccess,

    /// Reload failed (old config preserved)
    ReloadFailed(String),
}
```

Event Flow

Successful Reload:

```
1. FileChanged(~/.code/config.toml)  # File changed
2. [Debounce wait 2000ms]
3. [Parse TOML: OK]
4. [Validate: OK]
5. [Replace config]
6. ReloadSuccess  # Notify TUI
```

Failed Reload:

```
1. FileChanged(~/.code/config.toml)  # File changed
2. [Debounce wait 2000ms]
3. [Parse TOML: ERROR]
4. ReloadFailed("Invalid TOML: missing closing bracket")
5. [Old config preserved]
```

TUI Notifications

Success:

```
✅ Config reloaded successfully
   - 2 model configs changed
   - Quality gates updated
```

Failure:

```
❌ Config reload failed: Invalid TOML syntax at line 42
   Old configuration preserved.
```

Reload Performance

Latency Breakdown

End-to-end reload latency:

```
File save → Filesystem event → Debounce wait → Parse TOML → Validate
→ Write lock → Event
  0ms          ~10ms              2000ms          ~20ms        ~5ms
<1ms      ~1ms

Total: ~2036ms (p50)
       ~2120ms (p95)
```

Acceptable: Sub-3-second reload for manual config edits

Lock Performance

Read Lock (frequent, fast):

```rust
let config = watcher.get_config();  // Arc::clone
```

Timing:

```
Acquire read lock: <1µs
Clone Arc: <100ns
Release read lock: <100ns

Total: <1µs
```

Concurrency: Multiple readers allowed (RwLock)

Write Lock (rare, fast):

```rust
*config.write().unwrap() = new_config;
```

Timing:

```
Acquire write lock: <500µs (wait for readers to finish)
Replace config: <100ns
Release write lock: <100ns

Total: <1ms
```

Blocking: Briefly blocks readers (<1ms)

CPU Overhead

Idle (file watching):

```
CPU usage: <0.5%
Memory: ~2 MB (notify crate + debouncer)
```

During Reload:

```
CPU spike: ~10-20% for ~50ms (parsing + validation)
Memory spike: ~1 MB (temporary during validation)
```

Validation

Schema Validation

Checks: 1. TOML syntax validity 2. Required fields present 3. Type correctness (string, int, bool, array) 4. Enum values valid

Example Errors:

```
× Invalid TOML: unexpected character ']' at line 42
× Missing required field: model_providers.openai.base_url
× Type mismatch: quality_gates.plan expected array, got string
× Invalid enum value: approval_policy="unknown" (expected:
untrusted, on-failure, on-request, never)
```

Semantic Validation

Checks: 1. Model provider exists 2. Quality gate agents exist and are enabled 3. Evidence size limits reasonable 4. Debounce timing reasonable

Example Errors:

```
× Model provider 'unknown' not found in model_providers
× Quality gate agent 'gpt_pro' not found or disabled
× Evidence max_size_mb=5000 exceeds limit (1000 MB)
× Hot-reload debounce_ms=50 too low (minimum: 100ms)
```

Validation Failure Behavior

On validation failure: 1. **Preserve old config** (no changes applied) 2. **Emit `ReloadFailed` event** with error message 3. **Show TUI notification** with error details 4. **Log error** to `~/.code/debug.log`

User Action: Fix `config.toml` and save again (triggers new reload)

Deferring Reloads

When to Defer

Defer reload if: 1. Quality gate is active (don't interrupt validation) 2. Agents are running (don't interrupt execution) 3. Critical operation in progress (file write, git commit)

Implementation:

```
        pub fn should_defer_reload(quality_gate_active: bool, agent_running:
bool) -> bool {
            quality_gate_active || agent_running
        }
```

Deferred Reload Behavior

Scenario: User edits config while quality gate is running

Behavior:

```
1. FileChanged event received
```

```
2. Check if quality gate active: YES
3. Queue reload for later
4. Quality gate completes
5. Execute queued reload
```

Result: Config reloads after quality gate completes (no interruption)

Change Detection

Detecting Config Changes

Purpose: Show user what changed in TUI notification

Implementation:

```
        pub fn detect_config_changes(old: &AppConfig, new: &AppConfig) ->
(usize, bool, bool) {
            let models_changed = count_model_changes(old, new);
            let quality_gates_changed = old.quality_gates !=
new.quality_gates;
            let cost_changed = old.cost != new.cost;

            (models_changed, quality_gates_changed, cost_changed)
        }
```

Returns: `(models_changed, quality_gates_changed, cost_changed)`

TUI Notification with Changes

Example:

```
✅ Config reloaded successfully
   - 3 model configs changed (openai, anthropic, google)
   - Quality gates updated (plan: 3→2 agents)
   - Cost limits changed ($10/day → $20/day)
```

Debugging Hot-Reload

Enable Debug Logging

```
        export RUST_LOG=codex_spec_kit::config::hot_reload=debug
        code
```

Log Output:

```
[DEBUG] HotReloadWatcher initialized
[DEBUG] Watching: ~/.code/config.toml
[DEBUG] Debounce window: 2000ms
[DEBUG] FileChanged event: ~/.code/config.toml
[DEBUG] Debouncing... (waiting 2000ms)
[DEBUG] Debounce complete, attempting reload
[DEBUG] Parsing TOML: OK
[DEBUG] Validating config: OK
[DEBUG] Acquiring write lock...
```

```
[DEBUG] Write lock acquired (<1ms)
[DEBUG] Config replaced
[DEBUG] ReloadSuccess event emitted
```

Test Hot-Reload

Manual Test:

```
# Terminal 1: Run app with debug logging
export RUST_LOG=debug
code

# Terminal 2: Edit config
vim ~/.code/config.toml
# Make change and save

# Terminal 1: Check logs
[DEBUG] FileChanged event: ~/.code/config.toml
[DEBUG] Debouncing...
[DEBUG] Config reloaded successfully
```

Disable Hot-Reload (Troubleshooting)

```
[hot_reload]
enabled = false  # Disable hot-reload
```

Use Case: Debugging config loading issues, performance profiling

---

Best Practices

1. Use Default Debounce (2000ms)

Recommended:

```
[hot_reload]
debounce_ms = 2000  # 2 seconds
```

Reason: Balances responsiveness with reload frequency

---

2. Validate Config Before Saving

Workflow:

```
# Edit config
vim ~/.code/config.toml

# Validate locally (optional tool)
toml-lint ~/.code/config.toml

# Save (triggers hot-reload)
```

---

3. Monitor Reload Notifications

Good Practice: Check TUI notifications after config changes

Example:

```
✅ Config reloaded successfully
   - 2 agents enabled
   - Quality gates updated
```

Bad Sign:

```
❌ Config reload failed: Invalid agent name
   Old configuration preserved.
```

Action: Fix error and save again

---

4. Test Config Changes Incrementally

Good:

```
1. Change one section (e.g., model config)
2. Save and verify reload
3. Change next section (e.g., quality gates)
4. Save and verify reload
```

Bad:

```
1. Change 10 sections at once
2. Save
3. Error in section 7
4. Hard to debug which change caused error
```

Summary

Hot-Reload Features: - Instant config updates (<100ms latency) - 300ms debouncing (prevents reload storms) - Safe validation (old config preserved on error) - TUI notifications (success/failure) - Deferred reload (don't interrupt operations) - Change detection (show what changed)

Performance: - <0.5% CPU overhead (idle) - ~2036ms reload latency (p50) - <1µs read locks - <1ms write locks

Configuration:

```toml
[hot_reload]
enabled = true
debounce_ms = 2000
watch_paths = ["config.toml"]
```

Best Practices: - Use default 2000ms debounce - Validate config before saving - Monitor TUI notifications - Test changes incrementally

Next: MCP Servers

MCP Servers

MCP server configuration, custom servers, and lifecycle management.

Overview

MCP (Model Context Protocol) enables AI agents to access external tools and resources through standardized servers.

Use Cases: - Memory systems (local-memory for knowledge persistence) - Git operations (git-status for repository inspection) - Custom tools (HAL for policy validation) - External services (databases, APIs, file systems)

Configuration: `[mcp_servers.<name>]` sections in `config.toml`

MCP Server Configuration

Basic Configuration

```toml
# ~/.code/config.toml

[mcp_servers.local-memory]
command = "npx"
args = ["-y", "@modelcontextprotocol/server-memory"]
startup_timeout_ms = 10000  # 10 seconds
```

Configuration Fields

Field	Type	Required	Description
`command`	string	Yes	Executable command
`args`	array	No	Command arguments
`env`	table	No	Environment variables
`startup_timeout_ms`	integer	No	Startup timeout (default: 10000ms)

Built-in MCP Servers

local-memory (Knowledge Persistence)

Purpose: Store and retrieve high-value knowledge (architecture decisions, patterns, bug fixes)

Configuration:

```toml
[mcp_servers.local-memory]
command = "npx"
args = ["-y", "@modelcontextprotocol/server-memory"]
startup_timeout_ms = 10000
```

Installation:

```
# Auto-installed on first use via npx -y
# Or install globally:
npm install -g @modelcontextprotocol/server-memory
```

Tools Provided: - `mcp__local-memory__store_memory` - Store knowledge - `mcp__local-memory__search` - Search knowledge - `mcp__local-memory__analysis` - Analyze patterns

Usage:

```
Use mcp__local-memory__store_memory:
- content: "Routing bug fixed: SpecKitCommand wasn't passing config..."
- domain: "debugging"
- tags: ["type:bug-fix", "spec:SPEC-KIT-066"]
- importance: 9
```

Storage: `~/.code/mcp-memory/` (SQLite database)

---

git-status (Repository Inspection)

Purpose: Inspect Git repository state, history, changes

Configuration:

```
[mcp_servers.git-status]
command = "npx"
args = ["-y", "@just-every/mcp-server-git"]
env = { LOG_LEVEL = "info" }
```

Tools Provided: - `mcp__git-status__status` - Get git status - `mcp__git-status__diff` - Get diff for files - `mcp__git-status__log` - Get commit history

Use Case: Automated commit message generation, change analysis

---

HAL (Policy Validation)

Purpose: Validate spec-kit policies (storage policy, tag schema, quality gates)

Configuration:

```
[mcp_servers.hal]
command = "/path/to/hal-server"
args = ["--mode", "strict"]
env = { HAL_SECRET_KAVEDARR_API_KEY = "..." }
startup_timeout_ms = 15000
```

Tools Provided: - `mcp__hal__validate_storage_policy` - Check local-memory usage - `mcp__hal__validate_tag_schema` - Check tag naming - `mcp__hal__validate_quality_gates` - Check consensus

Note: HAL server is project-specific (not publicly available)

Custom MCP Servers

Creating a Custom Server

Example: Database query server

```toml
[mcp_servers.database]
command = "/path/to/db-mcp-server"
args = ["--connection-string", "postgres://localhost/mydb"]
env = { DB_PASSWORD = "secret" }
startup_timeout_ms = 20000  # Longer timeout for DB connection
```

Server Implementation: See MCP Server SDK

Custom Server Example (Node.js)

```javascript
// db-mcp-server.js
const { MCPServer } = require('@modelcontextprotocol/sdk');
const { Pool } = require('pg');

const server = new MCPServer({
  name: 'database',
  version: '1.0.0',
});

const pool = new Pool({
  connectionString: process.argv[2],
});

server.tool({
  name: 'query',
  description: 'Execute SQL query',
  parameters: {
    sql: { type: 'string', description: 'SQL query to execute' },
  },
  async handler({ sql }) {
    const result = await pool.query(sql);
    return { rows: result.rows };
  },
});

server.start();
```

Configuration:

```toml
[mcp_servers.database]
command = "node"
args = ["/path/to/db-mcp-server.js", "postgres://localhost/mydb"]
```

MCP Server Lifecycle

Startup Process

```
1. Config loaded → Parse [mcp_servers.*] sections
2. Spawn process → Execute command with args
```

```
3. Handshake → Initialize MCP protocol
4. List tools → Request tools/list from server
5. Cache tools → Store tool metadata
6. Ready → Server available for use
```

Timeout: `startup_timeout_ms` (default: 10000ms)

Lazy Loading

Default Behavior: MCP servers are **not** started until first use

Benefit: Save resources by only starting needed servers

Example:

```toml
# Configured but not started
[mcp_servers.database]
command = "node"
args = ["/path/to/db-server.js"]

# Only started when tool is called:
# Use mcp__database__query: "SELECT * FROM users"
```

Startup Optimization

Cache Tool List:

```
First session:
  1. Start MCP server (~500ms)
  2. Request tools/list (~100ms)
  3. Cache to ~/.code/mcp-cache/database.json
  4. Use tools

Subsequent sessions:
  1. Load cached tools from ~/.code/mcp-cache/database.json (~10ms)
  2. Lazy-start server only when tool is called
```

Benefit: Faster session startup (no waiting for MCP servers)

Shutdown Process

```
1. Session end → Send shutdown signal to all MCP servers
2. Wait for clean shutdown (max 5s)
3. Force kill if timeout
4. Clean up temp files
```

Environment Variables

Server-Specific Environment

```toml
[mcp_servers.custom]
command = "/path/to/server"
env = {
    API_KEY = "secret",
```

```
    LOG_LEVEL = "debug",
    FEATURE_FLAG = "experimental"
}
```

Scope: Only available to the MCP server process

Global Environment Variables

```
# Available to all MCP servers
export MCP_LOG_LEVEL="debug"
export MCP_TIMEOUT="30000"
```

Use Case: Global MCP debugging settings

Timeouts and Retries

Startup Timeout

Default: 10000ms (10 seconds)

Configuration:

```
[mcp_servers.slow-server]
command = "/path/to/slow-server"
startup_timeout_ms = 30000  # 30 seconds for slow startup
```

Behavior: If server doesn't respond within timeout, startup fails

---

Tool Call Timeout

Default: Inherited from validation.timeout_seconds

Override:

```
[validation]
timeout_seconds = 60  # 60 seconds for all MCP tool calls
```

Retry Logic

Startup Failures: No automatic retry (manual restart required)

Tool Call Failures: Retry up to 3 times with exponential backoff

Example:

```
1. Tool call fails (network error)
2. Wait 1s
3. Retry (1/3)
4. Wait 2s
5. Retry (2/3)
6. Wait 4s
7. Retry (3/3)
8. Give up, report error to agent
```

Debugging MCP Servers

Enable MCP Logging

```
export RUST_LOG=codex_mcp_client=debug
code
```

Log Output:

```
[DEBUG] Starting MCP server: local-memory
[DEBUG] Command: npx -y @modelcontextprotocol/server-memory
[DEBUG] Handshake complete
[DEBUG] Requesting tools/list...
[DEBUG] Received 3 tools: store_memory, search, analysis
[DEBUG] MCP server ready: local-memory
```

Test MCP Server Manually

MCP Inspector (official debugging tool):

```
npm install -g @modelcontextprotocol/inspector

# Test local-memory server
npx @modelcontextprotocol/inspector npx -y
@modelcontextprotocol/server-memory
```

Features: - Test tool calls - Inspect responses - Debug connection issues

---

Check MCP Server Status

```
code --mcp-status
```

Output:

```
MCP Servers (3 configured):

local-memory:
  Status: Running (PID: 12345)
  Command: npx -y @modelcontextprotocol/server-memory
  Uptime: 2h 15m
  Tools: 3 (store_memory, search, analysis)

git-status:
  Status: Not started (lazy-load)
  Command: npx -y @just-every/mcp-server-git
  Tools: 3 (cached)

database:
  Status: Failed (startup timeout)
  Command: /path/to/db-server
  Error: Connection timeout after 20000ms
```

Force Restart MCP Server

```
code --mcp-restart local-memory
```

Use Case: Server crashed, hung, or behaving incorrectly

Common MCP Servers

Filesystem Server

```
[mcp_servers.filesystem]
command = "npx"
args = ["-y", "@modelcontextprotocol/server-filesystem",
"/allowed/path"]
```

Tools: Read/write files in allowed directory

HTTP Server

```
[mcp_servers.http]
command = "npx"
args = ["-y", "@modelcontextprotocol/server-http"]
```

Tools: Make HTTP requests

Database Servers

PostgreSQL:

```
[mcp_servers.postgres]
command = "npx"
args = ["-y", "@modelcontextprotocol/server-postgres",
"postgres://localhost/mydb"]
```

SQLite:

```
[mcp_servers.sqlite]
command = "npx"
args = ["-y", "@modelcontextprotocol/server-sqlite",
"/path/to/db.sqlite"]
```

Custom API Server

```
[mcp_servers.custom-api]
command = "/path/to/custom-mcp-server"
args = ["--api-url", "https://api.example.com"]
env = { API_TOKEN = "secret" }
```

Security Considerations

1. Validate Command Paths

Good:

```
[mcp_servers.trusted]
```

```
command = "npx"  # Well-known command
args = ["-y", "@modelcontextprotocol/server-memory"]
```

Bad:

```
[mcp_servers.untrusted]
command = "/tmp/random-script.sh"  # ✗ Untrusted source
```

---

2. Avoid Secrets in Config

Good:

```
[mcp_servers.database]
command = "/path/to/db-server"
env = { DB_PASSWORD = "secret" }  # ⚠ Still visible in config

# Better: Use environment variable
env = { DB_PASSWORD = "$DB_PASSWORD_FROM_ENV" }
```

Best:

```
# Store secret in environment
export DB_PASSWORD="secret"

[mcp_servers.database]
command = "/path/to/db-server"
# Server reads $DB_PASSWORD from environment
```

---

3. Restrict Network Access

Sandbox Mode: MCP servers inherit sandbox restrictions

```
sandbox_mode = "read-only"  # MCP servers also read-only

[mcp_servers.filesystem]
command = "npx"
args = ["-y", "@modelcontextprotocol/server-filesystem",
"/safe/path"]
```

Best Practices

1. Use Lazy Loading

Default behavior (don't change): - Servers start on first use - Faster session startup - Lower resource usage

---

2. Set Appropriate Timeouts

Fast servers (in-memory):

```
[mcp_servers.memory]
startup_timeout_ms = 5000  # 5s
```

Slow servers (database, network):

```toml
[mcp_servers.database]
startup_timeout_ms = 30000  # 30s
```

---

3. Monitor MCP Server Logs

```bash
export RUST_LOG=debug
code

# Check logs for MCP errors
tail -f ~/.code/debug.log | grep MCP
```

---

4. Test Servers with MCP Inspector

```bash
npx @modelcontextprotocol/inspector <command> <args>
```

Benefit: Catch configuration errors before using in production

---

Summary

MCP Servers enable: - Knowledge persistence (local-memory) - Git operations (git-status) - Custom tools (database, API, filesystem) - External service integration

Configuration:

```toml
[mcp_servers.<name>]
command = "executable"
args = ["arg1", "arg2"]
env = { KEY = "value" }
startup_timeout_ms = 10000
```

Features: - Lazy loading (start on first use) - Tool caching (faster startup) - Automatic retry (tool call failures) - Hot-reload support (config changes)

Debugging: - MCP Inspector (test servers) - `--mcp-status` (check status) - `--mcp-restart` (force restart) - Debug logging (`RUST_LOG=debug`)

Next: Environment Variables

Model Configuration

Provider setup, reasoning effort, and model tuning.

Overview

Model configuration controls: 1. **Provider Selection** - Which AI service to use (OpenAI, Anthropic, Google, Ollama) 2. **Model Selection** - Which specific model (GPT-5, o3, Claude, Gemini) 3.

Reasoning Configuration - Effort level, summaries, verbosity 4.
Network Tuning - Retries, timeouts, streaming

Basic Model Configuration

Minimal Setup

```toml
# ~/.code/config.toml

model = "gpt-5"
model_provider = "openai"
```

Environment:

```bash
export OPENAI_API_KEY="sk-proj-..."
```

Model Selection

Available Models (OpenAI): - `gpt-5` - Default, balanced reasoning and cost - `gpt-5-codex` - Optimized for code generation - `o3` - Maximum reasoning capability (premium) - `o4-mini` - Fast reasoning model - `gpt-4o` - Legacy model - `gpt-4o-mini` - Fast, cheap legacy model

Configuration:

```toml
model = "o3"  # Use premium reasoning model
```

CLI Override:

```bash
code --model o3 "complex task"
```

Provider Configuration

OpenAI (Default)

```toml
model_provider = "openai"

[model_providers.openai]
name = "OpenAI"
base_url = "https://api.openai.com/v1"
env_key = "OPENAI_API_KEY"
wire_api = "responses"  # or "chat"
request_max_retries = 4
stream_max_retries = 10
stream_idle_timeout_ms = 300000  # 5 minutes
```

Environment Variables:

```bash
export OPENAI_API_KEY="sk-proj-..."

# Optional overrides
export OPENAI_BASE_URL="https://custom.openai.com/v1"
export OPENAI_WIRE_API="chat"  # Force chat completions API
```

Anthropic (Claude)

```toml
model_provider = "anthropic"
model = "claude-3-5-sonnet"

[model_providers.anthropic]
name = "Anthropic"
base_url = "https://api.anthropic.com"
env_key = "ANTHROPIC_API_KEY"
wire_api = "chat"
```

Environment:

```bash
export ANTHROPIC_API_KEY="sk-ant-..."
```

Google (Gemini)

```toml
model_provider = "google"
model = "gemini-2.0-flash-001"

[model_providers.google]
name = "Google"
base_url = "https://generativelanguage.googleapis.com/v1beta"
env_key = "GOOGLE_API_KEY"
wire_api = "chat"
```

Environment:

```bash
export GOOGLE_API_KEY="..."
```

Ollama (Local)

```toml
model_provider = "ollama"
model = "mistral"

[model_providers.ollama]
name = "Ollama"
base_url = "http://localhost:11434/v1"
# No env_key needed for local Ollama
```

Setup:

```bash
# Install Ollama
curl -fsSL https://ollama.com/install.sh | sh

# Pull model
ollama pull mistral

# Start server
ollama serve
```

Azure OpenAI

```toml
model_provider = "azure"
model = "gpt-5"

[model_providers.azure]
```

```toml
name = "Azure OpenAI"
base_url = "https://YOUR_PROJECT.openai.azure.com/openai"
env_key = "AZURE_OPENAI_API_KEY"
wire_api = "chat"
query_params = { api-version = "2025-04-01-preview" }
```

Environment:

```bash
export AZURE_OPENAI_API_KEY="..."
```

Custom Provider

```toml
[model_providers.custom]
name = "Custom Provider"
base_url = "https://custom.api.com/v1"
env_key = "CUSTOM_API_KEY"
wire_api = "chat"

# Optional: Static HTTP headers
http_headers = { "X-Custom-Header" = "value" }

# Optional: Dynamic HTTP headers from environment
env_http_headers = { "X-Features" = "CUSTOM_FEATURES" }

# Network tuning
request_max_retries = 3
stream_max_retries = 5
stream_idle_timeout_ms = 180000  # 3 minutes
```

Reasoning Configuration

Reasoning Effort

Controls how much computational effort the model uses for reasoning.

Options: - `minimal` - Fastest, least reasoning (previously “none”) - `low` - Light reasoning - `medium` - Balanced (default) - `high` - Maximum reasoning (premium cost)

Configuration:

```toml
model_reasoning_effort = "high"
```

Use Cases:

Effort	Use Case	Cost	Speed
`minimal`	Simple formatting, trivial tasks	Lowest	Fastest
`low`	Straightforward code changes	Low	Fast
`medium`	Moderate complexity tasks	Medium	Moderate
`high`	Complex refactoring, architecture	Highest	Slowest

Example:

```toml
# Premium profile for complex tasks
[profiles.premium]
```

```toml
model = "o3"
model_reasoning_effort = "high"

# Fast profile for simple tasks
[profiles.fast]
model = "gpt-4o-mini"
model_reasoning_effort = "minimal"
```

Reasoning Summary

Controls summarization of reasoning process.

Options: - `auto` - Model decides (default) - `concise` - Brief summary - `detailed` - Comprehensive summary - `none` - No summary

Configuration:

```toml
model_reasoning_summary = "detailed"
```

Example Output:

`auto`:

```
Reasoning: Analyzing code structure...
```

`concise`:

```
Reasoning: Identified 3 refactoring opportunities.
```

`detailed`:

```
Reasoning: Analyzed codebase structure. Identified 3 refactoring
opportunities:
1. Extract duplicate validation logic into shared function
2. Replace switch statement with strategy pattern
3. Simplify nested conditionals with early returns
```

`none`:

```
(No reasoning summary shown)
```

Model Verbosity (GPT-5 Only)

Controls output length/detail for GPT-5 family models.

Options: - `low` - Concise output - `medium` - Balanced (default) - `high` - Detailed explanations

Configuration:

```toml
model = "gpt-5"
model_verbosity = "low"
```

Example:

`low`:

```
Refactored validation logic. See main.rs:42.
```

`medium`:

```
Refactored validation logic into shared function `validate_input()`
in main.rs:42. Updated 3 call sites.
```

high:

```
Refactored validation logic to improve maintainability:
1. Extracted duplicate validation code into new function
`validate_input()`
   - Location: main.rs:42-58
   - Parameters: &str input, bool strict_mode
   - Returns: Result<(), ValidationError>
2. Updated call sites: handler.rs:15, api.rs:33, cli.rs:67
3. Added unit tests: tests/validation_test.rs:10-45
```

Context Window Configuration

Context Window Size

Default: Auto-detected based on model

Manual Override:

```
model_context_window = 128000  # 128K tokens
```

Use Case: New models not yet recognized by Codex

Max Output Tokens

Default: Auto-detected based on model

Manual Override:

```
model_max_output_tokens = 16384  # 16K tokens
```

Use Case: Limit output length for cost control

Network Tuning

Request Retries

Default: 4 retries

Configuration:

```
[model_providers.openai]
request_max_retries = 6  # Increase for unreliable networks
```

Behavior: Exponential backoff (1s, 2s, 4s, 8s, 16s, 32s)

Stream Retries

Default: 10 retries

Configuration:

```toml
[model_providers.openai]
stream_max_retries = 15  # Increase for flaky connections
```

Use Case: Unstable network, frequent disconnects

Stream Idle Timeout

Default: 300,000 ms (5 minutes)

Configuration:

```toml
[model_providers.openai]
stream_idle_timeout_ms = 600000  # 10 minutes for slow models
```

Use Case: Very slow models or complex tasks

Wire API Selection

Responses API (Default for GPT-5/o3)

Features: - Native reasoning support - Reasoning summaries - Verbosity control - Optimized for GPT-5 family

Configuration:

```toml
[model_providers.openai]
wire_api = "responses"
```

Chat Completions API (Legacy)

Features: - Compatible with all OpenAI models - Compatible with most third-party providers - Simpler protocol

Configuration:

```toml
[model_providers.openai]
wire_api = "chat"
```

Use Case: Third-party providers, older models

Advanced Configuration

Force Reasoning Support

Use Case: Custom models that support reasoning but aren't auto-detected

Configuration:

```toml
model_supports_reasoning_summaries = true
```

Disable Response Storage (ZDR Accounts)

Use Case: Zero Data Retention accounts

Configuration:

```toml
disable_response_storage = true
```

Effect: Forces Chat Completions API instead of Responses API

Configuration Examples

Premium Quality Setup

```toml
# Maximum reasoning quality
model = "o3"
model_provider = "openai"
model_reasoning_effort = "high"
model_reasoning_summary = "detailed"
model_verbosity = "high"

[model_providers.openai]
wire_api = "responses"
```

Fast Iteration Setup

```toml
# Speed over quality
model = "gpt-4o-mini"
model_provider = "openai"
model_reasoning_effort = "minimal"
model_reasoning_summary = "none"
model_verbosity = "low"

[model_providers.openai]
wire_api = "chat"
```

Local Development Setup

```toml
# Ollama for offline development
model = "mistral"
model_provider = "ollama"

[model_providers.ollama]
name = "Ollama"
base_url = "http://localhost:11434/v1"
```

Multi-Provider Setup

```toml
# Default to OpenAI
model = "gpt-5"
model_provider = "openai"

# OpenAI provider
```

```toml
[model_providers.openai]
name = "OpenAI"
base_url = "https://api.openai.com/v1"
env_key = "OPENAI_API_KEY"
wire_api = "responses"

# Anthropic provider
[model_providers.anthropic]
name = "Anthropic"
base_url = "https://api.anthropic.com"
env_key = "ANTHROPIC_API_KEY"
wire_api = "chat"

# Ollama provider (local)
[model_providers.ollama]
name = "Ollama"
base_url = "http://localhost:11434/v1"

# Profiles for quick switching
[profiles.openai]
model_provider = "openai"
model = "gpt-5"

[profiles.claude]
model_provider = "anthropic"
model = "claude-3-5-sonnet"

[profiles.local]
model_provider = "ollama"
model = "mistral"
```

Usage:

```
code --profile openai "task"
code --profile claude "task"
code --profile local "task"
```

Debugging Model Configuration

Check Effective Configuration

```
code --config-dump | grep -A 10 "model"
```

Output:

```
model = "o3"  # From: CLI flag
model_provider = "openai"  # From: config.toml
model_reasoning_effort = "high"  # From: profile 'premium'
model_reasoning_summary = "detailed"  # From: profile 'premium'
```

Test Provider Connection

```
# Enable debug logging
export RUST_LOG=debug
code "Hello world"
```

Log Output:

```
[DEBUG] Model provider: openai
[DEBUG] Base URL: https://api.openai.com/v1
[DEBUG] Wire API: responses
[DEBUG] Model: o3
[DEBUG] Reasoning effort: high
[INFO] Connection successful
```

Summary

Model Configuration covers: - Provider selection (OpenAI, Anthropic, Google, Ollama, Azure, custom) - Model selection (GPT-5, o3, Claude, Gemini, etc.) - Reasoning effort (minimal, low, medium, high) - Reasoning summaries (auto, concise, detailed, none) - Model verbosity (low, medium, high) - Network tuning (retries, timeouts) - Wire API selection (responses, chat)

Best Practices: - Use profiles for different quality/speed tradeoffs - Store API keys in environment variables - Tune network settings for your connection quality - Use local providers (Ollama) for offline development

Next: Agent Configuration

Precedence System

5-tier configuration precedence with examples.

Overview

The configuration system implements **5-tier precedence** (highest to lowest):

1. **CLI Flags** (highest priority) - Command-line arguments
2. **Shell Environment** - Environment variables
3. **Profile** - Named configuration sets
4. **Config File** - `~/.code/config.toml`
5. **Defaults** (lowest priority) - Built-in fallback values

Rule: Higher tiers override lower tiers

Precedence Order

Tier 1: CLI Flags (Highest)

Priority: Highest

Usage:

```
# Specific model flags
code --model o3 "task description"
code --profile premium "task"
```

```
        # Generic config flag
        code --config model="gpt-5"
        code --config approval_policy=never
        code -c model_reasoning_effort=high

        # Deep config paths (dot notation)
        code --config model_providers.openai.wire_api="chat"
        code --config shell_environment_policy.include_only='["PATH",
"HOME"]'
```

Characteristics: - Overrides all other tiers - Session-specific (not persisted) - Supports dot notation for nested values - Values in TOML format (not JSON)

Examples:

```
        # Override model
        code --model o3

        # Override approval policy
        code --config approval_policy=never

        # Override provider config
        code --config
model_providers.openai.base_url="https://custom.api.com"
```

Tier 2: Shell Environment

Priority: 2nd highest

Patterns: - `CODEX_HOME`, `CODE_HOME` - Installation directory - `<PROVIDER>_API_KEY` - API keys (e.g., `OPENAI_API_KEY`) - `OPENAI_BASE_URL` - Provider base URL override - `OPENAI_WIRE_API` - Wire protocol override (`"responses"` or `"chat"`) - `CODEX_MODEL`, `CODEX_PROVIDER` - Model/provider overrides

Usage:

```
        # API keys (most common)
        export OPENAI_API_KEY="sk-proj-..."
        export ANTHROPIC_API_KEY="sk-ant-..."
        export GOOGLE_API_KEY="..."

        # Home directory
        export CODEX_HOME="/custom/path"

        # Provider overrides
        export OPENAI_BASE_URL="https://custom.openai.com/v1"
        export OPENAI_WIRE_API="responses"

        # Model overrides
        export CODEX_MODEL="gpt-5"
        export CODEX_PROVIDER="anthropic"
```

Characteristics: - Persistent for session duration - Useful for secrets (API keys) - Environment-specific overrides - Case-insensitive for most values

---

Tier 3: Profile

Priority: 3rd highest

Activation:

```
# Via CLI
code --profile premium "task"

# Via config.toml
profile = "premium"
```

Definition:

```
# ~/.code/config.toml

[profiles.premium]
model = "o3"
model_reasoning_effort = "high"
model_reasoning_summary = "detailed"
approval_policy = "never"

[profiles.fast]
model = "gpt-4o-mini"
model_reasoning_effort = "low"
approval_policy = "never"

[profiles.ci]
model = "gpt-4o"
approval_policy = "never"
sandbox_mode = "read-only"
```

Characteristics: - Named configuration sets - Overrides config.toml base values - Can be selected per-session via CLI - Useful for different workflows

---

Tier 4: Config File

Priority: 4th highest

Location: `~/.code/config.toml`

Example:

```
model = "gpt-5"
model_provider = "openai"
approval_policy = "on-request"
sandbox_mode = "workspace-write"

[quality_gates]
plan = ["gemini", "claude", "code"]
tasks = ["gemini"]
```

Characteristics: - Persistent across sessions - User-specific configuration - Hot-reloadable (changes apply without restart) - TOML format (human-readable)

Tier 5: Defaults (Lowest)

Priority: Lowest

Source: Built-in code defaults

Example:

```rust
impl Default for AppConfig {
    fn default() -> Self {
        Self {
            model: "gpt-5-codex".to_string(),
            model_provider: "openai".to_string(),
            approval_policy: ApprovalPolicy::OnRequest,
            sandbox_mode: SandboxMode::ReadOnly,
            // ... 30+ more fields
        }
    }
}
```

Characteristics: - Fallback values when no other tier specifies - Hardcoded in Rust source - Guaranteed sensible defaults - Work out-of-the-box without configuration

---

Precedence Examples

Example 1: Model Selection

Setup:

```toml
# ~/.code/config.toml
model = "gpt-5"

[profiles.premium]
model = "o3"
```

```bash
export CODEX_MODEL="gpt-4o"
```

Scenarios:

Command	Effective Model	Why
`code "task"`	`gpt-4o`	Env var (Tier 2) > config.toml (Tier 4)
`code --profile premium "task"`	`o3`	Profile (Tier 3) > env var (Tier 2)
`code --model o1 "task"`	`o1`	CLI flag (Tier 1) > all others
`code --profile premium --model o1 "task"`	`o1`	CLI flag (Tier 1) wins

Example 2: API Key

Setup:

```
# ~/.code/config.toml
# (no API key specified)

export OPENAI_API_KEY="sk-proj-env-key"
```

Scenarios:

Command	Effective Key	Why
`code "task"`	`sk-proj-env-key`	Env var (Tier 2) > defaults (Tier 5)
`code --config model_providers.openai.env_key="sk-proj-cli-key" "task"`	`sk-proj-cli-key`	CLI flag (Tier 1) > env var (Tier 2)

Note: API keys should **always** be stored in environment variables, never in `config.toml`.

Example 3: Approval Policy

Setup:

```
# ~/.code/config.toml
approval_policy = "on-request"

[profiles.ci]
approval_policy = "never"

# No environment overrides
```

Scenarios:

Command	Effective Policy	Why
`code "task"`	`on-request`	config.toml (Tier 4) > defaults (Tier 5)
`code --profile ci "task"`	`never`	Profile (Tier 3) > config.toml (Tier 4)
`code --profile ci --config approval_policy=untrusted "task"`	`untrusted`	CLI flag (Tier 1) > profile (Tier 3)

Example 4: Complex Nested Config

Setup:

```toml
# ~/.code/config.toml
[model_providers.openai]
base_url = "https://api.openai.com/v1"
wire_api = "responses"

export OPENAI_BASE_URL="https://custom.openai.com/v1"
```

Scenarios:

Command	Effective URL
`code "task"`	`https://custom.openai.com/v1`
`code --config model_providers.openai.wire_api="chat" "task"`	`https://custom.openai.com/v1`

Special Cases

Shell Environment Policy Override

Warning: `shell_environment_policy.set` can override config values at runtime.

Example:

```toml
# ~/.code/config.toml
approval_policy = "always"

[shell_environment_policy]
set = { APPROVAL_POLICY = "never" }  # ⚠ OVERRIDES top-level setting!
```

Behavior: `APPROVAL_POLICY=never` wins at runtime (subprocess environment)

Best Practice: Avoid using `shell_environment_policy.set` for keys that exist as top-level config options.

Profile Selection Precedence

Priority: CLI `--profile` > `config.toml` `profile` field > no profile

Example:

```toml
# ~/.code/config.toml
profile = "fast"  # Default profile
```

```toml
[profiles.fast]
model = "gpt-4o-mini"

[profiles.premium]
model = "o3"
```

Command	Effective Profile	Model	Why
`code "task"`	`fast`	`gpt-4o-mini`	config.toml `profile` field
`code --profile premium "task"`	`premium`	`o3`	CLI `--profile` overrides config.toml

Precedence Table

Summary:

Tier	Source	Example	Persistence	Override M
1	CLI Flags	`--model o3`	Session-only	Command-lin
2	Environment	`OPENAI_API_KEY=...`	Session/shell	`export VAR=va`
3	Profile	`[profiles.premium]`	Persistent (in config.toml)	`--profile nam` `profile = "na`
4	Config File	`model = "gpt-5"`	Persistent	Edit `~/.code/confi`
5	Defaults	`"gpt-5-codex"`	Built-in	(Cannot over

Debugging Precedence

Check Effective Configuration

Command:

```bash
code --config-dump
```

Output:

```toml
# Effective configuration (after precedence resolution)
model = "o3"  # From: CLI flag (--model o3)
model_provider = "openai"  # From: config.toml
approval_policy = "never"  # From: profile 'premium'
# ... full effective config
```

Trace Configuration Source

Example:

```bash
# With verbose logging
```

```
export RUST_LOG=debug
code --model o3 "task"
```

Log Output:

```
[DEBUG] Config layer 5 (defaults): model=gpt-5-codex
[DEBUG] Config layer 4 (config.toml): model=gpt-5
[DEBUG] Config layer 3 (profile 'premium'): model=o3
[DEBUG] Config layer 1 (CLI flag): model=o3
[INFO] Effective model: o3 (source: CLI flag)
```

Best Practices

1. Use Environment Variables for Secrets

Good:

```
export OPENAI_API_KEY="sk-proj-..."
```

Bad:

```
# DON'T: API keys should NOT be in config.toml
[model_providers.openai]
api_key = "sk-proj-..."  # ✖ Security risk!
```

---

2. Use Profiles for Workflows

Example:

```
# Fast iteration
[profiles.fast]
model = "gpt-4o-mini"
approval_policy = "never"

# Premium quality
[profiles.premium]
model = "o3"
model_reasoning_effort = "high"

# CI/automation
[profiles.ci]
model = "gpt-4o"
approval_policy = "never"
sandbox_mode = "read-only"
```

Usage:

```
code --profile fast "quick formatting"
code --profile premium "complex refactor"
code --profile ci "generate report"
```

---

3. Use CLI Flags for One-Off Overrides

Example:

```
# One-time model override
```

```
code --model o3 "complex task"

# One-time approval policy override
code --config approval_policy=never "trusted script"
```

---

4. Keep config.toml for Persistent Preferences

Example:

```
# ~/.code/config.toml

# Personal preferences (persistent)
model = "gpt-5"
approval_policy = "on-request"
sandbox_mode = "workspace-write"
file_opener = "vscode"

[tui.theme]
name = "dark-carbon-night"
```

Summary

5-Tier Precedence (highest to lowest): 1. CLI Flags - Session-specific overrides 2. Environment Variables - Secrets and env-specific config 3. Profiles - Named configuration sets 4. Config File - Persistent user preferences 5. Defaults - Built-in fallback values

Rule: Higher tiers override lower tiers

Best Practices: - Secrets → Environment variables - Workflows → Profiles - One-off overrides → CLI flags - Persistent preferences → config.toml

Next: Model Configuration

Quality Gate Customization

Per-checkpoint agent selection and override rules.

Overview

Quality Gates are checkpoints in the spec-kit workflow that ensure standards are met before proceeding.

5 Quality Gates: 1. **Plan** - Architectural planning (multi-agent consensus) 2. **Tasks** - Task decomposition (single-agent) 3. **Validate** - Test strategy validation (multi-agent) 4. **Audit** - Security/compliance review (premium agents) 5. **Unlock** - Ship/no-ship decision (premium agents)

Configuration: `[quality_gates]` section in `config.toml`

Quality Gate Configuration

Basic Configuration

```toml
# ~/.code/config.toml

[quality_gates]
plan = ["gemini", "claude", "code"]        # Multi-agent planning
tasks = ["gemini"]                          # Single-agent task breakdown
validate = ["gemini", "claude", "code"]    # Multi-agent test validation
audit = ["gemini", "claude", "gpt_codex"]  # Security/compliance
unlock = ["gemini", "claude", "gpt_codex"] # Ship decision
```

Field Reference

Field	Purpose	Recommended Agents	Cost Tier
`plan`	Architectural decisions	3 agents (diverse)	Tier 2 (~$0.35)
`tasks`	Task breakdown	1 agent (cheap)	Tier 1 (~$0.10)
`validate`	Test strategy	3 agents (diverse)	Tier 2 (~$0.35)
`audit`	Security/compliance	3+ premium	Tier 3 (~$0.80)
`unlock`	Ship decision	3 premium	Tier 3 (~$0.80)

Agent Selection Strategy

Multi-Agent Consensus (Plan, Validate)

Purpose: Diverse perspectives on complex decisions

Recommended Setup:

```toml
[quality_gates]
plan = ["gemini", "claude", "code"]  # Fast + Balanced + Strategic
```

Agent Roles: - `gemini` - Fast consensus, broad coverage (12.5x cheaper) - `claude` - Balanced reasoning, edge case detection (12x cheaper) - `code` (GPT-5) - Strategic planning, complex reasoning (baseline)

Why 3 Agents: - 2 agents: Risk of tie (no consensus) - 3 agents: Majority vote possible - 4+ agents: Diminishing returns, higher cost

---

Single-Agent Deterministic (Tasks)

Purpose: Straightforward decomposition without opinion diversity

Recommended Setup:

```toml
[quality_gates]
tasks = ["gemini"]  # Single cheap agent
```

Why Single Agent: - Task breakdown is mechanical (not strategic) - No benefit from consensus - Cost savings (1 agent vs 3)

Premium Consensus (Audit, Unlock)

Purpose: Critical decisions requiring maximum reasoning

Recommended Setup:

```toml
[quality_gates]
audit = ["gemini", "claude", "gpt_codex"]  # Security-focused
unlock = ["gemini", "claude", "gpt_codex"] # Ship decision
```

Agent Selection: - `gemini` - Broad vulnerability scanning - `claude` - Edge case security analysis - `gpt_codex` - Code-specific security patterns

Why Premium: - Audit prevents security incidents ($1000s in damages) - Unlock prevents production bugs ($1000s in incidents) - $0.80 cost per stage justifiable for critical gates

---

Custom Configurations

Cost-Optimized Setup

Goal: Minimize cost while maintaining quality

```toml
[quality_gates]
plan = ["gemini", "claude"]  # 2 cheap agents (no GPT-5)
tasks = ["gemini"]            # Single cheap agent
validate = ["gemini", "claude"]  # 2 cheap agents
audit = ["gemini", "claude", "code"]  # 2 cheap + 1 mid-tier
unlock = ["gemini", "claude", "code"] # 2 cheap + 1 mid-tier
```

Cost Savings: ~60% reduction (from $2.70 to ~$1.08 per full pipeline)

Tradeoff: Less strategic depth (no GPT-5 on plan/validate)

Premium Quality Setup

Goal: Maximum quality, cost secondary

```toml
[quality_gates]
plan = ["gemini", "claude", "code", "gpt_pro"]  # 4 agents (premium)
tasks = ["code"]  # GPT-5 for task breakdown
validate = ["gemini", "claude", "code", "gpt_pro"]  # 4 agents
audit = ["gemini", "claude", "code", "gpt_codex", "gpt_pro"]  # 5
agents
unlock = ["gemini", "claude", "gpt_codex", "gpt_pro"]  # 4 premium
```

Cost: ~$4.50 per full pipeline (66% increase)

Benefit: Maximum reasoning, redundant validation

Specialist Configuration

Goal: Assign specialists per gate

```
        # Define specialized agents
        [[agents]]
        name = "security-specialist"
        canonical_name = "security"
        command = "claude"
        instructions = "Focus on OWASP Top 10, cryptography, auth/authz."

        [[agents]]
        name = "test-specialist"
        canonical_name = "test"
        command = "gemini"
        instructions = "Focus on test coverage, edge cases, property-based
tests."

        # Quality gates with specialists
        [quality_gates]
        plan = ["gemini", "claude", "code"]       # General agents
        tasks = ["gemini"]                         # General agent
        validate = ["test", "claude", "code"]     # Test specialist for
validation
        audit = ["security", "claude", "gpt_codex"]  # Security specialist
for audit
        unlock = ["gemini", "claude", "gpt_codex"]   # General agents
```

Per-Checkpoint Overrides

Override at Runtime

Quality gates can be overridden per-command:

```
        # Override plan agents
        /speckit.plan SPEC-KIT-065 --agents gemini,claude

        # Override validate agents (premium quality)
        /speckit.validate SPEC-KIT-065 --agents gemini,claude,code,gpt_pro

        # Override audit agents (cost-optimized)
        /speckit.audit SPEC-KIT-065 --agents gemini,claude
```

Use Case: One-off quality/cost tradeoffs

Environment Variable Overrides

```
        # Override plan agents via env var
        export SPECKIT_QUALITY_GATES_PLAN="gemini,claude,code,gpt_pro"
        /speckit.plan SPEC-KIT-065
```

```
# Override tasks agents
export SPECKIT_QUALITY_GATES_TASKS="code"
/speckit.tasks SPEC-KIT-065
```

Precedence: Env var > config.toml

Consensus Thresholds

Minimum Consensus

Default: 2/3 agents (66.7%)

Configuration:

```
[quality_gates]
plan = ["gemini", "claude", "code"]
consensus_threshold = 0.67  # 2/3 agents must agree
```

Example: - 3 agents, 2 agree → ✅ Pass (2/3 = 66.7%) - 3 agents, 1 agrees → ❌ Fail (1/3 = 33.3%)

Strict Consensus

Configuration:

```
[quality_gates]
unlock = ["gemini", "claude", "gpt_codex"]
consensus_threshold = 1.0  # 100% agreement required
```

Use Case: Critical ship decisions (unlock gate)

Behavior: All agents must agree to pass

Relaxed Consensus

Configuration:

```
[quality_gates]
plan = ["gemini", "claude", "code"]
consensus_threshold = 0.5  # 50% majority
```

Use Case: Exploratory planning (early stages)

Behavior: Simple majority sufficient

Degradation Handling

Agent Failure Behavior

Scenario: One agent fails (timeout, error)

Default Behavior: 1. Retry up to 3 times (AR-2) 2. If still fails, continue with remaining agents 3. Consensus valid if remaining agents ≥ threshold

Example:

```
[quality_gates]
plan = ["gemini", "claude", "code"]  # 3 agents
consensus_threshold = 0.67
```

If `code` agent fails: - Remaining: `gemini`, `claude` (2 agents) - If both agree: 2/2 = 100% ≥ 67% → ✅ Pass - If disagree: 1/2 = 50% < 67% → ❌ Fail

---

Empty Consensus Handling

Scenario: All agents fail

Behavior: Fall back to degraded mode

Example:

```
# All agents failed
❌ Quality gate failed: No agents returned valid consensus
⚠ Continuing in degraded mode (manual review required)
```

User Action: Manual PRD review and approval

Quality Gate Validation

Startup Validation

Validation Rules: 1. All agent names must exist in `[[agents]]` 2. Agent `canonical_name` must match quality gate references 3. Agents must be enabled 4. Minimum 1 agent per gate

Example Error:

```
Config validation error:
  quality_gates.plan: Agent 'unknown-agent' not found
  quality_gates.audit: Agent 'gpt_pro' exists but is disabled

Fix: Check [[agents]] configuration
```

Runtime Validation

Per-command validation:

```
/speckit.plan SPEC-KIT-065
```

Validation: 1. All specified agents are available 2. Agents can be spawned (commands exist) 3. Consensus threshold achievable

Example Error:

```
❌ Cannot execute /speckit.plan:
```

```
  - Agent 'claude' command not found
  - Consensus threshold 0.67 requires ≥2 agents, only 1 available

Fix: Install missing agent or adjust consensus_threshold
```

Example Configurations

Balanced (Default)

```toml
[quality_gates]
plan = ["gemini", "claude", "code"]        # 3 agents, diverse
tasks = ["gemini"]                          # 1 agent, cheap
validate = ["gemini", "claude", "code"]    # 3 agents, diverse
audit = ["gemini", "claude", "gpt_codex"]  # 3 agents, security-focused
unlock = ["gemini", "claude", "gpt_codex"] # 3 agents, ship decision
```

Cost: ~$2.70 per full pipeline

Cost-Optimized

```toml
[quality_gates]
plan = ["gemini", "claude"]     # 2 agents (no GPT-5)
tasks = ["gemini"]               # 1 agent
validate = ["gemini", "claude"] # 2 agents
audit = ["gemini", "claude"]    # 2 agents (no premium)
unlock = ["gemini", "claude"]   # 2 agents
```

Cost: ~$0.80 per full pipeline (70% reduction)

Premium Quality

```toml
[quality_gates]
plan = ["gemini", "claude", "code", "gpt_pro"]  # 4 agents
tasks = ["code"]  # GPT-5 for tasks
validate = ["gemini", "claude", "code", "gpt_pro"]  # 4 agents
audit = ["gemini", "claude", "code", "gpt_codex", "gpt_pro"]  # 5 agents
unlock = ["gemini", "claude", "gpt_codex", "gpt_pro"]  # 4 agents
```

Cost: ~$4.50 per full pipeline (66% increase)

Single-Agent (Development)

```toml
[quality_gates]
plan = ["gemini"]      # Fast iteration
tasks = ["gemini"]
validate = ["gemini"]
audit = ["gemini"]
unlock = ["gemini"]
```

Cost: ~$0.20 per full pipeline (93% reduction)

Use Case: Rapid prototyping, development iteration

Debugging Quality Gates

Check Quality Gate Configuration

```
code --quality-gates-dump
```

Output:

```
[quality_gates]
plan = ["gemini", "claude", "code"]  # 3 agents
tasks = ["gemini"]  # 1 agent
validate = ["gemini", "claude", "code"]  # 3 agents
audit = ["gemini", "claude", "gpt_codex"]  # 3 agents
unlock = ["gemini", "claude", "gpt_codex"]  # 3 agents

# Consensus thresholds (effective)
plan.consensus_threshold = 0.67
validate.consensus_threshold = 0.67
unlock.consensus_threshold = 1.0  # Strict (100%)
```

Validate Agent Availability

```
code --check-quality-gates
```

Output:

```
Validating quality gates...

plan:
  [✓] gemini (enabled, command found)
  [✓] claude (enabled, command found)
  [✓] code (enabled, command found)

tasks:
  [✓] gemini (enabled, command found)

validate:
  [✓] gemini (enabled, command found)
  [✓] claude (enabled, command found)
  [✓] code (enabled, command found)

audit:
  [✓] gemini (enabled, command found)
  [✓] claude (enabled, command found)
  [✗] gpt_codex (disabled)

unlock:
  [✓] gemini (enabled, command found)
  [✓] claude (enabled, command found)
  [✗] gpt_codex (disabled)

⚠ Warning: gpt_codex is disabled but referenced in audit, unlock
gates
```

Best Practices

1. Use 3 Agents for Consensus

Recommended: 3 agents for plan, validate, audit, unlock

Reason: Allows majority vote, avoids ties

---

2. Use 1 Agent for Deterministic Tasks

Recommended: 1 agent for tasks

Reason: Task breakdown is mechanical, no consensus needed

---

3. Reserve Premium Agents for Critical Gates

Good:

```
        [quality_gates]
        plan = ["gemini", "claude", "code"]  # Mid-tier for planning
        audit = ["gemini", "claude", "gpt_pro"]  # Premium for security
        unlock = ["gemini", "claude", "gpt_pro"]  # Premium for ship
decision
```

---

4. Test Quality Gate Configuration

```
        # Dry-run to validate config
        /speckit.plan SPEC-TEST-001 --dry-run
```

Summary

Quality Gate Customization covers: - 5 quality gates (plan, tasks, validate, audit, unlock) - Agent selection strategies (multi-agent, single-agent, premium) - Cost optimization (70-93% reduction possible) - Consensus thresholds (50-100%) - Degradation handling (agent failures) - Runtime overrides (CLI, env vars)

Best Practices: - 3 agents for consensus gates - 1 agent for deterministic gates - Premium agents for critical decisions - Test configuration with dry-run

Next: Hot-Reload

Template Customization

Installing, modifying, and versioning custom templates.

Overview

Templates provide pre-configured settings for common workflows.

Use Cases: - Team-specific default configurations - Project-specific quality gate settings - Environment-specific profiles (dev, staging, production) - Organization-wide standards

Location: `~/.code/templates/`

Template Structure

Template Format

```toml
# ~/.code/templates/premium-quality.toml

[template]
name = "Premium Quality"
version = "1.0.0"
description = "Premium quality configuration with maximum reasoning"
author = "Your Name"
created = "2025-11-17"

# Template configuration (will be merged with config.toml)
[config]
model = "o3"
model_reasoning_effort = "high"
model_reasoning_summary = "detailed"
approval_policy = "never"

[config.quality_gates]
plan = ["gemini", "claude", "code", "gpt_pro"]
tasks = ["code"]
validate = ["gemini", "claude", "code", "gpt_pro"]
audit = ["gemini", "claude", "code", "gpt_codex", "gpt_pro"]
unlock = ["gemini", "claude", "gpt_codex", "gpt_pro"]

[config.hot_reload]
enabled = true
debounce_ms = 2000

[[config.agents]]
name = "gpt_pro"
canonical_name = "gpt_pro"
command = "code"
args = ["--model", "o3", "--config", "model_reasoning_effort=high"]
enabled = true
```

Installing Templates

Method 1: Manual Installation

Steps:

```bash
# Create templates directory
mkdir -p ~/.code/templates
```

```
# Copy template file
cp premium-quality.toml ~/.code/templates/

# List installed templates
code --templates-list
```

Method 2: Install from URL

```
code --template-install https://example.com/templates/premium-quality.toml
```

Behavior: 1. Download template file 2. Validate template structure 3. Save to `~/.code/templates/` 4. Confirm installation

---

Method 3: Install from Git Repository

```
code --template-install github:theturtlecsz/code-templates/premium-quality.toml
```

Behavior: 1. Clone/fetch from GitHub 2. Extract template file 3. Validate and install

---

Using Templates

Apply Template Once

```
code --template premium-quality "task"
```

Behavior: Merges template config with `config.toml` for this session only

---

Set Default Template

```
# ~/.code/config.toml

template = "premium-quality"  # Apply on every session
```

Behavior: Template config merged on startup

Template Precedence

Precedence (highest to lowest): 1. CLI flags (`--model o3`) 2. Environment variables (`CODEX_MODEL=o3`) 3. **Template config** (new tier) 4. Profile (`[profiles.premium]`) 5. config.toml 6. Defaults

Example:

```
# ~/.code/config.toml
model = "gpt-5"

# ~/.code/templates/premium.toml
[config]
```

```toml
model = "o3"

# Usage:
code --template premium "task"
# Effective model: "o3" (template > config.toml)

code --template premium --model gpt-4o "task"
# Effective model: "gpt-4o" (CLI > template)
```

Creating Custom Templates

Step 1: Define Template Metadata

```toml
[template]
name = "My Custom Template"
version = "1.0.0"
description = "Custom configuration for my team"
author = "Team Lead"
created = "2025-11-17"
tags = ["team", "production"]  # Optional
```

Step 2: Define Configuration

```toml
[config]
# Model configuration
model = "gpt-5"
model_provider = "openai"
approval_policy = "on-request"

# Quality gates
[config.quality_gates]
plan = ["gemini", "claude", "code"]
tasks = ["gemini"]
validate = ["gemini", "claude", "code"]
audit = ["gemini", "claude", "gpt_codex"]
unlock = ["gemini", "claude", "gpt_codex"]

# Agents
[[config.agents]]
name = "gemini"
canonical_name = "gemini"
command = "gemini"
enabled = true

# ... more configuration
```

Step 3: Test Template

```bash
# Save template
cp my-template.toml ~/.code/templates/

# Test application
code --template my-template --dry-run "test task"

# Check effective configuration
```

```
code --template my-template --config-dump
```

Step 4: Version and Document

Version Incrementing: - Major: Breaking changes (agent names changed, quality gates restructured) - Minor: New features (new agents, new quality gates) - Patch: Bug fixes, clarifications

Documentation:

```
[template]
name = "My Template"
version = "2.1.0"  # Incremented version
changelog = """
2.1.0 (2025-11-17):
  - Added gpt_pro agent for premium reasoning
  - Increased audit quality gate to 4 agents

2.0.0 (2025-11-10):
  - BREAKING: Renamed 'code' agent to 'gpt_pro'
  - Added cost optimization profile

1.0.0 (2025-11-01):
  - Initial release
"""
```

Template Examples

Cost-Optimized Template

```
# ~/.code/templates/cost-optimized.toml

[template]
name = "Cost Optimized"
version = "1.0.0"
description = "Minimize cost while maintaining quality"

[config]
model = "gpt-4o-mini"
model_reasoning_effort = "low"
approval_policy = "never"

[config.quality_gates]
plan = ["gemini", "claude"]  # 2 cheap agents
tasks = ["gemini"]
validate = ["gemini", "claude"]
audit = ["gemini", "claude"]
unlock = ["gemini", "claude"]

[[config.agents]]
name = "gemini"
canonical_name = "gemini"
command = "gemini"
enabled = true

[[config.agents]]
```

```toml
name = "claude"
canonical_name = "claude"
command = "claude"
enabled = true
```

Usage:

```
code --template cost-optimized "task"
```

CI/CD Template

```toml
# ~/.code/templates/ci-cd.toml

[template]
name = "CI/CD"
version = "1.0.0"
description = "Configuration optimized for CI/CD pipelines"

[config]
model = "gpt-4o"
approval_policy = "never"
sandbox_mode = "read-only"
disable_response_storage = false

[config.quality_gates]
plan = ["gemini", "claude"]
tasks = ["gemini"]
validate = ["gemini", "claude"]
audit = ["gemini", "claude"]
unlock = ["gemini", "claude"]

[config.hot_reload]
enabled = false  # No hot-reload in CI

[config.history]
persistence = "none"  # Don't persist history in CI
```

Usage (in CI):

```
code --template ci-cd "generate report"
```

Team Standard Template

```toml
# ~/.code/templates/team-standard.toml

[template]
name = "Team Standard"
version = "1.2.0"
description = "Standard configuration for our team"
author = "Engineering Team"
organization = "ACME Corp"

[config]
model = "gpt-5"
model_provider = "openai"
approval_policy = "on-request"

# Custom quality gates for our workflow
```

```toml
[config.quality_gates]
plan = ["gemini", "claude", "code"]
tasks = ["gemini"]
validate = ["gemini", "claude", "code"]
audit = ["gemini", "claude", "gpt_codex"]
unlock = ["gemini", "claude", "gpt_codex"]

# Team-specific agents
[[config.agents]]
name = "team-security"
canonical_name = "security"
command = "claude"
instructions = """
Focus on ACME Corp security standards:
- OWASP Top 10 compliance
- PCI-DSS requirements for payment processing
- GDPR compliance for user data
"""
enabled = true

# Use team security agent for audit
[config.quality_gates]
audit = ["security", "gemini", "gpt_codex"]
```

Template Versioning

Version Schema

Format: `MAJOR.MINOR.PATCH`

Versioning Rules: - **MAJOR**: Breaking changes (incompatible with previous versions) - **MINOR**: New features (backward compatible) - **PATCH**: Bug fixes, documentation updates

---

Version Compatibility

Check Template Version:

```bash
code --template-info premium-quality
```

Output:

```
Template: Premium Quality
Version: 2.1.0
Compatible with: codex-rs >= 0.5.0
Author: Your Name
Description: Premium quality configuration with maximum reasoning

Changelog:
  2.1.0 (2025-11-17):
    - Added gpt_pro agent
    - Increased audit quality gate to 4 agents
  2.0.0 (2025-11-10):
    - BREAKING: Renamed agents
  1.0.0 (2025-11-01):
    - Initial release
```

Automatic Template Updates

Enable Auto-Update:

```
# ~/.code/config.toml

template_auto_update = true  # Check for updates on startup
template_update_channel = "stable"  # stable, beta, nightly
```

Manual Update:

```
code --template-update premium-quality
```

Output:

```
Checking for updates...
New version available: 2.2.0 (current: 2.1.0)

Changelog:
  2.2.0 (2025-11-20):
    - Added performance optimizations
    - Fixed quality gate configuration bug

Update? [Y/n]: y

Downloading... ✓
Installing... ✓
Template updated successfully.
```

Template Repositories

Official Template Repository

URL: https://github.com/theturtlecsz/code-templates

Templates: - `premium-quality.toml` - Maximum quality, high cost - `cost-optimized.toml` - Minimum cost, acceptable quality - `ci-cd.toml` - CI/CD pipelines - `team-standard.toml` - Team collaboration - `solo-developer.toml` - Individual productivity

---

Install from Repository

```
# Install from official repository
code --template-install official:premium-quality

# Install from GitHub
code --template-install github:theturtlecsz/code-templates/premium-
quality.toml

# Install from URL
code --template-install
https://raw.githubusercontent.com/.../template.toml
```

Create Your Own Repository

Structure:

```
my-templates/
├── README.md
├── templates.json  # Template index
└── templates/
    ├── premium.toml
    ├── cost.toml
    └── ci.toml
```

templates.json:

```
{
  "repository": "my-templates",
  "version": "1.0.0",
  "templates": [
    {
      "name": "premium",
      "file": "templates/premium.toml",
      "description": "Premium quality template",
      "version": "1.0.0"
    },
    {
      "name": "cost",
      "file": "templates/cost.toml",
      "description": "Cost-optimized template",
      "version": "1.0.0"
    }
  ]
}
```

Debugging Templates

Validate Template

```
code --template-validate ~/.code/templates/my-template.toml
```

Output:

```
Validating template...

Template Metadata:
  ✓ name: "My Template"
  ✓ version: "1.0.0"
  ✓ description: Present

Configuration:
  ✓ model: "gpt-5" (valid)
  ✓ quality_gates.plan: 3 agents (valid)
  ✓ agents: 3 configured (all valid)

Template is valid ✓
```

Dry-Run Template

```
code --template my-template --dry-run "task"
```

Output:

```
Dry-run mode: No actions will be executed

Effective configuration (with template "my-template"):
  model: o3 (from template)
  model_reasoning_effort: high (from template)
  quality_gates.plan: ["gemini", "claude", "code", "gpt_pro"] (from
template)

Would execute: [task description]
```

Compare Templates

```
code --template-diff premium-quality cost-optimized
```

Output:

```
Comparing templates:
  premium-quality v2.1.0
  cost-optimized v1.0.0

Differences:

model:
  - premium-quality: "o3"
  + cost-optimized: "gpt-4o-mini"

model_reasoning_effort:
  - premium-quality: "high"
  + cost-optimized: "low"

quality_gates.plan:
  - premium-quality: ["gemini", "claude", "code", "gpt_pro"] (4
agents)
  + cost-optimized: ["gemini", "claude"] (2 agents)
```

Best Practices

1. Version Templates Semantically

Good:

```toml
[template]
version = "2.1.0"
changelog = """
2.1.0: Added gpt_pro agent
2.0.0: BREAKING: Renamed agents
1.0.0: Initial release
"""
```

---

2. Document Template Usage

Include README:

```
# Premium Quality Template

**Purpose**: Maximum reasoning quality for critical projects

**Cost**: ~$4.50 per full pipeline (66% increase over default)

**When to Use**:
- Critical production features
- Security-sensitive code
- Architecture decisions

**When NOT to Use**:
- Simple formatting tasks
- Routine bug fixes
- Development iteration
```

---

3. Test Templates Before Distribution

```
# Validate template
code --template-validate my-template.toml

# Dry-run test
code --template my-template --dry-run "test task"

# Full test with real task
code --template my-template "simple test task"
```

---

4. Use Templates for Team Consistency

Team workflow:

```
# Install team template
code --template-install github:myorg/code-templates/team-
standard.toml

# Set as default
# Add to ~/.code/config.toml:
template = "team-standard"
```

Summary

Template Customization provides: - Pre-configured settings for common workflows - Team-specific default configurations - Environment-specific profiles (dev, staging, production) - Organization-wide standards

Features: - Template installation (URL, GitHub, local) - Template versioning (semantic versioning) - Template repositories (official + custom) - Automatic updates - Template validation and dry-run

Usage:

```
# Install template
code --template-install official:premium-quality

# Use template once
```

```
code --template premium-quality "task"

# Set as default
# Add to config.toml:
template = "premium-quality"
```

Best Practices: - Version templates semantically - Document template usage (purpose, cost, when to use) - Test templates before distribution - Use templates for team consistency

Next: Theme System

Theme System

TUI themes, color customization, and accessibility options.

Overview

The **theme system** provides visual customization for the TUI (Terminal User Interface).

Features: - 14 built-in themes (7 light + 7 dark) - Custom color overrides - Syntax highlighting themes - Accessibility options - Hot-reload support

Configuration: `[tui.theme]` section in `config.toml`

Theme Configuration

Basic Configuration

```
# ~/.code/config.toml

[tui.theme]
name = "dark-carbon-night"  # Built-in theme
```

Built-in Themes

Light Themes: 1. `light-photon` (default light) 2. `light-prism-rainbow` 3. `light-vivid-triad` 4. `light-porcelain` 5. `light-sandbar` 6. `light-glacier`

Dark Themes: 7. `dark-carbon-night` (default dark) 8. `dark-shinobi-dusk` 9. `dark-oled-black-pro` 10. `dark-amber-terminal` 11. `dark-aurora-flux` 12. `dark-charcoal-rainbow` 13. `dark-zen-garden` 14. `dark-paper-light-pro`

Theme Selection

Auto-Detection (default):

```toml
# Omit theme name to auto-detect based on terminal background
[tui.theme]
# name not specified - auto-detect
```

Behavior: Probes terminal background, selects appropriate light/dark theme

---

Manual Selection:

```toml
[tui.theme]
name = "dark-carbon-night"  # Explicitly select theme
```

Theme Previews

Light Photon (default light): - Background: Light gray (#F5F5F5) - Foreground: Dark gray (#333333) - Primary: Blue (#007ACC) - Secondary: Purple (#8B008B) - Success: Green (#28A745) - Warning: Orange (#FFA500) - Error: Red (#DC3545)

Dark Carbon Night (default dark): - Background: Very dark gray (#1E1E1E) - Foreground: Light gray (#D4D4D4) - Primary: Cyan (#00D4FF) - Secondary: Magenta (#FF00D4) - Success: Green (#4EC9B0) - Warning: Yellow (#DCDCAA) - Error: Red (#F48771)

Dark OLED Black Pro (true black for OLED displays): - Background: Pure black (#000000) - Foreground: White (#FFFFFF) - Primary: Bright cyan (#00FFFF) - Secondary: Bright magenta (#FF00FF) - Success: Bright green (#00FF00) - Warning: Bright yellow (#FFFF00) - Error: Bright red (#FF0000)

Custom Color Overrides

Override Individual Colors

```toml
[tui.theme]
name = "dark-carbon-night"

[tui.theme.colors]
primary = "#00D4FF"      # Override primary color
background = "#1A1A1A"   # Slightly darker background
border_focused = "#FFD700"  # Gold border for focused elements
```

Available Color Fields

Primary Colors: - `primary` - Primary accent color - `secondary` - Secondary accent color - `background` - Background color - `foreground` - Foreground (text) color

UI Elements: - `border` - Default border color - `border_focused` - Focused element border - `selection` - Selected item background - `cursor` - Cursor color

Status Colors: - `success` - Success messages (green) - `warning` - Warning messages (yellow/orange) - `error` - Error messages (red) - `info` - Info messages (blue)

Text Colors: - `text` - Primary text color - `text_dim` - Dimmed/secondary text - `text_bright` - Bright/emphasized text

Syntax Colors: - `keyword` - Syntax keywords (if, for, function) - `string` - String literals - `comment` - Code comments - `function` - Function names

Animation Colors: - `spinner` - Loading spinner color - `progress` - Progress bar color

---

Complete Custom Theme

```toml
[tui.theme]
name = "custom"  # Use 'custom' to define fully custom theme
label = "My Custom Theme"  # Display name
is_dark = true  # Dark theme hint

[tui.theme.colors]
# Primary colors
primary = "#0080FF"
secondary = "#FF0080"
background = "#1C1C1C"
foreground = "#E0E0E0"

# UI elements
border = "#444444"
border_focused = "#0080FF"
selection = "#2A2A2A"
cursor = "#FFFFFF"

# Status colors
success = "#00FF00"
warning = "#FFAA00"
error = "#FF0000"
info = "#00AAFF"

# Text colors
text = "#E0E0E0"
text_dim = "#808080"
text_bright = "#FFFFFF"

# Syntax colors
keyword = "#569CD6"
string = "#CE9178"
comment = "#6A9955"
function = "#DCDCAA"

# Animation colors
spinner = "#0080FF"
progress = "#00FF00"
```

Syntax Highlighting

Highlight Configuration

```toml
[tui.highlight]
theme = "auto"  # Auto-select based on UI theme
```

Options: - `"auto"` - Auto-detect (default) - `"<theme-name>"` - Specific syntect theme

Available Syntect Themes: - `base16-ocean.dark` - `base16-ocean.light` - `InspiredGitHub` - `Solarized (dark)` - `Solarized (light)` - `Monokai`

Custom Syntax Theme

```toml
[tui.highlight]
theme = "Monokai"  # Use Monokai theme for code blocks
```

Terminal Background Detection

Auto-Detection Process

```
1. Query terminal environment variables
   - $TERM (terminal type)
   - $TERM_PROGRAM (terminal program)
   - $COLORFGBG (foreground/background color hint)

2. Probe terminal background (if supported)
   - Send OSC 11 query
   - Parse RGB response
   - Determine if dark/light

3. Select appropriate theme
   - Dark background → dark-carbon-night
   - Light background → light-photon

4. Cache result
   - Store in ~/.code/config.toml
   - Skip probe on subsequent starts
```

Cached Terminal Background

Auto-Cached:

```toml
[tui]
[tui.cached_terminal_background]
is_dark = true
term = "xterm-256color"
term_program = "iTerm.app"
source = "osc11-probe"
rgb = "#1E1E1E"
```

Benefit: Faster startup (no terminal probe)

Force Re-Detection

```
        # Delete cached background
        code --clear-terminal-cache

        # Or manually edit config.toml and remove
[tui.cached_terminal_background]
```

Accessibility Options

High Contrast Mode

Enable via Custom Theme:

```toml
[tui.theme]
name = "custom"
label = "High Contrast"

[tui.theme.colors]
background = "#000000"  # Pure black
foreground = "#FFFFFF"  # Pure white
primary = "#00FFFF"     # Bright cyan
error = "#FF0000"       # Bright red
success = "#00FF00"     # Bright green
border_focused = "#FFFF00"  # Bright yellow
```

Large Text (Terminal Setting)

Increase Terminal Font Size:

```
Terminal Settings → Font Size → 16pt (or larger)
```

Note: TUI adapts to terminal font size automatically

Color Blindness Support

Protanopia/Deuteranopia (red-green color blindness):

```toml
[tui.theme]
name = "custom"
label = "Color Blind Friendly"

[tui.theme.colors]
# Avoid red/green distinction
success = "#0080FF"  # Blue instead of green
error = "#FF8800"    # Orange instead of red
warning = "#FFFF00"  # Yellow (safe)
info = "#00FFFF"     # Cyan (safe)
```

Tritanopia (blue-yellow color blindness):

```toml
[tui.theme.colors]
# Avoid blue/yellow distinction
primary = "#FF00FF"  # Magenta instead of blue
warning = "#FF8800"  # Orange instead of yellow
```

Theme Customization Examples

Solarized Dark

```toml
[tui.theme]
name = "custom"
label = "Solarized Dark"
is_dark = true

[tui.theme.colors]
background = "#002B36"  # base03
foreground = "#839496"  # base0
primary = "#268BD2"     # blue
secondary = "#D33682"   # magenta
success = "#859900"     # green
warning = "#B58900"     # yellow
error = "#DC322F"       # red
info = "#2AA198"        # cyan
```

Gruvbox Dark

```toml
[tui.theme]
name = "custom"
label = "Gruvbox Dark"
is_dark = true

[tui.theme.colors]
background = "#282828"  # dark0
foreground = "#EBDBB2"  # light1
primary = "#83A598"     # blue
secondary = "#D3869B"   # purple
success = "#B8BB26"     # green
warning = "#FABD2F"     # yellow
error = "#FB4934"       # red
info = "#8EC07C"        # aqua
```

Dracula

```toml
[tui.theme]
name = "custom"
label = "Dracula"
is_dark = true

[tui.theme.colors]
background = "#282A36"  # Background
foreground = "#F8F8F2"  # Foreground
primary = "#BD93F9"     # Purple
secondary = "#FF79C6"   # Pink
success = "#50FA7B"     # Green
warning = "#F1FA8C"     # Yellow
error = "#FF5555"       # Red
info = "#8BE9FD"        # Cyan
```

Debugging Themes

Test Theme

```
# Test theme without saving to config
code --theme dark-carbon-night
```

Preview All Themes

```
code --themes-preview
```

Output: Opens TUI showing all themes side-by-side

Dump Current Theme

```
code --theme-dump
```

Output:

```
[tui.theme]
name = "dark-carbon-night"

[tui.theme.colors]
primary = "#00D4FF"
background = "#1E1E1E"
foreground = "#D4D4D4"
# ... all effective colors
```

Validate Custom Theme

```
code --theme-validate ~/.code/config.toml
```

Output:

```
Validating theme...

Theme: custom (My Custom Theme)
  ✓ primary: #0080FF (valid hex)
  ✓ background: #1C1C1C (valid hex)
  ✓ foreground: #E0E0E0 (valid hex)
  ✓ All 24 color fields valid

Theme is valid ✓
```

Hot-Reload Support

Live Theme Changes

Edit `config.toml`:

```
[tui.theme]
name = "dark-carbon-night"  # Change to different theme
```

Save: TUI reloads theme within 2 seconds (debounced)

Notification:

```
✅ Config reloaded successfully
   - Theme changed: light-photon → dark-carbon-night
```

Live Color Tweaking

Edit `config.toml`:

```toml
[tui.theme.colors]
primary = "#FF0080"  # Change primary color
```

Save: Color updates instantly (hot-reload)

Use Case: Iterative theme customization

Spinner Customization

Built-in Spinners

Default: `"diamond"`

Available Spinners (from cli-spinners): - `dots`, `dots2`, `dots3` (simple dots) - `line`, `line2` (horizontal line) - `pipe`, `simpleDots` (classic spinners) - `star`, `star2` (star animation) - `flip`, `hamburger` (quirky animations) - `growVertical`, `growHorizontal` (growth animations) - `balloon`, `balloon2` (balloon animations) - `noise`, `bounce` (dynamic animations) - `boxBounce`, `boxBounce2` (box animations) - `triangle`, `arc` (geometric shapes) - `circle`, `circleQuarters`, `circleHalves` (circle animations) - `squish`, `toggle` (squish/toggle animations) - `layer`, `betaWave` (wave animations) - `fingerDance`, `fistBump` (emoji animations) - `soccerHeader`, `mindblown` (emoji animations) - `speaker`, `orangePulse` (pulse animations) - `bluePulse`, `orangeBluePulse` (multi-color pulses) - `timeTravel`, `aesthetic` (special effects) - `dqpb`, `weather` (themed spinners) - `christmas`, `grenade` (themed spinners) - `point`, `layer` (pointer animations) - `betaWave`, `shark` (wave/shark animations)

Spinner Configuration

```toml
[tui.spinner]
name = "dots"  # Simple dots spinner
```

Custom Spinner

```toml
[tui.spinner]
name = "my-spinner"

[tui.spinner.custom.my-spinner]
interval = 80  # Milliseconds between frames
frames = ["⠁", "⠂", "⠄", "⡀", "⢀", "⠠", "⠐", "⠈", "⠁", "⠂"]
label = "My Custom Spinner"  # Optional display name
```

Stream Animation

Stream Configuration

```toml
[tui.stream]
answer_header_immediate = false  # Show header before first text
show_answer_ellipsis = true      # Show "..." while waiting
commit_tick_ms = 50              # Animation speed (50ms default)
soft_commit_timeout_ms = 400     # Commit after 400ms idle
soft_commit_chars = 160          # Commit after 160 chars
relax_list_holdback = false      # Allow list line commits
relax_code_holdback = false      # Allow code block commits
responsive = false               # Enable snappier preset
```

Responsive Preset

Enable:

```toml
[tui.stream]
responsive = true  # Enable snappier preset
```

Effect: Overrides to: - `commit_tick_ms = 30` (faster animation) - `soft_commit_timeout_ms = 400` - `soft_commit_chars = 160`

Use Case: Users who prefer instant response over smooth animation

---

Best Practices

1. Use Built-in Themes When Possible

Reason: Pre-tested, well-balanced, maintained

Example:

```toml
[tui.theme]
name = "dark-carbon-night"  # Built-in theme
```

---

2. Override Colors Sparingly

Good (1-2 color overrides):

```toml
[tui.theme]
name = "dark-carbon-night"

[tui.theme.colors]
primary = "#00FFAA"  # Just change primary accent
```

Bad (override everything):

```toml
[tui.theme.colors]
# Defining all 24 colors - hard to maintain
primary = "..."
secondary = "..."
# ... 22 more fields
```

---

3. Test Themes in Different Scenarios

Test Cases: - Success messages (green) - Error messages (red) - Warning messages (yellow) - Info messages (blue) - Code syntax highlighting - Spinner animations - Border focus states

---

4. Consider Accessibility

Contrast Ratio: WCAG AA requires 4.5:1 for normal text

Check Contrast:

```
# Use online tool: https://webaim.org/resources/contrastchecker/

# Background: #1E1E1E
# Foreground: #D4D4D4
# Contrast: 12.63:1 ✓ (WCAG AAA)
```

Summary

Theme System provides: - 14 built-in themes (7 light + 7 dark) - Custom color overrides (24 color fields) - Syntax highlighting themes - Spinner customization (50+ built-in, custom support) - Stream animation tuning - Hot-reload support (live theme changes) - Accessibility options (high contrast, color blind support)

Configuration:

```toml
[tui.theme]
name = "dark-carbon-night"  # Built-in theme

[tui.theme.colors]
primary = "#00D4FF"  # Optional color override

[tui.highlight]
theme = "auto"  # Syntax highlighting

[tui.spinner]
name = "dots"  # Spinner style

[tui.stream]
responsive = false  # Animation speed
```

Best Practices: - Use built-in themes when possible - Override colors sparingly (1-2 overrides) - Test themes in different scenarios - Consider accessibility (contrast, color blindness)

Next: Configuration Reference (for complete schema)
