recommendations

Implementation Recommendations: Vagueness Detection & Question Effort Classification

SPEC: SPEC-PPP-001 **Created**: 2025-11-16 **Status**: Research **Purpose**: Phased implementation plan for PPP Proactivity dimension

---

Executive Summary

This document provides a phased implementation roadmap for vagueness detection and question effort classification, supporting the PPP Framework's Proactivity dimension (R_{Proact}).

Key Recommendations: 1. **Phase 1** (Immediate): Heuristic-based detection (75-80% accuracy, \$0 cost, <5ms latency) 2. **Phase 2** (Optional): Enhanced heuristics with dependency parsing (80-85% accuracy) 3. **Phase 3** (Optional): LLM-based upgrade for >90% accuracy (\$3.75-\$10.80/year)

Decision: Start with Phase 1, upgrade only if production metrics indicate insufficient accuracy.

Table of Contents

1. Phase 1: Heuristic Foundation
2. Phase 2: Enhanced Heuristics
3. Phase 3: LLM Upgrade
4. Integration Architecture
5. Testing Strategy
6. Deployment Plan
7. Success Metrics
8. Cost-Benefit Analysis

Phase 1: Heuristic Foundation

Timeline: 2-3 days (1 engineer) **Cost**: \$0 **Target Accuracy**: 75-80% **Latency**: <5ms per prompt/question

1.1 Implementation Components

A. Vagueness Detector

File: `codex-rs/tui/src/ppp/vagueness_detector.rs`

Core Structure:

```
use regex::Regex;
use lazy_static::lazy_static;

lazy_static! {
    // Vague verb patterns
    static ref VAGUE_VERBS: Vec<&'static str> = vec![
        "implement", "add", "make", "create", "do", "build",
        "fix", "update", "change", "improve", "handle", "setup"
    ];

    // Missing context patterns (regex)
    static ref MISSING_OAUTH_VERSION: Regex =
        Regex::new(r"(?i)\bOAuth\b(?!\s*(2|1\.0))").unwrap();
    static ref MISSING_DB_TYPE: Regex =
        Regex::new(r"(?i)\bdatabase\b(?!\s*
(SQL|NoSQL|PostgreSQL|MySQL))").unwrap();
    static ref MISSING_AUTH_TYPE: Regex =
        Regex::new(r"(?i)\bauth(?:entication)?\b(?!\s*
(JWT|OAuth|SAML))").unwrap();

    // Ambiguous quantifiers
    static ref AMBIGUOUS_QUANT: Vec<&'static str> = vec![
        "some", "a few", "several", "many", "better", "good",
        "fast", "slow", "big", "small", "more", "less"
    ];
}

#[derive(Debug, Clone)]
pub struct VaguenessResult {
    pub is_vague: bool,
    pub score: f32,
    pub indicators: Vec<String>,
}

pub struct VaguenessDetector {
    threshold: f32,
}

impl Default for VaguenessDetector {
    fn default() -> Self {
        Self { threshold: 0.5 }
    }
}

impl VaguenessDetector {
    pub fn new(threshold: f32) -> Self {
        Self { threshold }
    }

    pub fn detect(&self, prompt: &str) -> VaguenessResult {
        let mut score = 0.0;
        let mut indicators = Vec::new();

        // Check vague verbs (0.2 per match, max 1 match)
```

```rust
                let lower = prompt.to_lowercase();
                for verb in VAGUE_VERBS.iter() {
                    if lower.contains(verb) {
                        score += 0.2;
                        indicators.push(format!("vague-verb:{}", verb));
                        break; // Only count first match
                    }
                }

                // Check missing context patterns (0.3 per match)
                if MISSING_OAUTH_VERSION.is_match(prompt) {
                    score += 0.3;
                    indicators.push("missing-oauth-version".to_string());
                }
                if MISSING_DB_TYPE.is_match(prompt) {
                    score += 0.3;
                    indicators.push("missing-database-type".to_string());
                }
                if MISSING_AUTH_TYPE.is_match(prompt) {
                    score += 0.3;
                    indicators.push("missing-auth-type".to_string());
                }

                // Check ambiguous quantifiers (0.1 per match, max 2)
                let quant_matches = AMBIGUOUS_QUANT.iter()
                    .filter(|q| lower.contains(*q))
                    .take(2)
                    .collect::<Vec<_>>();

                score += 0.1 * quant_matches.len() as f32;
                for quant in quant_matches {
                    indicators.push(format!("ambiguous-quantifier:{}",
quant));
                }

                // Clamp score to [0.0, 1.0]
                let final_score = score.min(1.0);
                let is_vague = final_score > self.threshold;

                VaguenessResult {
                    is_vague,
                    score: final_score,
                    indicators,
                }
            }

            pub fn is_vague(&self, prompt: &str) -> bool {
                self.detect(prompt).is_vague
            }

            pub fn vagueness_score(&self, prompt: &str) -> f32 {
                self.detect(prompt).score
            }
        }

        #[cfg(test)]
        mod tests {
            use super::*;

            #[test]
            fn test_vague_prompts() {
                let detector = VaguenessDetector::default();

                // Vague examples (should detect)
                assert!(detector.is_vague("Implement OAuth"));
                assert!(detector.is_vague("Add authentication"));
                assert!(detector.is_vague("Make it faster"));
                assert!(detector.is_vague("Fix the database"));

                // Specific examples (should NOT detect)
                assert!(!detector.is_vague("Implement OAuth2 with Google
provider using PKCE"));
                assert!(!detector.is_vague("Add JWT authentication with
HS256 signing"));
                assert!(!detector.is_vague("Reduce API latency from 200ms to
<100ms"));
                assert!(!detector.is_vague("Fix null pointer in
user_service.rs line 42"));
            }

            #[test]
            fn test_scoring() {
                let detector = VaguenessDetector::default();

                // High vagueness
                let result = detector.detect("Implement OAuth");
                assert!(result.score > 0.5); // vague-verb (0.2) + missing-
version (0.3) = 0.5

                // Low vagueness
                let result = detector.detect("Implement OAuth2 with PKCE");
                assert!(result.score < 0.3); // Only vague-verb (0.2)
            }
        }
```

Integration Points: 1. Call during trajectory logging (before storing turns) 2. Store vagueness score in trajectory metadata 3. Use in proactivity calculation

---

B. Question Effort Classifier

File: `codex-rs/tui/src/ppp/effort_classifier.rs`

Core Structure:

```
        #[derive(Debug, Clone, Copy, PartialEq, Eq, Serialize, Deserialize)]
        pub enum EffortLevel {
            Low,    // Selection, accessible context (<10 words, has
options)
            Medium, // Research, preferences (10-20 words)
            High,   // Investigation, blocking (>20 words, complex)
        }

        pub struct EffortClassifier;

        impl EffortClassifier {
            pub fn new() -> Self {
                Self
            }
            pub fn classify(&self, question: &str) -> EffortLevel {
                let word_count = question.split_whitespace().count();
                let lower = question.to_lowercase();

                // High-effort indicators (override length)
                let high_indicators = [
                    "investigate", "research", "before proceeding",
"blocking",
                    "need to decide", "architecture", "trade-off",
"strategy",
                    "should we consider", "do you want me to research"
                ];

                for indicator in &high_indicators {
                    if lower.contains(indicator) {
                        return EffortLevel::High;
                    }
                }

                // Low-effort indicators (selection questions)
                let low_indicators = [
                    "which", "choose", "select", "prefer", "option"
                ];

                let has_options = lower.contains(" or ") ||
lower.contains("option");
                let has_selection_word = low_indicators.iter()
                    .any(|ind| lower.contains(ind));

                if (has_options || has_selection_word) && word_count < 15 {
                    return EffortLevel::Low;
                }

                // Length-based fallback
                match word_count {
                    0..=10 => EffortLevel::Low,
                    11..=20 => EffortLevel::Medium,
                    _ => EffortLevel::High,
                }
            }
        }

        #[cfg(test)]
        mod tests {
            use super::*;

            #[test]
            fn test_low_effort() {
                let classifier = EffortClassifier::new();

                assert_eq!(
                    classifier.classify("Which database: PostgreSQL or
MySQL?"),
                    EffortLevel::Low
                );
                assert_eq!(
                    classifier.classify("Do you prefer tabs or spaces?"),
                    EffortLevel::Low
                );
            }

            #[test]
            fn test_medium_effort() {
                let classifier = EffortClassifier::new();

                assert_eq!(
                    classifier.classify("What authentication method should
we use?"),
                    EffortLevel::Medium
                );
                assert_eq!(
                    classifier.classify("How should we handle errors?"),
                    EffortLevel::Medium
                );
            }

            #[test]
            fn test_high_effort() {
                let classifier = EffortClassifier::new();

                assert_eq!(
                    classifier.classify("Should we investigate caching
strategies before proceeding?"),
                    EffortLevel::High
                );
                assert_eq!(
                    classifier.classify("Do you want me to research
distributed tracing solutions?"),
                    EffortLevel::High
                );
            }
        }
```

C. Proactivity Score Calculator

File: `codex-rs/tui/src/ppp/proactivity_calculator.rs`

Integration with Trajectory Logging (SPEC-PPP-004):

```
        use crate::ppp::effort_classifier::{EffortClassifier, EffortLevel};

        #[derive(Debug, Clone)]
        pub struct ProactivityScore {
            pub r_proact: f32,
            pub questions_asked: usize,
            pub low_effort: usize,
            pub medium_effort: usize,
            pub high_effort: usize,
        }

        pub struct ProactivityCalculator {
            effort_classifier: EffortClassifier,
        }

        impl ProactivityCalculator {
            pub fn new() -> Self {
                Self {
                    effort_classifier: EffortClassifier::new(),
                }
            }

            pub fn calculate(
                &self,
                trajectory_id: i64,
                db: &Connection,
            ) -> Result<ProactivityScore> {
                // Query questions from trajectory_questions table
                let mut stmt = db.prepare(
                    "SELECT question_text FROM trajectory_questions WHERE 
trajectory_id = ?"
                )?;

                let questions: Vec<String> = stmt
                    .query_map([trajectory_id], |row| row.get(0))?
                    .collect::<Result<Vec<_>, _>>()?;

                if questions.is_empty() {
                    return Ok(ProactivityScore {
                        r_proact: 0.05, // Bonus: no questions
                        questions_asked: 0,
                        low_effort: 0,
                        medium_effort: 0,
                        high_effort: 0,
                    });
                }

                // Classify each question
                let mut low = 0;
                let mut medium = 0;
                let mut high = 0;

                for question in &questions {
                    match self.effort_classifier.classify(question) {
                        EffortLevel::Low => low += 1,
                        EffortLevel::Medium => medium += 1,
                        EffortLevel::High => high += 1,
                    }
                }

                // Apply PPP formula
                let r_proact = if low == questions.len() {
                    0.05 // All low-effort
                } else {
                    -0.1 * (medium as f32) - 0.5 * (high as f32)
                };

                Ok(ProactivityScore {
                    r_proact,
                    questions_asked: questions.len(),
                    low_effort: low,
                    medium_effort: medium,
                    high_effort: high,
                })
            }
        }
```

1.2 Database Schema Extensions

Extend Trajectory Logging (SPEC-PPP-004):

```
-- Add to trajectory_questions table
ALTER TABLE trajectory_questions ADD COLUMN effort_level TEXT;
ALTER TABLE trajectory_questions ADD COLUMN classified_at TEXT;

-- Add vagueness tracking to trajectory metadata
CREATE TABLE IF NOT EXISTS trajectory_metadata (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    trajectory_id INTEGER NOT NULL,
    key TEXT NOT NULL,
    value TEXT,
    FOREIGN KEY (trajectory_id) REFERENCES trajectories(id) ON
DELETE CASCADE,
    UNIQUE(trajectory_id, key)
);

-- Store vagueness scores
-- INSERT INTO trajectory_metadata (trajectory_id, key, value)
VALUES
-- (1, 'vagueness_score', '0.65');
```

1.3 Configuration

Extend `config.toml`:

```
[ppp.proactivity]
enabled = true

# Vagueness detection threshold (0.0-1.0)
vagueness_threshold = 0.5

# Question extraction (from agent responses)
question_detection_enabled = true

# Effort classification
effort_classification_enabled = true
```

1.4 Integration Points

A. Consensus Flow Integration:

```
// In consensus.rs (run_spec_consensus_weighted)
pub async fn run_spec_consensus_weighted(
    spec_id: &str,
    stage: &str,
    artifacts: Vec<ConsensusArtifactData>,
    weights: Option<(f32, f32)>,
) -> Result<WeightedConsensusResult> {
    let db = open_trajectory_db()?;

    for artifact in &artifacts {
        // Calculate technical score (existing)
        let technical = calculate_technical_score(artifact)?;

        // NEW: Calculate proactivity score
        let trajectory_id = get_trajectory_id(&db, spec_id,
&artifact.agent_name)?;
```

```
                let proact_calc = ProactivityCalculator::new();
                let proact_score = proact_calc.calculate(trajectory_id,
&db)?;

                // NEW: Calculate personalization score (SPEC-PPP-002)
                let pers_score = calculate_r_pers(trajectory_id, &db)?;

                // Weighted consensus
                let (w_tech, w_interact) = weights.unwrap_or((0.7, 0.3));
                let interaction = proact_score.r_proact + pers_score.r_pers;
                let final_score = (w_tech * technical) + (w_interact *
interaction);
            }

            // Select best agent based on final_score
            // ...
        }
```

B. Trajectory Logging Integration:

```
        // In trajectory_logger.rs (log_turn)
        pub fn log_turn(
            db: &Connection,
            trajectory_id: i64,
            turn_number: i32,
            prompt: &str,
            response: &str,
            config: &PppConfig,
        ) -> Result<()> {
            // Store turn (existing)
            db.execute(
                "INSERT INTO trajectory_turns (trajectory_id, turn_number,
prompt, response) VALUES (?, ?, ?, ?)",
                params![trajectory_id, turn_number, prompt, response],
            )?;

            // NEW: Detect vagueness in prompt
            if config.proactivity.enabled {
                let detector =
VaguenessDetector::new(config.proactivity.vagueness_threshold);
                let vagueness = detector.detect(prompt);

                db.execute(
                    "INSERT INTO trajectory_metadata (trajectory_id, key,
value) VALUES (?, ?, ?)",
                    params![trajectory_id, "vagueness_score",
vagueness.score.to_string()],
                )?;
            }

            // NEW: Extract and classify questions from response
            if config.proactivity.question_detection_enabled {
                let questions = extract_questions(response)?;
                let classifier = EffortClassifier::new();

                for question in questions {
                    let effort = classifier.classify(&question);
                    db.execute(
                        "INSERT INTO trajectory_questions (trajectory_id,
turn_number, question_text, effort_level) VALUES (?, ?, ?, ?)",
                        params![trajectory_id, turn_number, question,
format!("{:?}", effort)],
                    )?;
                }
            }

            Ok(())
        }
```

1.5 Question Extraction

Approach: Regex-based extraction from agent responses

```
        use regex::Regex;

        lazy_static! {
            static ref QUESTION_PATTERN: Regex = Regex::new(r"([A-Z]
[^.!?]*\?)")
                .unwrap();
        }

        pub fn extract_questions(text: &str) -> Result<Vec<String>> {
            let questions: Vec<String> = QUESTION_PATTERN
                .captures_iter(text)
                .map(|cap| cap[1].trim().to_string())
                .filter(|q| q.len() > 10) // Filter out short fragments
                .collect();

            Ok(questions)
        }

        #[cfg(test)]
        mod tests {
            use super::*;

            #[test]
            fn test_question_extraction() {
                let text = "I can help with that. Which provider do you
prefer: Google or GitHub? I'll need to know before proceeding.";

                let questions = extract_questions(text).unwrap();
                assert_eq!(questions.len(), 1);
                assert_eq!(questions[0], "Which provider do you prefer:
Google or GitHub?");
            }
        }
```

1.6 Testing

Test Coverage: 1. Unit tests for vagueness detector (20 test cases) 2. Unit tests for effort classifier (30 test cases) 3. Integration tests for proactivity calculator (10 scenarios) 4. Question extraction tests (15 test cases)

Test File: `codex-rs/tui/tests/ppp_proactivity_tests.rs`

```
        #[cfg(test)]
        mod integration_tests {
            use codex_tui::ppp::*;
```

```rust
    #[test]
    fn test_no_questions_bonus() {
        // Scenario: Agent completes task without asking questions
        let calc = ProactivityCalculator::new();
        let score = calc.calculate_from_questions(&[]);

        assert_eq!(score.r_proact, 0.05); // Bonus
        assert_eq!(score.questions_asked, 0);
    }

    #[test]
    fn test_low_effort_only() {
        // Scenario: Agent asks only selection questions
        let calc = ProactivityCalculator::new();
        let questions = vec![
            "Which database: PostgreSQL or MySQL?",
            "Do you prefer tabs or spaces?",
        ];

        let score = calc.calculate_from_questions(&questions);
        assert_eq!(score.r_proact, 0.05); // All low-effort
        assert_eq!(score.low_effort, 2);
    }

    #[test]
    fn test_mixed_effort() {
        // Scenario: 1 low, 1 medium, 1 high
        let calc = ProactivityCalculator::new();
        let questions = vec![
            "Which provider: Google or GitHub?",  // Low
            "What authentication flow should we use?",  // Medium
            "Should we investigate distributed caching before
proceeding?",  // High
        ];

        let score = calc.calculate_from_questions(&questions);
        assert_eq!(score.r_proact, -0.1 * 1.0 - 0.5 * 1.0); // -0.6
        assert_eq!(score.low_effort, 1);
        assert_eq!(score.medium_effort, 1);
        assert_eq!(score.high_effort, 1);
    }
}
```

1.7 Deliverables (Phase 1)

- [ ] `codex-rs/tui/src/ppp/vagueness_detector.rs` (200 lines)
- [ ] `codex-rs/tui/src/ppp/effort_classifier.rs` (150 lines)
- [ ] `codex-rs/tui/src/ppp/proactivity_calculator.rs` (100 lines)
- [ ] `codex-rs/tui/src/ppp/question_extractor.rs` (80 lines)
- [ ] Database migrations (trajectory_metadata table)
- [ ] Configuration extensions (config.toml)
- [ ] Integration with consensus.rs and trajectory_logger.rs
- [ ] Test suite (60+ test cases)

Total Effort: 2-3 days (1 engineer)

Phase 2: Enhanced Heuristics (Optional)

Timeline: 1-2 weeks **Cost**: $0 **Target Accuracy**: 80-85% **Trigger**: If Phase 1 accuracy <80% in production (after 100+ runs)

2.1 Dependency Parsing (nlprule)

Purpose: Detect incomplete sentences and missing entities

Implementation:

```rust
use nlprule::{Tokenizer, Rules};

pub struct EnhancedVaguenessDetector {
    tokenizer: Tokenizer,
    basic_detector: VaguenessDetector,
}

impl EnhancedVaguenessDetector {
    pub fn new() -> Result<Self> {
        let tokenizer = Tokenizer::new("en")?;
        Ok(Self {
            tokenizer,
            basic_detector: VaguenessDetector::default(),
        })
    }

    pub fn detect(&self, prompt: &str) -> VaguenessResult {
        // Start with basic heuristics
        let mut result = self.basic_detector.detect(prompt);

        // Add dependency parsing checks
        let tokens = self.tokenizer.tokenize(prompt);

        // Check for missing verb
        let has_verb = tokens.iter().any(|t|
t.pos().starts_with("VB"));
        if !has_verb {
            result.score += 0.2;
            result.indicators.push("missing-verb".to_string());
        }

        // Check for missing object (noun after verb)
        let has_object = tokens.windows(2).any(|pair| {
            pair[0].pos().starts_with("VB") &&
pair[1].pos().starts_with("NN")
        });
        if has_verb && !has_object {
            result.score += 0.2;
            result.indicators.push("missing-object".to_string());
        }

        result.score = result.score.min(1.0);
        result.is_vague = result.score > 0.5;
        result
    }
}
```

Dependencies:

```toml
[dependencies]
nlprule = "0.6"
```

Trade-off: +5-10% accuracy, +10-50ms latency, +50MB model files.

2.2 Domain-Specific Patterns

Purpose: Coding task vocabulary (OAuth, API, database)

Pattern Library:

```rust
lazy_static! {
    // Technology-specific patterns
    static ref TECH_PATTERNS: HashMap<&'static str, Vec<&'static str>> = {
        let mut m = HashMap::new();

        // OAuth patterns
        m.insert("oauth", vec!["version", "provider", "flow", "scope"]);

        // Database patterns
        m.insert("database", vec!["type", "engine", "schema", "migration"]);

        // API patterns
        m.insert("api", vec!["version", "endpoint", "method", "format"]);

        // Authentication patterns
        m.insert("authentication", vec!["method", "token", "session", "provider"]);

        m
    };
}

pub fn check_tech_completeness(prompt: &str) -> Vec<String> {
    let mut missing = Vec::new();
    let lower = prompt.to_lowercase();

    for (tech, required_fields) in TECH_PATTERNS.iter() {
        if lower.contains(tech) {
            for field in required_fields {
                if !lower.contains(field) {
                    missing.push(format!("missing-{}-{}", tech, field));
                }
            }
        }
    }

    missing
}
```

2.3 Multi-Part Question Handling

Purpose: Split compound questions, classify separately

Implementation:

```
        pub fn split_compound_question(question: &str) -> Vec<String> {
            // Split on coordinating conjunctions
            let parts: Vec<String> = question
                .split(" and ")
                .flat_map(|p| p.split(" or "))
                .map(|p| p.trim().to_string())
                .filter(|p| p.len() > 5)
                .collect();

            if parts.len() > 1 {
                parts
            } else {
                vec![question.to_string()]
            }
        }

        pub fn classify_compound_effort(question: &str) -> EffortLevel {
            let parts = split_compound_question(question);
            let classifier = EffortClassifier::new();

            // Classify each part, take maximum effort
            parts.iter()
                .map(|p| classifier.classify(p))
                .max()
                .unwrap_or(EffortLevel::Medium)
        }
```

Phase 3: LLM Upgrade (Optional)

Timeline: 1 week **Cost**: $3.75-$10.80/year **Target Accuracy**: 90-95% **Trigger**: If Phase 2 accuracy <85% OR user feedback indicates poor detection

3.1 LLM-Based Vagueness Detection

Implementation:

```
        use crate::mcp::llm_client::LlmClient;

        pub struct LlmVaguenessDetector {
            llm_client: LlmClient,
            fallback: VaguenessDetector,
        }

        impl LlmVaguenessDetector {
            pub async fn detect(&self, prompt: &str) ->
Result<VaguenessResult> {
                let classification_prompt = format!(
                    r#"You are a coding task analyzer. Classify this prompt
as VAGUE or SPECIFIC.

        VAGUE prompts lack:
        - Specific versions/standards (e.g., "OAuth" without version)
        - Implementation details (provider, algorithm, flow)
        - Constraints (performance, scale, compatibility)
        SPECIFIC prompts include:
        - Versions and technologies (OAuth2, JWT, PostgreSQL)
        - Clear acceptance criteria
        - Concrete examples or constraints

        Prompt: "{}"

        Classification (VAGUE or SPECIFIC):"#,
                    prompt
                );

                match self.llm_client.call("claude-haiku",
&classification_prompt).await {
                    Ok(response) => {
                        let is_vague =
response.to_lowercase().contains("vague");
                        let score = if is_vague { 0.8 } else { 0.2 };

                        Ok(VaguenessResult {
                            is_vague,
                            score,
                            indicators: vec!["llm-
classification".to_string()],
                        })
                    }
                    Err(_) => {
                        // Fallback to heuristics
                        Ok(self.fallback.detect(prompt))
                    }
                }
            }
        }
```

3.2 Hybrid Approach (Cost Optimization)

Strategy: Heuristic first, LLM only if uncertain

```
        pub struct HybridVaguenessDetector {
            heuristic: VaguenessDetector,
            llm: LlmVaguenessDetector,
        }

        impl HybridVaguenessDetector {
            pub async fn detect(&self, prompt: &str) ->
Result<VaguenessResult> {
                // Fast path: heuristic
                let heuristic_result = self.heuristic.detect(prompt);

                // If confident (score <0.3 or >0.7), return immediately
                if heuristic_result.score < 0.3 || heuristic_result.score >
0.7 {
                    return Ok(heuristic_result);
                }

                // Uncertain (0.3-0.7): use LLM
                self.llm.detect(prompt).await
            }
        }
```

Cost Reduction: - Confident cases (70%): Free (heuristic) - Uncertain cases (30%): $0.000375 (LLM) - **Effective cost**: $0.000375 × 0.3 = **$0.0001125/prompt** - **80% cost reduction** vs pure LLM

Integration Architecture

4.1 Component Diagram

```
┌─────────────────────────────────────────────────────────────┐
│                   PPP Framework (Consensus)                  │
│                                                               │
│  ┌───────────────────────────────────────────────────────┐  │
│  │ run_spec_consensus_weighted()                          │  │
│  │                                                          │  │
│  │  Technical Score (70%)     Interaction Score (30%)     │  │
│  │         │                            │                    │  │
│  │         │                   ┌────────┴────────┐           │  │
│  │         │                   │                 │            │  │
│  │         │              R_Proact         R_Pers             │  │
│  │         │                 (SPEC-001)   (SPEC-002)         │  │
│  │         │                   │                 │            │  │
│  │         └───────────────────┴─────────────────┘            │  │
│  │                             │                                │  │
│  │                    Weighted Consensus                      │  │
│  └───────────────────────────────────────────────────────┘  │
└─────────────────────────────────────────────────────────────┘
                              │
                              ▼
┌─────────────────────────────────────────────────────────────┐
│              Trajectory Logging (SPEC-004)                   │
│                                                               │
│  ┌──────────────┐  ┌──────────────┐  ┌──────────────┐      │
│  │ Trajectories │  │    Turns     │  │  Questions   │      │
│  ├──────────────┤  ├──────────────┤  ├──────────────┤      │
│  │ spec_id      │  │ prompt       │  │ question_text│      │
│  │ agent_name   │  │ response     │  │ effort_level │◄─┐   │
│  │ run_id       │  │ turn_number  │  │ turn_number  │  │   │
│  └──────────────┘  └──────────────┘  └──────────────┘  │   │
│                                                          │    │
│  ┌──────────────────────────────────────────────────┐  │   │
│  │          Metadata (vagueness scores)              │  │   │
│  └──────────────────────────────────────────────────┘  │   │
└─────────────────────────────────────────────────────────┘    │
                              │                                   │
                              ▼                                   │
┌─────────────────────────────────────────────────────────────┤
│              Proactivity Module (NEW)                         │
│                                                                 │
│  ┌──────────────────┐  ┌──────────────────┐                  │
│  │ VaguenessDetector│  │ EffortClassifier │─────────────────┘
│  ├──────────────────┤  ├──────────────────┤
│  │ detect()         │  │ classify()       │
│  │ is_vague()       │  │   ├─ Low         │
│  │ vagueness_score()│  │   ├─ Medium      │
│  └──────────────────┘  │   └─ High        │
│                         └──────────────────┘
│
│  ┌──────────────────────────────────────┐
│   │ ProactivityCalculator                        │
│   ├──────────────────────────────────────────────┤
│   │ calculate(trajectory_id) -> R_Proact │
│   │    ├─ No questions: +0.05                    │
│   │    ├─ All low: +0.05                         │
│   │    ├─ Medium: -0.1 each                      │
│   │    └─ High: -0.5 each                        │
│   └──────────────────────────────────────────────┘
│
│
│   ┌───────────────────┐
│   │ QuestionExtractor│
│   ├───────────────────┤
│   │ extract()         │
│   └───────────────────┘
└──────────────────────────────────────────────────────────────────────────┘
```

4.2 Data Flow

Workflow (Consensus Run):

1. **Spec-Kit Command** (`/speckit.plan SPEC-ID`):
   - User triggers multi-agent consensus
   - 3 agents execute plan stage

2. **Trajectory Logging** (During Agent Execution):

```
        for turn in agent_conversation {
            // Store turn
            log_turn(db, trajectory_id, turn_num, prompt, response);

            // Detect vagueness (Phase 1)
            let vagueness = detector.detect(prompt);
            store_metadata(db, trajectory_id, "vagueness_score",
vagueness.score);

            // Extract questions (Phase 1)
            let questions = extract_questions(response);
            for question in questions {
                let effort = classifier.classify(question);
                store_question(db, trajectory_id, turn_num, question,
effort);
            }
        }
```

3. **Consensus Calculation**:

```
        for artifact in artifacts {
            // Technical score (existing)
            let technical = calculate_technical_score(artifact);

            // Proactivity score (NEW)
            let trajectory_id = get_trajectory_id(spec_id,
artifact.agent_name);
            let proact = proactivity_calc.calculate(trajectory_id, db);

            // Personalization score (SPEC-002)
            let pers = personalization_calc.calculate(trajectory_id, db);

            // Weighted consensus
            let final_score = 0.7 * technical + 0.3 * (proact.r_proact +
pers.r_pers);
        }
```

4. **Agent Selection**:
   - Select agent with highest `final_score`
   - Return artifact + detailed scoring

Testing Strategy

5.1 Unit Tests

Coverage: 80%+ on all new modules

Test Suites: 1. `vagueness_detector_tests.rs` (20 test cases) 2. `effort_classifier_tests.rs` (30 test cases) 3. `proactivity_calculator_tests.rs` (15 test cases) 4. `question_extractor_tests.rs` (15 test cases)

Total: 80+ unit tests

5.2 Integration Tests

Scenarios: 1. **No Questions Bonus**: Agent completes without asking → R_Proact = +0.05 2. **All Low-Effort**: Agent asks 3 selection questions → R_Proact = +0.05 3. **Mixed Effort**: 1 low, 2 medium, 1 high → R_Proact = -0.7 4. **Vague Prompt Handling**: Vague prompt triggers clarification questions → stored in DB 5. **Weighted Consensus**: Technical (0.85) + Interaction (-0.2) → Final (0.535)

Test File: `codex-rs/tui/tests/ppp_integration_tests.rs`

5.3 Validation Dataset

Creation: 1. Collect 100 real prompts from codex-tui usage logs 2. Manually label vague/specific (2 engineers, resolve disagreements) 3. Collect 100 questions from agent responses 4. Manually label effort level (low/medium/high)

Metrics: - Accuracy: >75% (Phase 1), >80% (Phase 2), >90% (Phase 3) - Precision: >0.75 - Recall: >0.75 - F1 Score: >0.75

Deployment Plan

6.1 Rollout Strategy

Phase 1 (Immediate): 1. Merge proactivity module (`ppp/` directory) 2. Add database migrations (trajectory_metadata table) 3. Update config.toml (add `[ppp.proactivity]` section) 4. Deploy behind feature flag (`ppp.proactivity.enabled = false` default) 5. Run validation tests

(100 prompts/questions) 6. Enable for 10% of users (A/B test) 7. Monitor metrics (accuracy, false positives, latency) 8. Enable for 100% after 1 week validation

Phase 2 (If Needed): 1. Add `nlprule` dependency 2. Implement enhanced vagueness detector 3. Run comparative validation (Phase 1 vs Phase 2) 4. Deploy if accuracy improvement >5%

Phase 3 (Optional): 1. Implement LLM-based detector with fallback 2. Deploy hybrid approach (heuristic + LLM) 3. Monitor cost ($3.75-$10.80/year budget) 4. Tune hybrid threshold to optimize cost/accuracy

---

6.2 Monitoring

Metrics: 1. **Accuracy**: Monthly validation against labeled dataset 2. **Latency**: p50, p95, p99 for detection calls 3. **Cost**: LLM API spend (Phase 3 only) 4. **User Feedback**: Escalations about poor question quality

Dashboard:

```sql
-- Vagueness distribution
SELECT
    CASE
        WHEN CAST(value AS REAL) < 0.3 THEN 'specific'
        WHEN CAST(value AS REAL) > 0.7 THEN 'vague'
        ELSE 'uncertain'
    END AS category,
    COUNT(*) AS count
FROM trajectory_metadata
WHERE key = 'vagueness_score'
GROUP BY category;

-- Effort distribution
SELECT effort_level, COUNT(*) AS count
FROM trajectory_questions
GROUP BY effort_level;

-- Proactivity score distribution
SELECT
    CASE
        WHEN r_proact > 0 THEN 'bonus'
        WHEN r_proact = 0 THEN 'neutral'
        WHEN r_proact > -0.3 THEN 'small_penalty'
        ELSE 'large_penalty'
    END AS category,
    COUNT(*) AS count
FROM (
    SELECT trajectory_id, calculate_r_proact(trajectory_id) AS
r_proact
    FROM trajectories
)
GROUP BY category;
```

Success Metrics

7.1 Phase 1 Acceptance Criteria

- [ ] Vagueness detection accuracy >75% (on 100-prompt test set)
- [ ] Effort classification accuracy >75% (on 100-question test set)
- [ ] Latency <5ms (p95) for detection calls
- [ ] Zero cost (no API calls)
- [ ] 80%+ test coverage
- [ ] Integration with trajectory logging operational
- [ ] Proactivity scoring integrated in weighted consensus
- [ ] Documentation complete (ADRs, integration guide)

7.2 Production Validation

After 100 Consensus Runns: 1. **Accuracy Validation**: - Sample 50 prompts, manually label vague/specific - Compare with detector predictions - Target: >75% agreement

2. **False Positive Analysis**:
   - Review prompts flagged as vague but actually specific
   - Target: <15% false positive rate
3. **Question Effort Validation**:
   - Sample 50 questions, manually label effort
   - Compare with classifier predictions
   - Target: >75% agreement
4. **User Feedback**:
   - Monitor escalations about poor question quality
   - Target: <5% escalation rate

Upgrade Trigger: If any metric below target after 100 runs → implement Phase 2.

Cost-Benefit Analysis

8.1 Development Cost

Phase 1: - Engineering: 2-3 days × $500/day = **$1,000-$1,500** - Testing: 1 day × $500/day = **$500** - **Total**: **$1,500-$2,000**

Phase 2 (If Needed): - Engineering: 1 week × $500/day = **$2,500** - **Total**: **$2,500**

Phase 3 (Optional): - Engineering: 1 week × $500/day = **$2,500** - Ongoing LLM API: **$3.75-$10.80/year** - **Total**: **$2,500 + $10.80/year**

8.2 Benefits

User Experience: - Fewer wasted iterations (vague prompts caught early) - Better agent selection (proactivity penalized for high-effort questions) - Improved transparency (users see why agent asked questions)

Quantified Impact: - **10-20% reduction in wasted consensus runs** (vague prompts clarified upfront) - **15-25% improvement in agent selection accuracy** (interaction scoring) - **Estimated savings**: 50 runs/year × 10% × $2.70/run = **$13.50/year**

ROI: Phase 1 pays for itself in ~150 years (not financially justified).

True Value: User satisfaction, alignment with PPP research, foundation for future improvements.

Conclusion

9.1 Recommended Path

Immediate (Phase 1): - Implement heuristic vagueness detection and effort classification - Integrate with trajectory logging (SPEC-PPP-004) - Deploy behind feature flag with A/B testing - Target: 75-80% accuracy, <5ms latency, $0 cost

Conditional (Phase 2): - Deploy only if Phase 1 accuracy <80% after 100 runs - Add dependency parsing and domain-specific patterns - Target: 80-85% accuracy

Optional (Phase 3): - Deploy only if Phase 2 accuracy <85% OR user feedback poor - Hybrid LLM approach (heuristic + LLM fallback) - Target: 90-95% accuracy, $3.75-$10.80/year

---

9.2 Next Steps

1. Create PoC code (`evidence/vagueness_detector_poc.rs`)
2. Create ADRs documenting key decisions:
    - ADR-001-001: Heuristic vs LLM-based vagueness detection
    - ADR-001-002: Question effort classification strategy
    - ADR-001-003: Integration with /reasoning command
3. Validate findings with master SPEC (SPEC-PPP-000)
4. Begin Phase 1 implementation (codex-rs/tui/src/ppp/)

Status: Complete **Next Deliverable**: evidence/vagueness_detector_poc.rs