recommendations

SPEC-PPP-002: Implementation Recommendations

Target: 95%+ PPP Framework Compliance (Personalization Dimension) **Phased Rollout**: 3 phases over 4-6 months **Total Effort**: 68 hours (~2 weeks for 1 engineer)

✅ Phase 1: Foundation (12/20 Preferences, 60% Compliance)

Duration: 1 week **Effort**: 34 hours **Compliance Target**: 60% (12/20 preferences) **Risk Level**: LOW

Scope

Include (LOW-MEDIUM complexity only): 1. ✅ `no_preference` (baseline) 2. ✅ `concise_question` (prompt injection) 3. ✅ `detail_question` (prompt injection) 4. ✅ `answer_more` (prompt injection) 5. ✅ `no_ask` (prompt injection) 6. ✅ `professional` (prompt injection) 7. ✅ `amateur` (prompt injection) 8. ✅ `capital` (post-processing) 9. ✅ `commas` (post-processing) 10. ✅ `json` (validation + retry) 11. ✅ `length` (post-processing) 12. ✅ `snippet` (validation)

Defer to Phase 2: - `do_selection`, `only_begin`, `ask_many`, `one_question`, `first_try` (require trajectory tracking) - `lang_ita`, `lang_multi` (require translation service) - `joke` (complex content generation)

Implementation Tasks

Task 1: Configuration Schema (8 hours)

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

        #[derive(Deserialize, Serialize, Debug, Clone, PartialEq)]
        pub struct UserPreferences {
            // Phase 1: Core preferences
            pub interaction_mode: InteractionMode,       // PPP #1, #4, #6
            pub question_detail: QuestionDetail,         // PPP #2, #3
            pub expertise_level: ExpertiseLevel,         // PPP #8, #9
            pub format_constraints: FormatConstraints,   // PPP #15, #16,
#17, #20
            pub content_requirements: ContentRequirements, // PPP #19
        }

        impl UserPreferences {
            pub fn validate(&self) -> Result<(), Vec<PreferenceConflict>>;
            pub fn apply_to_prompt(&self, base: &str) -> String;
```

```
        pub fn validate_output(&self, output: &str) -> ValidationResult;
    }
```

Task 2: Prompt Injection Layer (4 hours)

```
    // File: codex-rs/tui/src/chatwidget/spec_kit/consensus.rs

    // In artifact processing (line ~220):
    let personalized_prompt = if let Some(prefs) =
&config.user_preferences {
        prefs.apply_to_prompt(&base_prompt)
    } else {
        base_prompt
    };
```

Task 3: Output Validation Module (8 hours)

```
    // File: codex-rs/tui/src/output_formatter.rs (NEW)

    pub struct OutputFormatter {
        preferences: UserPreferences,
    }

    impl OutputFormatter {
        pub fn format_and_validate(&self, output: &str) ->
FormattedOutput {
            // 1. Validate against preferences
            // 2. Apply transformations if needed
            // 3. Return formatted output + validation result
        }
    }
```

Task 4: Post-Processing Transformations (6 hours)

```
    impl OutputFormatter {
        fn enforce_no_commas(&self, text: &str) -> String {
            text.replace(',', " and").replace(", ", " and ")
        }

        fn enforce_all_caps(&self, text: &str) -> String {
            text.to_uppercase()
        }

        fn enforce_sentence_count(&self, text: &str, target: usize) ->
String {
            let sentences = split_sentences(text);
            if sentences.len() == target {
                text.to_string()
            } else if sentences.len() > target {
                sentences[..target].join(". ") + "."
            } else {
                // Pad with filler sentences (e.g., "More details
available upon request.")
                pad_sentences(text, target)
            }
        }
    }
```

Task 5: Validation + Retry Logic (6 hours)

```
    // In consensus flow:
```

```rust
let mut attempts = 0;
let max_retries = 1;

loop {
    let output = agent.call(&prompt).await?;
    let validation = preferences.validate_output(&output);

    if validation.valid || attempts >= max_retries {
        break output;
    }

    // Strengthen constraints for retry
    prompt = format!(
        "{}\n\n⚠ CRITICAL: Previous response violated:\n{}",
        original_prompt,
        validation.violations.iter()
            .map(|v| format!("- {}", v.description))
            .collect::<Vec<_>>()
            .join("\n")
    );
    attempts += 1;
}
```

Task 6: Unit Tests (2 hours)

```rust
#[cfg(test)]
mod tests {
    #[test]
    fn test_no_commas_enforcement() {
        let prefs = UserPreferences {
            format_constraints: FormatConstraints {
                no_commas: true,
                ..Default::default()
            },
            ..Default::default()
        };

        let input = "Hello, world! How are you, today?";
        let result = prefs.validate_output(input);
        assert!(!result.valid);
        assert_eq!(result.violations.len(), 1);
    }

    #[test]
    fn test_json_validation() {
        let prefs = UserPreferences {
            format_constraints: FormatConstraints {
                require_json: true,
                ..Default::default()
            },
            ..Default::default()
        };

        let valid = r#"{"message": "Hello world"}"#;
        let invalid = "Hello world";

        assert!(prefs.validate_output(valid).valid);
        assert!(!prefs.validate_output(invalid).valid);
    }
}
```

Success Metrics (Phase 1)

☒ 12/20 preferences implemented (60%)
☒ Validation + retry achieves 90%+ compliance
☒ <10% latency overhead for non-retry cases
☒ All unit tests pass
☒ No breaking changes to existing consensus flow

⏳ Phase 2: Advanced Features (18/20 Preferences, 90% Compliance)

Duration: 2-3 weeks **Effort**: 24 hours **Compliance Target**: 90% (18/20 preferences) **Risk Level**: MEDIUM

Scope

Add (require trajectory tracking): 13. ✅ `do_selection` (A/B/C format validation) 14. ✅ `only_begin` (state: has_asked_before) 15. ✅ `ask_many` (state: questions_in_current_turn) 16. ✅ `one_question` (state: questions_in_current_turn) 17. ✅ `first_try` (state: attempt_count) 18. ✅ `lang_ita` (translation service - LibreTranslate)

Defer to Phase 3: - `lang_multi` (complex: 5+ languages) - `joke` (complex: humor detection)

Implementation Tasks

Task 7: Trajectory Tracking Integration (8 hours) - Depends on SPEC-PPP-004 (trajectory schema) - Track turn number, questions asked, timing - Store in SQLite `trajectories` table

Task 8: State-Based Preference Enforcement (8 hours)

```
        impl UserPreferences {
            pub fn check_state_constraints(&self, trajectory:
&AgentTrajectory) -> Result<()> {
                // Check only_begin: already asked?
                if self.question_timing == QuestionTiming::OnlyBegin
                    && trajectory.turns.len() > 1
                    && !trajectory.questions_asked.is_empty()
                {
                    return Err("only_begin violated: agent already asked
questions".into());
                }

                // Check one_question: multiple questions in current turn?
                if self.question_format == QuestionFormat::OnePerTurn {
                    let current_turn_questions = trajectory.turns.last()
                        .map(|t| count_questions(&t.response))
                        .unwrap_or(0);

                    if current_turn_questions > 1 {
                        return Err("one_question violated: multiple
questions in turn".into());
                    }
                }
```

```
        Ok(())
    }
}
```

Task 9: Translation Service Integration (6 hours)

```rust
// File: codex-rs/tui/src/translation_service.rs (NEW)

pub enum TranslationBackend {
    LibreTranslate { url: String },
    DeepL { api_key: String },
    LlmNative,
}

pub async fn translate(
    text: &str,
    target_lang: &str,
    backend: &TranslationBackend,
) -> Result<String> {
    match backend {
        TranslationBackend::LibreTranslate { url } => {
            // HTTP POST to LibreTranslate API
        },
        TranslationBackend::DeepL { api_key } => {
            // HTTP POST to DeepL API
        },
        TranslationBackend::LlmNative => {
            // Meta-prompt: "Translate to {lang}: {text}"
        },
    }
}
```

Task 10: Self-Hosted LibreTranslate Setup (2 hours)

```bash
# Docker deployment
docker run -d -p 5000:5000 \
  libretranslate/libretranslate:latest

# Test
curl -X POST http://localhost:5000/translate \
  -H "Content-Type: application/json" \
  -d '{"q":"Hello","source":"en","target":"it"}'
```

Success Metrics (Phase 2)

☒ 18/20 preferences implemented (90%)
☒ Trajectory-dependent preferences validated
☒ Translation service operational (LibreTranslate)
☒ <500ms translation latency (self-hosted)

🚀 Phase 3: Full Compliance (20/20 Preferences, 100% Compliance)

Duration: 1-2 weeks **Effort**: 10 hours **Compliance Target**: 100% (20/20 preferences) **Risk Level**: MEDIUM-HIGH

Scope

Add (complex features): 19. ✅ `lang_multi` (5+ languages in single response) 20. ✅ `joke` (humor detection and injection)

Implementation Tasks

Task 11: Multi-Language Support (6 hours)

```
        async fn enforce_multilingual(
            text: &str,
            languages: &[String],
            translator: &TranslationBackend,
        ) -> Result<String> {
            // Strategy: Split response into sections, translate each to
different language
            let sentences = split_sentences(text);
            let mut result = Vec::new();

            for (i, sentence) in sentences.iter().enumerate() {
                let target_lang = &languages[i % languages.len()];
                let translated = translate(sentence, target_lang,
translator).await?;
                result.push(format!("[{}] {}", target_lang.to_uppercase(),
translated));
            }

            Ok(result.join(" "))
        }
```

Task 12: Humor Detection (4 hours)

```
        fn contains_joke(text: &str) -> bool {
            // Heuristics:
            // - Contains "haha", "lol", emoji 😂
            // - Has punchline structure (setup → punchline)
            // - Contains wordplay patterns

            let joke_markers = ["haha", "lol", "😂", "🤣", "joke", "pun"];
            joke_markers.iter().any(|m| text.to_lowercase().contains(m))

            // TODO: Improve with LLM-based joke detection
        }

        fn inject_joke(text: &str) -> String {
            // Append a programming joke if none detected
            let jokes = [
                "Why do programmers prefer dark mode? Because light attracts
bugs!",
                "There are 10 types of people: those who understand binary,
and those who don't.",
                "A SQL query walks into a bar, walks up to two tables and
asks... 'Can I join you?'",
            ];

            if contains_joke(text) {
                text.to_string()
            } else {
                format!("{}\n\nHere's a joke: {}", text,
jokes[rand::random::<usize>() % jokes.len()])
```

```
        }
    }
```

Success Metrics (Phase 3)

☒ 20/20 preferences implemented (100%)
☒ Multi-language responses validated
☒ Joke injection/detection functional
☒ Full PPP framework compliance achieved

🔧 Technical Recommendations

Recommendation 1: Use TOML for Configuration

Rationale: - Human-readable (better UX than JSON) - Native Rust support (`serde` + `toml` crates) - Supports complex nesting (enums, structs, arrays) - Widely used in Rust ecosystem (Cargo.toml)

Example:

```
[user_preferences]
interaction_mode = "no_preference"
question_detail = "concise"
expertise_level = "professional"

[user_preferences.format_constraints]
no_commas = true
require_json = false
exact_sentence_count = 3
```

Recommendation 2: Implement 3-Layer Enforcement

Layer 1: Prompt Injection (70-85% compliance, 0ms overhead)

```
fn apply_to_prompt(&self, base: &str) -> String {
    format!("{}\n\nUSER PREFERENCES:\n{}", base,
self.constraint_text())
}
```

Layer 2: Validation + Retry (90-95% compliance, +20-30% latency)

```
if !prefs.validate_output(&output).valid {
    retry_with_stronger_constraints();
}
```

Layer 3: Post-Processing Fallback (100% compliance, may degrade quality)

```
if still_invalid {
    force_transformation(&output);  // e.g., strip commas, convert
to JSON
}
```

Recommendation 3: Defer Translation to Phase 2

Rationale: - Translation adds significant complexity (service integration, latency) - Only 2/20 preferences require translation (lang_ita, lang_multi) - Can validate 90% of framework without translation - Phase 1 proves core concept, Phase 2 adds polish

Migration Path: 1. Phase 1: English-only, 12 preferences 2. Phase 2: Add LibreTranslate (self-hosted), 6 more preferences 3. Phase 3: Optimize with DeepL (optional premium), 2 final preferences

---

Recommendation 4: Prioritize Format Constraints

Reason: Format constraints (json, no_commas, capital, length) are: - Easy to implement (post-processing) - High user value (ensure output compatibility) - Deterministic (100% enforceable)

Implementation Order: 1. ✅ `no_commas` (simplest: regex replace) 2. ✅ `capital` (trivial: `.to_uppercase()`) 3. ✅ `length` (moderate: sentence counting) 4. ✅ `json` (moderate: validation + retry)

Recommendation 5: Validate Conflicts Early

Problem: Users may specify contradictory preferences - `no_ask` + `do_selection` (can't ask questions AND require A/B/C format) - `all_caps` + `require_json` (JSON requires lowercase keywords)

Solution: Pre-parse validation in config loading

```
        impl UserPreferences {
            pub fn validate(&self) -> Result<(), Vec<PreferenceConflict>> {
                let mut conflicts = Vec::new();

                if self.interaction_mode == InteractionMode::NoQuestions
                    && self.question_format != QuestionFormat::Freeform
                {
                    conflicts.push(PreferenceConflict {
                        description: "no_ask conflicts with
question_format".into(),
                        severity: ConflictSeverity::Error,
                    });
                }

                if conflicts.is_empty() { Ok(()) } else { Err(conflicts) }
            }
        }
```

📊 Success Metrics & Validation

Quantitative Metrics

Metric	Phase 1 Target	Phase 2 Target	Phase 3 Target
Preference Coverage	12/20 (60%)	18/20 (90%)	20/20 (100%)
Validation Compliance	90%+	95%+	98%+

Latency Overhead	<10% (no retry)	<10% (no retry)	<10% (no retry)
Retry Rate	<15%	<10%	<5%
Translation Latency	N/A	<500ms (self-hosted)	<300ms (DeepL)
Unit Test Coverage	80%+	90%+	95%+

Qualitative Metrics

- ☐ **User Acceptance**: 80%+ users find preferences useful (survey)
- ☐ **Agent Compliance**: Agents respect preferences without quality degradation
- ☐ **Error Messages**: Clear, actionable conflict warnings
- ☐ **Documentation**: Comprehensive guide for config.toml setup

---

🚨 Risk Mitigation

Risk 1: Agent Non-Compliance

Risk: Agents may ignore preference constraints despite prompt injection

Mitigation: - Use validation + retry (Layer 2) for critical preferences - Fall back to post-processing (Layer 3) if 2 retries fail - Log violations to identify which preferences need stronger enforcement

Acceptance: Some preferences (e.g., `joke`) may have lower compliance (~80%) - acceptable trade-off

---

Risk 2: Translation Quality Degradation

Risk: LibreTranslate may produce poor translations for technical content

Mitigation: - Offer multiple backends (LibreTranslate, DeepL, LLM-native) - Default to LLM-native for low-volume use cases (better technical accuracy) - Allow users to configure translation backend in config.toml

Fallback: If translation fails, return English with warning

Risk 3: Performance Regression

Risk: Validation + translation adds latency to agent execution

Mitigation: - Measure baseline latency before implementation - Set performance budgets (<10% overhead for 90% of requests) - Optimize hot paths (cache regex compilation, reuse translators) - Make translation async (don't block agent execution)

Monitoring: Add telemetry to track P50/P95/P99 latencies

🛠 Implementation Checklist

Phase 1 (Week 1)

- ☐ Extend `config_types.rs` with `UserPreferences` struct
- ☐ Implement `validate()` for conflict detection
- ☐ Implement `apply_to_prompt()` for prompt injection
- ☐ Create `output_formatter.rs` module
- ☐ Implement post-processing transformations (no_commas, capital, length)
- ☐ Implement JSON validation + retry logic
- ☐ Write unit tests (80%+ coverage)
- ☐ Integration test with `/speckit.plan`
- ☐ Update `config.toml.example` with preferences section
- ☐ Document in CLAUDE.md

Phase 2 (Weeks 2-4)

- ☐ Integrate with trajectory tracking (SPEC-PPP-004)
- ☐ Implement state-based preferences (only_begin, one_question, etc.)
- ☐ Deploy LibreTranslate (self-hosted Docker)
- ☐ Implement translation service integration
- ☐ Add `lang_ita` support
- ☐ Write integration tests for trajectory-dependent preferences
- ☐ Benchmark translation latency
- ☐ Update documentation

Phase 3 (Weeks 5-6)

- ☐ Implement multi-language support (`lang_multi`)
- ☐ Implement joke detection and injection
- ☐ Final compliance validation (20/20 preferences)
- ☐ Performance tuning
- ☐ User acceptance testing
- ☐ Production rollout (feature flag: `ppp.enabled = true`)

📚 References

1. **PPP Framework**: arXiv:2511.02208
2. **Feasibility Analysis**: `docs/ppp-framework-feasibility-analysis.md`
3. **SPEC-PPP-000**: Master coordination SPEC
4. **Consensus System**: `codex-rs/tui/src/chatwidget/spec_kit/consensus.rs`
5. **Config Types**: `codex-rs/core/src/config_types.rs`
6. **LibreTranslate**: https://github.com/LibreTranslate/LibreTranslate
7. **DeepL API**: https://www.deepl.com/docs-api

End of Recommendations ✅