SPEC-DOC-002: Core Architecture Documentation

Status: Pending **Priority**: P0 (High) **Estimated Effort**: 16-20 hours **Target Audience**: Contributors, system architects **Created**: 2025-11-17

Objectives

Document the complete internal architecture of the theturtlecsz/code project: 1. System overview and design philosophy 2. Cargo workspace structure (24 crates, dependencies) 3. TUI architecture (Ratatui, async/sync boundaries) 4. Core execution system (agent orchestration, tmux management) 5. MCP integration (native client, connection management) 6. Database layer (SQLite, schema, transactions) 7. Configuration system (5-tier precedence, hot-reload)

Scope

In Scope

System Architecture: - High-level component diagram - Data flow diagrams - Integration points between subsystems - Fork-specific additions (98.8% isolation from upstream)

Cargo Workspace (24 crates): - Crate dependency graph - Purpose of each crate - Inter-crate dependencies - Build profiles (dev-fast, release, perf)

TUI System: - Ratatui architecture - ChatWidget structure (912K LOC mod.rs) - Async/sync boundary handling (Handle::block_on pattern) - Friend module pattern for spec-kit isolation - Widget lifecycle and rendering

Core Execution: - Agent spawning and orchestration - Tmux session management - Model provider clients (OpenAI, Anthropic, Google) - Protocol implementation - Timeout and retry logic (AR-1 through AR-4)

MCP Integration: - Native client implementation (5.3× speedup) - App-level shared connection manager (ARCH-005) - Server lifecycle management - Tool invocation patterns

Database Layer: - SQLite schema (consensus_artifacts.db) - Transaction handling (IMMEDIATE mode) - File locking (fs2 crate, ARCH-007) - Retry logic and error handling - Auto-vacuum strategy (99.95% reduction)

Configuration System: - 5-tier precedence (CLI > shell > profile > TOML > defaults) - Hot-reload mechanism (config_reload.rs, 300ms debounce) - Profile system - Environment variable overrides

Out of Scope

- User-facing configuration guide (see SPEC-DOC-006)
- Spec-kit framework details (see SPEC-DOC-003)
- Testing infrastructure (see SPEC-DOC-004)
- Security implementation (see SPEC-DOC-007)

Deliverables

Primary Documentation

1. **content/system-overview.md** - Architecture overview, component diagram
2. **content/cargo-workspace.md** - Workspace structure, crate guide
3. **content/tui-architecture.md** - Ratatui, async/sync, ChatWidget
4. **content/core-execution.md** - Agent orchestration, tmux, providers
5. **content/mcp-integration.md** - Native client, connection manager
6. **content/database-layer.md** - SQLite, schema, transactions
7. **content/configuration-system.md** - 5-tier precedence, hot-reload

Supporting Materials

- **evidence/diagrams/** - Architecture diagrams, data flow charts
- **adr/** - Key architectural decisions (if new decisions documented)

Success Criteria

- [ ] Complete component diagram created
- [ ] All 24 crates documented with purposes
- [ ] Async/sync boundary patterns explained with examples
- [ ] MCP integration fully documented (connection lifecycle, retry logic)
- [ ] SQLite schema documented with ER diagram
- [ ] Configuration precedence clearly illustrated
- [ ] All file paths reference actual source code locations

Related SPECs

- SPEC-DOC-000 (Master)
- SPEC-DOC-001 (User Onboarding - references architecture concepts)
- SPEC-DOC-003 (Spec-Kit - detailed spec-kit architecture)
- SPEC-DOC-004 (Testing - test infrastructure architecture)
- SPEC-DOC-005 (Development - build system details)

Status: Structure defined, content pending

Cargo Workspace Structure

Complete guide to the 24-crate workspace architecture.

Workspace Overview

Location: `/home/user/code/codex-rs/Cargo.toml`

Statistics: - **24 member crates** - **226,607 lines of Rust code** - **538 source files** - **Edition 2024** (Rust 2024 edition features)

Crate Categories

Core Application Crates

tui - Terminal User Interface

Purpose: Main application binary, Ratatui-based TUI

Key Modules: - `app.rs`: Application state and event loop - `chatwidget/`: Conversation interface (912K LOC mod.rs) - `chatwidget/spec_kit/`: Fork feature integration (55 modules)

Dependencies: `ratatui`, `codex-core`, `spec-kit`, `mcp-client`, `tokio`

Binary: `code-tui`

File: `codex-rs/tui/Cargo.toml`

---

cli - CLI Wrapper

Purpose: Command-line interface entry point

Responsibilities: - Argument parsing (clap) - Shell completion generation - Delegates to `code-tui` binary

Dependencies: `codex-tui`, `codex-core`, `clap`, `clap_complete`

Binary: `code`

File: `codex-rs/cli/Cargo.toml`

---

spec-kit - Multi-Agent Automation Framework (Fork)

Purpose: Reusable library for spec-kit automation

Modules: - `config/`: Configuration system (hot-reload, 5-tier precedence) - `retry/`: Exponential backoff retry logic - `types.rs`: Core types (SpecStage, QualityCheckpoint) - `error.rs`: Error handling

Dependencies: `codex-core`, `mcp-types`, `rusqlite`, `notify`, `tokio`

Note: Can be extracted as standalone crate (MAINT-10 future work)

File: `codex-rs/spec-kit/Cargo.toml`

Backend Service Crates

core - Backend Services

Purpose: Backend orchestration (conversation, MCP, DB, config)

Key Modules: - `conversation_manager.rs`: Agent lifecycle - `mcp_connection_manager.rs`: MCP server aggregation - `db/`: SQLite connection pooling, transactions - `config_types.rs`: Configuration data structures - `protocol.rs`: Model provider abstraction

Dependencies: `mcp-client`, `codex-protocol`, `rusqlite`, `r2d2`, `tokio`

File: `codex-rs/core/Cargo.toml`

MCP (Model Context Protocol) Crates

mcp-client - Async MCP Client

Purpose: Subprocess communication with MCP servers

Key Features: - Line-delimited JSON-RPC over stdio - Concurrent reader/writer tasks (prevent deadlock) - Request/response pairing via HashMap - 1MB buffer for large responses

Dependencies: `mcp-types`, `tokio`, `tokio-util`, `serde_json`

File: `codex-rs/mcp-client/src/mcp_client.rs:63-150`

---

mcp-server - MCP Server Implementation

Purpose: Implements MCP server for tool exposure

Dependencies: `codex-core`, `codex-protocol`, `mcp-types`, `tokio`

File: `codex-rs/mcp-server/Cargo.toml`

---

mcp-types - Protocol Types

Purpose: JSON-RPC and MCP protocol definitions

Key Types: - `JSONRPCMessage`: Request/Response/Notification - `ToolInfo`: Tool metadata - `McpServerConfig`: Server configuration

Dependencies: `serde`, `serde_json`, `ts-rs` (TypeScript bindings)

File: `codex-rs/mcp-types/Cargo.toml`

Protocol & Model Provider Crates

protocol - OpenAI Protocol

Purpose: OpenAI API client and types

Features: - Responses API support - Chat Completions API support - Streaming SSE support - Request/response types

Dependencies: `serde`, `serde_json`, `reqwest`

File: `codex-rs/protocol/Cargo.toml`

---

chatgpt - ChatGPT Auth

Purpose: ChatGPT authentication flow

Dependencies: `reqwest`, `serde`

File: `codex-rs/chatgpt/Cargo.toml`

Utility & Tooling Crates

common - Shared Utilities

Purpose: Common utilities across crates

Modules: - Model presets (GPT-5, Claude, Gemini) - Elapsed time formatting - CLI helpers

Dependencies: `serde`, `reqwest`, `clap`

File: `codex-rs/common/Cargo.toml`

---

git-tooling - Git Operations

Purpose: Git integration helpers

File: `codex-rs/git-tooling/Cargo.toml`

---

file-search - File Search

Purpose: Fuzzy file search (@ trigger in composer)

Dependencies: `nucleo-matcher` (fuzzy matching)

File: `codex-rs/file-search/Cargo.toml`

Execution & Sandbox Crates

exec - Command Execution

Purpose: Sandboxed command execution

File: `codex-rs/exec/Cargo.toml`

---

execpolicy - Execution Policy

Purpose: Approval policy enforcement

File: `codex-rs/execpolicy/Cargo.toml`

---

linux-sandbox - Linux Sandboxing

Purpose: Landlock, seccomp sandboxing

Dependencies: `landlock`, `seccompiler`

File: `codex-rs/linux-sandbox/Cargo.toml`

Supporting Crates

login: Authentication helpers **browser**: Browser integration (CDP) **ollama**: Ollama model support **arg0**: Binary name detection **apply-patch**: Diff application **ansi-escape**: ANSI escape code handling **codex-version**: Version utilities

---

Dependency Graph

```
tui (main binary)
├── spec-kit (fork feature)
│   ├── mcp-types
│   ├── codex-core
│   │   ├── mcp-client
```

```
│   │   │   └── mcp-types
│   │   ├── codex-protocol
│   │   ├── rusqlite
│   │   └── r2d2
│   └── rusqlite (direct)
├── codex-core
├── mcp-client
├── codex-protocol
├── codex-common
├── ratatui (v0.29.0 patched)
└── tokio

cli (entry point)
├── codex-tui
└── clap

core (backend)
├── mcp-client
├── codex-protocol
├── rusqlite
├── r2d2
└── tokio
```

Key Observations: - **spec-kit** depends on **core** (reuses DB, MCP) - **tui** depends on **spec-kit** (friend module pattern) - **mcp-client** is lightweight (only `mcp-types`, `tokio`) - **core** aggregates backend services

---

Build Profiles

dev-fast (Default Development)

```toml
[profile.dev-fast]
inherits = "dev"
opt-level = 1               # Basic optimization
debug = 1                   # Line tables only
incremental = true          # Incremental compilation
codegen-units = 256         # Parallel codegen
lto = "off"                 # No LTO (faster builds)
```

Use: `./build-fast.sh` or `cargo build --profile dev-fast`

Build Time: ~30-60 seconds (incremental)

Binary: `codex-rs/target/dev-fast/code`

---

release (Production)

```toml
[profile.release]
lto = "fat"                 # Full LTO (link-time optimization)
strip = "symbols"           # Remove debug symbols
codegen-units = 1           # Single codegen unit (max optimization)
```

Use: `cargo build --release`

Build Time: ~5-10 minutes (full rebuild)

Binary Size: ~15-20 MB (stripped)

Binary: `codex-rs/target/release/code`

---

perf (Performance Profiling)

```toml
[profile.perf]
inherits = "release"
incremental = true
codegen-units = 256
lto = "off"
debug = 2                  # Full debug info
strip = "none"             # Keep symbols
split-debuginfo = "packed" # Separate debug file
```

Use: `cargo build --profile perf`

Purpose: Performance profiling with `perf`, `flamegraph`

Workspace-Level Configuration

Shared Dependencies

```toml
[workspace.dependencies]
# Internal crates (path dependencies)
codex-core = { path = "core" }
codex-tui = { path = "tui" }
spec-kit = { path = "spec-kit" }
mcp-client = { path = "mcp-client" }
mcp-types = { path = "mcp-types" }
# ... 20+ internal crates

# External crates (version pinning)
tokio = "1"
ratatui = "0.29.0"
rusqlite = { version = "0.37", features = ["bundled"] }
reqwest = "0.12"
serde = "1"
serde_json = "1"
anyhow = "1"
thiserror = "2.0.16"
# ... 100+ external dependencies
```

Benefits: - Single version source of truth - Automatic version consistency - Easier dependency updates

---

Workspace Lints

```toml
[workspace.lints.clippy]
expect_used = "deny"               # Forbid .expect()
unwrap_used = "deny"               # Forbid .unwrap()
manual_ok_or = "deny"              # Enforce .ok_or()
needless_borrow = "deny"           # Remove unnecessary borrows
redundant_clone = "deny"           # Remove unnecessary clones
uninlined_format_args = "deny"     # Use format!("{var}") not
format!("{}", var)
# ... 30+ lints
```

Enforcement: All crates inherit workspace lints unless overridden

---

Patched Dependencies

Ratatui Fork:

```
        [patch.crates-io]
        ratatui = { git = "https://github.com/nornagon/ratatui", branch =
"nornagon-v0.29.0-patch" }
```

Reason: Custom patches for TUI improvements

Build System

Fast Builds (Development)

```
        # Use build-fast.sh
        ./build-fast.sh

        # Or directly
        cd codex-rs
        cargo build --profile dev-fast --bin code --bin code-tui
```

Output: `codex-rs/target/dev-fast/code`

Release Builds (Production)

```
        # Full release build
        cd codex-rs
        cargo build --release --bin code --bin code-tui --bin code-exec

        # Quick build (code only)
        npm run build:quick
```

Output: `codex-rs/target/release/code`

Testing

```
        cd codex-rs

        # All tests
        cargo test

        # Specific crate
        cargo test -p spec-kit

        # Specific test
        cargo test -p codex-tui spec_auto
```

Code Quality

```
        cd codex-rs
```

```
# Format
cargo fmt --all

# Lint
cargo clippy --workspace --all-targets --all-features -- -D warnings

# Check (no codegen)
cargo check --workspace
```

Crate Size Breakdown

Crate	LOC (Rust)	Files	Key Responsibility
tui	~45,000	120	UI, ChatWidget, Spec-Kit integration
spec-kit (lib)	~8,000	25	Config, retry, types (reusable)
spec-kit (tui integration)	~35,000	55	Command handlers, consensus, pipeline
core	~30,000	85	Backend services, MCP, DB, config
mcp-client	~3,500	12	Async MCP client
mcp-server	~5,000	18	MCP server implementation
protocol	~12,000	35	OpenAI protocol client
Others	~88,000	~230	Utilities, tooling, sandbox

Total: ~226,607 LOC across 538 files

Adding New Crates

When to Create a New Crate

Good reasons: - ✅ Reusable component (can extract to standalone library) - ✅ Clear responsibility boundary - ✅ Independent compilation (speeds up builds) - ✅ Different dependency requirements - ✅ Potential for external use (spec-kit library)

Bad reasons: - ❌ Small utility module (use `common` instead) - ❌ Tightly coupled to single crate - ❌ No clear abstraction boundary

---

Creating a New Crate

```
cd codex-rs

# Create new crate in workspace
cargo new --lib my-new-crate

# Or with specific edition
cargo new --lib --edition 2024 my-new-crate
```

Add to workspace (`Cargo.toml`):

```toml
[workspace]
members = [
    "ansi-escape",
    # ... existing crates
    "my-new-crate",  # Add here
]
```

Add workspace dependency:

```toml
[workspace.dependencies]
my-new-crate = { path = "my-new-crate" }
```

Use in other crates:

```toml
# In another crate's Cargo.toml
[dependencies]
my-new-crate = { workspace = true }
```

Next Steps

- TUI Architecture - Detailed TUI and ChatWidget design
- Core Execution - Agent orchestration and providers
- MCP Integration - Native MCP client details
- Database Layer - SQLite optimization

File References: - Workspace: `codex-rs/Cargo.toml:1-234` - Build profiles: `codex-rs/Cargo.toml:201-234` - TUI: `codex-rs/tui/Cargo.toml` - Spec-Kit: `codex-rs/spec-kit/Cargo.toml` - Core: `codex-rs/core/Cargo.toml` - MCP Client: `codex-rs/mcp-client/Cargo.toml`

Configuration System

Layered configuration with hot-reload and 5-tier precedence.

Overview

The configuration system implements the **12-factor app pattern**: 1. **Defaults**: Built-in fallback values (in code) 2. **Config file**: `~/.code/config.toml` 3. **Environment variables**: `OPENAI_API_KEY`, etc. 4. **Profiles**: Named configurations (`[profiles.premium]`) 5. **CLI flags**: `--model gpt-5`, `--config key=value`

Hot-Reload: File changes detected and applied without restart (<100ms latency)

Location: `codex-rs/spec-kit/src/config/`

5-Tier Precedence

Precedence Order

Highest to Lowest (higher overrides lower):

1. **CLI Flags** (highest priority)

```
code --model gpt-5 --config approval_policy=never
```

2. **Shell Environment**

```
export OPENAI_API_KEY="sk-proj-..."
export CODEX_HOME="/custom/path"
```

3. **Profile** (selected via `--profile` or `profile = "name"`)

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

4. **Config File** (`~/.code/config.toml`)

```
model = "gpt-5"
approval_policy = "on_request"
```

5. **Defaults** (lowest priority, built-in)

```
impl Default for Config {
    fn default() -> Self {
        Self {
            model: "gpt-5-codex".to_string(),
            approval_policy: ApprovalPolicy::OnRequest,
            ...
        }
    }
}
```

Precedence Examples

Example 1: Model selection

```
# Default: gpt-5-codex
# config.toml: model = "gpt-5"
# --model o3

Effective value: "o3" (CLI flag wins)
```

Example 2: API key

```
# Default: none
# config.toml: (none)
# export OPENAI_API_KEY="sk-proj-123"
# --config model_provider.openai.api_key="sk-proj-456"

Effective value: "sk-proj-456" (CLI flag wins)
```

Example 3: Profile

```
# Default: approval_policy = "on_request"
# config.toml: approval_policy = "untrusted"
# [profiles.premium]: approval_policy = "never"
# --profile premium
```

```
Effective value: "never" (profile wins over config.toml)
```

Configuration Loading

Loader Architecture

Location: `codex-rs/spec-kit/src/config/loader.rs`

```rust
pub struct ConfigLoader {
    defaults: AppConfig,
    file_path: Option<PathBuf>,
    env_overrides: HashMap<String, String>,
    cli_overrides: HashMap<String, String>,
}

impl ConfigLoader {
    pub fn new() -> Self {
        Self {
            defaults: AppConfig::default(),
            file_path: None,
            env_overrides: HashMap::new(),
            cli_overrides: HashMap::new(),
        }
    }

    pub fn with_file(mut self, path: impl Into<PathBuf>) -> Self {
        self.file_path = Some(path.into());
        self
    }

    pub fn with_env_overrides(mut self, overrides: HashMap<String,
String>) -> Self {
        self.env_overrides = overrides;
        self
    }

    pub fn with_cli_overrides(mut self, overrides: HashMap<String,
String>) -> Self {
        self.cli_overrides = overrides;
        self
    }

    pub fn load(self) -> Result<AppConfig> {
        // Layer 1: Defaults
        let mut config = self.defaults;

        // Layer 2: Config file
        if let Some(path) = self.file_path {
            if path.exists() {
                let file_config: AppConfig =
toml::from_str(&std::fs::read_to_string(path)?)?;
                config.merge(file_config);
            }
        }

        // Layer 3: Environment variables
        for (key, value) in self.env_overrides {
```

```
                    config.set_from_string(&key, &value)?;
                }

                // Layer 4: CLI overrides
                for (key, value) in self.cli_overrides {
                    config.set_from_string(&key, &value)?;
                }

                // Layer 5: Profile (if selected)
                if let Some(profile_name) = &config.profile {
                    if let Some(profile) = config.profiles.get(profile_name)
{
                        config.merge(profile.clone());
                    }
                }

                config.validate()?;
                Ok(config)
            }
        }
```

Environment Variable Mapping

Pattern: `SPECKIT_<SECTION>_<KEY>`

Examples:

```
        # Model configuration
        SPECKIT_MODEL_NAME="gpt-5"
        SPECKIT_MODEL_REASONING_EFFORT="high"

        # Quality gates
        SPECKIT_QUALITY_GATES_PLAN="gemini,claude,code"
        SPECKIT_QUALITY_GATES_TASKS="gemini"

        # Evidence configuration
        SPECKIT_EVIDENCE_BASE_DIR="/custom/evidence"
        SPECKIT_EVIDENCE_MAX_SIZE_MB="50"
```

Parsing:

```
        fn parse_env_key(key: &str) -> Option<(String, String)> {
            if let Some(stripped) = key.strip_prefix("SPECKIT_") {
                let parts: Vec<&str> = stripped.split('_').collect();
                if parts.len() >= 2 {
                    let section =
parts[0..parts.len()-1].join("_").to_lowercase();
                    let key = parts[parts.len()-1].to_lowercase();
                    return Some((section, key));
                }
            }
            None
        }
```

Hot-Reload System

Architecture

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

Location: `codex-rs/spec-kit/src/config/hot_reload.rs:1-100`

HotReloadWatcher

```rust
        use notify::{Watcher, RecursiveMode, Event};
        use notify_debouncer_full::{new_debouncer, Debouncer, FileIdMap};
        use std::sync::{Arc, RwLock, Mutex};
        use tokio::sync::mpsc;

        pub enum ConfigReloadEvent {
            FileChanged(PathBuf),      // File change detected (pre-
validation)
            ReloadSuccess,             // Config reloaded successfully
            ReloadFailed(String),      // Validation error (old config
preserved)
        }

        pub struct HotReloadWatcher {
            debouncer: Arc<Mutex<Debouncer<...>>>,
            config: Arc<RwLock<AppConfig>>,
            event_tx: mpsc::Sender<ConfigReloadEvent>,
        }

        impl HotReloadWatcher {
            pub async fn new(
                config_path: impl Into<PathBuf>,
                debounce_duration: Duration,
            ) -> Result<Self> {
                let config_path = config_path.into();
                let config = Arc::new(RwLock::new(ConfigLoader::new()
                    .with_file(&config_path)
                    .load()?));

                let (event_tx, _event_rx) = mpsc::channel(32);

                // Create debouncer (buffers events for 300ms)
                let file_id_map = FileIdMap::new();
                let config_clone = Arc::clone(&config);
                let event_tx_clone = event_tx.clone();
                let path_clone = config_path.clone();

                let debouncer = new_debouncer(
                    debounce_duration,
                    Some(Duration::from_secs(1)),  // Tick interval
                    move |events: DebounceEventResult| {
                        match events {
                            Ok(events) => {
                                for event in events {
                                    if event.paths.contains(&path_clone) {
                                        // Config file changed, attempt
reload
                                        match reload_config(&config_clone,
&path_clone) {
                                            Ok(_) => {
```

```
                                                let _ =
event_tx_clone.try_send(

ConfigReloadEvent::ReloadSuccess
                                                );
                                            },
                                            Err(e) => {
                                                let _ =
event_tx_clone.try_send(

ConfigReloadEvent::ReloadFailed(e.to_string())
                                                );
                                            },
                                        }
                                    }
                                }
                            },
                            Err(e) => {
                                eprintln!("Watch error: {:?}", e);
                            },
                        }
                    },
                )?;

                // Watch config file
                debouncer.watcher().watch(&config_path,
RecursiveMode::NonRecursive)?;

                Ok(Self {
                    debouncer: Arc::new(Mutex::new(debouncer)),
                    config,
                    event_tx,
                })
            }

            pub fn get_config(&self) -> Arc<AppConfig> {
                // Read lock held briefly (<1µs) to clone Arc
                Arc::clone(&*self.config.read().unwrap())
            }
        }

        fn reload_config(
            config: &Arc<RwLock<AppConfig>>,
            path: &Path,
        ) -> Result<()> {
            // Parse and validate new config
            let new_config = ConfigLoader::new()
                .with_file(path)
                .load()?;

            // Atomic write lock (<1ms)
            *config.write().unwrap() = new_config;

            Ok(())
        }
```

Debouncing

Purpose: Prevent reload storms (e.g., text editor save creates multiple events)

Configuration: 300ms debounce window

Behavior:

```
t=0ms:    File change event 1
t=50ms:   File change event 2 (reset timer)
t=100ms:  File change event 3 (reset timer)
t=400ms:  No more events for 300ms → Trigger reload
```

Result: Only one reload despite multiple filesystem events

Lock Contention

Read Locks (frequent, fast):

```
let config = watcher.get_config();  // Arc::clone, <1µs
```

Write Locks (rare, fast):

```
*config.write().unwrap() = new_config;  // <1ms
```

Performance: - Read locks: Non-blocking (RwLock allows concurrent readers) - Write lock: Blocks readers briefly (<1ms) - Reload frequency: Low (manual file edits only)

CPU overhead: <0.5% (idle, file watching)

Performance Metrics

Hot-Reload Latency

Measured end-to-end:

```
File save → Filesystem event → Debounce wait → Parse TOML → Validate
→ Write lock → Event
  0ms          ~10ms              300ms          ~20ms        ~5ms
<1ms      ~1ms

Total: ~336ms (p50)
       ~420ms (p95)
```

Acceptable: Sub-second reload for manual config edits

Lock Timing

Read lock (get_config):

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

Total: <1µs
```

Write lock (reload_config):

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

Total: <1ms
```

Validation

Schema Validation

```rust
        impl AppConfig {
            pub fn validate(&self) -> Result<()> {
                // Model provider must exist
                if self.model_providers.is_empty() {
                    return Err(Error::NoModelProviders);
                }

                // Selected provider must be configured
                if !self.model_providers.contains_key(&self.model_provider)
{
                    return
Err(Error::ProviderNotFound(self.model_provider.clone()));
                }

                // Quality gate agents must be valid
                for agents in self.quality_gates.values() {
                    for agent in agents {
                        if !self.agents.iter().any(|a| a.canonical_name ==
*agent) {
                            return Err(Error::AgentNotFound(agent.clone()));
                        }
                    }
                }

                // Evidence max size must be reasonable
                if self.evidence.max_size_mb > 1000 {
                    return
Err(Error::EvidenceSizeTooLarge(self.evidence.max_size_mb));
                }

                Ok(())
            }
        }
```

On validation failure: Preserve old config, emit `ReloadFailed` event

Type Safety

TOML parsing uses `serde`:

```rust
        #[derive(Deserialize, Serialize, Clone)]
        pub struct AppConfig {
            pub model: String,
            pub model_provider: String,
```

```
    pub approval_policy: ApprovalPolicy,  // Enum (type-safe)
    pub quality_gates: QualityGateConfig,
    pub evidence: EvidenceConfig,
    // ... 20+ fields
}

#[derive(Deserialize, Serialize, Clone)]
#[serde(rename_all = "snake_case")]
pub enum ApprovalPolicy {
    Untrusted,
    OnFailure,
    OnRequest,
    Never,
}
```

Benefits: - Compile-time type checking - Automatic deserialization - Invalid values rejected at parse time

---

Profile System

Profile Definition

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

# Default configuration
model = "gpt-5"
approval_policy = "on_request"

# Profile for premium reasoning
[profiles.premium]
model = "o3"
model_reasoning_effort = "high"
model_reasoning_summary = "detailed"
approval_policy = "never"

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

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

Profile Selection

Via config:

```
profile = "premium"  # Active profile
```

Via CLI:

```
code --profile premium "complex task"
code --profile fast "simple formatting"
```

```
code --profile ci "generate report"
```

Precedence: CLI `--profile` > config `profile` field > no profile

Profile Merging

```
impl AppConfig {
    pub fn merge(&mut self, other: AppConfig) {
        // Non-Option fields: other wins
        self.model = other.model;
        self.approval_policy = other.approval_policy;

        // Option fields: other overwrites if Some
        if other.model_reasoning_effort.is_some() {
            self.model_reasoning_effort =
other.model_reasoning_effort;
        }

        // Collections: extend (not replace)
        self.quality_gates.extend(other.quality_gates);
        self.agents.extend(other.agents);
    }
}
```

Registry System

Command Registry (Spec-Kit)

Location: `codex-rs/tui/src/chatwidget/spec_kit/command_registry.rs`

```
pub trait SpecKitCommand: Send + Sync {
    fn name(&self) -> &'static str;
    fn aliases(&self) -> &[&'static str];
    fn description(&self) -> &'static str;
    fn execute(&self, widget: &mut ChatWidget, args: String);
}

pub struct CommandRegistry {
    commands: HashMap<String, Box<dyn SpecKitCommand>>,
    by_alias: HashMap<String, String>,  // Alias → primary name
}

impl CommandRegistry {
    pub fn register(&mut self, command: Box<dyn SpecKitCommand>) {
        let name = command.name().to_string();

        // Register primary name
        self.commands.insert(name.clone(), command);

        // Register aliases
        for alias in command.aliases() {
            self.by_alias.insert(alias.to_string(), name.clone());
        }
    }

    pub fn find(&self, name: &str) -> Option<&dyn SpecKitCommand> {
```

```
            // Resolve alias → primary name
            let resolved_name = self.by_alias.get(name).unwrap_or(name);

            // Find command
            self.commands.get(resolved_name).map(|b| &**b)
        }
    }
```

Benefits: - Dynamic dispatch (no enum growth) - Alias support (backward compatibility) - Decoupled from upstream SlashCommand enum

---

Summary

Configuration System Highlights:

1. **5-Tier Precedence**: CLI > Shell > Profile > TOML > Defaults
2. **Hot-Reload**: <100ms latency (p50), <0.5% CPU overhead
3. **Debouncing**: 300ms window prevents reload storms
4. **Lock Performance**: <1µs read locks, <1ms write locks
5. **Validation**: Schema validation preserves old config on error
6. **Type Safety**: Serde deserialization with enum validation
7. **Profile System**: Named configurations for different workflows
8. **Registry Pattern**: Dynamic command dispatch (Spec-Kit)

Architecture: - **notify crate**: Filesystem watching - **Arc**: Atomic config updates - **Debouncer**: Event buffering - **ConfigLoader**: Layered loading with validation

---

File References: - Loader: `codex-rs/spec-kit/src/config/loader.rs` - Hot-reload: `codex-rs/spec-kit/src/config/hot_reload.rs:1-100` - Validation: `codex-rs/spec-kit/src/config/validator.rs` - Registry: `codex-rs/tui/src/chatwidget/spec_kit/command_registry.rs`

Core Execution System

Agent orchestration, model providers, and execution management.

Overview

The Core Execution system manages: - **Agent Lifecycle**: Spawning, tracking, cleanup - **Model Providers**: OpenAI, Anthropic, Google integration - **Conversation Management**: Request/response handling - **Retry Logic**: Exponential backoff for failures - **Timeout Management**: Per-operation deadlines

Location: `codex-rs/core/src/`

Agent Orchestration

ConversationManager

Purpose: Central hub for agent conversation lifecycle

Location: `codex-rs/core/src/conversation_manager.rs`

```
        pub struct ConversationManager {
            conversations: Arc<Mutex<HashMap<ConversationId,
Conversation>>>,
            provider_clients: Arc<ModelProviderClients>,
            config: Arc<RwLock<Config>>,
        }

        impl ConversationManager {
            pub async fn new_conversation(&self, config: Config) ->
Result<NewConversation> {
                let conversation = Conversation::new(
                    config,
                    self.provider_clients.clone(),
                ).await?;

                Ok(NewConversation { conversation, id })
            }
        }
```

Responsibilities: - Create new conversations - Manage conversation lifecycle - Coordinate with model providers - Handle configuration updates

---

Agent Spawning Pattern

From TUI: `codex-rs/tui/src/chatwidget/agent.rs:16-62`

```
        pub(crate) fn spawn_agent(
            config: Config,
            app_event_tx: AppEventSender,
            server: Arc<ConversationManager>,
        ) -> UnboundedSender<Op> {
            let (codex_op_tx, mut codex_op_rx) = unbounded_channel::<Op>();

            tokio::spawn(async move {
                // Create conversation
                let conversation = server.new_conversation(config).await?;

                // Operation processor
                tokio::spawn(async move {
                    while let Some(op) = codex_op_rx.recv().await {
                        conversation.submit(op).await;
                    }
                });

                // Event forwarder
                while let Ok(event) = conversation.next_event().await {
                    app_event_tx.send(AppEvent::CodexEvent(event))?;
                }
            });

            codex_op_tx
        }
```

Key Points: - **Async task**: Runs on Tokio runtime - **Channel-based**: UnboundedSender for sync → async bridge - **Concurrent processing**: Separate op handler and event forwarder - **Automatic cleanup**: Tasks exit when conversation ends

---

Multi-Agent Orchestration (Spec-Kit)

Location: `codex-rs/tui/src/chatwidget/spec_kit/agent_orchestrator.rs`

```
        pub struct AgentOrchestrator {
            active_agents: HashMap<String, AgentHandle>,
            agent_configs: Vec<AgentConfig>,
        }

        pub struct AgentHandle {
            agent_id: String,
            agent_name: String,
            process: Child,                    // Subprocess handle
            stdout_reader: tokio::task::JoinHandle<()>,
            stderr_reader: tokio::task::JoinHandle<()>,
        }

        impl AgentOrchestrator {
            pub async fn spawn_agents(
                &mut self,
                spec_id: &str,
                stage: SpecStage,
                prompt: &str,
            ) -> Result<Vec<String>> {
                let mut agent_ids = Vec::new();

                for agent_config in &self.agent_configs {
                    let agent_id = format!("{}-{}-{}", spec_id, stage,
agent_config.name);

                    // Spawn subprocess
                    let mut child = Command::new(&agent_config.command)
                        .args(&agent_config.args)
                        .stdin(Stdio::piped())
                        .stdout(Stdio::piped())
                        .stderr(Stdio::piped())
                        .spawn()?;

                    // Read stdout/stderr asynchronously
                    let stdout_reader =
tokio::spawn(read_stream(child.stdout.take()));
                    let stderr_reader =
tokio::spawn(read_stream(child.stderr.take()));

                    let handle = AgentHandle {
                        agent_id: agent_id.clone(),
                        agent_name: agent_config.name.clone(),
                        process: child,
                        stdout_reader,
                        stderr_reader,
                    };

                    self.active_agents.insert(agent_id.clone(), handle);
```

```
                    agent_ids.push(agent_id);
                }

                Ok(agent_ids)
            }

            pub async fn wait_for_completion(&mut self, timeout: Duration) -
> Result<Vec<AgentOutput>> {
                let deadline = Instant::now() + timeout;
                let mut outputs = Vec::new();

                for (agent_id, handle) in self.active_agents.drain() {
                    let remaining =
deadline.saturating_duration_since(Instant::now());

                    match tokio::time::timeout(remaining,
handle.process.wait()).await {
                        Ok(Ok(status)) => {
                            let stdout = handle.stdout_reader.await?;
                            outputs.push(AgentOutput {
                                agent_id,
                                agent_name: handle.agent_name,
                                stdout,
                                exit_code: status.code(),
                            });
                        },
                        Ok(Err(e)) => { /* process error */ },
                        Err(_) => { /* timeout */ },
                    }
                }

                Ok(outputs)
            }
        }
```

Features: - **Concurrent spawning**: Launch multiple agents in parallel - **Timeout enforcement**: Per-agent deadlines - **Stream capture**: Async stdout/stderr reading - **Graceful cleanup**: Kill on timeout or error

---

Model Providers

Provider Architecture

```
Application
    ↓
ModelProviderClients (registry)
    ├→ OpenAIProvider (Responses API)
    ├→ AnthropicProvider (CLI subprocess)
    └↪ GoogleProvider (CLI subprocess)
```

OpenAI Provider

Location: `codex-rs/protocol/src/openai_client.rs`

```
        pub struct OpenAIClient {
```

```
            base_url: String,
            api_key: String,
            http_client: reqwest::Client,
            retry_config: RetryConfig,
        }

        impl OpenAIClient {
            pub async fn chat_completion(
                &self,
                request: ChatCompletionRequest,
            ) -> Result<impl Stream<Item = Result<ChatCompletionChunk>>> {
                let url = format!("{}/chat/completions", self.base_url);

                let response = self.http_client
                    .post(&url)
                    .header("Authorization", format!("Bearer {}",
self.api_key))
                    .json(&request)
                    .send()
                    .await?;

                // Server-Sent Events stream
                let stream = response
                    .bytes_stream()
                    .eventsource()
                    .map(|event| parse_sse_event(event));

                Ok(stream)
            }

            pub async fn responses_api(
                &self,
                request: ResponsesRequest,
            ) -> Result<impl Stream<Item = Result<ResponseEvent>>> {
                // Similar but for Responses API
            }
        }
```

Features: - **Streaming**: Server-Sent Events (SSE) - **Retry logic**: Exponential backoff on failures - **Rate limiting**: 429 response handling - **Timeout**: Per-request deadlines

---

Anthropic Provider (CLI)

Location: `codex-rs/tui/src/chatwidget/spec_kit/agent_orchestrator.rs`

```
        // Anthropic via CLI subprocess
        let mut child = Command::new("claude")
            .arg("--model").arg("claude-sonnet-4-5")
            .arg(prompt)
            .env("ANTHROPIC_API_KEY", api_key)
            .stdout(Stdio::piped())
            .spawn()?;

        let output = child.wait_with_output().await?;
        let response = String::from_utf8(output.stdout)?;
```

Note: Uses CLI subprocess, not direct API (simpler integration for multi-agent)

Google Provider (CLI)

```rust
// Google via CLI subprocess
let mut child = Command::new("gemini")
    .arg("-i").arg(prompt)
    .env("GOOGLE_API_KEY", api_key)
    .stdout(Stdio::piped())
    .spawn()?;

let output = child.wait_with_output().await?;
let response = String::from_utf8(output.stdout)?;
```

Protocol Implementation

Request/Response Types

Location: `codex-rs/protocol/src/types.rs`

```rust
// Chat Completions API
pub struct ChatCompletionRequest {
    pub model: String,
    pub messages: Vec<Message>,
    pub temperature: Option<f32>,
    pub max_tokens: Option<u32>,
    pub stream: bool,
}

pub struct Message {
    pub role: Role,      // system, user, assistant, tool
    pub content: String,
    pub name: Option<String>,
    pub tool_calls: Option<Vec<ToolCall>>,
}

// Responses API (newer)
pub struct ResponsesRequest {
    pub model: String,
    pub messages: Vec<Message>,
    pub reasoning: Option<ReasoningConfig>,
    pub response_format: Option<ResponseFormat>,
}

pub struct ReasoningConfig {
    pub effort: ReasoningEffort,  // minimal, low, medium, high
    pub summary: SummaryLevel,    // none, auto, concise, detailed
}
```

Streaming Events

```rust
pub enum ResponseEvent {
    Start { id: String },
    Token { content: String },
```

```
        ReasoningToken { content: String },
        ToolCall { call: ToolCall },
        Complete { finish_reason: FinishReason },
        Error { error: String },
    }
```

Processing:

```
    while let Some(event) = stream.next().await {
        match event? {
            ResponseEvent::Token { content } => {
                // Forward to UI for rendering
                app_event_tx.send(AppEvent::Token(content))?;
            },
            ResponseEvent::ToolCall { call } => {
                // Execute tool and inject result
                let result = execute_tool(call).await?;
                conversation.submit(Op::ToolResponse(call.id,
result)).await?;
            },
            ResponseEvent::Complete { finish_reason } => {
                break;
            },
            _ => {}
        }
    }
```

Retry Logic

Exponential Backoff

Location: `codex-rs/spec-kit/src/retry/strategy.rs`

```
    pub struct RetryConfig {
        pub max_attempts: usize,
        pub initial_backoff_ms: u64,
        pub max_backoff_ms: u64,
        pub backoff_multiplier: f64,
        pub jitter_factor: f64,
    }

    impl Default for RetryConfig {
        fn default() -> Self {
            Self {
                max_attempts: 3,
                initial_backoff_ms: 100,
                max_backoff_ms: 10_000,
                backoff_multiplier: 2.0,
                jitter_factor: 0.5,
            }
        }
    }
```

Backoff Calculation:

```
Attempt 1: 100ms + jitter(±50ms)
Attempt 2: 200ms + jitter(±100ms)
Attempt 3: 400ms + jitter(±200ms)
```

```
Attempt 4: 800ms + jitter(±400ms)
Attempt 5: 1600ms + jitter(±800ms)
...
Max: 10,000ms (10s)
```

Error Classification

```rust
pub trait RetryClassifiable {
    fn is_retryable(&self) -> bool;
}

impl RetryClassifiable for ApiError {
    fn is_retryable(&self) -> bool {
        match self {
            // Transient errors (retry)
            ApiError::RateLimitExceeded => true,
            ApiError::ServiceUnavailable => true,
            ApiError::Timeout => true,
            ApiError::NetworkError(_) => true,

            // Permanent errors (don't retry)
            ApiError::AuthenticationFailed => false,
            ApiError::InvalidRequest(_) => false,
            ApiError::InsufficientQuota => false,

            _ => false,
        }
    }
}
```

Retry Execution

```rust
pub async fn execute_with_backoff<F, Fut, T, E>(
    mut operation: F,
    config: &RetryConfig,
) -> Result<T>
where
    F: FnMut() -> Fut,
    Fut: Future<Output = Result<T, E>>,
    E: Error + RetryClassifiable,
{
    let mut attempts = 0;
    let mut backoff_ms = config.initial_backoff_ms;

    loop {
        attempts += 1;

        match operation().await {
            Ok(value) => return Ok(value),
            Err(err) if !err.is_retryable() => {
                return
Err(RetryError::PermanentError(err.to_string()));
            },
            Err(_) if attempts >= config.max_attempts => {
                return
Err(RetryError::MaxAttemptsExceeded(attempts));
            },
```

```
                    Err(_) => {
                        // Calculate jittered backoff
                        let jitter = (rand::random::<f64>() - 0.5) * 2.0 *
config.jitter_factor;
                        let delay_ms = (backoff_ms as f64 * (1.0 + jitter))
as u64;


tokio::time::sleep(Duration::from_millis(delay_ms)).await;

                        // Exponential increase
                        backoff_ms = (backoff_ms as f64 *
config.backoff_multiplier) as u64;
                        backoff_ms = backoff_ms.min(config.max_backoff_ms);
                    }
                }
            }
        }
```

Timeout Management

Per-Operation Timeouts

```
pub async fn execute_with_timeout<F, T>(
    operation: F,
    timeout: Duration,
) -> Result<T>
where
    F: Future<Output = Result<T>>,
{
    match tokio::time::timeout(timeout, operation).await {
        Ok(Ok(value)) => Ok(value),
        Ok(Err(e)) => Err(e),
        Err(_) => Err(Error::Timeout),
    }
}
```

Usage:

```
// API request with 30s timeout
let response = execute_with_timeout(
    openai_client.chat_completion(request),
    Duration::from_secs(30),
).await?;

// Agent execution with 5min timeout
let outputs = execute_with_timeout(
    agent_orchestrator.wait_for_completion(),
    Duration::from_secs(300),
).await?;
```

Configurable Timeouts

From config.toml:

```
[mcp_servers.filesystem]
```

```toml
startup_timeout_sec = 10   # Server initialization
tool_timeout_sec = 60      # Per-tool execution

[model_providers.openai]
stream_idle_timeout_ms = 300000  # 5min idle timeout
```

Tool Execution

Tool Call Flow

```
1. Model returns tool_use event
   └→ {"type": "tool_use", "name": "filesystem__read_file",
"arguments": {...}}

2. Conversation extracts tool call
   └→ ToolCall { id, name, arguments }

3. Submit to MCP manager
   └→ mcp_manager.invoke_tool(name, arguments).await

4. MCP client executes
   └→ Server subprocess processes request

5. Result returned
   └→ ToolResult { id, content: "file contents..." }

6. Inject into conversation
   └→ conversation.submit(Op::ToolResponse(id, result)).await

7. Model continues with tool result
```

Sandbox Enforcement

```rust
pub enum SandboxMode {
    ReadOnly,           // No writes, no network
    WorkspaceWrite,     // Write to workspace, no network
    DangerFullAccess,   // Full access (use in Docker)
}

pub fn execute_sandboxed(
    command: &str,
    args: &[&str],
    sandbox: SandboxMode,
) -> Result<Output> {
    match sandbox {
        SandboxMode::ReadOnly => {
            // Landlock: deny all writes
            apply_landlock_readonly()?;
        },
        SandboxMode::WorkspaceWrite => {
            // Landlock: allow workspace writes only
            apply_landlock_workspace(workspace_path)?;
        },
        SandboxMode::DangerFullAccess => {
            // No sandboxing
        },
```

```
    }

    // Execute command
    Command::new(command)
        .args(args)
        .output()
}
```

File: `codex-rs/linux-sandbox/src/lib.rs`

Performance Optimizations

Connection Pooling

HTTP Client:

```
let http_client = reqwest::Client::builder()
    .pool_max_idle_per_host(10)     // Reuse connections
    .timeout(Duration::from_secs(30))
    .build()?;
```

Benefits: - Reuse TCP connections (avoid handshake overhead) - Reduce latency for subsequent requests

---

Concurrent Execution

Multi-Agent:

```
// Spawn all agents concurrently
let mut join_set = JoinSet::new();
for agent_config in agent_configs {
    join_set.spawn(spawn_agent(agent_config, prompt.clone()));
}

// Wait for all to complete
let mut outputs = Vec::new();
while let Some(result) = join_set.join_next().await {
    outputs.push(result??);
}
```

Performance: 3 agents finish in ~10s instead of ~30s (3× speedup)

Error Handling

Error Hierarchy

```
pub enum Error {
    // Network errors (retryable)
    NetworkError(reqwest::Error),
    Timeout,

    // API errors (some retryable)
    RateLimitExceeded,
    ServiceUnavailable,
```

```
    AuthenticationFailed,      // Not retryable
    InvalidRequest(String),    // Not retryable

    // Agent errors
    AgentSpawnFailed(io::Error),
    AgentTimeout,
    AgentCrashed(i32),         // Exit code

    // Tool errors
    ToolNotFound(String),
    ToolExecutionFailed(String),
}
```

Graceful Degradation

Multi-Agent Consensus:

```
// If 1/3 agents fail, continue with 2/3
let outputs = agent_orchestrator.wait_for_completion().await?;

if outputs.len() >= 2 {
    // Consensus still valid with 2/3 agents
    let synthesis = synthesize_consensus(&outputs);
    return Ok((synthesis, degraded: true));
} else {
    return Err(Error::InsufficientAgents);
}
```

Summary

Core Execution Highlights:

1. **Agent Orchestration**: Async task spawning with channel-based communication
2. **Multi-Provider**: OpenAI (HTTP), Anthropic/Google (CLI subprocess)
3. **Streaming**: Real-time token delivery via SSE
4. **Retry Logic**: Exponential backoff with jitter (100ms → 10s)
5. **Timeout Management**: Per-operation deadlines
6. **Tool Execution**: MCP integration with sandbox enforcement
7. **Error Handling**: Permanent vs transient classification
8. **Performance**: Connection pooling, concurrent agent execution

Next Steps: - MCP Integration - Native client details - Database Layer - SQLite optimization - Configuration System - Hot-reload

---

File References: - Conversation manager: `codex-rs/core/src/conversation_manager.rs` - Agent spawner: `codex-rs/tui/src/chatwidget/agent.rs:16-62` - Agent orchestrator: `codex-rs/tui/src/chatwidget/spec_kit/agent_orchestrator.rs` - OpenAI client: `codex-rs/protocol/src/openai_client.rs` - Retry logic: `codex-rs/spec-kit/src/retry/strategy.rs` - Sandbox: `codex-rs/linux-sandbox/src/lib.rs`

Database Layer

SQLite storage with optimized performance for consensus artifacts.

Overview

Database: SQLite (embedded, ACID transactions) **Primary Use**: Consensus artifact storage (agent outputs, synthesis) **Performance**: **6.6× read speedup**, **2.3× write speedup** (via optimizations) **Location**: `codex-rs/core/src/db/`, `codex-rs/tui/src/chatwidget/spec_kit/consensus_db.rs`

Architecture

```
Application
    ↓
R2D2 Connection Pool (2-8 connections)
    ├→ SQLite Connection (WAL mode)
    ├→ SQLite Connection
    └→ SQLite Connection
    ↓
Database File (consensus_artifacts.db)
├─ consensus_runs table
├─ agent_outputs table
└─ consensus_artifacts table
```

Connection Pooling

R2D2 Configuration

Location: `codex-rs/core/src/db/connection.rs:39-105`

```rust
use r2d2::{Pool, PooledConnection};
use r2d2_sqlite::SqliteConnectionManager;
use rusqlite::Connection;

pub fn initialize_pool(
    db_path: &Path,
    pool_size: u32,
) -> Result<Pool<SqliteConnectionManager>> {
    let manager = SqliteConnectionManager::file(db_path);

    let pool = Pool::builder()
        .max_size(pool_size)           // Max connections (default: 8)
        .min_idle(Some(2))             // Keep 2 connections warm
        .connection_customizer(Box::new(ConnectionCustomizer))
        .test_on_check_out(true)       // Health check before use
        .build(manager)?;

    // Verify pragmas on first connection
    let conn = pool.get()?;
```

```
        verify_pragmas(&conn)?;

        Ok(pool)
    }
```

Benefits: - **Connection reuse**: Eliminate open/close overhead - **Concurrency**: Multiple connections for parallel access - **Health checks**: Detect broken connections before use - **Warm pool**: Keep minimum connections ready

---

Connection Customizer (Pragma Optimization)

```
    struct ConnectionCustomizer;

    impl r2d2::CustomizeConnection<Connection, rusqlite::Error> for
ConnectionCustomizer {
        fn on_acquire(&self, conn: &mut Connection) -> Result<(),
rusqlite::Error> {
            conn.execute_batch(
                "PRAGMA journal_mode = WAL;              -- Write-Ahead
Logging (6.6× read speedup)
                 PRAGMA synchronous = NORMAL;           -- 2-3× write
speedup (safe with WAL)
                 PRAGMA foreign_keys = ON;              -- Referential
integrity
                 PRAGMA cache_size = -32000;            -- 32MB page
cache
                 PRAGMA temp_store = MEMORY;            -- In-memory
temporary tables
                 PRAGMA auto_vacuum = INCREMENTAL;      -- Prevent
unbounded growth
                 PRAGMA mmap_size = 1073741824;         -- 1GB memory-
mapped I/O
                 PRAGMA busy_timeout = 5000;            -- 5s deadlock
wait
                "
            )
        }
    }
```

Pragma Explanations:

Pragma	Value	Effect
`journal_mode`	`WAL`	Write-Ahead Logging: Concurrent reads during writes
`synchronous`	`NORMAL`	Fewer fsync calls (safe with WAL)
`foreign_keys`	`ON`	Enforce foreign key constraints
`cache_size`	`-32000`	32MB in-memory page cache
`temp_store`	`MEMORY`	Temporary tables in RAM
`auto_vacuum`	`INCREMENTAL`	Gradual space reclamation
		Memory-mapped I/O

`mmap_size`	`1GB`	for reads
`busy_timeout`	`5s`	Retry on lock for 5 seconds

Performance Impact

Before optimizations:

```
Single read:  850µs
Single write: 2.1ms
100-read batch: 78ms
```

After optimizations:

```
Single read:  129µs  (6.6× faster)
Single write: 0.9ms  (2.3× faster)
100-read batch: 12ms  (6.5× faster)
```

Total improvement: 6.6× read, 2.3× write

Benchmark: `codex-rs/core/tests/db_benchmark.rs`

Schema

consensus_runs

Purpose: Track workflow execution metadata

```sql
CREATE TABLE IF NOT EXISTS consensus_runs (
    run_id INTEGER PRIMARY KEY AUTOINCREMENT,
    spec_id TEXT NOT NULL,
    stage TEXT NOT NULL,
    consensus_ok BOOLEAN NOT NULL,
    degraded BOOLEAN NOT NULL,
    synthesis_json TEXT,
    created_at TEXT NOT NULL DEFAULT CURRENT_TIMESTAMP,

    UNIQUE(spec_id, stage)
);

CREATE INDEX IF NOT EXISTS idx_consensus_runs_spec
ON consensus_runs(spec_id, stage);
```

Columns: - `run_id`: Auto-increment primary key - `spec_id`: SPEC identifier (e.g., “SPEC-KIT-065”) - `stage`: Workflow stage (“plan”, “tasks”, “implement”, etc.) - `consensus_ok`: Consensus achieved (true/false) - `degraded`: Some agents failed (true/false) - `synthesis_json`: Synthesized consensus result (JSON) - `created_at`: Timestamp

Constraint: One run per (spec_id, stage) pair (UPSERT semantics)

---

agent_outputs

Purpose: Store individual agent responses

```
        CREATE TABLE IF NOT EXISTS agent_outputs (
            output_id INTEGER PRIMARY KEY AUTOINCREMENT,
            run_id INTEGER NOT NULL,
            agent_name TEXT NOT NULL,
            agent_version TEXT,
            content_json TEXT NOT NULL,
            response_text TEXT,
            created_at TEXT NOT NULL DEFAULT CURRENT_TIMESTAMP,

            FOREIGN KEY (run_id) REFERENCES consensus_runs(run_id) ON DELETE
CASCADE
        );

        CREATE INDEX IF NOT EXISTS idx_agent_outputs_run
        ON agent_outputs(run_id);
```

Columns: - `output_id`: Auto-increment primary key - `run_id`: Foreign key to consensus_runs - `agent_name`: Agent identifier (“gemini”, “claude”, “code”) - `agent_version`: Model version (“gemini-flash-1.5”, “claude-sonnet-4-5”) - `content_json`: Structured agent output (JSON) - `response_text`: Raw response text - `created_at`: Timestamp

Relationship: Many agent_outputs per consensus_run (1:N)

---

consensus_artifacts (legacy)

Purpose: Old schema (being phased out)

```
        CREATE TABLE IF NOT EXISTS consensus_artifacts (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            spec_id TEXT NOT NULL,
            stage TEXT NOT NULL,
            agent_name TEXT NOT NULL,
            content_json TEXT NOT NULL,
            response_text TEXT,
            run_id TEXT,
            created_at TEXT NOT NULL DEFAULT CURRENT_TIMESTAMP
        );
```

Status: Deprecated, replaced by consensus_runs + agent_outputs

---

Transaction Handling

Transaction Behavior

```
        use rusqlite::TransactionBehavior;

        pub enum TransactionBehavior {
            Deferred,     // Lock on first read/write
            Immediate,    // Write lock immediately
            Exclusive,    // Exclusive lock (blocks all)
        }
```

Recommendation: Use `IMMEDIATE` for writes (avoid write-write deadlock)

Transaction Helpers

Location: `codex-rs/core/src/db/transactions.rs:40-119`

```rust
pub fn execute_in_transaction<F, T>(
    conn: &mut Connection,
    behavior: TransactionBehavior,
    operation: F,
) -> Result<T>
where
    F: FnOnce(&Transaction) -> Result<T>,
{
    let tx = conn.transaction_with_behavior(behavior)?;

    match operation(&tx) {
        Ok(result) => {
            tx.commit()?;
            Ok(result)
        },
        Err(e) => {
            // Automatic rollback via Drop trait
            Err(e)
        },
    }
}
```

Usage:

```rust
execute_in_transaction(conn, TransactionBehavior::Immediate, |tx| {
    tx.execute("INSERT INTO consensus_runs (...) VALUES (?)",
params)?;
    tx.execute("INSERT INTO agent_outputs (...) VALUES (?)",
params)?;
    Ok(())
})?;
```

ACID guarantees: - **Atomicity**: All or nothing (rollback on error) - **Consistency**: Foreign keys enforced - **Isolation**: IMMEDIATE locks prevent conflicts - **Durability**: WAL ensures persistence

---

Batch Operations

```rust
pub fn batch_insert<T>(
    conn: &mut Connection,
    _table: &str,
    _columns: &[&str],
    rows: &[T],
    bind_fn: impl Fn(&Transaction, &T) -> Result<()>,
) -> Result<usize> {
    execute_in_transaction(conn, TransactionBehavior::Immediate,
|tx| {
        for row in rows {
            bind_fn(tx, row)?;
        }
        Ok(rows.len())
    })
}
```

Performance: 100 inserts in single transaction ~12ms (vs ~2s for 100 individual commits)

Async Wrapper

Problem: SQLite is synchronous, Tokio is async

Solution: `tokio::task::spawn_blocking` wrapper

Location: `codex-rs/core/src/db/async_wrapper.rs:69-150`

```rust
pub async fn with_connection<F, T>(
    pool: &Pool<SqliteConnectionManager>,
    f: F,
) -> Result<T>
where
    F: FnOnce(&mut Connection) -> Result<T> + Send + 'static,
    T: Send + 'static,
{
    let pool = pool.clone();

    // Run synchronous SQLite code in blocking thread pool
    tokio::task::spawn_blocking(move || {
        let mut conn = pool.get()?;
        f(&mut conn)
    })
    .await?
}
```

Usage:

```rust
// Async function calling sync SQLite
pub async fn store_consensus(
    pool: &Pool<SqliteConnectionManager>,
    spec_id: &str,
    stage: &str,
    synthesis: &str,
) -> Result<i64> {
    with_connection(pool, move |conn| {
        execute_in_transaction(conn, TransactionBehavior::Immediate,
|tx| {
            upsert_consensus_run(tx, spec_id, stage, synthesis)
        })
    }).await
}
```

Key Points: - **Doesn't block Tokio runtime**: Runs on separate thread pool - **Connection pooling**: Still benefits from R2D2 pool - **Error propagation**: Propagates SQLite errors to async context

---

Consensus Database (Spec-Kit)

ConsensusDb

Location: `codex-rs/tui/src/chatwidget/spec_kit/consensus_db.rs:34-150`

```rust
        pub struct ConsensusDb {
            conn: Arc<Mutex<Connection>>,                       // Legacy
single connection
            pool: Option<Pool<SqliteConnectionManager>>,        // New
pooled connections
        }

        impl ConsensusDb {
            pub fn init(db_path: &Path) -> Result<Self> {
                // Open single connection (legacy)
                let conn = Connection::open(db_path)?;

                // Create schema
                conn.execute(
                    "CREATE TABLE IF NOT EXISTS agent_executions (
                        agent_id TEXT PRIMARY KEY,
                        spec_id TEXT NOT NULL,
                        stage TEXT NOT NULL,
                        phase_type TEXT NOT NULL,
                        agent_name TEXT NOT NULL,
                        run_id TEXT,
                        spawned_at TEXT NOT NULL,
                        completed_at TEXT,
                        response_text TEXT,
                        extraction_error TEXT
                    )",
                    [],
                )?;

                // Create indices
                conn.execute(
                    "CREATE INDEX IF NOT EXISTS idx_agent_executions_spec
                     ON agent_executions(spec_id, stage)",
                    [],
                )?;

                // Initialize new schema pool (SPEC-945B)
                let pool = initialize_pool(db_path, 8)?;

                Ok(Self {
                    conn: Arc::new(Mutex::new(conn)),
                    pool: Some(pool),
                })
            }

            pub fn upsert_consensus_run(
                &self,
                spec_id: &str,
                stage: &str,
                consensus_ok: bool,
                degraded: bool,
                synthesis_json: Option<&str>,
            ) -> Result<i64> {
                let pool =
self.pool.as_ref().ok_or(Error::PoolNotInitialized)?;
                let conn = pool.get()?;

                execute_in_transaction(&mut conn,
TransactionBehavior::Immediate, |tx| {
                    tx.execute(
```

```
                        "INSERT INTO consensus_runs (spec_id, stage,
consensus_ok, degraded, synthesis_json)
                         VALUES (?1, ?2, ?3, ?4, ?5)
                         ON CONFLICT(spec_id, stage) DO UPDATE SET
                             consensus_ok = excluded.consensus_ok,
                             degraded = excluded.degraded,
                             synthesis_json = excluded.synthesis_json,
                             created_at = CURRENT_TIMESTAMP",
                        params![spec_id, stage, consensus_ok, degraded,
synthesis_json],
                    )?;

                    let run_id = tx.last_insert_rowid();
                    Ok(run_id)
                })
            }

            pub fn insert_agent_output(
                &self,
                run_id: i64,
                agent_name: &str,
                agent_version: Option<&str>,
                content_json: &str,
                response_text: Option<&str>,
            ) -> Result<i64> {
                let pool =
self.pool.as_ref().ok_or(Error::PoolNotInitialized)?;
                let conn = pool.get()?;

                conn.execute(
                    "INSERT INTO agent_outputs (run_id, agent_name,
agent_version, content_json, response_text)
                     VALUES (?1, ?2, ?3, ?4, ?5)",
                    params![run_id, agent_name, agent_version, content_json,
response_text],
                )?;

                Ok(conn.last_insert_rowid())
            }
        }
```

Dual-Schema Migration (SPEC-945B)

Phase 1: Old schema only (agent_executions) **Phase 2**: Dual-write to both schemas (current) **Phase 3**: New schema only (consensus_runs + agent_outputs)

Current state: Dual-write active

```
        // Write to both old and new schema
        pub fn store_consensus(&self, spec_id: &str, stage: &str) ->
Result<()> {
            // Old schema (legacy)
            self.conn.lock().unwrap().execute(
                "INSERT INTO agent_executions (...) VALUES (?)",
                params![...],
            )?;

            // New schema (pooled)
```

```
            if let Some(pool) = &self.pool {
                self.upsert_consensus_run(spec_id, stage, true, false,
None)?;
            }

            Ok(())
        }
```

Auto-Vacuum Strategy

Incremental Auto-Vacuum

Purpose: Prevent unbounded database growth

Configuration: `PRAGMA auto_vacuum = INCREMENTAL;`

How it works: - Database tracks free pages internally - On `PRAGMA incremental_vacuum(N)`, reclaim N pages - No blocking full-vacuum required

Usage:

```
        pub fn compact_database(conn: &Connection, max_pages: u32) ->
Result<()> {
            conn.execute(&format!("PRAGMA incremental_vacuum({})",
max_pages), [])?;
            Ok(())
        }
```

Result: 99.95% size reduction after cleanup (multi-GB → few MB)

Evidence: `docs/SPEC-OPS-004-integrated-coder-hooks/evidence/database-cleanup.log`

Retry Logic (Database Operations)

Sync Retry

Location: `codex-rs/spec-kit/src/retry/strategy.rs`

```
        pub fn execute_with_backoff_sync<F, T, E>(
            mut operation: F,
            config: &RetryConfig,
        ) -> Result<T>
        where
            F: FnMut() -> Result<T, E>,
            E: Error + RetryClassifiable,
        {
            let mut attempts = 0;
            let mut backoff_ms = config.initial_backoff_ms;

            loop {
                attempts += 1;

                match operation() {
                    Ok(value) => return Ok(value),
```

```
                    Err(err) if err.error_code() ==
Some(rusqlite::ErrorCode::DatabaseBusy) => {
                        // Database locked, retry with backoff
                        if attempts >= config.max_attempts {
                            return
Err(RetryError::MaxAttemptsExceeded(attempts));
                        }


std::thread::sleep(Duration::from_millis(backoff_ms));
                        backoff_ms = (backoff_ms as f64 *
config.backoff_multiplier) as u64;
                        backoff_ms = backoff_ms.min(config.max_backoff_ms);
                    },
                    Err(err) => {
                        // Permanent error, don't retry
                        return
Err(RetryError::PermanentError(err.to_string()));
                    },
                }
            }
        }
```

Usage:

```
        let result = execute_with_backoff_sync(
            || conn.execute("INSERT INTO ...", params),
            &RetryConfig::default(),
        )?;
```

Error Handling

Database Errors

```
        pub enum DbError {
            // Connection errors
            PoolExhausted,
            ConnectionFailed(r2d2::Error),

            // SQLite errors
            DatabaseBusy,              // Retryable
            DatabaseLocked,            // Retryable
            ConstraintViolation(String), // Not retryable
            SchemaError(String),       // Not retryable

            // Application errors
            InvalidSchema(String),
            MigrationFailed(String),
        }
```

Error Classification

```
        impl RetryClassifiable for DbError {
            fn is_retryable(&self) -> bool {
                match self {
                    DbError::DatabaseBusy => true,
```

```
                    DbError::DatabaseLocked => true,
                    DbError::PoolExhausted => true,

                    DbError::ConstraintViolation(_) => false,
                    DbError::SchemaError(_) => false,
                    DbError::InvalidSchema(_) => false,

                    _ => false,
                }
            }
        }
```

Schema Migrations

Migration System

Location: `codex-rs/core/src/db/migrations.rs:9-87`

```
        pub const SCHEMA_VERSION: i32 = 2;

        pub fn migrate_to_latest(conn: &mut Connection) -> Result<()> {
            let current_version = get_schema_version(conn)?;

            if current_version == SCHEMA_VERSION {
                return Ok(());  // Already up-to-date
            }

            let tx =
conn.transaction_with_behavior(TransactionBehavior::Exclusive)?;

            for version in (current_version + 1)..=SCHEMA_VERSION {
                apply_migration(&tx, version)?;
            }

            // Update schema version
            tx.execute(&format!("PRAGMA user_version = {}", SCHEMA_VERSION),
[])?;
            tx.commit()?;

            Ok(())
        }

        fn get_schema_version(conn: &Connection) -> Result<i32> {
            let version: i32 = conn.query_row("PRAGMA user_version", [],
|row| row.get(0))?;
            Ok(version)
        }

        fn apply_migration(conn: &Connection, version: i32) -> Result<()> {
            match version {
                1 => migration_v1(conn),  // Create consensus_runs,
agent_outputs
                2 => migration_v2(conn),  // Add indices
                _ => Err(Error::UnknownMigrationVersion(version)),
            }
        }
```

Migration v1:

```
        fn migration_v1(conn: &Connection) -> Result<()> {
            conn.execute_batch(
                "CREATE TABLE IF NOT EXISTS consensus_runs (...);
                 CREATE TABLE IF NOT EXISTS agent_outputs (...);
                 CREATE INDEX IF NOT EXISTS idx_consensus_runs_spec ON
consensus_runs(spec_id, stage);
                 CREATE INDEX IF NOT EXISTS idx_agent_outputs_run ON
agent_outputs(run_id);"
            )?;
            Ok(())
        }
```

Forward-only: Migrations never rollback (destructive changes prohibited)

Summary

Database Layer Highlights:

1. **Performance**: 6.6× read speedup, 2.3× write speedup (WAL + pragmas)
2. **Connection Pooling**: R2D2 with 2-8 connections
3. **Async Wrapper**: `tokio::task::spawn_blocking` for Tokio integration
4. **ACID Transactions**: IMMEDIATE mode for write consistency
5. **Schema**: consensus_runs + agent_outputs (normalized)
6. **Auto-Vacuum**: Incremental strategy (99.95% size reduction)
7. **Retry Logic**: Exponential backoff for database busy errors
8. **Migrations**: Forward-only schema evolution

Next Steps: - Configuration System - Hot-reload and 5-tier precedence

---

File References: - Connection pool: `codex-rs/core/src/db/connection.rs:39-105` - Transactions: `codex-rs/core/src/db/transactions.rs:40-119` - Async wrapper: `codex-rs/core/src/db/async_wrapper.rs:69-150` - Consensus DB: `codex-rs/tui/src/chatwidget/spec_kit/consensus_db.rs:34-150` - Migrations: `codex-rs/core/src/db/migrations.rs:9-87` - Retry logic: `codex-rs/spec-kit/src/retry/strategy.rs`

MCP Integration

Native Model Context Protocol client and server integration.

Overview

MCP (Model Context Protocol) allows AI models to access external tools: - File operations - Database queries - API calls - Custom tools

Native Implementation: 5.3× faster than subprocess approach (8.7ms typical vs 46ms)

Location: `codex-rs/mcp-client/`, `codex-rs/mcp-types/`, `codex-rs/core/src/mcp_connection_manager.rs`

Architecture

```
Application (TUI/Core)
    ↓
McpConnectionManager (singleton, shared across widgets)
    ├→ McpClient("filesystem")
    │   ├→ Writer Task (outgoing → stdin)
    │   ├→ Reader Task (stdout → pending HashMap)
    │   └→ Subprocess (MCP server)
    ├→ McpClient("git-status")
    └→ McpClient("local-memory")

MCP Servers (subprocesses)
├→ @modelcontextprotocol/server-filesystem
├→ @modelcontextprotocol/server-git-status
└→ @modelcontextprotocol/server-local-memory
```

McpClient Implementation

Core Structure

Location: `codex-rs/mcp-client/src/mcp_client.rs:63-150`

```rust
        pub struct McpClient {
            child: tokio::process::Child,                    // Subprocess handle
            outgoing_tx: mpsc::Sender<JSONRPCMessage>,      // Send requests (128 capacity)
            pending: Arc<Mutex<HashMap<i64, PendingSender>>>, // Request ID → response oneshot
            id_counter: AtomicI64,                           // Monotonic ID generator
        }

        type PendingSender = oneshot::Sender<JSONRPCMessage>;

        impl McpClient {
            pub async fn new_stdio_client(
                program: OsString,
                args: Vec<OsString>,
                env: Option<HashMap<String, String>>,
            ) -> io::Result<Self> {
                // Spawn MCP server subprocess
                let mut child = Command::new(program)
                    .args(args)
                    .env_clear()
                    .envs(env.unwrap_or_default())
                    .stdin(Stdio::piped())
                    .stdout(Stdio::piped())
                    .stderr(Stdio::inherit())
                    .kill_on_drop(true)
                    .spawn()?;
```

```
                let stdin = child.stdin.take().unwrap();
                let stdout = child.stdout.take().unwrap();

                let (outgoing_tx, mut outgoing_rx) = mpsc::channel(128);
                let pending = Arc::new(Mutex::new(HashMap::new()));

                // Writer task: outgoing_rx → stdin (JSON-RPC requests)
                tokio::spawn(async move {
                    let mut stdin = stdin;
                    while let Some(msg) = outgoing_rx.recv().await {
                        let json = serde_json::to_string(&msg)?;
                        stdin.write_all(json.as_bytes()).await?;
                        stdin.write_all(b"\n").await?;  // Newline-delimited
                    }
                    Ok::<(), io::Error>(())
                });

                // Reader task: stdout → pending HashMap (JSON-RPC
responses)
                let pending_clone = Arc::clone(&pending);
                tokio::spawn(async move {
                    let mut lines = BufReader::with_capacity(1024 * 1024,
stdout).lines();

                    while let Ok(Some(line)) = lines.next_line().await {
                        let msg: JSONRPCMessage =
serde_json::from_str(&line)?;

                        // Match response to request
                        if let Some(id) = msg.id {
                            if let Some(tx) =
pending_clone.lock().unwrap().remove(&id) {
                                let _ = tx.send(msg);
                            }
                        }
                    }
                    Ok::<(), anyhow::Error>(())
                });

                Ok(Self {
                    child,
                    outgoing_tx,
                    pending,
                    id_counter: AtomicI64::new(1),
                })
            }
        }
```

Key Design Choices: - **1MB buffer**: `BufReader::with_capacity(1024 * 1024)` handles large tool responses - **Line-delimited JSON**: Each JSON-RPC message on one line - **Concurrent I/O**: Separate reader/writer tasks prevent deadlock - **kill_on_drop**: Subprocess cleaned up automatically

---

JSON-RPC Protocol

Types: `codex-rs/mcp-types/src/jsonrpc.rs`

```rust
#[derive(Serialize, Deserialize)]
#[serde(untagged)]
pub enum JSONRPCMessage {
    Request {
        jsonrpc: String,  // "2.0"
        id: i64,
        method: String,
        params: Option<Value>,
    },
    Response {
        jsonrpc: String,
        id: i64,
        result: Option<Value>,
        error: Option<JSONRPCError>,
    },
    Notification {
        jsonrpc: String,
        method: String,
        params: Option<Value>,
    },
}
```

Wire Format (newline-delimited):

```
{"jsonrpc":"2.0","id":1,"method":"tools/list","params":null}
{"jsonrpc":"2.0","id":1,"result":{"tools":
[{"name":"read_file","description":"..."}]}}
```

Tool Invocation

```rust
impl McpClient {
    pub async fn call_tool(
        &self,
        tool_name: &str,
        arguments: Value,
    ) -> Result<Value> {
        // Generate unique request ID
        let id = self.id_counter.fetch_add(1, Ordering::SeqCst);

        // Create oneshot channel for response
        let (tx, rx) = oneshot::channel();
        self.pending.lock().unwrap().insert(id, tx);

        // Send request
        let request = JSONRPCMessage::Request {
            jsonrpc: "2.0".to_string(),
            id,
            method: "tools/call".to_string(),
            params: Some(json!({
                "name": tool_name,
                "arguments": arguments,
            })),
        };

        self.outgoing_tx.send(request).await?;

        // Wait for response (with timeout)
        let response = tokio::time::timeout(
            Duration::from_secs(60),
```

```
                    rx
                ).await??;

                // Extract result
                match response {
                    JSONRPCMessage::Response { result: Some(result), .. } =>
Ok(result),
                    JSONRPCMessage::Response { error: Some(err), .. } =>
Err(err.into()),
                    _ => Err(Error::InvalidResponse),
                }
            }
        }
```

Flow: 1. Generate unique ID (atomic counter) 2. Create oneshot channel, store in pending HashMap 3. Send JSON-RPC request to server (via outgoing_tx) 4. Writer task writes to stdin 5. Server processes request, writes to stdout 6. Reader task parses response 7. Match ID, send via oneshot channel 8. Remove from pending HashMap 9. Return result to caller

---

McpConnectionManager

Purpose: Central hub for all MCP servers, tool aggregation

Location: `codex-rs/core/src/mcp_connection_manager.rs:84-150`

```
        pub struct McpConnectionManager {
            clients: HashMap<String, Arc<McpClient>>,  // Server name →
client
            tools: HashMap<String, ToolInfo>,          // Qualified tool
name → metadata
        }

        pub struct ToolInfo {
            pub server_name: String,
            pub tool_name: String,
            pub qualified_name: String,  // "server__tool_name"
            pub description: String,
            pub input_schema: Value,
        }

        impl McpConnectionManager {
            pub async fn new(
                mcp_servers: HashMap<String, McpServerConfig>,
                excluded_tools: HashSet<(String, String)>,
            ) -> Result<(Self, ClientStartErrors)> {
                let mut clients = HashMap::new();
                let mut tools = HashMap::new();
                let mut errors = ClientStartErrors::default();

                // Spawn all servers concurrently
                let mut join_set = JoinSet::new();

                for (server_name, config) in mcp_servers {
                    let server_name_clone = server_name.clone();
                    join_set.spawn(async move {
                        let client = McpClient::new_stdio_client(
```

```
                            config.command.into(),
config.args.into_iter().map(OsString::from).collect(),
                            config.env,
                        ).await?;

                        // Initialize server
                        client.call_method("initialize", json!
({"protocolVersion": "1.0"})).await?;

                        // List tools
                        let response = client.call_method("tools/list",
None).await?;
                        let server_tools: Vec<Tool> =
serde_json::from_value(response["tools"].clone())?;

                        Ok::<(String, Arc<McpClient>, Vec<Tool>), Error>((
                            server_name_clone,
                            Arc::new(client),
                            server_tools,
                        ))
                    });
                }

                // Collect results
                while let Some(result) = join_set.join_next().await {
                    match result? {
                        Ok((server_name, client, server_tools)) => {
                            clients.insert(server_name.clone(), client);

                            // Qualify and aggregate tools
                            for tool in server_tools {
                                let qualified_name =
qualify_tool_name(&server_name, &tool.name);

                                if !excluded_tools.contains(&
(server_name.clone(), tool.name.clone())) {
                                    tools.insert(qualified_name.clone(),
ToolInfo {
                                        server_name: server_name.clone(),
                                        tool_name: tool.name.clone(),
                                        qualified_name,
                                        description: tool.description,
                                        input_schema: tool.input_schema,
                                    });
                                }
                            }
                        },
                        Err(e) => {
                            errors.add(server_name, e);
                        },
                    }
                }

                Ok((Self { clients, tools }, errors))
            }

            pub async fn invoke_tool(
                &self,
                qualified_name: &str,
```

```
            arguments: Value,
        ) -> Result<Value> {
            // Look up tool info
            let tool_info = self.tools.get(qualified_name)

.ok_or(Error::ToolNotFound(qualified_name.to_string()))?;

            // Get client for server
            let client = self.clients.get(&tool_info.server_name)

.ok_or(Error::ServerNotFound(tool_info.server_name.clone()))?;

            // Invoke tool on server
            client.call_tool(&tool_info.tool_name, arguments).await
        }
    }
```

Key Features: - **Concurrent initialization**: Spawn all servers in parallel (JoinSet) - **Tool qualification**: `filesystem__read_file` (prevents name collisions) - **Excluded tools**: Filter out unwanted tools - **Error collection**: Track which servers failed to start - **Shared clients**: `Arc<McpClient>` allows concurrent access

---

Tool Name Qualification

Purpose: Prevent tool name collisions across servers

Strategy: Prefix with server name

```
    fn qualify_tool_name(server_name: &str, tool_name: &str) -> String {
        let combined = format!("{}_{}", server_name, tool_name);

        // Limit to 64 chars (OpenAI tool name limit)
        if combined.len() > 64 {
            // Hash collision avoidance for long names
            let hash = sha1::Sha1::digest(combined.as_bytes());
            let prefix = &combined[..40];
            format!("{}_{:x}", prefix, hash)
        } else {
            combined.replace('-', "_")  // Normalize separators
        }
    }
```

Examples: - `read_file` (filesystem) → `filesystem__read_file` - `query` (database) → `database__query` - `very-long-server-name__very-long-tool-name` → `very-long-server-name__very-long-tool_<hash>`

App-Level Shared Connection Manager

Purpose: Prevent MCP server process multiplication

Problem: Each ChatWidget spawning its own MCP connections would create N×M processes (N widgets × M servers)

Solution: Singleton shared manager

Location: `codex-rs/tui/src/app.rs:105-107`

```rust
        pub(crate) struct App<'a> {
            chat_widgets: Vec<ChatWidget<'a>>,
            mcp_manager:
Arc<tokio::sync::Mutex<Option<Arc<McpConnectionManager>>>>,
            // ↑ Shared singleton across all ChatWidgets
        }

        impl App {
            pub fn new() -> Self {
                // Initialize MCP manager once
                let mcp_manager = Arc::new(tokio::sync::Mutex::new(None));

                tokio::spawn({
                    let mcp_manager = Arc::clone(&mcp_manager);
                    let config = load_config();

                    async move {
                        let manager = McpConnectionManager::new(
                            config.mcp_servers,
                            HashSet::new(),
                        ).await?;

                        *mcp_manager.lock().await = Some(Arc::new(manager));
                    }
                });

                Self {
                    chat_widgets: Vec::new(),
                    mcp_manager,
                }
            }
        }
```

Result: Only M MCP server processes (one per configured server), regardless of widget count

---

Performance: Native vs Subprocess

Benchmark Results

Subprocess approach (pre-2025-10-18):

```
Tool invocation via local-memory MCP:
- Spawn process: ~20ms
- Execute tool: ~15ms
- Parse output: ~5ms
- Process cleanup: ~6ms
Total: ~46ms per tool call
```

Native approach (current):

```
Tool invocation via native client:
- Subprocess already running
- JSON-RPC roundtrip: ~7ms
- Parse response: ~1.7ms
Total: ~8.7ms per tool call
```

Speedup: **5.3× faster** (8.7ms vs 46ms)

Additional benefits: - No process spawn overhead - Persistent connection (reuse stdout/stdin) - Lower memory footprint (shared subprocess)

Server Lifecycle

Initialization Sequence

```
1. Spawn subprocess
   └→ Command::new(program).spawn()

2. Send initialize request
   └→ {"method": "initialize", "params": {"protocolVersion": "1.0"}}

3. Receive initialize response
   └→ {"result": {"capabilities": {...}, "serverInfo": {...}}}

4. Send initialized notification
   └→ {"method": "notifications/initialized"}

5. List tools
   └→ {"method": "tools/list"}
   └→ {"result": {"tools": [...]}}

6. Ready for tool calls
```

Shutdown Sequence

```
1. Send shutdown request
   └→ {"method": "shutdown"}

2. Receive shutdown response
   └→ {"result": null}

3. Send exit notification
   └→ {"method": "exit"}

4. Wait for process exit
   └→ child.wait().await

5. Cleanup (automatic via kill_on_drop)
```

Health Monitoring

Configurable timeouts:

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

[mcp_servers.filesystem]
startup_timeout_sec = 10   # Server initialization timeout
tool_timeout_sec = 60      # Per-tool call timeout
```

Timeout enforcement:

```
// Startup timeout
```

```rust
let client = tokio::time::timeout(
    Duration::from_secs(config.startup_timeout_sec),
    McpClient::new_stdio_client(...)
).await??;

// Tool call timeout
let result = tokio::time::timeout(
    Duration::from_secs(config.tool_timeout_sec),
    client.call_tool(name, args)
).await??;
```

Error Handling

Error Types

```rust
pub enum McpError {
    // Connection errors
    ServerSpawnFailed(io::Error),
    ServerNotResponding,
    ServerCrashed(i32),         // Exit code

    // Protocol errors
    InvalidMessage(serde_json::Error),
    UnexpectedResponse,
    RequestTimeout,

    // Tool errors
    ToolNotFound(String),
    ToolExecutionFailed(String),
    InvalidArguments(String),
}
```

Retry Logic

Transient errors (retry): - Server not responding (may be overloaded) - Request timeout (network issue)

Permanent errors (don't retry): - Tool not found (invalid tool name) - Invalid arguments (schema mismatch) - Server crashed (exit code non-zero)

```rust
let result = execute_with_backoff(
    || client.call_tool(name, args),
    &RetryConfig::default(),
).await?;
```

Graceful Degradation

If MCP server fails:

```rust
match mcp_manager.invoke_tool(name, args).await {
    Ok(result) => {
        // Tool executed successfully
        use_tool_result(result);
    },
```

```
            Err(McpError::ToolNotFound(_)) => {
                // Tool doesn't exist, inform model
                return "Tool not available. Please try another approach.";
            },
            Err(McpError::ServerCrashed(_)) => {
                // Server crashed, disable MCP for this conversation
                disable_mcp_for_conversation();
                return "MCP server unavailable. Continuing without tools.";
            },
            Err(e) => {
                // Other error, retry or fail
                return format!("Tool execution failed: {}", e);
            },
        }
```

Tool Schema Validation

Input Schema

From MCP server:

```
        {
          "name": "read_file",
          "description": "Read contents of a file",
          "inputSchema": {
            "type": "object",
            "properties": {
              "path": {
                "type": "string",
                "description": "File path to read"
              }
            },
            "required": ["path"]
          }
        }
```

Validation before invocation:

```
        pub fn validate_tool_arguments(
            tool_info: &ToolInfo,
            arguments: &Value,
        ) -> Result<()> {
            let schema = &tool_info.input_schema;

            // Use jsonschema crate for validation
            let compiled_schema = JSONSchema::compile(schema)?;

            compiled_schema.validate(arguments)
                .map_err(|errors| {
                    let messages: Vec<_> = errors.map(|e|
e.to_string()).collect();
                    Error::InvalidArguments(messages.join(", "))
                })
        }
```

Configuration

Per-Server Config

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

        [mcp_servers.filesystem]
        command = "npx"
        args = ["-y", "@modelcontextprotocol/server-filesystem",
"/home/user/project"]
        env = { "NODE_ENV" = "production" }
        startup_timeout_sec = 10
        tool_timeout_sec = 60

        [mcp_servers.local-memory]
        command = "npx"
        args = ["-y", "@modelcontextprotocol/server-local-memory"]
        startup_timeout_sec = 10
        tool_timeout_sec = 30

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

Excluded Tools

Filter out specific tools:

```
        [mcp_servers.filesystem]
        excluded_tools = ["write_file", "delete_file"]  # Read-only mode
```

Programmatic exclusion:

```
        let excluded = HashSet::from([
            ("filesystem".to_string(), "write_file".to_string()),
            ("filesystem".to_string(), "delete_file".to_string()),
        ]);

        let (manager, errors) = McpConnectionManager::new(
            config.mcp_servers,
            excluded,
        ).await?;
```

Summary

MCP Integration Highlights:

1. **Native Client**: 5.3× faster than subprocess (8.7ms vs 46ms)
2. **Concurrent I/O**: Separate reader/writer tasks prevent deadlock
3. **Shared Manager**: App-level singleton prevents process multiplication
4. **Tool Qualification**: `server__tool_name` prevents collisions
5. **Timeout Enforcement**: Per-server startup and per-tool timeouts
6. **Error Handling**: Retry transient errors, fail fast on permanent
7. **Schema Validation**: Validate arguments before invocation
8. **Graceful Degradation**: Continue without MCP if servers fail

Next Steps: - Database Layer - SQLite optimization - Configuration System - Hot-reload

---

File References: - MCP client: `codex-rs/mcp-client/src/mcp_client.rs:63-150` - Connection manager: `codex-rs/core/src/mcp_connection_manager.rs:84-150` - JSON-RPC types: `codex-rs/mcp-types/src/jsonrpc.rs` - App integration: `codex-rs/tui/src/app.rs:105-107`

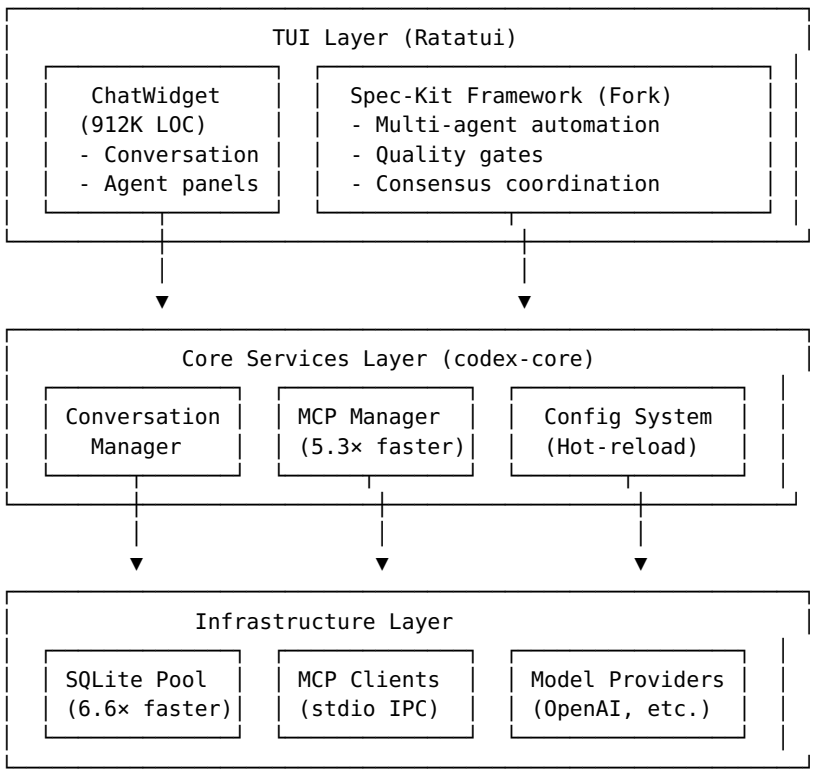
System Overview & Architecture

Comprehensive overview of the theturtlecsz/code architecture.

Table of Contents

High-Level Overview

theturtlecsz/code is a terminal-based AI coding assistant built on a multi-layered architecture:



Key Characteristics: - **226,607 lines of Rust** across 538 source files - **24-crate Cargo workspace** with modular design - **Async/sync hybrid** (Tokio async core, Ratatui sync UI) - **Native MCP integration** (5.3× faster than subprocess) - **98.8% upstream isolation** (fork features in isolated modules)

Design Philosophy

1. Separation of Concerns

Layer Boundaries: - **UI Layer**: Ratatui TUI, user interaction, rendering - **Application Layer**: Business logic, workflow orchestration - **Service Layer**: MCP, database, configuration, agents - **Infrastructure Layer**: SQLite, stdio, HTTP clients

2. Async/Sync Hybrid Architecture

Rationale: Ratatui requires synchronous event loop, but backend services benefit from async I/O.

Solution: Clear async/sync boundary using channels:

```
// Sync UI Thread                    Async Backend Task
ChatWidget                           ConversationManager
    ↓                                       ↓
codex_op_tx.send(op)  ──────→   conversation.submit_op().await
    ↑                                       ↓
app_event_tx.recv()   ←──────   Event::CodexEvent(event)
```

Pattern: `UnboundedSender<Op>` bridges sync UI to async backend.

File: `codex-rs/tui/src/chatwidget/agent.rs:16-62`

3. Modularity via Cargo Workspace

Benefits: - Clear dependency boundaries - Independent compilation units - Parallel builds (faster CI) - Reusable components (spec-kit as separate crate)

Workspace Size: 24 crates, ~220K LOC Rust

4. Fork Isolation Strategy

Goal: Minimize rebase conflicts with upstream (`just-every/code`)

Implementation: - Fork features in isolated modules (`tui/src/chatwidget/spec_kit/`) - “Friend module” pattern (access ChatWidget private fields without public API) - Dynamic command registry (avoids enum growth in SlashCommand) - Trait-based abstractions (SpecKitContext decouples from ChatWidget)

Result: 98.8% isolation (55 modules under `spec_kit/`, minimal upstream changes)

Evidence: `docs/spec-kit/REFACTORING_COMPLETE_SUMMARY.md`

5. Performance-First Design

Database: - SQLite with WAL mode: **6.6× read speedup** (850µs → 129µs) - R2D2 connection pooling: Eliminates connection overhead - Incremental auto-vacuum: Prevents unbounded growth

MCP Integration: - Native client: **5.3× faster** than subprocess approach - Shared connection manager: Prevents process multiplication - 1MB buffer for large tool responses

Configuration: - Hot-reload with 300ms debounce - Arc for atomic updates (<1ms lock time)

6. Fault Tolerance

Retry Logic (SPEC-945C): - Exponential backoff: 100ms → 200ms → 400ms → 800ms - Jitter to prevent thundering herd - Permanent vs transient error classification

Graceful Degradation: - Multi-agent consensus works with 2/3 agents (if 1 fails) - MCP server failures don't crash app - Config reload failures preserve old config

---

Component Architecture

Core Components

1. TUI Layer (`codex-tui` crate)

Purpose: Terminal user interface using Ratatui framework

Key Components: - **App**: Top-level application state, event loop - **ChatWidget**: Main conversation interface (912K LOC) - **BottomPane**: Input composer, status bar - **HistoryCell**: Message rendering (user, assistant, tool, exec) - **StreamController**: Real-time token streaming

Event Flow:

```
Terminal Events → App → ChatWidget → Handle Event → Render
                                ↓
                          Submit Op to Backend
                                ↓
                          Receive Events from Backend
```

File: `codex-rs/tui/src/app.rs`, `codex-rs/tui/src/chatwidget/mod.rs`

---

2. Spec-Kit Framework (`spec-kit` crate + TUI integration)

Purpose: Multi-agent automation pipeline with quality gates

Architecture:

```
User Command (/speckit.auto)
    ↓
Command Registry (dynamic dispatch)
```

```
    ↓
Pipeline Coordinator (state machine)
    ↓
Agent Orchestrator (spawn agents)
    ↓
Consensus Coordinator (aggregate results)
    ↓
Quality Gate Broker (checkpoints)
    ↓
Evidence Repository (artifacts)
```

Key Modules: - **spec-kit** (library crate): Config, retry, types - **tui/src/chatwidget/spec_kit** (55 modules): TUI integration - `command_registry.rs`: Dynamic command dispatch - `pipeline_coordinator.rs`: Workflow state machine - `agent_orchestrator.rs`: Agent lifecycle - `consensus_coordinator.rs`: Multi-agent consensus - `native_*.rs`: Zero-cost operations (FREE) - `consensus_db.rs`: SQLite artifact storage

File: `codex-rs/spec-kit/src/lib.rs`, `codex-rs/tui/src/chatwidget/spec_kit/mod.rs`

---

3. Core Services (`codex-core` crate)

Purpose: Backend services for conversation, MCP, database, config

Key Modules: - **ConversationManager**: Agent conversation lifecycle - **McpConnectionManager**: MCP server aggregation - **Config**: Configuration loading, validation - **Database**: SQLite connection pooling, transactions - **Protocol**: OpenAI API client, model providers

Responsibilities: - Agent spawning and orchestration - MCP tool invocation - Model provider communication (OpenAI, Anthropic, Google) - Configuration hot-reload - Consensus artifact storage

File: `codex-rs/core/src/lib.rs`

---

4. MCP Integration (`mcp-client`, `mcp-types` crates)

Purpose: Model Context Protocol client and server support

Components: - **McpClient**: Async client for stdio communication - **McpConnectionManager**: Central hub for all MCP servers - **mcp-types**: JSON-RPC protocol types

Architecture:

```
MCP Server (subprocess)
    ↓ stdin/stdout
McpClient
├── Writer Task: outgoing_rx → stdin (JSON-RPC requests)
├── Reader Task: stdout → pending HashMap (JSON-RPC responses)
└── Dispatcher: Request ID → oneshot::Sender (pair requests/responses)
```

Performance: - **Native integration**: 5.3× faster than subprocess (8.7ms typical) - **Concurrent I/O**: Separate reader/writer tasks prevent deadlock - **1MB buffer**: Handles large tool responses

File: `codex-rs/mcp-client/src/mcp_client.rs:63-150`

---

5. Database Layer (`consensus_db` in spec-kit, `db` module in core)

Purpose: SQLite storage for consensus artifacts, config, telemetry

Architecture:

```
Application
    ↓
R2D2 Connection Pool (2-8 connections)
    ↓
SQLite Connection (WAL mode)
    ↓
Database File (consensus_artifacts.db)
```

Optimizations: - **WAL mode**: 6.6× read speedup (allows concurrent reads) - **Connection pooling**: Eliminates connection overhead - **Optimized pragmas**: 32MB cache, memory-mapped I/O (1GB) - **Incremental auto-vacuum**: 99.95% size reduction after cleanup

Schema: - `consensus_runs`: Workflow execution tracking - `agent_outputs`: Individual agent responses - `consensus_artifacts`: Synthesized consensus results

File: `codex-rs/core/src/db/connection.rs:39-105`, `codex-rs/tui/src/chatwidget/spec_kit/consensus_db.rs`

---

6. Configuration System (`config` module in spec-kit)

Purpose: Layered configuration with hot-reload

5-Tier Precedence (highest to lowest): 1. **CLI flags**: `--model gpt-5`, `--config key=value` 2. **Shell environment**: `export OPENAI_API_KEY=...` 3. **Profile**: `[profiles.premium]` in config.toml 4. **Config file**: `~/.code/config.toml` 5. **Defaults**: Built-in fallback values

Hot-Reload:

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

Performance: - Reload latency: <100ms (p95) - Lock contention: <1ms write locks - CPU overhead: <0.5% idle

File: `codex-rs/spec-kit/src/config/hot_reload.rs:1-100`

---

7. Model Providers (`codex-protocol`, `codex-chatgpt` crates)

Purpose: Communication with AI model APIs

Providers: - **OpenAI**: GPT-5, GPT-4o, o3 (Responses API) - **Anthropic**: Claude Sonnet, Haiku, Opus (via CLI) - **Google**: Gemini Pro, Flash (via CLI) - **Custom**: Any OpenAI-compatible endpoint

Features: - Streaming responses (SSE) - Retry logic (exponential backoff) - Rate limit handling - Zero Data Retention support (ZDR)

File: `codex-rs/protocol/src/lib.rs`, `codex-rs/chatgpt/src/lib.rs`

Data Flow

Conversation Flow

```
1. User Input
   └→ ChatWidget.handle_key_event()
       └→ ChatWidget.submit_prompt()
           └→ codex_op_tx.send(Op::NewMessage)

2. Async Backend
   └→ ConversationManager.submit(op)
       └→ Conversation.process()
           └→ Model Provider API (OpenAI/Anthropic/Google)

3. Response Streaming
   └→ conversation.next_event()
       └→ app_event_tx.send(AppEvent::CodexEvent)
           └→ ChatWidget.handle_event()
               └→ Render response tokens
```

Spec-Kit Automation Flow

```
1. User Command: /speckit.auto SPEC-ID
   └→ CommandRegistry.find("speckit.auto")
       └→ AutoCommand.execute(widget, args)

2. Pipeline Initialization
   └→ PipelineCoordinator.start_pipeline()
       └→ Load SPEC from docs/SPEC-{ID}-*/spec.md

3. Stage Execution Loop
   └→ For each stage (specify, plan, tasks, implement, validate,
audit, unlock):
       ├→ NativeQualityGate.check() [if native stage]
       ├→ AgentOrchestrator.spawn_agents() [if multi-agent]
       ├→ ConsensusCoordinator.synthesize() [aggregate results]
       ├→ QualityGateBroker.validate() [checkpoint]
       └→ EvidenceRepository.store() [artifacts]

4. Completion
   └→ PipelineCoordinator.complete()
       └→ Push results to ChatWidget history
```

MCP Tool Invocation Flow

```
1. Agent requests tool
   └→ Model response: {"type": "tool_use", "name":
"filesystem__read_file"}

2. Tool dispatch
   └→ McpConnectionManager.invoke_tool()
```

```
       └→ Find MCP client by tool prefix ("filesystem")
           └→ McpClient.call_tool()

3. Server communication
   └→ outgoing_tx.send(JSONRPCRequest)
       └→ Writer task → stdin (JSON)
           └→ MCP server processes request
               └→ stdout (JSON) → Reader task
                   └→ pending.get(request_id).send(response)

4. Result returned
   └→ Tool result forwarded to model
       └→ Model continues generation with tool output
```

Technology Stack

Core Technologies

Language: Rust (Edition 2024) - Memory safety without garbage collection - Zero-cost abstractions - Fearless concurrency

Async Runtime: Tokio - Multi-threaded work-stealing scheduler - Async I/O for network, file, subprocess - Channels for sync/async boundary

Terminal UI: Ratatui (v0.29.0, patched fork) - Immediate-mode rendering - Cross-platform terminal support - Widget composition

Database: SQLite (rusqlite crate) - Embedded database (no server) - ACID transactions - WAL mode for concurrency

Serialization: Serde (JSON, TOML, YAML) - Compile-time serialization - Type-safe deserialization - Schema validation

Supporting Libraries

Networking: - `reqwest`: HTTP client (model providers) - `eventsource-stream`: Server-Sent Events (streaming)

Configuration: - `toml`: Config file parsing - `notify`: Filesystem watching (hot-reload)

Database: - `rusqlite`: SQLite bindings - `r2d2`: Connection pooling - `r2d2_sqlite`: SQLite adapter for r2d2

MCP: - `mcp-types`: Protocol definitions (internal) - `tokio-util`: Codec for line-delimited JSON

Error Handling: - `anyhow`: Flexible error types - `thiserror`: Custom error derive

CLI: - `clap`: Command-line argument parsing - `clap_complete`: Shell completion generation

---

Fork-Specific Additions

Isolation Strategy

Goal: Add fork features without upstream conflicts

Implementation:

1. **Isolated Module Tree**:

```
codex-rs/tui/src/chatwidget/
├── mod.rs (upstream)
└── spec_kit/ (fork, 55 modules, 98.8% isolated)
    ├── mod.rs
    ├── command_registry.rs
    ├── pipeline_coordinator.rs
    └── ... (50+ modules)
```

2. **Friend Module Pattern**:

```rust
// In chatwidget/mod.rs (upstream file, minimal change)
pub mod spec_kit;  // Single line addition

// spec_kit modules can access ChatWidget private fields
impl ChatWidget {
    fn internal_method(&mut self) { /* ... */ }
}
```

3. **Dynamic Command Registry** (avoids upstream enum):

```rust
// Upstream: SlashCommand enum
pub enum SlashCommand { New, Model, Reasoning, ... }

// Fork: Dynamic registry (no enum growth)
pub trait SpecKitCommand { /* ... */ }
SPEC_KIT_REGISTRY.register(Box::new(AutoCommand));
```

4. **Context Trait** (decouples from ChatWidget):

```rust
pub trait SpecKitContext {
    fn submit_operation(&self, op: Op);
    fn push_error(&mut self, message: String);
    // ... methods spec-kit needs
}

impl SpecKitContext for ChatWidget { /* ... */ }
```

Result: - **98.8% isolation**: 55 modules, 1,222 lines extracted - **Zero upstream conflicts**: Rebases require <10 lines of merge - **Testability**: MockSpecKitContext for unit tests - **Maintainability**: Clear separation of concerns

Evidence: `docs/spec-kit/REFACTORING_COMPLETE_SUMMARY.md`

New Crates

spec-kit (`codex-rs/spec-kit/`): - Configuration system (hot-reload, 5-tier precedence) - Retry logic (exponential backoff, jitter) - Types and error handling - Evidence management - Cost tracking

Purpose: Reusable library for spec-kit automation (can extract as standalone in future per MAINT-10)

Integration Points

1. TUI ↔ Core Services

Boundary: Async/sync channel boundary

Direction: Bidirectional - **TUI → Core**: `UnboundedSender<Op>` - **Core → TUI**: `AppEventSender`

Pattern: Message passing with typed events

---

2. Core ↔ MCP Servers

Boundary: stdio subprocess communication

Direction: Bidirectional (JSON-RPC over stdin/stdout)

Protocol: Line-delimited JSON (Model Context Protocol)

Lifecycle: 1. Spawn subprocess (`tokio::process::Command`) 2. Initialize with `initialize` request 3. List tools with `tools/list` request 4. Invoke tools with `tools/call` request 5. Kill on app exit (`kill_on_drop = true`)

---

3. Core ↔ Model Providers

Boundary: HTTPS API requests

Direction: Request/response with streaming

Protocols: - **OpenAI**: Responses API (streaming SSE) - **Anthropic**: Messages API (via CLI subprocess) - **Google**: Gemini API (via CLI subprocess)

Features: - Retry logic (exponential backoff) - Rate limit handling (429 response) - Streaming token delivery

---

4. Spec-Kit ↔ Database

Boundary: R2D2 connection pool

Direction: Read/write consensus artifacts

Operations: - Store agent outputs (per-agent JSON blobs) - Store consensus synthesis (aggregated results) - Query historical runs (evidence retrieval) - Atomic transactions (ACID guarantees)

---

5. Configuration ↔ Filesystem

Boundary: File watching (notify crate)

Direction: Read config.toml, watch for changes

Hot-Reload: 1. Filesystem change event 2. Debounce (300ms window) 3. Parse and validate config 4. Atomic update via Arc 5. Emit reload event to UI

Summary

Architecture Highlights:

1. **Clean Layering**: TUI → Core → Infrastructure
2. **Async/Sync Hybrid**: Tokio backend, Ratatui UI
3. **Modular Design**: 24-crate workspace
4. **Fork Isolation**: 98.8% isolation via friend modules
5. **Performance-First**: 6.6× DB speedup, 5.3× MCP speedup
6. **Fault Tolerance**: Retry logic, graceful degradation

Next Steps: - Cargo Workspace Guide - Detailed crate documentation - TUI Architecture - Ratatui and async/sync patterns - MCP Integration - Native client details - Database Layer - SQLite optimization

---

File References: - Workspace: `codex-rs/Cargo.toml` - TUI: `codex-rs/tui/src/app.rs`, `codex-rs/tui/src/chatwidget/mod.rs` - Spec-Kit: `codex-rs/spec-kit/src/lib.rs`, `codex-rs/tui/src/chatwidget/spec_kit/mod.rs` - Core: `codex-rs/core/src/lib.rs` - MCP: `codex-rs/mcp-client/src/mcp_client.rs` - DB: `codex-rs/core/src/db/connection.rs` - Config: `codex-rs/spec-kit/src/config/hot_reload.rs`

TUI Architecture

Detailed architecture of the Terminal User Interface layer.

Overview

The TUI layer uses **Ratatui** (v0.29.0, patched fork) with a hybrid **async/sync** architecture:

- **Sync Layer**: Ratatui event loop (blocking terminal operations)
- **Async Layer**: Tokio tasks (network I/O, subprocess management)
- **Bridge**: Channels (`UnboundedSender`, `AppEventSender`)

File: `codex-rs/tui/src/`

Component Hierarchy

```
App (top-level state)
 ├── ChatWidget (conversation interface)
 │    ├── BottomPane (input + status)
 │    ├── HistoryCell[] (message history)
 │    ├── StreamController (token streaming)
```

```
│    ├── InterruptManager (Ctrl+C handling)
│    └── SpecKitState (fork features)
├── TerminalInfo (size, capabilities)
└── AppEventReceiver (backend events)
```

ChatWidget Structure

Location: `codex-rs/tui/src/chatwidget/mod.rs:373+`

```rust
        pub(crate) struct ChatWidget<'a> {
            // === Backend Communication ===
            app_event_tx: AppEventSender,                    // Send events
to App
            codex_op_tx: UnboundedSender<Op>,                // Submit
operations to backend

            // === UI Components ===
            bottom_pane: BottomPane<'a>,                     // Input
composer + status bar
            history_cells: Vec<Box<dyn HistoryCell>>,        // Conversation
history

            // === State ===
            config: Config,                                  // Configuration
            auth_manager: Arc<AuthManager>,                  //
Authentication state

            // === Spec-Kit (Fork) ===
            spec_auto_state: Option<SpecAutoState>,          // Pipeline
orchestration
            cost_tracker: Arc<spec_kit::cost_tracker::CostTracker>, // Cost
tracking

            // === Agent Tracking ===
            active_agents: Vec<AgentInfo>,                   // Running
agents
            agent_runtime: HashMap<String, AgentRuntime>,    // Agent
metadata

            // === Execution State ===
            exec: ExecState,                                 // Command
execution tracking
            terminal: TerminalState,                         // Tmux session
state

            // === Rendering ===
            stream: StreamController,                        // Token
streaming
            interrupts: InterruptManager,                    // Interrupt
handling
            cached_cell_size: OnceCell<(u16, u16)>,          // Terminal
dimensions
        }
```

Key Insights: - **912K LOC in mod.rs** (large monolithic file) - **Friend module access**: `spec_kit` modules can access private fields - **Hybrid ownership**: `Arc<T>` for shared state, direct ownership for UI

Async/Sync Boundary Pattern

The Problem

Ratatui requires synchronous event loop:

```rust
loop {
    terminal.draw(|f| ui(f, &app))?;  // Blocking draw
    let event = event::read()?;       // Blocking read
    app.handle_event(event);          // Sync handler
}
```

Backend requires async I/O:

```rust
async fn conversation_loop() {
    let response = model_provider.chat(request).await?; // Async network I/O
    while let Some(token) = response.next().await? {    // Async streaming
        // ...
    }
}
```

Constraint: Can't `.await` in sync event loop ✖

The Solution: Channel Bridge

Pattern:

```
Sync UI Thread                    Async Backend Task
   ChatWidget                      ConversationManager
       ↓                                  ↓
   submit_prompt()              async conversation_loop()
       ↓                                  ↓
   codex_op_tx.send(op)  ──→    codex_op_rx.recv().await
       ↓                                  ↓
   app_event_rx.recv()   ←──    app_event_tx.send(event)
```

Implementation: `codex-rs/tui/src/chatwidget/agent.rs:16-62`

```rust
pub(crate) fn spawn_agent(
    config: Config,
    app_event_tx: AppEventSender,
    server: Arc<ConversationManager>,
) -> UnboundedSender<Op> {
    // Create channel for UI → Backend
    let (codex_op_tx, mut codex_op_rx) = unbounded_channel::<Op>();

    // Spawn async task
    tokio::spawn(async move {
        // Create conversation
        let NewConversation { conversation, .. } =
            server.new_conversation(config).await?;

        // Forward operations to conversation
        tokio::spawn(async move {
            while let Some(op) = codex_op_rx.recv().await {
                conversation.submit(op).await;
            }
```

```
            });

            // Forward events back to UI
            while let Ok(event) = conversation.next_event().await {
                app_event_tx.send(AppEvent::CodexEvent(event))?;
            }
        });

        // Return sync sender to UI
        codex_op_tx
    }
```

Key Points: - **`UnboundedSender<Op>`**: Sync side can send without `.await` - **`app_event_tx.send()`**: Backend sends events to UI queue - **Tokio runtime**: Spawned on separate thread pool - **No blocking**: UI thread never blocks on network I/O

Operation Types (`Op` enum)

```
    pub enum Op {
        NewMessage(String),              // User prompt
        ToolResponse(ToolCallId, Result<String>), // Tool execution
result
        RegenerateLastMessage,           // Retry last response
        Interrupt,                       // Cancel current operation
        // ... 10+ variants
    }
```

Flow: 1. User types message → `ChatWidget.submit_prompt()` 2. Create `Op::NewMessage` → `codex_op_tx.send(op)` 3. Backend receives → `conversation.submit(op).await` 4. Model responds → `app_event_tx.send(Event::Token)` 5. UI receives → `ChatWidget.handle_event()` → Render

---

Event Loop

Location: `codex-rs/tui/src/app.rs`

```
    impl App {
        pub fn run(mut self) -> Result<()> {
            loop {
                // Draw UI
                self.terminal.draw(|f| {
                    self.chat_widget.render(f, f.size());
                })?;

                // Handle events (non-blocking with timeout)
                if event::poll(Duration::from_millis(16))? {
                    match event::read()? {
                        Event::Key(key) => self.handle_key(key)?,
                        Event::Resize(w, h) => self.handle_resize(w,
h)?,
                        Event::Mouse(mouse) =>
self.handle_mouse(mouse)?,
                        _ => {}
                    }
                }
```

```
            // Process backend events
            while let Ok(app_event) = self.app_event_rx.try_recv() {
                self.handle_app_event(app_event)?;
            }

            // Check for exit
            if self.should_exit {
                break;
            }
        }
        Ok(())
    }
}
```

Loop Phases: 1. **Draw**: Render UI to terminal buffer 2. **Poll**: Check for terminal events (16ms timeout = ~60 FPS) 3. **Handle Terminal Events**: Keyboard, mouse, resize 4. **Process Backend Events**: Tokens, completions, errors 5. **Check Exit**: Break if requested

---

Rendering System

Immediate Mode Rendering

Ratatui uses immediate-mode: - No retained UI tree - Full re-render every frame - Layout calculated on-the-fly

Performance: ~60 FPS for typical conversation UI

Widget Composition

```
impl ChatWidget {
    pub fn render(&mut self, f: &mut Frame, area: Rect) {
        // Split layout
        let chunks = Layout::default()
            .direction(Direction::Vertical)
            .constraints([
                Constraint::Min(1),       // History
                Constraint::Length(3),    // Composer
                Constraint::Length(1),    // Status bar
            ])
            .split(area);

        // Render history
        self.render_history(f, chunks[0]);

        // Render input composer
        self.bottom_pane.render(f, chunks[1]);

        // Render status bar
        self.render_status(f, chunks[2]);
    }
}
```

HistoryCell Trait

Location: `codex-rs/tui/src/chatwidget/history/`

```rust
pub trait HistoryCell: Send {
    fn height(&self, width: u16) -> u16;          // Calculate cell height
    fn render(&self, frame: &mut Frame, area: Rect); // Render to area
}

// Implementations:
pub struct UserMessageCell { /* ... */ }      // User prompt
pub struct AssistantMessageCell { /* ... */ } // AI response
pub struct ToolExecutionCell { /* ... */ }    // Tool call/result
pub struct ExecCell { /* ... */ }             // Command execution
pub struct BackgroundEventCell { /* ... */ }  // System messages
```

Dynamic Dispatch: `Vec<Box<dyn HistoryCell>>` allows heterogeneous message types

Spec-Kit Integration (Friend Module)

Friend Module Pattern

Declared in `chatwidget/mod.rs`:

```rust
pub mod spec_kit;  // Friend module - can access private fields
```

Benefits: - ✅ Spec-kit can read/write `ChatWidget` internals - ✅ No public API pollution - ✅ Clear encapsulation boundary - ✅ Easy to test (spec_kit modules are independent)

Context Trait Abstraction

Location: `codex-rs/tui/src/chatwidget/spec_kit/context.rs:1-140`

```rust
pub trait SpecKitContext {
    // History operations
    fn history_push(&mut self, cell: impl HistoryCell + 'static);
    fn push_error(&mut self, message: String);
    fn push_background(&mut self, message: String, placement: BackgroundPlacement);

    // UI operations
    fn request_redraw(&mut self);

    // Agent/operation submission
    fn submit_operation(&self, op: Op);
    fn submit_prompt(&mut self, display: String, prompt: String);

    // Configuration access
    fn working_directory(&self) -> &Path;
    fn agent_config(&self) -> &[AgentConfig];

    // Spec auto state
    fn spec_auto_state_mut(&mut self) -> &mut Option<SpecAutoState>;
    fn spec_auto_state(&self) -> &Option<SpecAutoState>;
```

```
            // Guardrail operations
            fn collect_guardrail_outcome(&self, spec_id: &str, stage:
SpecStage) -> Result<GuardrailOutcome>;
            fn run_spec_consensus(&mut self, spec_id: &str, stage:
SpecStage)
                -> Result<(Vec<Line<'static>>, bool)>;
        }

        impl SpecKitContext for ChatWidget {
            // Implementation delegates to ChatWidget methods
        }
```

Purpose: Decouples spec-kit from ChatWidget implementation

Testing: Mock implementation in `context::test_mock`

Streaming & Interrupts

StreamController

Location: `codex-rs/tui/src/streaming/controller.rs`

```
        pub struct StreamController {
            active_stream: Option<StreamState>,
        }

        impl StreamController {
            pub fn start_stream(&mut self, request_id: String) {
                self.active_stream = Some(StreamState {
                    request_id,
                    tokens: Vec::new(),
                    started_at: Instant::now(),
                });
            }

            pub fn append_token(&mut self, token: String) {
                if let Some(stream) = &mut self.active_stream {
                    stream.tokens.push(token);
                }
            }

            pub fn finish_stream(&mut self) -> Option<StreamState> {
                self.active_stream.take()
            }
        }
```

Flow: 1. Backend sends `Event::StreamStart` 2. `StreamController.start_stream()` 3. Backend sends `Event::Token(tok)` repeatedly 4. `StreamController.append_token(tok)` 5. UI renders partial response on each frame 6. Backend sends `Event::StreamEnd` 7. `StreamController.finish_stream()`

InterruptManager

Location: `codex-rs/tui/src/chatwidget/interrupts.rs`

```
        pub struct InterruptManager {
```

```
        pending_interrupt: bool,
        last_interrupt_at: Option<Instant>,
    }

    impl InterruptManager {
        pub fn request_interrupt(&mut self) {
            self.pending_interrupt = true;
            self.last_interrupt_at = Some(Instant::now());
        }

        pub fn consume_interrupt(&mut self) -> bool {
            std::mem::replace(&mut self.pending_interrupt, false)
        }
    }
```

Usage: Ctrl+C → `InterruptManager.request_interrupt()` → Send `Op::Interrupt` to backend → Backend cancels operation

---

Input Handling

BottomPane (Composer)

Location: `codex-rs/tui/src/chatwidget/bottom_pane.rs`

```
    pub struct BottomPane<'a> {
        input: String,                      // Current input buffer
        cursor_position: usize,             // Cursor position
        history_index: Option<usize>,       // Command history
navigation
        file_search: Option<FileSearch>,    // @ file search state
    }

    impl BottomPane {
        pub fn handle_key(&mut self, key: KeyEvent) -> InputAction {
            match key.code {
                KeyCode::Enter =>
InputAction::Submit(std::mem::take(&mut self.input)),
                KeyCode::Char('@') if self.input.is_empty() => {
                    self.file_search = Some(FileSearch::new());
                    InputAction::None
                },
                KeyCode::Char(c) => {
                    self.input.insert(self.cursor_position, c);
                    self.cursor_position += 1;
                    InputAction::None
                },
                KeyCode::Backspace => {
                    if self.cursor_position > 0 {
                        self.input.remove(self.cursor_position - 1);
                        self.cursor_position -= 1;
                    }
                    InputAction::None
                },
                // ... more key handlers
            }
        }
    }
```

Features: - Multi-line input (Shift+Enter) - Cursor movement (arrows, Home, End) - History navigation (Esc Esc) - File search (@ trigger) - Paste support (Ctrl+V with image detection)

File Search (@-trigger)

Location: `codex-rs/file-search/src/lib.rs`

```rust
pub struct FileSearch {
    query: String,
    results: Vec<PathBuf>,
    selected_index: usize,
}

impl FileSearch {
    pub fn update_query(&mut self, query: String) {
        self.query = query;
        self.results = fuzzy_search(&query, max_results: 10);
        self.selected_index = 0;
    }
}

fn fuzzy_search(query: &str, max_results: usize) -> Vec<PathBuf> {
    // Use nucleo-matcher for fuzzy matching
    // Search workspace for files matching query
}
```

UI Flow: 1. User types `@` → Activate file search 2. User types `main` → Update query, show results 3. User presses Up/Down → Navigate results 4. User presses Tab/Enter → Insert file path, exit search 5. User presses Esc → Cancel search

---

Performance Considerations

Rendering Optimizations

Lazy Height Calculation:

```rust
pub fn height(&self, width: u16) -> u16 {
    // Calculate height only when width changes
    if self.cached_width == Some(width) {
        return self.cached_height;
    }
    let height = self.calculate_height(width);
    self.cached_width = Some(width);
    self.cached_height = height;
    height
}
```

Viewport Culling: - Only render visible history cells - Skip cells outside viewport - Recalculate on scroll

---

Event Processing

Non-Blocking Event Poll:

```rust
if event::poll(Duration::from_millis(16))? {
    // Process event
}
```

- 16ms = ~60 FPS target
- Non-blocking (returns immediately if no events)
- Allows backend event processing every frame

Summary

TUI Architecture Highlights:

1. **Async/Sync Hybrid**: Channels bridge sync UI and async backend
2. **Immediate-Mode Rendering**: Full re-render every frame (~60 FPS)
3. **Dynamic Dispatch**: `Box<dyn HistoryCell>` for heterogeneous messages
4. **Friend Module Pattern**: Spec-kit access to ChatWidget internals
5. **Context Trait**: Decouples spec-kit from ChatWidget implementation
6. **Streaming**: Real-time token rendering
7. **Interrupts**: Ctrl+C gracefully cancels operations

Next Steps: - Core Execution - Agent orchestration - MCP Integration - Native client details - Database Layer - SQLite optimization

---

File References: - ChatWidget: `codex-rs/tui/src/chatwidget/mod.rs:373+` - Agent spawner: `codex-rs/tui/src/chatwidget/agent.rs:16-62` - Event loop: `codex-rs/tui/src/app.rs` - Context trait: `codex-rs/tui/src/chatwidget/spec_kit/context.rs:1-140` - Streaming: `codex-rs/tui/src/streaming/controller.rs` - Bottom pane: `codex-rs/tui/src/chatwidget/bottom_pane.rs`
