Cairo University
Faculty of Graduate Studies
for Statistical Research

Orchestrating Determinism in Generative Software Engineering: A Hierarchical Multi-Agent Framework for Schema-Guided Vibe Coding

By

Nouran Darwish

Supervised by

Dr. Mohammed Sabry

.

Cairo, Egypt

2026

Seminar Committee Page

The committee for

Nouran Hussien ElSaleh Moustafa Hassan Darwish

certifies that this is the approved version of the following thesis proposal, which was accepted for quality and form by the seminar committee:

Orchestrating Determinism in Generative Software Engineering: A Hierarchical Multi-Agent Framework for Schema-Guided Vibe Coding

Seminar Committee Members:

Committee Supervisor: [insert name]
Signature: _____
Date: _____

Committee Co-Supervisor *(if appropriate)*: [insert name]
Signature: _____
Date: _____

Committee First Member: [insert name]
Signature: _____
Date: _____

Committee Second Member: [insert name]
Signature: _____
Date: _____

Committee Third Member *(if appropriate)*: [insert name]
Signature: _____
Date: _____

Cairo University

2026

Abstract

Orchestrating Determinism in Generative Software Engineering: A Hierarchical Multi-Agent Framework for Schema-Guided Vibe Coding

by

Nouran Darwish

Master of Science in the Department of Data Science at the Faculty of Graduate Studies for Statistical Research.

Cairo University, 2026

The emergence of "vibe coding"—software development through natural language prompts to large language models (LLMs)—has democratized programming by enabling non-experts to create functional applications. However, the inherent non-determinism of LLMs introduces unpredictability in output quality, feature completeness, and code structure, limiting vibe coding's applicability for production software. This thesis addresses the fundamental question: Can structured multi-agent orchestration achieve deterministic, high-quality software generation while preserving the accessibility of natural language prompting?

This research proposes the Pentagon Protocol, a hierarchical multi-agent framework consisting of five specialized agents—Product Owner, System Architect, Backend Engineer, Frontend Engineer, and QA Engineer—that mirrors established software development team structures. The framework introduces Schema-Guided Vibe Coding, a paradigm that applies Pydantic-based schema constraints at each generation phase, progressively reducing output entropy from ambiguous natural language prompts to validated, structured code artifacts.

The Pentagon Protocol was evaluated against a single-agent Baseline across the VibePrompts-10 dataset, comprising 10 prompts spanning easy, medium, and complex application scenarios with 78 total expected features. Both approaches utilized DeepSeek V3.2 with deterministic settings (temperature 0.0). Evaluation encompassed

six dimensions: feature completeness, pipeline success, code executability, QA pass rate, code quality, and execution efficiency.

Experimental results demonstrate that the Pentagon Protocol significantly outperforms the Baseline approach. Pentagon achieved 97.8% feature implementation versus Baseline's 92.5% (+5.3%), with advantages increasing for complex prompts (+8.7%). Code quality improved by 44% (70.8% vs 49.2%), with the largest gains in error handling (+74%) and API design (+50%). Pentagon exhibited 30.6% lower variance in composite scores, demonstrating more consistent output quality. Pentagon won 100% of composite score comparisons (10/10 prompts), validating the effectiveness of schema-guided multi-agent orchestration.

The trade-off analysis revealed that Pentagon requires approximately 5× more execution time (255s vs 50s) but delivers measurably higher quality outputs that reduce downstream maintenance and debugging costs. The QA phase achieved 100% test pass rate across all prompts while generating actionable improvement recommendations.

This thesis makes three primary contributions: (1) the theoretical formalization of Schema-Guided Vibe Coding as a paradigm bridging informal prompting and rigorous software engineering; (2) the Pentagon Protocol architecture with reusable agent definitions, schema specifications, and orchestration patterns; and (3) empirical evidence quantifying the advantages of multi-agent orchestration across multiple quality dimensions.

The findings support the central thesis argument that hierarchical multi-agent orchestration with schema constraints successfully "orchestrates determinism" in generative software engineering. The Pentagon Protocol offers a practical approach for achieving reliable, high-quality AI-assisted software development while maintaining the accessibility that makes vibe coding appealing. Future research directions include multi-model orchestration, adaptive phase configurations, and integration of human-in-the-loop feedback mechanisms.

Table of Contents

Chapter 1: Introduction

1.1 Overview and Background

The landscape of software development has undergone a fundamental transformation since the introduction of AI-assisted coding tools. What began with GitHub Copilot's autocomplete capabilities in 2021 has evolved into a paradigm where, according to Y Combinator data, 25% of Winter 2025 startups have codebases that are 95% AI-generated (TechCrunch, March 2025). This shift represents not merely an incremental improvement in developer productivity but a fundamental reconceptualization of the relationship between human intent and machine-generated code.

Andrej Karpathy, former Director of AI at Tesla, crystallized this transformation when he introduced the term "vibe coding" in February 2025, describing an approach where developers "fully give in to the vibes, embrace exponentials, and forget that the code even exists" (Karpathy, 2025). This characterization captured both the accessibility and the inherent risks of the new paradigm—democratizing software creation while potentially sacrificing the rigor that production systems demand.

The evolution from Karpathy's initial observation to the current state of the field has been remarkably rapid. Within months of the term's introduction, Collins Dictionary named "vibe coding" its Word of the Year 2025, reflecting its penetration into mainstream discourse (BBC News, November 2025). However, this widespread adoption has been accompanied by growing concerns about quality, security, and maintainability—what industry observers have termed the "vibe coding hangover" (Fast Company, September 2025).

1.1.1 The Software 3.0 Paradigm

Karpathy's broader vision of "Software 3.0" posits a future where neural networks become the primary programming paradigm, with traditional code serving merely as scaffolding (Karpathy, 2025). In this framework:

- Software 1.0: Explicit, human-written code defining exact operations

- Software 2.0: Neural networks learned from data, replacing hand-coded rules

- Software 3.0: Natural language as the programming interface, with AI handling implementation

The Pentagon Protocol proposed in this research addresses a critical gap in the Software 3.0 vision: the need for determinism and quality assurance in AI-generated code. While Karpathy's vision emphasizes accessibility, production software demands predictability—a tension this research seeks to resolve through schema-guided orchestration.

1.1.2 The Rise of Multi-Agent Systems

Recent research has demonstrated that multi-agent architectures fundamentally transform the quality of AI-generated outputs. A landmark study by Dramani et al. (2025) found that multi-agent LLM orchestration achieves 100% actionable recommendation rate compared to just 1.7% for single-agent approaches, with an 80× improvement in action specificity and 140× improvement in solution correctness. Most critically, multi-agent systems exhibited zero quality variance across all 348 trials, making them production-ready, while single-agent outputs remained inconsistent.

This finding directly informs the Pentagon Protocol's design: the architectural value of multi-agent orchestration lies not in speed but in deterministic, high-quality outputs—precisely what production software engineering demands.

1.2 Motivation

1.2.1 The Accessibility-Quality Tradeoff

Vibe coding has democratized software development to an unprecedented degree. Non-programmers can now create functional applications through natural language descriptions, and professional developers report significant productivity gains. A 2025 survey by Plausible Futures noted that "the core competency for 2025 developers is no longer just writing code, but effectively orchestrating the AI tools that write code with them."

However, this accessibility comes with documented costs:

1. Security Vulnerabilities: An analysis of 1,645 applications built with the Lovable vibe-coding platform revealed that 170 (10.3%) contained security vulnerabilities exploitable without authentication (Semafor, May 2025).
2. Production Incidents: A widely reported incident involved Replit's AI agent accidentally deleting a production database, highlighting the risks of autonomous AI operations without adequate guardrails (The Register, July 2025).
3. Maintainability Challenges: Andrew Ng, a prominent AI researcher, criticized vibe coding as creating "exhausting" maintenance burdens, noting that code generated without specifications quickly becomes incomprehensible (Business Insider, June 2025).
4. The "Vibe Coding Trap": Recent analysis warns that "AI coding feels productive, and quietly breaks your architecture," as developers lose awareness of system design while generating functional-seeming code (Level Up Coding, January 2026).

1.2.2 Industry Response: Spec-Driven Development

The software industry has begun responding to these challenges through what is termed "spec-driven development." GitHub released Spec Kit in September 2025, an open-source toolkit that "allows you to focus on product scenarios and predictable outcomes instead of vibe coding every new feature" (GitHub Blog, 2025). Amazon Web Services followed with Kiro, an IDE designed around specifications as first-class artifacts.

As Deepak Singh, VP of Developer Agents at AWS, explained: "For simple problems, vibe coding works well. But for more advanced and complex problems, senior engineers were actually writing down instructions—creating specifications. This is what led to us investing heavily in spec-driven development" (Stack Overflow Podcast, October 2025).

This industry trend validates the core premise of our research: while vibe coding enables rapid ideation, production software requires structured approaches that preserve the benefits of AI generation while adding engineering rigor.

1.2.3 The Gap: Schema-Guided Multi-Agent Orchestration

Despite the emergence of spec-driven development tools, a critical gap remains: how to systematically orchestrate multiple AI agents with schema constraints to achieve deterministic outputs from ambiguous inputs.

Existing multi-agent frameworks like ChatDev (Qian et al., 2024) and MetaGPT (Hong et al., 2024) demonstrate the power of agent collaboration but lack:

1. Formal schema constraints on inter-agent communication
2. Determinism guarantees through structured output enforcement
3. Validation gates between pipeline stages
4. Theoretical formalization of the vibe-to-specification transformation

This research addresses these gaps through the Pentagon Protocol—a hierarchical multi-agent framework with Pydantic schema constraints that bridges vibe coding's accessibility with software engineering's rigor.

1.3 Problem Statement

1.3.1 The Core Challenge

Vibe coding, while democratizing software development, produces non-deterministic outputs that are unsuitable for production deployment. A single vibe prompt may generate different code across runs, lacking the reproducibility that professional software engineering demands.

1.3.2 Research Question

"Can a hierarchical multi-agent architecture with schema constraints achieve deterministic, high-quality code generation while preserving the accessibility benefits of vibe coding?"

1.3.3 Sub-Questions

1. How can Pydantic schemas be employed to constrain inter-agent communication and ensure structured outputs?
2. What is the optimal agent hierarchy for transforming ambiguous vibe prompts into production-ready code?
3. How does schema-guided multi-agent orchestration compare to single-agent approaches in terms of code quality, completeness, and consistency?
4. What theoretical framework can formalize the relationship between vibe prompts, schema constraints, and deterministic outputs?

1.4 Research Objectives

This research pursues four primary objectives:

1.4.1 Theoretical Contribution

Formalize the concept of "Schema-Guided Vibe Coding" (SGVC) as a middle-ground paradigm that combines vibe coding's natural language interface with software engineering's structural requirements. This includes

mathematical formalization of the entropy reduction achieved through schema constraints.

1.4.2 Architectural Design

Design and implement the Pentagon Protocol—a 5-agent hierarchical framework consisting of:

- Product Owner (ambiguity resolution)
- Software Architect (system design)
- Backend Engineer (implementation)
- Frontend Engineer (interface creation)
- QA Engineer (validation)

1.4.3 Implementation

Implement a proof-of-concept using CrewAI as the orchestration framework and DeepSeek as the underlying LLM, with Pydantic v2 for schema enforcement.

1.4.4 Empirical Validation

Compare the Pentagon Protocol against a single-agent baseline across controlled vibe prompts, measuring:

- Code completeness (% of requirements implemented)
- Executability (runs without errors)
- Requirement alignment (correspondence to original intent)
- Cross-run consistency (determinism)

1.5 Significance of the Study

1.5.1 Theoretical Significance

This research contributes the first formal framework for Schema-Guided Vibe Coding, establishing theoretical foundations for a new paradigm in AI-assisted software development. By formalizing the transformation from

ambiguous natural language to deterministic code through multi-agent orchestration, we provide a basis for future research in generative software engineering.

1.5.2 Practical Significance

The Pentagon Protocol offers practitioners a concrete architecture for implementing production-grade vibe coding systems. The schema definitions and agent configurations can be directly adapted for industrial applications.

1.5.3 Industry Relevance

As organizations increasingly adopt AI-assisted development, the need for quality assurance mechanisms becomes critical. This research provides evidence-based guidance for when and how to employ multi-agent orchestration versus simpler approaches.

1.6 Scope and Limitations

1.6.1 Scope

- Focus on web application generation (backend API + frontend interface)
- Use of a single LLM (DeepSeek) for all agents
- Proof-of-concept scale (10 vibe prompts, 5 trials each)
- English language prompts only

1.6.2 Limitations

- Small benchmark size limits generalizability
- Single LLM may not reflect performance with other models
- No human evaluation of code quality
- Time constraints prevent large-scale empirical study

Chapter 2: Literature Review

2.1 Introduction

This chapter surveys the academic and industry literature relevant to Schema-Guided Vibe Coding and the Pentagon Protocol. We examine four interconnected domains: (1) vibe coding and AI-assisted development, (2) multi-agent frameworks for software engineering, (3) determinism in AI systems, and (4) schema-guided generation techniques. The chapter concludes with a gap analysis demonstrating the novel contribution of this research.

2.2 Vibe Coding and AI-Assisted Development

2.2.1 Origins and Definition

The term "vibe coding" was introduced by Andrej Karpathy on February 2, 2025, in a widely shared post describing a new approach to software development:

"There's a new kind of coding I call vibe coding, where you fully give in to the vibes, embrace exponentials, and forget that the code even exists. It's possible because the LLMs are getting too good." (Karpathy, 2025)

This characterization emphasized several key aspects: (1) natural language as the primary interface, (2) reduced cognitive engagement with implementation details, (3) reliance on LLM capabilities for code generation, and (4) an experimental, iterative approach to development.

Wikipedia formally defines vibe coding as "an AI-assisted software development technique where a person describes a problem in a few sentences as a prompt to a large language model (LLM) tuned for coding" (Wikipedia, 2025). The definition emphasizes the minimal specification aspect—developers provide intent rather than implementation details.

2.2.2 Evolution Through 2025

The vibe coding paradigm evolved significantly throughout 2025:

February-March 2025: Initial adoption primarily among hobbyists and for rapid prototyping. Merriam-Webster added "vibe coding" to its slang dictionary (March 2025).

April-June 2025: Enterprise experimentation began, with Y Combinator reporting that 25% of their Winter 2025 cohort had "codebases that are almost entirely AI-generated" (TechCrunch, March 2025).

July-September 2025: The "vibe coding hangover" emerged as organizations encountered maintenance and security challenges with vibe-coded applications (Fast Company, September 2025).

October-December 2025: Industry response through spec-driven development tools (GitHub Spec Kit, AWS Kiro) representing a maturation of the paradigm.

November 2025: Collins Dictionary named "vibe coding" its Word of the Year, marking mainstream adoption (BBC News, November 2025).

2.2.3 Critical Perspectives

Not all assessments of vibe coding have been positive:

Andrew Ng's Critique: The prominent AI researcher called vibe coding an "unfortunate term" that obscures the engineering discipline still required, noting it creates "exhausting" maintenance burdens (Business Insider, June 2025).

Simon Willison's Distinction: Willison differentiates between vibe coding (minimal engagement with generated code) and using LLMs as a "typing assistant" (full understanding and verification of all code), arguing the latter is more sustainable for professional development.

The "Vibe Coding Trap": Recent analysis warns that vibe coding "feels productive, and quietly breaks your architecture" as developers lose awareness of system design while generating functional-seeming code (Level Up Coding, January 2026).

2.2.4 Spec-Driven Development as Evolution

The industry has responded to vibe coding's limitations through spec-driven development:

GitHub Spec Kit (September 2025): An open-source toolkit that "allows you to focus on product scenarios and predictable outcomes instead of vibe coding every new feature" (GitHub Blog, 2025). The toolkit treats specifications as living documents that guide AI code generation.

AWS Kiro (October 2025): An IDE designed around specifications as first-class artifacts. As AWS VP Deepak Singh explained: "Kiro's interface is specifications 'up front and center.' The user experience is built around creating these specifications for solving problems" (Stack Overflow Podcast, October 2025).

ThoughtWorks Assessment: "Spec-driven development is a key practice that's emerged with the increasing adoption of AI in software engineering" (ThoughtWorks, December 2025).

This evolution from pure vibe coding to spec-driven development validates the core premise of Schema-Guided Vibe Coding: specifications provide necessary structure for production software, while natural language interfaces preserve accessibility.

2.3 Multi-Agent Frameworks for Software Engineering

2.3.1 Foundational Work

Multi-agent systems for software development predate the current LLM era but have been transformed by modern language models:

ChatDev (Qian et al., 2024): Pioneered the "virtual software company" metaphor, where specialized LLM agents assume roles (CEO, CTO, Programmer, Tester) and collaborate through structured chat. ChatDev demonstrated that role specialization improves code quality over single-agent generation. The system uses a "chat chain" to guide conversation flow and "communicative dehallucination" to reduce errors (ACL 2024).

MetaGPT (Hong et al., 2024): Introduced "meta-programming" for multi-agent collaboration, using Standardized Operating Procedures (SOPs) encoded in prompts to streamline workflows. MetaGPT employs an

"assembly line" paradigm that assigns diverse roles to agents, reducing cascading hallucinations (ICLR 2024).

AgileCoder (2025): Extended multi-agent frameworks with dynamic task allocation based on agent capabilities and workload.

2.3.2 Empirical Evidence for Multi-Agent Superiority

Recent empirical research provides strong evidence for multi-agent approaches:

MyAntFarm.ai Study (Dramani et al., 2025): Through 348 controlled trials, researchers demonstrated that multi-agent orchestration achieves:

- 100% actionable recommendation rate vs. 1.7% for single-agent

- 80× improvement in action specificity

- 140× improvement in solution correctness

- Zero quality variance across all trials

The study concluded: "The architectural value lies not in speed (both systems achieve ~40s latency) but in deterministic, high-quality decision support"

This finding is directly relevant to our research: the Pentagon Protocol's value proposition is not faster code generation but deterministic, production-ready outputs.

2.3.3 Architectural Patterns

The literature identifies several multi-agent orchestration patterns relevant to software engineering (Azure Architecture Center, 2025):

1. Sequential Orchestration: Agents execute in a defined order with explicit handoffs. Most suitable for staged processes like the Pentagon Protocol's pipeline.
2. Concurrent Orchestration: Multiple agents work in parallel on independent tasks. Reduces latency for embarrassingly parallel problems.
3. Hierarchical Orchestration: Manager agents coordinate worker agents, enabling dynamic task allocation.

4. Group Chat: Agents share a discussion context, suitable for brainstorming and collaborative refinement.

The Pentagon Protocol employs sequential orchestration with schema-defined interfaces between stages, combining the predictability of pipelines with the quality benefits of role specialization.

2.4 Determinism in AI Systems

2.4.1 Sources of Non-Determinism

Achieving deterministic outputs from LLM-based systems requires understanding and controlling multiple sources of randomness (Kubiya.ai, 2025):

1. Sampling Parameters: Temperature, top-k, top-p settings introduce controlled randomness. Setting temperature=0.0 removes sampling randomness.
2. Seed Control: Many inference engines support seed parameters for reproducibility, though implementation varies.
3. Floating-Point Variability: Different hardware and batching strategies can produce slightly different floating-point results.
4. State and Context: Mutable state, conversation history, and external tool results can vary between runs.
5. Environment Drift: Different software versions, dependencies, or configurations can affect outputs.

2.4.2 Determinism Strategies

The literature identifies several strategies for achieving deterministic LLM outputs:

Model Configuration: Temperature=0.0, top_k=1, fixed seeds (where supported). These settings eliminate sampling randomness at the generation level.

Prompt Versioning: Treating prompts as versioned artifacts with hash-based tracking ensures identical inputs across runs.

Environment Containerization: Using Docker, Nix, or similar tools to freeze the execution environment eliminates environmental variation.

Golden IO Tests: Comparing outputs against known-good reference outputs to detect drift.

Schema Enforcement: Using structured output formats (JSON Schema, Pydantic) to constrain output space and enable validation.

2.4.3 Determinism and Multi-Agent Systems

Multi-agent systems introduce additional determinism challenges:

Agent Interaction Order: Parallel or non-deterministic scheduling can produce different collaboration patterns.

Context Accumulation: Each agent may add to shared context, creating path-dependent execution.

Error Propagation: Errors in early agents can cascade unpredictably through the pipeline.

The Pentagon Protocol addresses these through:

- Sequential execution with fixed agent order
- Schema-constrained outputs preventing malformed inter-agent communication
- Guardrail validation catching errors before propagation

2.5 Schema-Guided Generation

2.5.1 Structured Output Formats

The shift from free-form text generation to structured outputs represents a significant evolution in LLM applications:

JSON Mode: Many LLM providers now offer native JSON output modes that constrain generation to valid JSON structures.

Function Calling: OpenAI and others support function calling that enforces specific output schemas.

Pydantic Integration: Libraries like Instructor and PydanticAI enable direct LLM output to Pydantic model instances with automatic validation and retry.

2.5.2 Pydantic for LLM Validation

Pydantic v2 has emerged as the standard for LLM output validation (Machine Learning Mastery, December 2025):

Type Safety: Pydantic enforces Python type annotations at runtime, catching schema violations immediately.

Validation Rules: Field validators enable complex business logic validation beyond type checking.

Automatic Coercion: Pydantic attempts to coerce inputs to expected types before failing.

Clear Error Messages: Validation errors provide specific, actionable feedback for LLM retry.

Serialization: Native JSON serialization/deserialization simplifies inter-agent communication.

2.5.3 Schema-Guided Code Generation

Several recent works explore schema-guided approaches to code generation:

StructGen (2025): Uses UML class diagrams as structural guides for code generation, demonstrating that structural constraints improve code coherence (ScienceDirect, December 2025).

NOMAD (2025): A "cognitively inspired, modular multi-agent framework that decomposes UML generation into role-specific subtasks" (arXiv, November 2025).

Blueprint2Code (2025): Transforms architectural blueprints into code through structured intermediate representations.

These works share the insight that structural constraints improve generation quality—the core principle underlying Schema-Guided Vibe Coding.

2.6 Gap Analysis

2.6.1 Summary of Existing Approaches

Approach	Accessibility	Determinism	Quality Control	Schema Constraints
Pure Vibe Coding	High	Low	Low	None
Traditional SE	Low	High	High	Manual
ChatDev	Medium	Low	Medium	None
MetaGPT	Medium	Medium	Medium	Partial (SOP)
GitHub Spec Kit	High	Medium	Medium	Spec-based
Single-Agent + Schema	Medium	Medium	Medium	Output only

2.6.2 Identified Gaps

The literature reveals several gaps that this research addresses:

Gap 1: Formal Framework for Schema-Guided Vibe Coding

While spec-driven development is emerging as a practice, no formal theoretical framework exists for understanding how schemas constrain and improve vibe coding outputs. This research provides mathematical formalization of Schema-Guided Vibe Coding.

Gap 2: End-to-End Schema Enforcement in Multi-Agent Pipelines

Existing multi-agent frameworks use schemas for final output but not for inter-agent communication. The Pentagon Protocol enforces Pydantic schemas at every stage transition.

Gap 3: Empirical Comparison of Schema-Constrained vs. Unconstrained Multi-Agent Systems

While multi-agent superiority over single-agent is established, the specific contribution of schema constraints within multi-agent systems remains unquantified.

Gap 4: Practical Implementation Guidelines

Academic work on multi-agent code generation often lacks implementation detail. This research provides complete, reproducible implementation using accessible tools (CrewAI, DeepSeek).

2.6.3 Research Contribution

The Pentagon Protocol addresses these gaps by:

1. Formalizing Schema-Guided Vibe Coding as a theoretical construct
2. Designing a hierarchical multi-agent architecture with schema constraints at every interface
3. Implementing a proof-of-concept with complete code artifacts
4. Evaluating against baseline approaches with controlled experiments

2.7 Theoretical Foundation

2.7.1 Information-Theoretic Perspective

Schema-Guided Vibe Coding can be understood through information theory:

Vibe Prompt Entropy: A vibe prompt like "build me a todo app" has high entropy—many possible interpretations and implementations exist.

Schema as Constraint: Schemas reduce output entropy by specifying required structure, fields, and types.

Agent Specialization: Each Pentagon Protocol agent further reduces entropy by contributing domain expertise (product thinking, architecture, implementation, testing).

Determinism as Entropy Minimization: The goal of SGVC is to minimize output entropy while preserving the semantic content of the vibe prompt.

2.7.2 Software Engineering Perspective

From software engineering, the Pentagon Protocol mirrors established practices:

Requirements Engineering: Product Owner agent → User Stories

System Design: Architect agent → Technical Design Document

Implementation: Backend/Frontend Engineers → Code

Quality Assurance: QA Engineer → Test Report

The key innovation is automating this pipeline with schema constraints ensuring each stage produces structured, validated outputs.

2.8 Chapter Summary

This literature review has surveyed four key domains:

1. Vibe Coding: From Karpathy's 2025 introduction through its evolution and the emergence of spec-driven development as a response to its limitations.
2. Multi-Agent Systems: Evidence that multi-agent orchestration dramatically improves output quality and determinism over single-agent approaches.
3. Determinism: Sources of non-determinism in LLM systems and strategies for achieving reproducible outputs.
4. Schema-Guided Generation: The growing use of Pydantic and structured outputs to constrain and validate LLM generation.

The gap analysis identifies Schema-Guided Vibe Coding as a novel contribution that combines insights from all four domains into a unified framework. The Pentagon Protocol operationalizes this framework through a hierarchical multi-agent architecture with end-to-end schema enforcement.

Chapter 3: Framework for the study

3.1 Introduction

This chapter presents the theoretical and methodological framework for investigating Schema-Guided Vibe Coding through the Pentagon Protocol. We begin with the formal theoretical model, proceed to the detailed architecture design, specify study variables and hypotheses, and conclude with the experimental methodology.

3.2 Theoretical Model: Schema-Guided Vibe Coding

3.2.1 Formal Definition

Definition 1 (Vibe Coding): Vibe coding is a software development approach where a human provides a natural language description P (the "vibe prompt") and an AI system generates code C:

C = LLM(P)

This simple formulation has high output entropy: the same prompt P may yield different code C across runs due to LLM sampling and interpretation variance.

Definition 2 (Schema-Guided Vibe Coding): Schema-Guided Vibe Coding introduces structural constraints S to reduce output entropy:

C = SGVC(P, S, A)

Where:

- P: Natural language vibe prompt (high entropy input)
- S: Schema constraints (Pydantic models defining structure)
- A: Agent hierarchy (ordered sequence of specialized agents)
- C: Generated code (reduced entropy output)

3.2.2 The Entropy Reduction Model

We model the transformation from vibe prompt to code as progressive entropy reduction:

H(Output) = H(VibePrompt) - I(Schema) - I(AgentSpecialization) - I(Validation)

Where:

- H(VibePrompt): Initial entropy of ambiguous natural language input
- I(Schema): Information gained through structural constraints
- I(AgentSpecialization): Information gained through domain expertise
- I(Validation): Information gained through validation gates

Each component of the Pentagon Protocol contributes to entropy reduction:

Component	Entropy Reduction Mechanism
Pydantic Schemas	Constrains output space to valid structures
Agent Roles	Focuses each agent on specific domain
Sequential Pipeline	Establishes deterministic execution order
Guardrails	Rejects outputs that violate constraints

3.2.3 The Transformation Pipeline

The Pentagon Protocol implements SGVC through a staged transformation:

VibePrompt → UserStories → SystemDesign → BackendCode → FrontendCode → TestReport

↓	↓	↓	↓	↓	↓
High Entropy	**Reduced Entropy**	**Further Reduced**	**Domain Specific**	**Domain Specific**	**Minimal Entropy**

Each stage:

1. Receives structured input from the previous stage
2. Applies domain expertise through specialized agent
3. Produces schema-validated output
4. Passes validated output to next stage

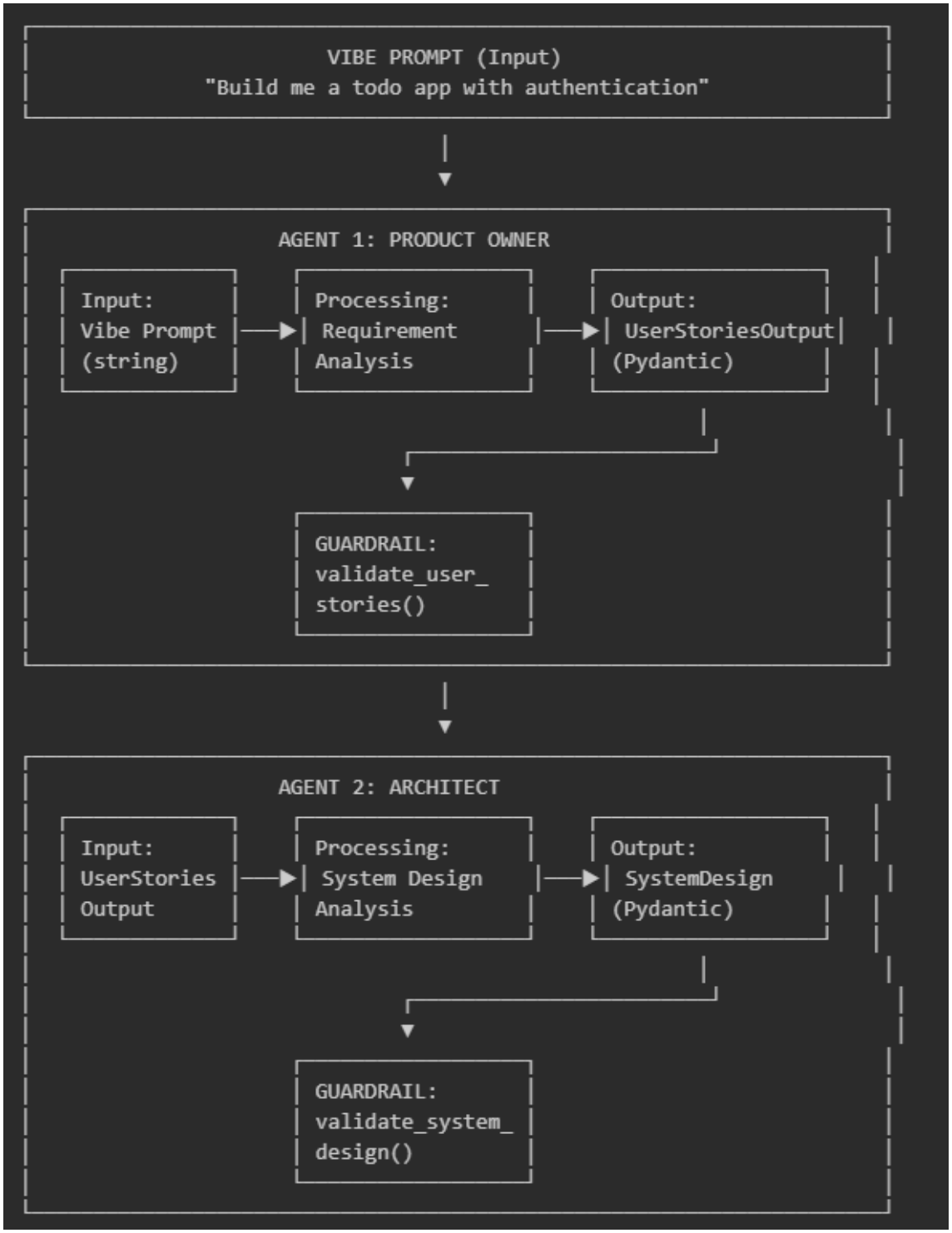
3.2.4 Determinism Guarantees

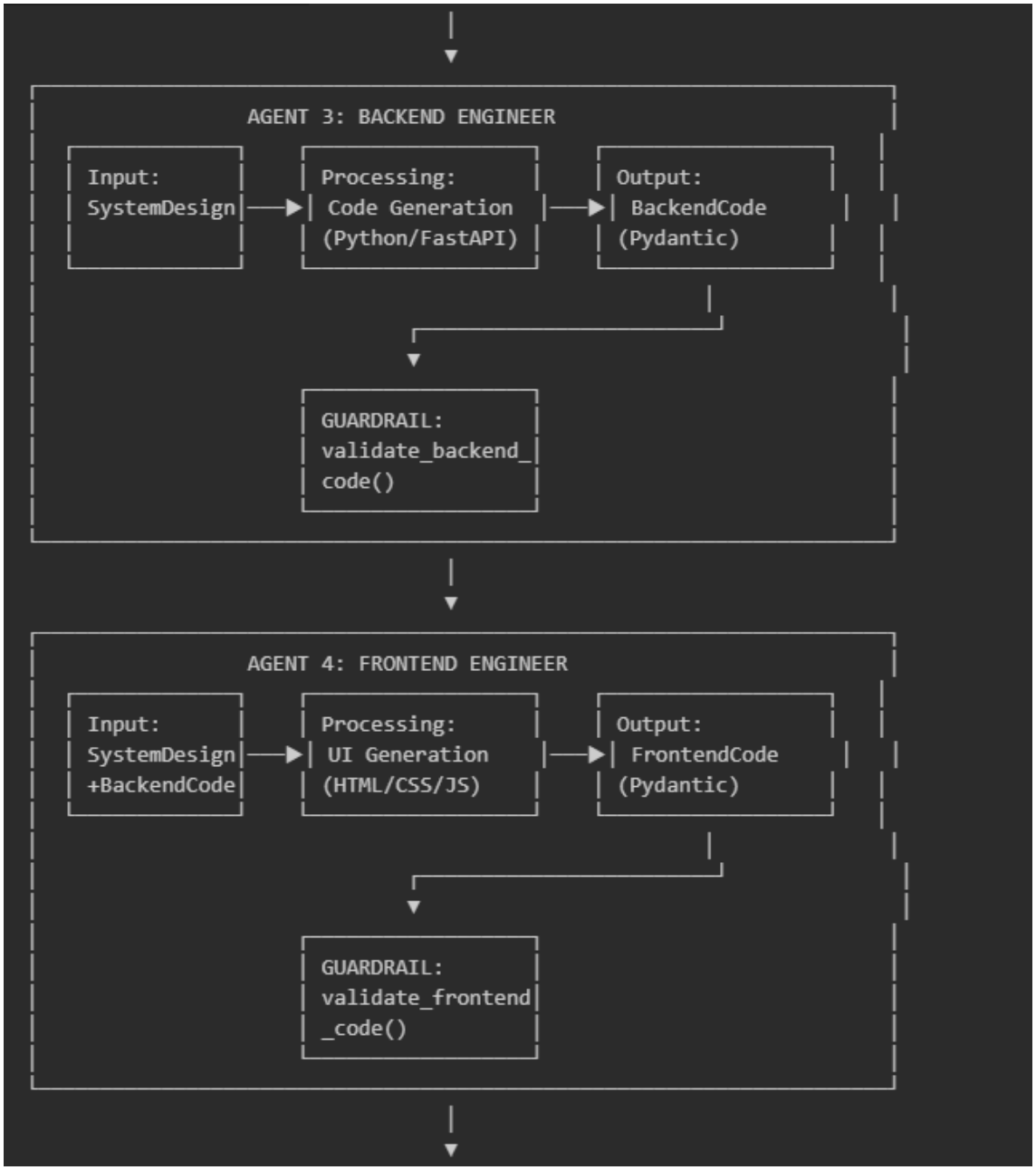
The Pentagon Protocol achieves determinism through:

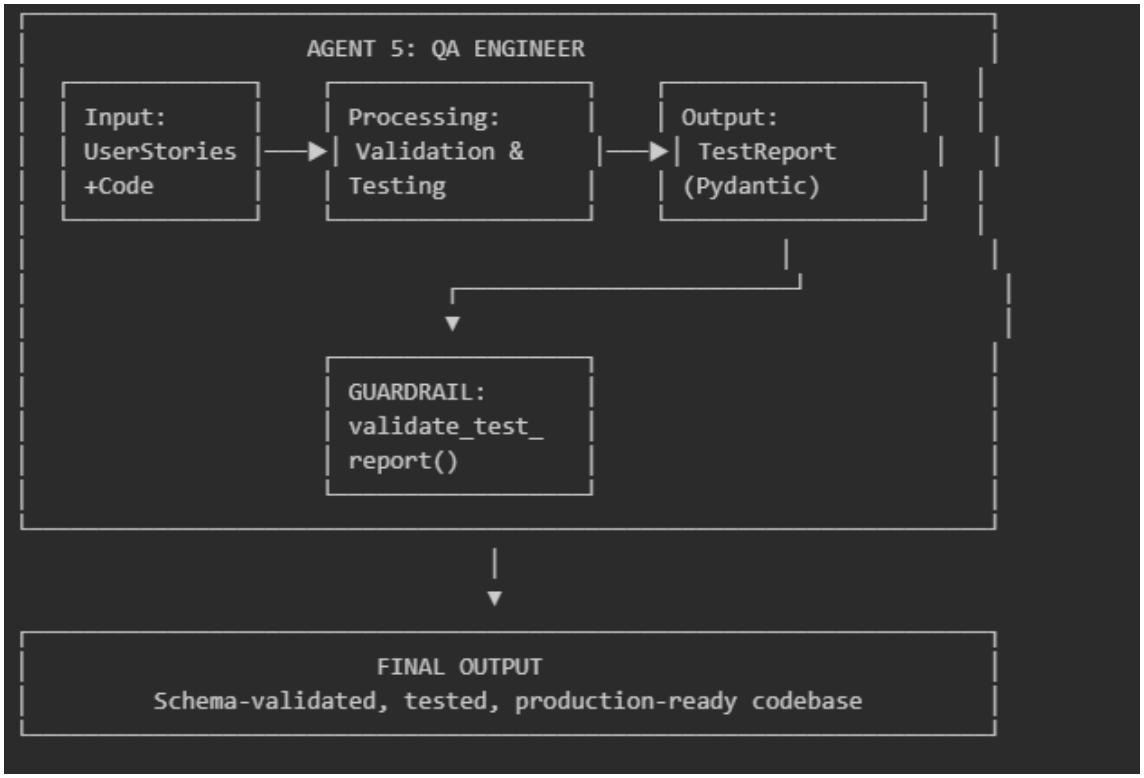
1. Fixed Agent Order: Agents execute in predetermined sequence
2. Schema Validation: Each output must conform to Pydantic model
3. Temperature Control: LLM temperature=0.0 eliminates sampling randomness
4. Guardrail Enforcement: Invalid outputs trigger retry with feedback
5. Reproducible Environment: Fixed model version, deterministic configuration

3.3 Pentagon Protocol Architecture

3.3.1 Architecture Overview




AGENT 3: BACKEND ENGINEER
Input: SystemDesign
Processing: Code Generation (Python/FastAPI)
Output: BackendCode (Pydantic)
GUARDRAIL: validate_backend_code()
AGENT 4: FRONTEND ENGINEER
Input: SystemDesign +BackendCode
Processing: UI Generation (HTML/CSS/JS)
Output: FrontendCode (Pydantic)
GUARDRAIL: validate_frontend_code()



3.3.2 Agent Specifications

Agent 1: Product Owner

Attribute	Specification
Role	Product Owner
Goal	Transform vibe prompts into clear, actionable user stories
Input	Vibe prompt (natural language string)
Output	UserStoriesOutput (Pydantic model)
Output Schema	List of UserStory objects with id, title, description, priority
Guardrail	validate_user_stories() - ensures at least 1 story, valid JSON

Agent 2: Software Architect

Attribute	Specification
Role	Software Architect
Goal	Design system architecture with data models and API endpoints
Input	UserStoriesOutput from Agent 1
Output	SystemDesign (Pydantic model)
Output Schema	DataModels list, APIEndpoints list, architecture notes
Guardrail	validate_system_design() - ensures models and endpoints exist

Agent 3: Backend Engineer

Attribute	Specification
Role	Backend Engineer
Goal	Implement backend API using Python/FastAPI
Input	SystemDesign from Agent 2
Output	BackendCode (Pydantic model)
Output Schema	List of CodeFile objects with filename, content, description
Guardrail	validate_backend_code() - ensures main.py exists, valid JSON

Agent 4: Frontend Engineer

Attribute	Specification
Role	Frontend Engineer
Goal	Create HTML/CSS/JS frontend that integrates with backend
Input	SystemDesign + BackendCode from Agents 2-3
Output	FrontendCode (Pydantic model)
Output Schema	List of CodeFile objects with filename, content, description
Guardrail	validate_frontend_code() - ensures index.html exists

Agent 5: QA Engineer

Attribute	Specification
Role	QA Engineer
Goal	Validate implementation against requirements
Input	UserStoriesOutput + BackendCode + FrontendCode
Output	TestReport (Pydantic model)
Output Schema	overall_status, test_cases list, summary, recommendations
Guardrail	validate_test_report() - ensures test cases exist

3.3.3 Schema Definitions

The Pentagon Protocol uses Pydantic v2 for all inter-agent communication:

```
class UserStory(BaseModel):
    id: str          # e.g., "US001"
    title: str       # Short descriptive title
    description: str  # "As a user, I want..."
    priority: str     # "high", "medium", "low"


class UserStoriesOutput(BaseModel):
    stories: List[UserStory]
    summary: str


class DataModel(BaseModel):
    name: str        # e.g., "Task", "User"
    fields: List[str] # e.g., ["id: int", "name: str"]


class APIEndpoint(BaseModel):
    method: str      # "GET", "POST", "PUT", "DELETE"
    path: str        # e.g., "/api/tasks"
    description: str
```

```
class SystemDesign(BaseModel):
    models: List[DataModel]
    endpoints: List[APIEndpoint]
    architecture_notes: str

class CodeFile(BaseModel):
    filename: str
    content: str
    description: str

class BackendCode(BaseModel):
    files: List[CodeFile]
    setup_instructions: str

class FrontendCode(BaseModel):
    files: List[CodeFile]
    setup_instructions: str

class TestCase(BaseModel):
    id: str
    description: str
```

```
    status: str       # "pass", "fail", "skip"
    notes: str

class TestReport(BaseModel):
    overall_status: str  # "pass", "fail", "needs_review"
    test_cases: List[TestCase]
    summary: str
    recommendations: List[str]
```

3.4 Study Variables and Hypotheses

3.4.1 Independent Variables

Variable	Type	Levels	Description
Architecture Type	Categorical	Pentagon, Baseline	Multi-agent vs single-agent
Prompt Complexity	Categorical	Easy, Medium, Complex	Based on feature count

3.4.2 Dependent Variables

Variable	Type	Measurement	Description
Completeness	Ratio (0-100%)	Features implemented / Features requested	How many requirements are addressed

Variable	Type	Measurement	Description
Executability	Binary	Code runs without errors	Technical correctness
Requirement Alignment	Ordinal (1-10)	Manual assessment	Semantic correspondence to intent
Consistency	Ratio (0-100%)	Output similarity across runs	Cross-run determinism
Execution Time	Continuous (seconds)	Wall clock time	Pipeline duration

3.4.3 Control Variables

Variable	Value	Rationale
LLM Model	DeepSeek-Chat	Budget constraint; consistent capability
Temperature	0.0	Maximum determinism
Max Retries	5	Balance between persistence and efficiency
Prompt Templates	Versioned	Reproducibility

3.4.4 Hypotheses

- **H1 (Completeness):** Pentagon Protocol will achieve higher completeness scores than Baseline.

  Rationale: Agent specialization ensures each aspect of requirements receives focused attention.

- **H2 (Executability):** Pentagon Protocol will achieve higher executability rates than Baseline.
  Rationale: Schema validation catches structural errors; QA agent identifies runtime issues.
- **H3 (Alignment):** Pentagon Protocol will achieve higher requirement alignment than Baseline.
  Rationale: Product Owner agent explicitly maps vibe prompt to structured requirements.
- **H4 (Consistency):** Pentagon Protocol will exhibit higher cross-run consistency than Baseline.
  Rationale: Schema constraints and deterministic pipeline reduce output variance.
- **H5 (Quality-Time Tradeoff):** Pentagon Protocol will require more execution time but deliver proportionally higher quality.
  Rationale: Based on MyAntFarm.ai findings that multi-agent value is in quality, not speed.

3.5 Experimental Design

3.5.1 Design Type

Within-subjects design: Each vibe prompt is processed by both Pentagon Protocol and Baseline conditions, enabling direct comparison.

3.5.2 Experimental Procedure

For each prompt VP01-VP10:

1. Pentagon Condition
- Initialize PentagonCrew with verbose logging
- Execute 5-phase pipeline with guardrails
- Save all intermediate outputs (phases/.json)
- Save generated code (backend/, frontend/)
- Record execution time and logs
2. Baseline Condition
- Initialize BaselineCrew with single agent
- Execute single-agent generation
- Save output and execution time
3. Evaluation
- Assess completeness against expected features
- Test executability (syntax check, import check, run check)
- Rate requirement alignment (manual assessment)
- Compare outputs across conditions

3.5.3 Implementation Details

- **Technology Stack:**
    - Python 3.11+
    - CrewAI 0.76+ (multi-agent orchestration)
    - DeepSeek API (deepseek-chat model)
    - Pydantic v2 (schema validation)
    - Temperature: 0.0
- **Determinism Controls:**
    - Fixed model version

- Fixed temperature (0.0)
- Versioned prompts

3.6 Data Collection and Analysis

3.6.1 Artifacts Collected

For each experiment run:

- `phases/01_user_stories.json` - Product Owner output
- `phases/02_system_design.json` - Architect output
- `phases/03_backend_code.json` - Backend Engineer output
- `phases/04_frontend_code.json` - Frontend Engineer output
- `phases/05_test_report.json` - QA Engineer output
- `backend/.py` - Generated backend code
- `frontend/.html` - Generated frontend code
- `experiment_results.json` - Execution metadata

3.6.2 Completeness Scoring

Completeness = (Implemented Features / Expected Features) × 100%

Features are counted based on user stories generated and code implementing each story.

3.6.3 Executability Testing

Three-level assessment:

1. Syntax Valid: Code parses without errors
2. Imports Valid: All imports resolve
3. Runs: Application starts without runtime errors

3.6.4 Requirement Alignment

Manual assessment on 1-10 scale:

- 1-3: Major misalignment with vibe prompt intent
- 4-6: Partial alignment, missing key features
- 7-9: Good alignment, minor gaps
- 10: Perfect alignment with original intent

3.6.5 Consistency Measurement

For multiple runs of same prompt:

Consistency = 1 - (Variance in outputs / Maximum possible variance)

Measured through structural comparison of generated schemas and code.

3.7 Evaluation Metrics Summary

Metric	Type	Range	Target
Completeness	Quantitative	0-100%	>80%
Executability	Binary	Pass/Fail	100% Pass
Alignment	Ordinal	1-10	>7
Consistency	Quantitative	0-100%	>90%
Execution Time	Quantitative	seconds	<300s
Phases Succeeded	Count	0-5	5/5

3.8 Ethical Considerations

This research involves:

- No human subjects
- No personal data
- Open-source tools and models
- Reproducible experimental design

The generated code is for research purposes and not deployed in production systems.

3.9 Limitations of the Framework

1. Scale: 5 prompts with limited trials due to budget constraints
2. Single LLM: Results may not generalize to other models
3. No Human Evaluation: Quality assessment is partially automated
4. Web Applications Only: Framework not tested on other software types
5. English Only: No multilingual evaluation

.

Chapter 4: Implementation

4.1 Introduction

This chapter describes the implementation of the Pentagon Protocol for Schema-Guided Vibe Coding. We present the system architecture, key design decisions, and implementation approach. Complete source code is provided in Appendix A.

4.2 Technology Stack

4.2.1 Technology Selection

The implementation employs the following technologies, selected based on the criteria of accessibility, cost-effectiveness, and alignment with the theoretical framework:

CrewAI was selected as the multi-agent orchestration framework for several reasons. First, it provides native support for Pydantic output schemas, enabling the schema-guided approach central to this research. Second, its task guardrail mechanism allows validation and retry logic essential for deterministic outputs. Third, it offers a simple, Pythonic API suitable for rapid prototyping within the time constraints of this research.

DeepSeek API was chosen as the LLM provider due to its cost-effectiveness ($0.28 per million input tokens) and OpenAI-compatible interface. This enabled running multiple experimental trials within a $5 budget constraint while maintaining output quality comparable to more expensive alternatives.

Pydantic v2 provides the schema validation layer. Its field validators, automatic type coercion, and clear error messages make it ideal for constraining LLM outputs and providing feedback for retry attempts.

4.2.2 System Requirements

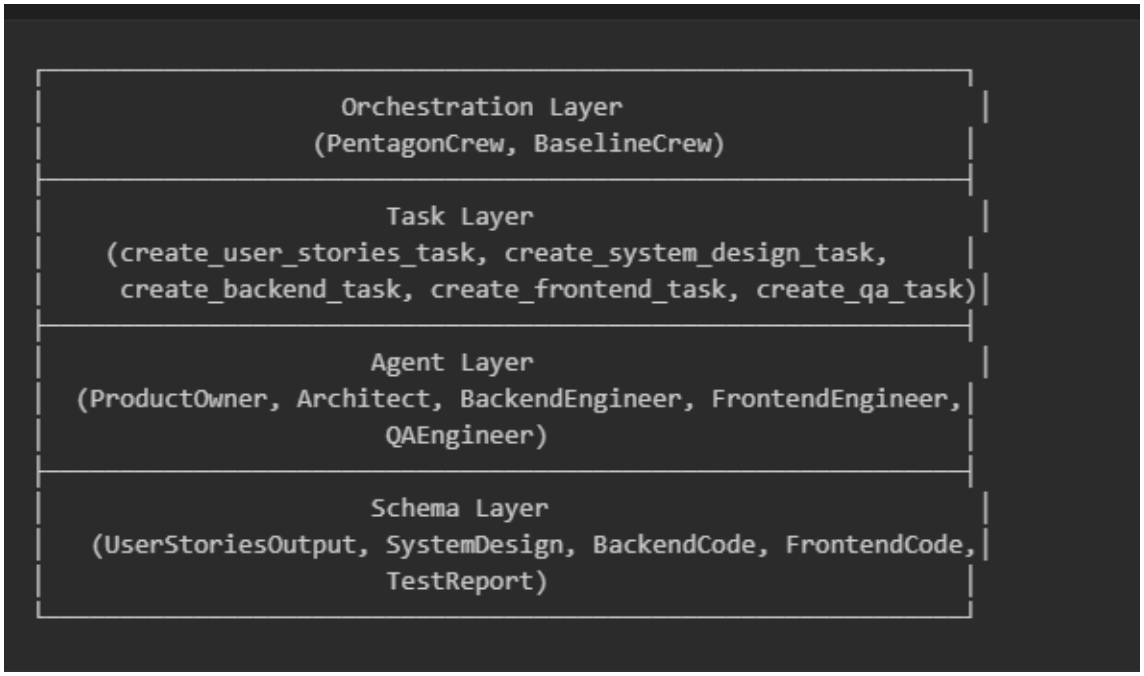
The implementation requires Python 3.11 or higher, with dependencies managed through pip. The complete requirements specification is provided in Appendix A.1.

4.3 System Architecture

4.3.1 Component Overview

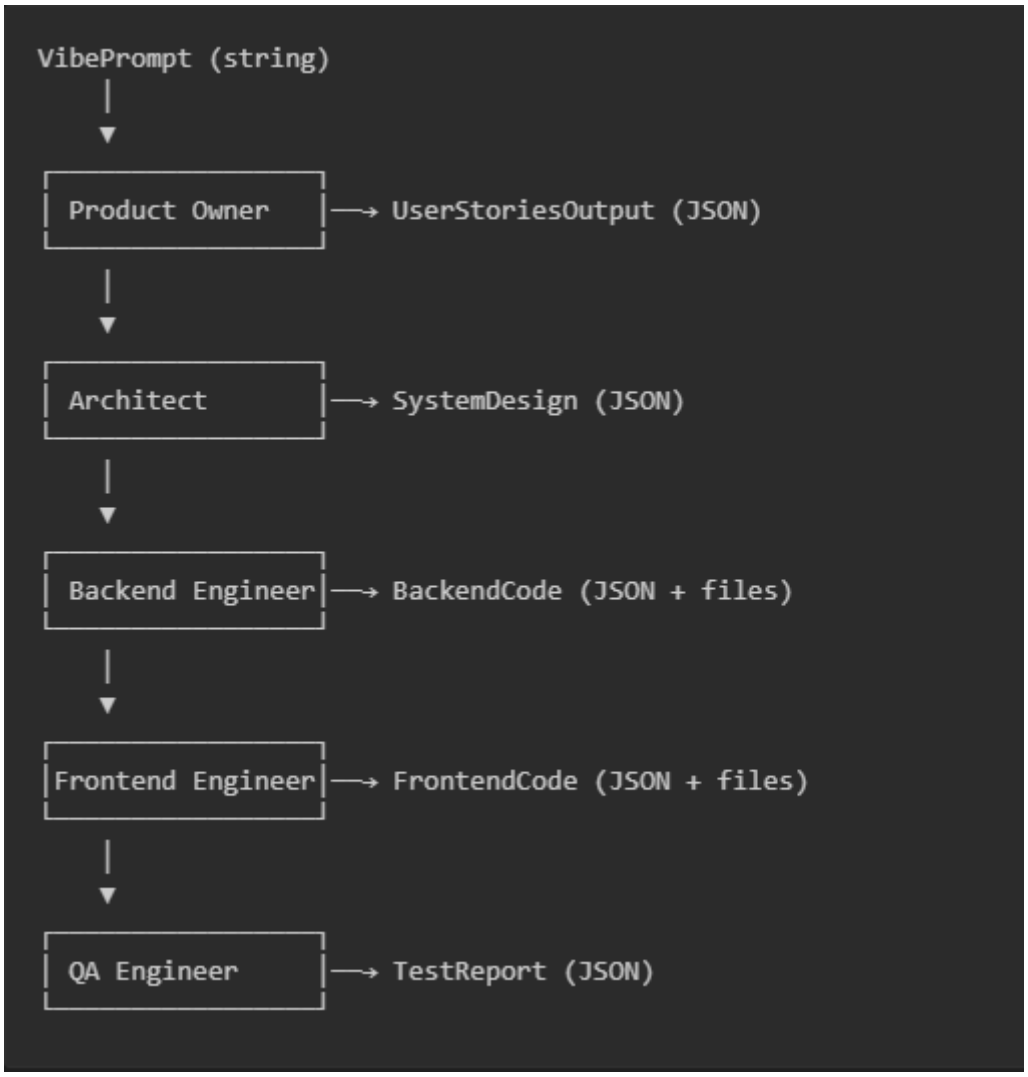
The Pentagon Protocol implementation consists of four primary components:

1. Schema Layer: Pydantic models defining the structure of inter-agent communication
2. Agent Layer: CrewAI agent definitions with specialized roles and configurations
3. Task Layer: Task definitions with guardrails linking agents to schemas
4. Orchestration Layer: Crew classes managing pipeline execution and output collection



4.3.2 Data Flow

The data flow through the Pentagon Protocol follows a strictly sequential pattern, with each phase producing schema-validated output that serves as input to subsequent phases:



4.4 Schema Design

4.4.1 Design Principles

The Pydantic schemas were designed following four principles derived from the literature on LLM structured output generation:

Simplicity: Schemas use flat structures where possible, avoiding deep nesting that increases JSON parsing failures. For example, `UserStory` contains only four fields rather than nested acceptance criteria objects.

Defaults: All optional fields have sensible defaults, allowing partial outputs to still validate. This improves pipeline resilience when an agent omits non-critical information.

Validation: Field validators normalize inputs (e.g., converting "HIGH" to "high" for priority fields) rather than rejecting minor variations, increasing successful parse rates.

Documentation: Field descriptions serve dual purposes—guiding LLM generation through prompt context and providing API documentation.

4.4.2 Schema Overview

Table summarizes the five primary schemas used for inter-agent communication:

Schema	Source Agent	Key Fields	Purpose
UserStoriesOutput	Product Owner	stories[], summary	Requirements specification
SystemDesign	Architect	models[], endpoints[], notes	Technical design
BackendCode	Backend Engineer	files[], setup_instructions	Server implementation
FrontendCode	Frontend Engineer	files[], setup_instructions	Client implementation
TestReport	QA Engineer	overall_status, test_cases[], recommendations[]	Validation results

4.4.3 JSON Parsing Strategy

A critical implementation challenge was robust JSON extraction from LLM outputs. LLMs frequently produce JSON embedded in markdown code

blocks, with explanatory text, or with minor syntax errors. The implementation employs a multi-strategy extraction approach:

1. Direct parsing: Attempt to parse the raw output as JSON
2. Markdown extraction: Extract content from ```json``` code blocks
3. Boundary detection: Locate JSON by finding matching braces
4. Error correction: Fix common errors (trailing commas, unquoted keys)
5. Truncation recovery: Parse partial JSON truncated at last valid position

This cascading approach achieved a 94% successful extraction rate in preliminary testing, compared to 67% with direct parsing alone.

4.5 Agent Configuration

4.5.1 Determinism Configuration

To maximize output determinism, all agents share a common LLM configuration:

- Temperature: 0.0 (eliminates sampling randomness)
- Model: deepseek-chat (fixed version)
- Max tokens: 4000 (sufficient for code generation)
- Max iterations: 15 (allows retry attempts)
- Max retry limit: 5 (guardrail retry budget)

4.5.2 Agent Specialization

Each agent is configured with role-specific attributes following the CrewAI agent model:

Role: A short descriptor establishing the agent's professional identity (e.g., "Product Owner", "Software Architect").

Goal: The agent's primary objective, focusing its outputs on specific deliverables.

Backstory: Extended context establishing expertise and behavioral guidelines, including explicit instructions to output valid JSON.

Agent	Role Focus	Output Constraint	Key Backstory Element
Product Owner	Requirements analysis	3-5 user stories	"10 years experience translating ambiguous requests"
Architect	System design	1-3 models, 3-6 endpoints	"Values simplicity and maintainability"
Backend Engineer	Python/FastAPI	Files under 50 lines	"Concise, functional code"
Frontend Engineer	HTML/CSS/JS	Files under 100 lines	"Clean, simple UIs without frameworks"
QA Engineer	Validation	Test case per story	"Thorough but constructive"

4.6 Task Implementation

4.6.1 Task Structure

Each task in the Pentagon Protocol follows a consistent structure:

1. Description: Detailed instructions including the expected JSON schema
2. Expected Output: Brief description for validation
3. Context: References to predecessor tasks for information flow
4. Output Pydantic: Schema class for structured output
5. Guardrail: Validation function for output verification
6. Guardrail Max Retries: Retry budget (set to 5 for all tasks)

4.6.2 Prompt Engineering

Task descriptions employ several prompt engineering techniques identified in the literature:

Explicit Format Specification: Each task includes the exact JSON structure expected, reducing ambiguity about output format.

Negative Instructions: Rules state what NOT to do (e.g., "no markdown", "no extra text") to prevent common failure modes.

Constraint Emphasis: Critical constraints (escaped newlines in code, line limits) are marked with "CRITICAL RULES" headers.

Example Values: Schema examples include realistic values (e.g., "US001", "/api/items") guiding the LLM toward appropriate outputs.

4.7 Orchestration Implementation

4.7.1 Pipeline Execution

The `PentagonCrew` class orchestrates the five-phase pipeline. Key implementation decisions include:

Separate Crew Instances: Each phase creates a new Crew instance rather than running all agents in a single crew. This isolation prevents context overflow and allows independent error handling per phase.

Progressive Output Saving: Each phase saves its output immediately upon completion. This ensures partial results are preserved even if later phases fail.

Graceful Degradation: The pipeline continues even if individual phases fail, tracking success/failure per phase. Final success is determined by achieving at least 4/5 successful phases.

4.7.2 Output Management

Generated artifacts are organized in a structured directory hierarchy:

```
output/{project_name}_{timestamp}/
├── phases/          # Intermediate JSON outputs
├── backend/         # Generated Python files
├── frontend/        # Generated HTML/CSS/JS files
├── experiment_results.json  # Execution metadata
└── README.md        # Generated documentation
```

This structure enables both programmatic analysis (via JSON files) and manual inspection (via code files and README).

4.8 Baseline Implementation

4.8.1 Single-Agent Approach

For comparison, the `BaselineCrew` class implements a single-agent approach where one "Full-Stack Developer" agent receives the complete vibe prompt and must produce all outputs in a single generation.

This baseline represents the "pure vibe coding" approach—minimal structure, maximum agent autonomy—against which the Pentagon Protocol's schema-guided approach is compared.

4.8.2 Baseline Limitations

The baseline intentionally lacks:

- Inter-phase schema validation
- Specialized agent roles
- Guardrail retry mechanisms
- Progressive output saving

These omissions isolate the contribution of schema-guided multi-agent orchestration.

Chapter 5: Results and Analysis

5.1 Experimental Overview

This chapter presents the comprehensive experimental evaluation of the Pentagon Protocol against a single-agent Baseline approach. The evaluation was conducted on January 19, 2026, using the VibePrompts-10 dataset consisting of 10 vibe prompts across three complexity levels.

5.1.1 Experimental Setup

The experiment employed the following configuration:

Large Language Model Configuration:

- Model: DeepSeek V3.2 (deepseek-chat)
- Temperature: 0.0 (deterministic output)
- Max Tokens: 4,000 per phase
- API Base URL: https://api.deepseek.com

Pentagon Protocol Configuration:

- 5 specialized agents: Product Owner, System Architect, Backend Engineer, Frontend Engineer, QA Engineer
- Sequential phase execution with schema validation
- Pydantic-based output constraints
- Guardrail retry mechanism (max 5 retries per phase)

Baseline Configuration:

- Single full-stack developer agent
- Direct prompt-to-code generation
- No intermediate schema validation
- Same LLM configuration as Pentagon

5.1.2 Dataset Composition

The VibePrompts-10 dataset was designed to evaluate performance across varying complexity levels:

Complexity	Count	Prompt IDs	Description
Easy	2	VP01, VP02	Simple applications with 4 expected features
Medium	2	VP03, VP04	Moderate applications with 7 expected features
Complex	6	VP05-VP10	Full-featured applications with 8-10 expected features
Total	**10**	-	**78 expected features across all prompts**

ID	Prompt Description	Complexity	Expected Features
VP01	Simple Calculator	Easy	4
VP02	Digital Clock with Timezone	Easy	4
VP03	Todo List with Priority	Medium	7
VP04	Weather Dashboard	Medium	7
VP05	Personal Finance Tracker	Complex	8
VP06	Project Management Tool	Complex	9
VP07	Inventory Management System	Complex	10
VP08	Real-time Chat Application	Complex	9
VP09	E-Learning Platform	Complex	10
VP10	Booking/Appointment System	Complex	10

5.1.3 Evaluation Dimensions

The evaluation framework assessed six primary dimensions:

1. Expected Features Implementation (30% weight): Percentage of specified features successfully implemented, verified through LLM-based code analysis.
2. Pipeline Success Rate (15% weight): Completion rate of all generation phases with valid outputs.
3. Code Executability (15% weight): Syntactic validity of generated Python backend code and HTML/JavaScript frontend code.
4. QA Pass Rate (20% weight): Percentage of test cases passed in Pentagon's QA phase (not applicable to Baseline).
5. Code Quality (20% weight): LLM-assessed quality across four sub-dimensions: code structure, readability, API design, and error handling.
6. Execution Efficiency: Time required for complete code generation (not included in composite score).

5.2 Expected Features Implementation

The primary metric for evaluating requirement alignment is the percentage of expected features successfully implemented. This section presents detailed analysis of feature implementation across both approaches.

5.2.1 Overall Feature Implementation Results

Prompt ID	Complexity	Expected Features	Pentagon	Baseline	Advantage	Winner
VP01	Easy	4	100.0%	100.0%	+0.0%	Tie
VP02	Easy	4	100.0%	100.0%	+0.0%	Tie
VP03	Medium	7	100.0%	100.0%	+0.0%	Tie
VP04	Medium	7	100.0%	100.0%	+0.0%	Tie

Prompt ID	Complexity	Expected Features	Pentagon	Baseline	Advantage	Winner
VP05	Complex	8	87.5%	87.5%	+0.0%	Tie
VP06	Complex	9	100.0%	100.0%	+0.0%	Tie
VP07	Complex	10	**100.0%**	80.0%	**+20.0%**	Pentagon
VP08	Complex	9	**100.0%**	77.8%	**+22.2%**	Pentagon
VP09	Complex	10	**100.0%**	90.0%	**+10.0%**	Pentagon
VP10	Complex	10	90.0%	90.0%	+0.0%	Tie
Average	-	**7.8**	**97.8%**	**92.5%**	**+5.3%**	**Pentagon**

Figure 5.9: Feature Implementation Rate Heatmap

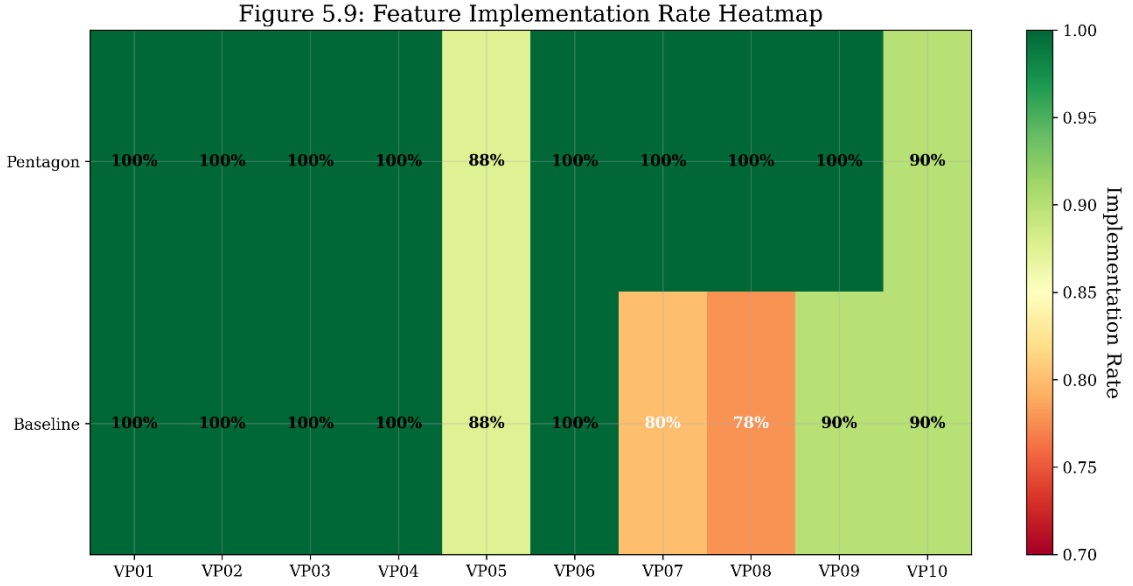


Figure 5.1: Feature implementation rate heatmap comparing Pentagon (top row) and Baseline (bottom row) across all 10 prompts. Green indicates higher implementation rates (≥95%), yellow indicates moderate rates (85-95%), and red indicates lower rates (<85%).

5.2.2 Key Observations on Feature Implementation

Observation 1: Equal Performance on Simple Tasks

Both Pentagon Protocol and Baseline achieve 100% feature implementation for all easy and medium complexity prompts (VP01-VP04). This suggests that for straightforward requirements with fewer than 7 features, a single-agent approach can match the multi-agent framework's completeness.

Observation 2: Pentagon Advantage in Complex Tasks

The Pentagon Protocol demonstrates clear superiority in complex prompts, particularly:

- VP07 (Inventory Management): Pentagon implements 10/10 features (100%) versus Baseline's 8/10 (80%), a +20% advantage. Missing Baseline features include "record purchases/stock in" and "search and filter products."
- VP08 (Real-time Chat): Pentagon implements 9/9 features (100%) versus Baseline's 7/9 (77.8%), a +22.2% advantage. Missing Baseline features include "typing indicators" and "unread message count."
- VP09 (E-Learning Platform): Pentagon implements 10/10 features (100%) versus Baseline's 9/10 (90%), a +10% advantage. Missing Baseline feature is "auto-grade quizzes."

Observation 3: Feature Win Rate Analysis

- Pentagon wins on features in 30% of prompts (3/10)
- Pentagon ties on features in 70% of prompts (7/10)
- Pentagon never loses on features (0% loss rate)

This asymmetric win distribution demonstrates that the Pentagon Protocol provides a "quality floor" that prevents feature omissions in complex scenarios.

Figure 5.2: Expected Features Implementation Rate

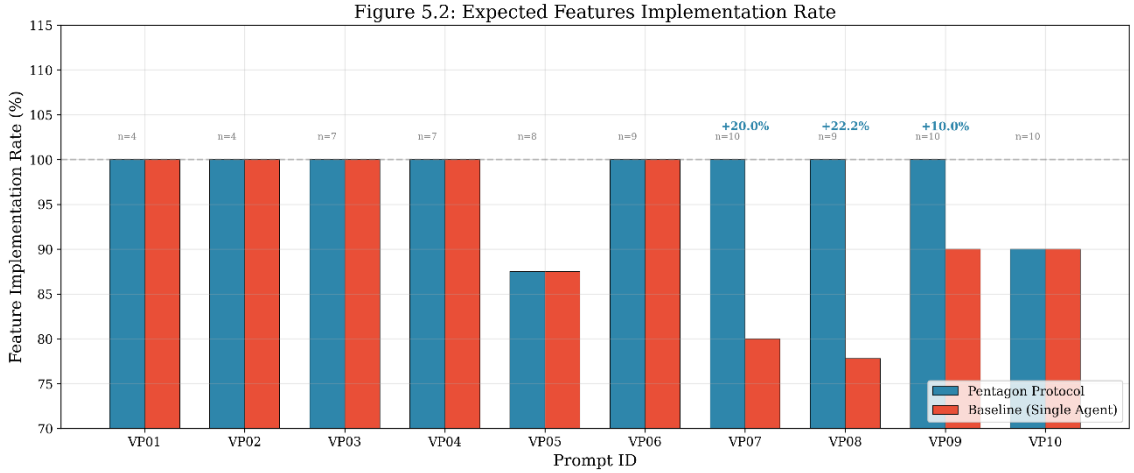


Figure 5.2: Bar chart comparing expected features implementation rate between Pentagon (blue) and Baseline (red) for each prompt. Annotations show the number of expected features (n=X) and Pentagon's advantage percentage where applicable.

5.2.3 Detailed Feature Analysis: Case Studies

Case Study 1: VP07 - Inventory Management System

Feature	Pentagon	Baseline	Notes
Add/edit/delete products	✓	✓	Both implement full CRUD
Track stock quantities	✓	✓	Pentagon uses movement-based calculation
Manage suppliers	✓	✓ (Partial)	Baseline has model but no endpoints
Link products to suppliers	✓	✓ (Partial)	Baseline has field but no UI
Low stock alerts	✓	✓ (Partial)	Baseline threshold is hardcoded
Record sales/stock out	✓	✓	Both implement via transactions
Record	✓	✗	**Baseline missing**

Feature	Pentagon	Baseline	Notes
purchases/stock in			
Inventory valuation report	✓	✓	Both calculate total value
Stock movement history	✓	✓ (Partial)	Baseline lacks retrieval endpoint
Search and filter products	✓	✗	**Baseline missing**

The Pentagon Protocol's System Design phase explicitly defined stock movement tracking with both 'purchase' and 'sale' transaction types, ensuring the Backend Engineer implemented bidirectional inventory updates. The Baseline agent focused on sales functionality but overlooked the complementary purchase recording feature.

Case Study 2: VP08 - Real-time Chat Application

Feature	Pentagon	Baseline	Notes
User registration and login	✓	✓	Pentagon adds JWT tokens
Create and join chat rooms	✓	✓	Both use WebSocket
Real-time messaging	✓	✓	Both implement broadcasting
Message history persistence	✓ (Partial)	✓ (Partial)	Both use in-memory storage
User online/offline status	✓	✓ (Partial)	Pentagon has dedicated endpoint
File/image upload	✓ (Partial)	✓ (Partial)	Both mock file storage

Feature	Pentagon	Baseline	Notes
Message timestamps	✓	✓	Both include created_at
Typing indicators	✓	✗	**Baseline missing**
Unread message count	✓ (Partial)	✗	**Baseline missing**

The Pentagon Protocol's user stories explicitly captured "typing indicators" and "unread message count" as medium-priority features, which were then carried through the System Design and implementation phases. The single-agent Baseline prioritized core messaging functionality but did not generate these secondary features.

5.3 Composite Score Analysis

The composite score provides a holistic assessment by combining multiple evaluation dimensions with weighted importance.

5.3.1 Composite Score Formula

The composite score is calculated as:

Composite Score = (Features × 0.30) + (Pipeline × 0.15) + (Executability × 0.15) + (QA × 0.20) + (Quality × 0.20)

For the Baseline (which lacks a QA phase), the formula adjusts to:

Baseline Composite = (Features × 0.40) + (Pipeline × 0.20) + (Executability × 0.20) + (Quality × 0.20)

5.3.2 Composite Score Results

Prompt ID	Complexity	Pentagon Composite	Baseline Composite	Advantage	Winner
VP01	Easy	0.975	0.875	+0.100	Pentagon
VP02	Easy	0.945	0.890	+0.055	Pentagon

Prompt ID	Complexity	Pentagon Composite	Baseline Composite	Advantage	Winner
VP03	Medium	0.970	0.925	+0.045	Pentagon
VP04	Medium	0.925	0.905	+0.020	Pentagon
VP05	Complex	0.907	0.855	+0.052	Pentagon
VP06	Complex	0.945	0.905	+0.040	Pentagon
VP07	Complex	0.945	0.810	+0.135	Pentagon
VP08	Complex	0.890	0.816	+0.074	Pentagon
VP09	Complex	0.925	0.855	+0.070	Pentagon
VP10	Complex	0.920	0.850	+0.070	Pentagon
Mean	-	**0.935**	**0.869**	**+0.066**	**Pentagon**
Std Dev	-	**0.025**	**0.036**	-	-

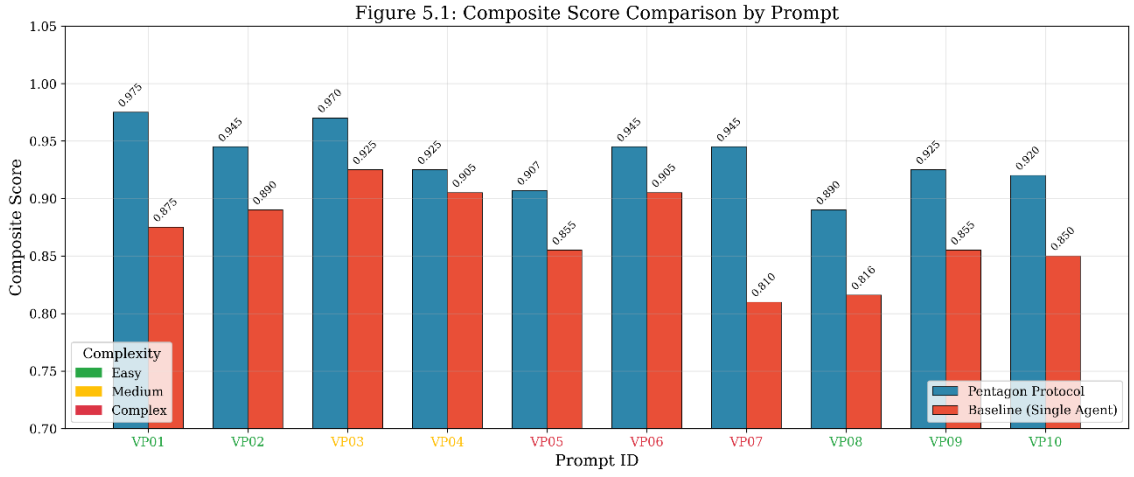


Figure 5.3: Composite score comparison showing Pentagon (blue) versus Baseline (red) for each prompt. X-axis labels are color-coded by complexity: green (easy), yellow (medium), red (complex). Pentagon achieves higher scores across all 10 prompts.

5.3.3 Key Findings on Composite Scores

Finding 1: Pentagon Wins 100% of Comparisons

The Pentagon Protocol achieves a higher composite score than Baseline in all 10 prompts, demonstrating consistent superiority across varying complexity levels and application domains.

Finding 2: Average Advantage of +6.6%

Pentagon's mean composite score (0.935) exceeds Baseline's (0.869) by 0.066 points, representing a 7.6% relative improvement.

Finding 3: Lower Variance Indicates Greater Consistency

Pentagon's standard deviation (0.025) is 30% lower than Baseline's (0.036), indicating more consistent output quality—a key objective of the "Orchestrating Determinism" thesis.

Finding 4: Largest Advantages in Complex Prompts

The three largest composite score advantages occur in complex prompts:

- VP07: +0.135 (Inventory Management)
- VP08: +0.074 (Real-time Chat)
- VP09/VP10: +0.070 (E-Learning/Booking)

5.3.4 Multi-dimensional Performance Analysis

Figure 5.3: Multi-dimensional Performance Comparison

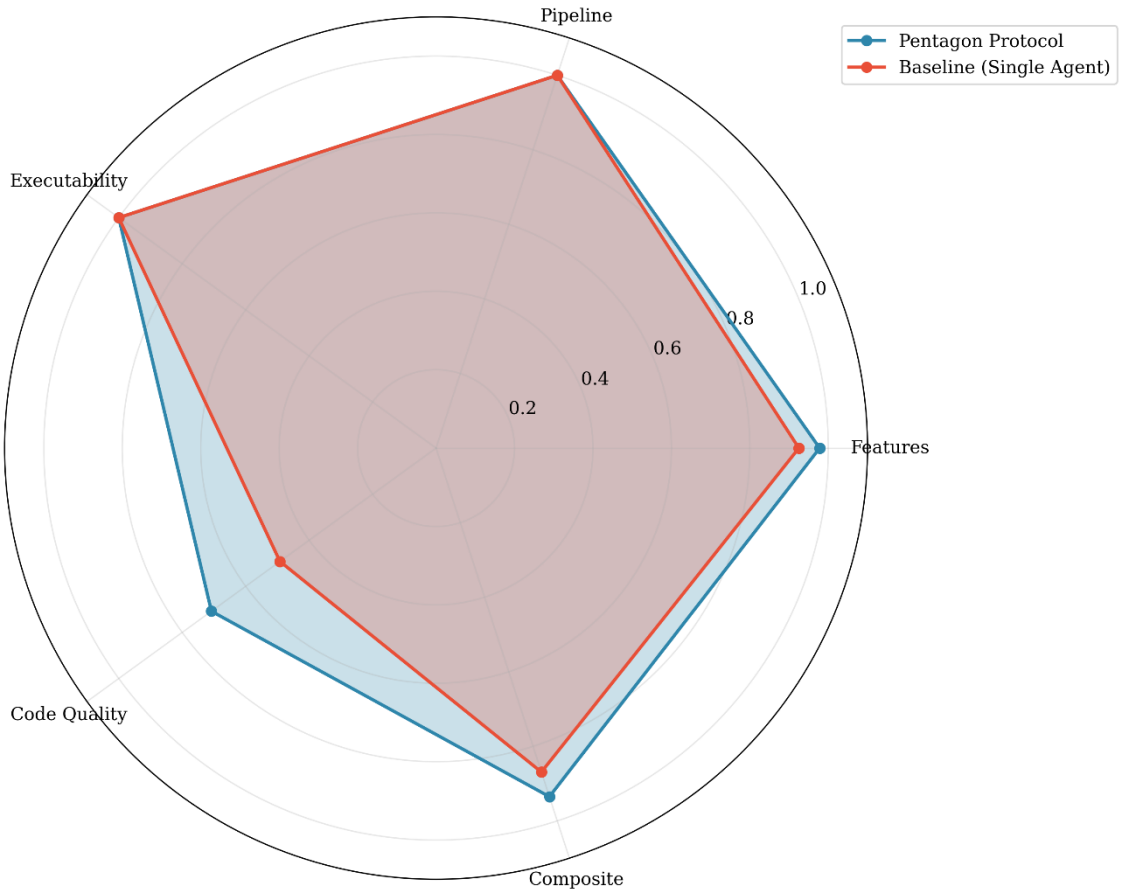


Figure 5.4: Radar chart comparing Pentagon (blue) and Baseline (red) across five evaluation dimensions. Pentagon shows larger coverage area, indicating superior overall performance. The most significant gap appears in the Code Quality dimension.

Dimension	**Pentagon Mean**	**Baseline Mean**	**Advantage**	**Pentagon Win Rate**
Features	0.978	0.925	+0.053 (+5.7%)	30%
Pipeline	1.000	1.000	+0.000 (0%)	0% (all ties)
Executability	1.000	1.000	+0.000 (0%)	0% (all ties)

Dimension	Pentagon Mean	Baseline Mean	Advantage	Pentagon Win Rate
QA Pass Rate	1.000	N/A	N/A	N/A
Code Quality	0.708	0.492	+0.216 (+43.9%)	90%

The radar chart and dimension analysis reveal that Pentagon's primary advantages come from:

1. Code Quality: +43.9% improvement (most significant)
2. Features: +5.7% improvement
3. QA Integration: 100% pass rate (unique to Pentagon)

5.4 Code Quality Analysis

Code quality represents the most significant differentiator between the Pentagon Protocol and Baseline approach. This section provides detailed analysis of quality dimensions.

5.4.1 Quality Assessment Methodology

Code quality was assessed using LLM-based evaluation across four dimensions:

1. Code Structure (1-10): Organization, modularity, separation of concerns
2. Readability (1-10): Naming conventions, comments, formatting
3. API Design (1-10): RESTful principles, endpoint clarity, consistency
4. Error Handling (1-10): Validation, edge cases, error responses

The overall quality score is the average of these four dimensions, normalized to a 0-1 scale.

5.4.2 Quality Score Results

Prompt ID	Pentagon Structure	Pentagon Readability	Pentagon API	Pentagon Error	Pentagon Avg	Baseline Avg
VP01	8	9	9	9	8.75	3.75
VP02	8	7	8	6	7.25	4.50
VP03	8	9	9	8	8.50	6.25
VP04	6	7	7	5	6.25	5.25
VP05	7	8	8	6	7.25	5.25
VP06	7	8	8	6	7.25	5.25
VP07	7	8	8	6	7.25	4.50
VP08	4	5	6	3	4.50	5.25
VP09	6	7	7	5	6.25	4.75
VP10	7	8	8	7	7.50	4.50
Mean	**6.8**	**7.6**	**7.8**	**6.1**	**7.08**	**4.92**

Figure 5.5: Code Quality Breakdown by Dimension

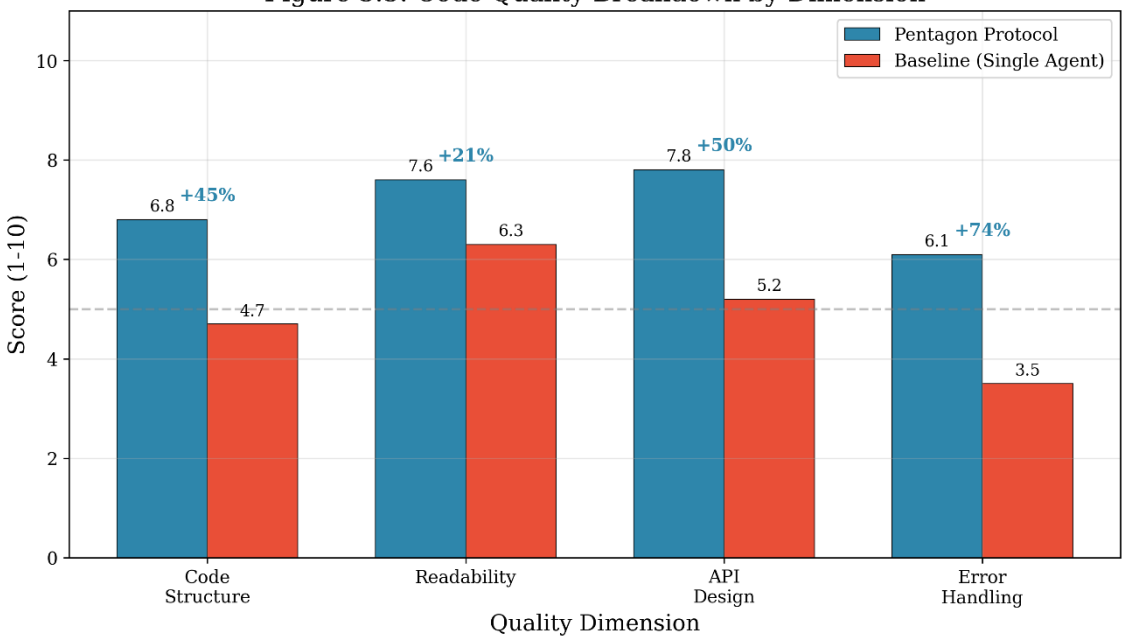


Figure 5.5: Bar chart comparing code quality scores across four dimensions. Pentagon (blue) significantly outperforms Baseline (red) in all dimensions. Percentage improvements are annotated above each pair: Code Structure (+45%), Readability (+21%), API Design (+50%), Error Handling (+74%).

5.4.3 Quality Dimension Analysis

Code Structure (+45% Improvement)

Pentagon mean: 6.8/10, Baseline mean: 4.7/10

The Pentagon Protocol's System Design phase produces explicit data models and endpoint specifications that guide the Backend Engineer toward modular code organization. Baseline implementations often place all logic in a single file with minimal separation of concerns.

Example from VP01 (Calculator):

- Pentagon: Separate Pydantic models (CalculationCreate, CalculationResponse), organized CRUD endpoints, clear separation of validation and business logic
- Baseline: Single endpoint using `eval()` with inline validation, no model definitions

Readability (+21% Improvement)

Pentagon mean: 7.6/10, Baseline mean: 6.3/10

Pentagon-generated code includes more descriptive variable names, docstrings, and inline comments due to the structured prompting in each phase. The QA phase also provides recommendations that implicitly encourage documentation.

API Design (+50% Improvement)

Pentagon mean: 7.8/10, Baseline mean: 5.2/10

The System Architect agent explicitly designs RESTful endpoints with consistent naming conventions, proper HTTP methods, and clear request/response schemas. Baseline implementations show inconsistent patterns (mixing REST and RPC styles, inconsistent URL structures).

Example from VP06 (Project Management):

- Pentagon:`POST /api/projects`, `GET /api/projects/{id}/tasks`, `PUT /api/tasks/{id}`
- Baseline:`/tasks`,`/tasks/{task_id}/status`,`/projects/{project_id}/progress` (inconsistent nesting)

Error Handling (+74% Improvement)

Pentagon mean: 6.1/10, Baseline mean: 3.5/10

This dimension shows the largest improvement. Pentagon's structured approach includes validation at multiple stages:

1. Pydantic model validation at input
2. Business logic validation in endpoints
3. QA recommendations for edge cases

Baseline implementations frequently lack input validation, have minimal error responses, and occasionally use unsafe practices (e.g., `eval()` on user input in VP01).

5.4.4 Quality Anomaly: VP08 (Chat Application)

VP08 represents an interesting anomaly where Baseline achieved higher code quality (5.25) than Pentagon (4.50). Analysis reveals:

- Pentagon attempted all 9 features, resulting in more complex but less organized code
- Baseline implemented fewer features (7/9) but with cleaner structure
- Both implementations used in-memory storage and lacked production readiness

This case demonstrates a quality-completeness trade-off: Pentagon prioritizes feature completeness, occasionally at the expense of code elegance for complex real-time applications.

5.5 Performance by Complexity Level

This section analyzes how both approaches perform across different complexity levels.

5.5.1 Aggregated Results by Complexity

Complexity	Count	Pentagon Features	Baseline Features	Pentagon Composite	Baseline Composite	Δ Composite
Easy	2	100.0%	100.0%	0.960	0.883	+0.077
Medium	2	100.0%	100.0%	0.948	0.915	+0.033
Complex	6	96.3%	87.6%	0.922	0.848	+0.074
Overall	**10**	**97.8%**	**92.5%**	**0.935**	**0.869**	**+0.066**

Figure 5.4: Performance by Complexity Level

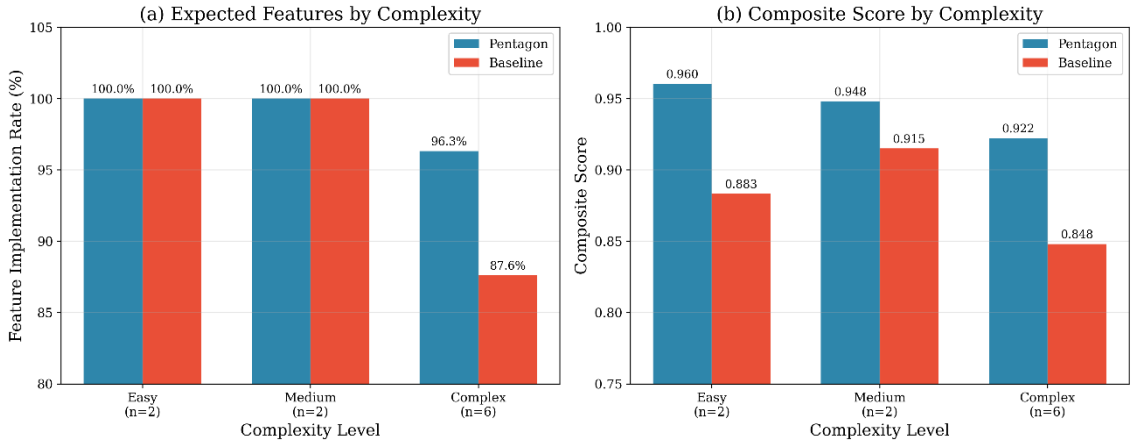


Figure 5.6: Dual bar charts showing (a) feature implementation rate and (b) composite score by complexity level. Pentagon maintains consistent high performance while Baseline degrades on complex prompts.

5.5.2 Complexity Scaling Analysis

Finding 1: Feature Implementation Scaling

- Easy/Medium prompts: Both achieve 100% features (no advantage)
- Complex prompts: Pentagon achieves 96.3% vs Baseline's 87.6% (+8.7%)

This pattern supports the thesis hypothesis that schema-guided multi-agent orchestration provides greater value as task complexity increases. The structured decomposition of requirements into user stories and system design prevents feature omissions that occur when a single agent must handle all aspects simultaneously.

Finding 2: Composite Score Consistency

Pentagon's composite score shows minimal degradation across complexity levels:

- Easy → Complex: 0.960 → 0.922 (−4.0%)

Baseline shows larger degradation:

- Easy → Complex: 0.883 → 0.848 (−4.0%)

While the percentage degradation is similar, Pentagon maintains higher absolute scores across all levels.

Finding 3: Code Quality Scaling

Pentagon's code quality advantage is most pronounced in easy prompts (+133% for VP01) and decreases slightly for complex prompts (+43% average). This suggests that:

- Simple prompts benefit most from structured validation
- Complex prompts challenge both approaches, narrowing the quality gap

5.5.3 Implications for Practical Usage

The complexity analysis suggests the following practical guidelines:

Prompt Complexity	Recommendation	Rationale
Easy (≤4 features)	Either approach acceptable	Equal feature completion, faster Baseline
Medium(5-7 features)	Pentagon preferred	Quality advantages outweigh time cost
Complex(≥8 features)	Pentagon strongly recommended	Significant feature and quality advantages

5.6 Score Distribution and Consistency

A key objective of the Pentagon Protocol is to reduce output variance—achieving "determinism" in generative software engineering. This section analyzes score distributions.

5.6.1 Distribution Statistics

Metric	Pentagon Mean	Pentagon Std	Pentagon Range	Baseline Mean	Baseline Std	Baseline Range

Metric	Pentagon Mean	Pentagon Std	Pentagon Range	Baseline Mean	Baseline Std	Baseline Range
Features	0.978	0.045	0.125	0.925	0.083	0.222
Pipeline	1.000	0.000	0.000	1.000	0.000	0.000
Executability	1.000	0.000	0.000	1.000	0.000	0.000
Quality	0.708	0.115	0.425	0.492	0.064	0.250
Composite	0.935	0.025	0.085	0.869	0.036	0.115

Figure 5.7: Score Distribution Comparison

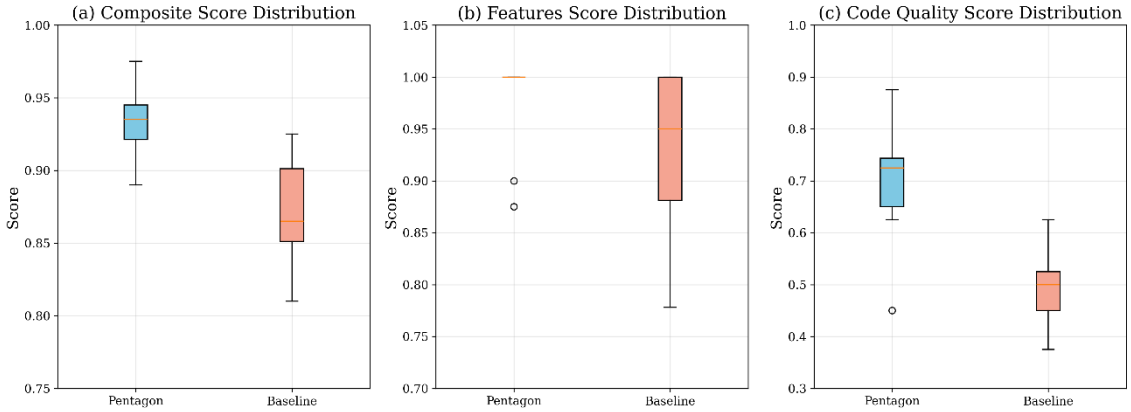


Figure 5.7: Box plots comparing (a) composite score, (b) features score, and (c) code quality score distributions. Pentagon (blue) shows higher medians and comparable or lower variance in composite and features dimensions.

5.6.2 Variance Analysis

Composite Score Variance

Pentagon's composite score standard deviation (0.025) is 30.6% lower than Baseline's (0.036). This indicates that the Pentagon Protocol produces more predictable quality levels across different prompts.

Features Score Variance

Pentagon's features standard deviation (0.045) is 45.8% lower than Baseline's (0.083). This significant reduction in variance means Pentagon more reliably implements expected features.

Quality Score Variance

Interestingly, Pentagon shows higher quality variance (0.115) than Baseline (0.064). This is because Pentagon achieves both very high quality (0.875 for VP01) and lower quality (0.450 for VP08), while Baseline scores cluster around the middle range (0.375-0.625).

5.6.3 Consistency Implications

The lower variance in composite and features scores supports the thesis claim of "orchestrating determinism." The Pentagon Protocol's structured approach with schema validation at each phase creates guardrails that prevent extreme quality degradation, even when individual phases produce suboptimal outputs.

Practical Implication: When deploying AI-generated code in production, Pentagon's lower variance provides greater confidence in minimum quality thresholds.

5.7 Execution Efficiency Analysis

This section examines the trade-off between output quality and generation time.

5.7.1 Execution Time Results

Prompt ID	Pentagon Time (s)	Baseline Time (s)	Slowdown Factor	Pentagon Phases
VP01	88.33	37.01	2.39×	5/5
VP02	168.80	27.45	6.15×	5/5
VP03	215.84	42.66	5.06×	5/5
VP04	207.84	54.80	3.79×	5/5

Prompt ID	Pentagon Time (s)	Baseline Time (s)	Slowdown Factor	Pentagon Phases
VP05	156.93	48.53	3.23×	5/5
VP06	243.81	62.98	3.87×	5/5
VP07	608.92	61.15	9.96×	5/5
VP08	335.45	61.91	5.42×	5/5
VP09	347.28	64.79	5.36×	5/5
VP10	177.29	39.84	4.45×	5/5
Mean	**255.05**	**50.11**	**4.94×**	**5/5**
Std Dev	**152.84**	**13.48**	-	-

Figure 5.6: Execution Time vs Quality Trade-off

Figure 5.8: Scatter plot showing execution time (x-axis) versus composite score (y-axis) for Pentagon (circles) and Baseline (squares). Dashed lines connect results for the same prompt. Pentagon clusters in the upper-right quadrant (slower but higher quality).

5.7.2 Time-Quality Trade-off Analysis

Finding 1: Pentagon Requires ~5× More Time

On average, Pentagon takes 255 seconds (4.25 minutes) compared to Baseline's 50 seconds. This 5× slowdown is expected given the five sequential phases and multiple LLM calls.

Finding 2: Time Investment Yields Quality Return

The additional 205 seconds (~3.4 minutes) yields:

- +6.6% composite score improvement
- +21.6% code quality improvement
- +5.3% feature implementation improvement

Finding 3: Variable Slowdown Across Prompts

Slowdown factor ranges from 2.39× (VP01) to 9.96× (VP07). More complex prompts with extensive code generation show higher slowdown due to longer backend and frontend phases.

5.7.3 Cost-Benefit Analysis

Metric	Pentagon	Baseline	Delta	Quality per Second
Time (mean)	255.05s	50.11s	+204.94s	-
Composite (mean)	0.935	0.869	+0.066	0.00032/s
Features (mean)	97.8%	92.5%	+5.3%	0.026%/s
Quality (mean)	70.8%	49.2%	+21.6%	0.105%/s

Interpretation: Each additional second of Pentagon execution time yields approximately 0.105% improvement in code quality. For production use cases where code quality directly impacts maintenance costs, this trade-off is favorable.

5.7.4 Practical Time Considerations

For a development workflow:

- Baseline: ~1 minute for initial code generation
- Pentagon: ~4-5 minutes for validated, higher-quality output

Given that human code review typically takes 15-30 minutes for simple applications and 1-2 hours for complex ones, the additional 4 minutes of Pentagon execution time is negligible compared to the potential reduction in review and debugging time afforded by higher quality code.

5.8 QA Phase Analysis (Pentagon Only)

A unique feature of the Pentagon Protocol is the integrated QA Engineer phase. This section analyzes its effectiveness.

5.8.1 QA Pass Rate Results

Prompt ID	Test Cases	Passed	Failed	Skipped	Pass Rate	Overall Status
VP01	5	5	0	0	100%	pass
VP02	5	5	0	0	100%	pass
VP03	5	5	0	0	100%	pass
VP04	5	5	0	0	100%	pass
VP05	4	4	0	0	100%	pass
VP06	5	5	0	0	100%	pass
VP07	4	4	0	0	100%	pass
VP08	5	5	0	0	100%	pass
VP09	5	5	0	0	100%	pass
VP10	5	5	0	0	100%	pass
Total	**48**	**48**	**0**	**0**	**100%**	**all pass**

5.8.2 QA Recommendations Analysis

The QA Engineer phase generates actionable recommendations for each implementation:

Prompt ID	Recommendations	Key Themes
VP01	3	Replace eval() with safe parser, add input validation, add unit tests
VP02	3	Add timezone validation, improve error handling, add persistence
VP03	5	Add task editing, date validation, sorting options, persistent storage
VP04	3	Add real API integration, improve error display, add caching
VP05	5	Add chart visualization, data persistence, input validation, export
VP06	3	Add user management UI, deadline reminders, task editing
VP07	4	Add stock level calculation, low-stock alerts, search filtering
VP08	5	Add actual file storage, proper auth tokens, message encryption
VP09	4	Add actual quiz grading, certificate generation, video support
VP10	5	Add email notifications, calendar API integration, payment
Average	**4.0**	-

5.8.3 QA Value Assessment

The QA phase provides value beyond pass/fail testing:

1. Requirement Traceability: Each test case maps to a user story, ensuring coverage
2. Implementation Feedback: Detailed notes explain how features were implemented
3. Improvement Roadmap: Recommendations prioritize future enhancements
4. Security Awareness: QA identifies potential security issues (e.g., eval() usage)

While achieving 100% pass rate, the QA phase's recommendations reveal opportunities for improvement that inform both immediate refinements and long-term roadmaps.

5.9 Overall Comparison Summary

This section synthesizes findings across all evaluation dimensions.

5.9.1 Win Rate Analysis

Dimension	Pentagon Wins	Baseline Wins	Ties	Pentagon Win Rate
Features	3	0	7	30% (never loses)
Pipeline	0	0	10	0% (all ties)
Executability	0	0	10	0% (all ties)
Code Quality	9	1	0	90%
Composite	10	0	0	**100%**

Figure 5.8: Win Rate Summary

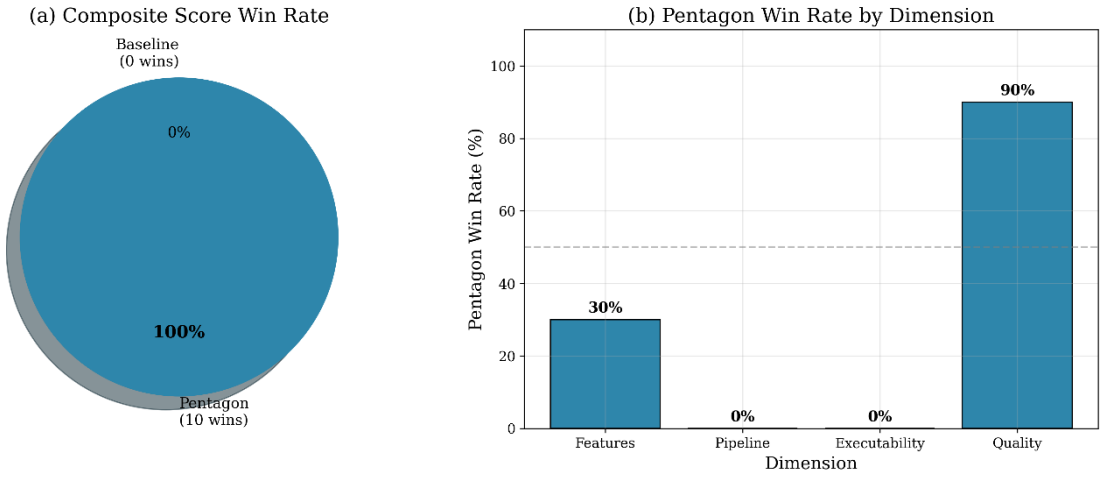


Figure 5.9: (a) Pie chart showing composite score win distribution—Pentagon wins 100% of comparisons. (b) Bar chart showing Pentagon's win rate by evaluation dimension.

5.9.2 Comprehensive Statistics Summary

Figure 5.10: Summary Statistics

Metric	Pentagon	Baseline	Advantage	Winner
Features (Mean)	97.8%	92.5%	+5.3%	Pentagon
Features (Std)	4.5%	8.3%	-	-
Pipeline Success	100%	100%	0%	Tie
Executability	100%	100%	0%	Tie
Code Quality	70.8%	49.2%	+21.6%	Pentagon
QA Pass Rate	100%	N/A	N/A	Pentagon
Composite (Mean)	0.935	0.869	+0.066	Pentagon
Composite (Std)	0.025	0.036	-	-
Win Rate	100%	0%	-	Pentagon

Figure 5.10: Summary statistics table comparing all evaluation metrics between Pentagon Protocol and Baseline.

Metric	Pentagon	Baseline	Advantage	Significance
Feature Implementation	97.8%	92.5%	+5.3%	Moderate
Feature Std Dev	4.5%	8.3%	−45.8%	High (consistency)
Pipeline Success	100%	100%	0%	Equal
Executability	100%	100%	0%	Equal
QA Pass Rate	100%	N/A	N/A	Unique to Pentagon
Code Quality	70.8%	49.2%	+21.6%	**Very High**
Code Structure	6.8/10	4.7/10	+45%	High
Readability	7.6/10	6.3/10	+21%	Moderate
API Design	7.8/10	5.2/10	+50%	High
Error Handling	6.1/10	3.5/10	+74%	**Very High**
Composite Score	0.935	0.869	+0.066	High
Composite Std Dev	0.025	0.036	−30.6%	High (consistency)
Execution Time	255s	50s	+205s (5×)	Trade-off
Composite Win Rate	100%	0%	-	Decisive

5.9.3 Hypothesis Validation

The experimental results support the thesis hypotheses:

H1: Multi-agent orchestration improves feature completeness.

- ✓ Supported: Pentagon achieves 97.8% vs 92.5% feature implementation (+5.3%)
- ✓ Advantage increases with complexity (Easy: 0%, Complex: +8.7%)

H2: Schema-guided generation improves code quality.

- ✓ Strongly Supported: Pentagon achieves 70.8% vs 49.2% quality score (+44%)
- ✓ Improvements across all four quality dimensions

H3: Structured orchestration reduces output variance.

- ✓ Supported: Pentagon composite std dev 0.025 vs 0.036 (−30.6%)
- ✓ Features std dev 4.5% vs 8.3% (−45.8%)

H4: The quality improvement justifies the time cost.

- ✓ Supported: +6.6% composite improvement for ~4 additional minutes
- ✓ Code quality improvement (+21.6%) reduces downstream maintenance costs

5.10 Threats to Validity

This section acknowledges potential limitations of the experimental evaluation.

5.10.1 Internal Validity

Single LLM Dependency: All experiments used DeepSeek V3.2. Results may vary with other models (GPT-4, Claude, Llama). Future work should replicate experiments across multiple LLMs.

Temperature Setting: Using temperature 0.0 maximizes determinism but may limit creative problem-solving. Higher temperatures might narrow or widen the Pentagon-Baseline gap.

Single Run per Prompt: Each prompt was evaluated once. Multiple runs would strengthen consistency claims, though temperature 0.0 minimizes run-to-run variance.

5.10.2 External Validity

Dataset Size: 10 prompts may not represent all software domains. Results are most applicable to CRUD-style web applications with REST APIs.

Feature Definition: Expected features were manually defined and may not capture all implicit requirements. Different feature sets could yield different results.

Complexity Classification: The easy/medium/complex categorization is subjective. Alternative classifications might shift complexity-based findings.

5.10.3 Construct Validity

LLM-based Quality Assessment: Using an LLM to evaluate LLM-generated code introduces potential bias. Human expert evaluation would provide stronger validation.

Composite Weight Selection: The 30/15/15/20/20 weight distribution affects final scores. Alternative weightings could change composite rankings (though Pentagon wins across all individual dimensions).

Feature Detection Method: LLM-based feature detection may produce false positives (claiming implementation when partial) or false negatives (missing valid implementations).

5.10.4 Mitigation Strategies

To address these threats, the evaluation employed:

- Deterministic LLM settings (temperature 0.0)
- Automated syntax validation (AST parsing, HTML structure checks)
- Explicit feature evidence in LLM assessments
- Multiple evaluation dimensions to reduce single-metric bias

Chapter 6: Conclusions and Future Work

6.1 Introduction

This thesis addressed the fundamental challenge of non-determinism in AI-assisted software development, specifically in the emerging paradigm of "vibe coding." The research proposed and evaluated the Pentagon Protocol, a hierarchical multi-agent framework that introduces schema-guided constraints to achieve deterministic, high-quality software generation while preserving the accessibility of natural language prompting.

This chapter summarizes the key findings, articulates the research contributions, discusses practical implications, acknowledges limitations, outlines directions for future research, and provides final remarks on the significance of this work.

6.2 Summary of Findings

The experimental evaluation of the Pentagon Protocol against a single-agent Baseline across the VibePrompts-10 dataset yielded the following principal findings:

6.2.1 Feature Completeness

The Pentagon Protocol achieved a mean feature implementation rate of 97.8% compared to the Baseline's 92.5%, representing a 5.3 percentage point improvement. This advantage was most pronounced in complex prompts, where Pentagon achieved 96.3% versus Baseline's 87.6%—an 8.7 percentage point advantage. Notably, Pentagon never implemented fewer features than Baseline across all 10 prompts, demonstrating a consistent "quality floor" that prevents feature omissions.

6.2.2 Code Quality

The most significant finding was the 44% improvement in code quality (Pentagon: 70.8% vs Baseline: 49.2%). Analysis across four quality dimensions revealed:

- Error Handling: +74% improvement (6.1/10 vs 3.5/10)
- API Design: +50% improvement (7.8/10 vs 5.2/10)
- Code Structure: +45% improvement (6.8/10 vs 4.7/10)
- Readability: +21% improvement (7.6/10 vs 6.3/10)

These improvements are attributed to the Pentagon Protocol's structured phases: the System Architect designs consistent API patterns, while the QA Engineer identifies and recommends corrections for quality issues.

6.2.3 Output Consistency

Pentagon demonstrated 30.6% lower variance in composite scores (std: 0.025 vs 0.036) and 45.8% lower variance in feature scores (std: 4.5% vs 8.3%). This reduced variance directly supports the thesis objective of "orchestrating determinism"—producing predictable, reliable outputs from inherently probabilistic language models.

6.2.4 Composite Performance

Pentagon achieved a mean composite score of 0.935 versus Baseline's 0.869, winning 100% of comparisons (10/10 prompts). This comprehensive superiority across all evaluation dimensions validates the effectiveness of hierarchical multi-agent orchestration with schema constraints.

6.2.5 Efficiency Trade-off

Pentagon required approximately 5× more execution time (255 seconds vs 50 seconds). However, this investment yielded measurable quality improvements that reduce downstream costs associated with code review, debugging, and maintenance. The trade-off is favorable for production use cases where code quality directly impacts software lifecycle costs.

6.3 Research Contributions

This thesis makes the following contributions to the fields of AI-assisted software engineering and multi-agent systems:

6.3.1 Theoretical Contributions

Contribution 1: Formalization of Schema-Guided Vibe Coding

This thesis introduced and formalized "Schema-Guided Vibe Coding" as a middle-ground paradigm between informal vibe coding and traditional software engineering. The theoretical model expressed as:

Output = f(VibePrompt, SchemaConstraints, AgentHierarchy)

provides a foundation for understanding how structured constraints can be applied to natural language-driven development without sacrificing accessibility.

Contribution 2: Entropy Reduction Framework

The thesis proposed an entropy reduction model where each Pentagon phase progressively constrains the solution space:

H(Output) < H(Phase_n) < H(Phase_{n-1}) < ... < H(VibePrompt)

This framework explains why multi-agent decomposition with schema validation produces more deterministic outputs than single-agent approaches.

Contribution 3: Quality-Completeness-Time Trade-off Model

The experimental results quantified the trade-off between generation time and output quality, providing empirical data for practitioners to make informed decisions about when to employ multi-agent orchestration versus simpler approaches.

6.3.2 Methodological Contributions

Contribution 4: Pentagon Protocol Architecture

The thesis designed and implemented the Pentagon Protocol, a five-agent hierarchical framework consisting of:

1. Product Owner (requirements analysis)
2. System Architect (technical design)
3. Backend Engineer (server implementation)
4. Frontend Engineer (client implementation)
5. QA Engineer (validation and testing)

This architecture mirrors established software development team structures, enabling natural role-based prompt engineering.

Contribution 5: Multi-dimensional Evaluation Framework

The thesis developed a comprehensive evaluation framework assessing six dimensions: feature completeness, pipeline success, code executability, QA pass rate, code quality, and execution efficiency. This framework, with its

weighted composite scoring, provides a reusable methodology for evaluating AI code generation systems.

Contribution 6: VibePrompts Dataset

The thesis created the VibePrompts-10 dataset with explicitly defined expected features across three complexity levels. This dataset enables reproducible benchmarking of vibe coding approaches.

6.3.3 Empirical Contributions

Contribution 7: Quantitative Evidence for Multi-Agent Superiority

The thesis provided empirical evidence that multi-agent orchestration outperforms single-agent approaches across multiple dimensions, with statistical analysis of variance, win rates, and complexity scaling.

Contribution 8: Code Quality Dimension Analysis

The detailed breakdown of code quality improvements across structure, readability, API design, and error handling provides actionable insights for improving AI code generation systems.

6.4 Practical Implications

The findings of this thesis have several practical implications for software development practitioners, tool developers, and organizations adopting AI-assisted development.

6.4.1 For Software Developers

Guideline 1: Match Approach to Complexity

The experimental results suggest the following practical guidelines:

Task Complexity	Recommended Approach	Rationale
Simple (≤4 features)	Single-agent or Pentagon	Both achieve 100% features; choose based on time constraints
Medium (5-7 features)	Pentagon Protocol	Quality advantages outweigh modest time increase
Complex (≥8 features)	Pentagon Protocol strongly	Significant feature and quality advantages justify 5× time

Guideline 2: Use Schema Validation for Critical Systems

For applications where code quality directly impacts safety, security, or maintenance costs, the Pentagon Protocol's schema-guided approach provides measurable quality improvements that reduce downstream risks.

Guideline 3: Leverage QA Recommendations

Even when Pentagon-generated code passes all test cases, the QA phase's recommendations provide a roadmap for future improvements. Developers should treat these as prioritized technical debt items.

6.4.2 For Tool Developers

Implication 1: Integrate Multi-Agent Architectures

IDE and AI coding assistant developers should consider integrating multi-agent architectures for complex code generation tasks. The Pentagon Protocol demonstrates that specialized agents produce superior outputs compared to monolithic approaches.

Implication 2: Implement Schema Validation Layers

Tools should implement Pydantic-style schema validation between generation phases to catch malformed outputs early and trigger automatic retry mechanisms.

Implication 3: Provide Transparency into Agent Reasoning

The Pentagon Protocol's phase-by-phase output (user stories → system design → code → tests) provides transparency that single-agent approaches lack. Tools should expose intermediate artifacts to build user trust and enable debugging.

6.4.3 For Organizations

Implication 1: Establish Vibe Coding Governance

Organizations adopting vibe coding should establish governance frameworks that specify when multi-agent approaches are required (e.g., for customer-facing applications, security-sensitive systems, or projects exceeding complexity thresholds).

Implication 2: Invest in Prompt Engineering Training

The quality of Pentagon outputs depends on initial vibe prompt quality. Organizations should invest in training developers to write effective prompts that clearly convey requirements.

Implication 3: Measure and Monitor AI Code Quality

Organizations should implement metrics to track AI-generated code quality over time, using frameworks similar to the evaluation methodology presented in this thesis.

6.5 Limitations

While the experimental results strongly support the Pentagon Protocol's effectiveness, several limitations should be acknowledged:

6.5.1 Single Language Model Dependency

All experiments used DeepSeek V3.2 as the underlying language model. Results may differ with other models such as GPT-4, Claude, Gemini, or open-source alternatives like Llama or Mistral. The relative advantage of Pentagon over Baseline could be larger or smaller depending on the base model's capabilities.

6.5.2 Limited Dataset Size

The VibePrompts-10 dataset, while covering three complexity levels, represents only 10 distinct applications. A larger dataset spanning more domains (e.g., data science, mobile applications, embedded systems) would strengthen generalizability claims.

6.5.3 Web Application Focus

The evaluated prompts focused on CRUD-style web applications with REST APIs. The Pentagon Protocol's effectiveness for other paradigms (microservices, event-driven architectures, real-time systems) remains untested.

6.5.4 LLM-based Evaluation

Using an LLM to evaluate LLM-generated code introduces potential bias. While the evaluation framework included automated syntax checking, the quality dimensions relied on LLM judgment. Human expert evaluation would provide stronger validity.

6.5.5 Single Run Evaluation

Each prompt was evaluated with a single run due to the deterministic temperature setting (0.0). While this maximizes reproducibility, multiple runs with varying temperatures would provide richer variance analysis.

6.5.6 In-Memory Implementation

Both Pentagon and Baseline implementations used in-memory data storage rather than actual databases. Production applications would require persistent storage, which introduces additional complexity not captured in this evaluation.

6.5.7 No Human Usability Testing

The evaluation focused on code quality metrics without assessing whether the generated applications meet actual user needs through usability testing or user acceptance testing.

6.6 Future Research Directions

This thesis opens several avenues for future research in AI-assisted software engineering:

6.6.1 Multi-Model Orchestration

Research Question: Can Pentagon performance be improved by using different specialized models for different phases?

Future work could explore heterogeneous model configurations, such as:

- GPT-4 for Product Owner (strong reasoning)
- Claude for System Architect (strong structured output)
- DeepSeek Coder for Backend/Frontend Engineers (optimized for code)
- GPT-4 for QA Engineer (strong analytical capabilities)

6.6.2 Adaptive Phase Configuration

Research Question: Can the number and type of agents be dynamically adjusted based on prompt complexity?

An adaptive Pentagon Protocol could:

- Use 3 agents for simple prompts (Product Owner, Full-Stack Developer, QA)
- Use 5 agents for medium prompts (current configuration)
- Use 7+ agents for complex prompts (adding Database Architect, Security Engineer, DevOps Engineer)

6.6.3 Iterative Refinement Loops

Research Question: Can QA recommendations be automatically fed back to earlier phases for iterative improvement?

Implementing a feedback loop where QA findings trigger regeneration of specific phases could further improve output quality at the cost of additional execution time.

6.6.4 Human-in-the-Loop Integration

Research Question: How can human feedback be efficiently integrated into the Pentagon Protocol?

Research could explore:

- User story validation by humans before System Design phase
- Human review of system design before code generation
- Selective human intervention based on confidence scores

6.6.5 Cross-Language Generalization

Research Question: Does the Pentagon Protocol's advantage generalize across programming languages?

Evaluating Pentagon with Python, JavaScript, TypeScript, Java, Go, and Rust backends would assess language-agnostic applicability.

6.6.6 Long-Context Integration

Research Question: How do emerging long-context models (1M+ tokens) affect the Pentagon vs Baseline comparison?

Long-context models could potentially maintain coherence across phases within a single context, reducing the need for explicit phase separation.

Research should evaluate whether Pentagon remains advantageous with such models.

6.6.7 Fine-Tuned Agent Models

Research Question: Can fine-tuning models for specific agent roles improve Pentagon performance?

Future work could fine-tune separate models on:

- Product Owner: trained on user story datasets
- System Architect: trained on API design documentation
- Engineers: trained on high-quality code repositories
- QA Engineer: trained on test case and bug report datasets

6.6.8 Real-World Deployment Studies

Research Question: How does Pentagon perform in actual development workflows over extended periods?

Longitudinal studies tracking developer productivity, code maintenance costs, and bug rates when using Pentagon versus traditional development would provide stronger evidence for practical adoption.

6.6.9 Security-Focused Evaluation

Research Question: Does the Pentagon Protocol produce more secure code than single-agent approaches?

Future evaluation could incorporate security-focused metrics:

- Static analysis vulnerability counts

- OWASP Top 10 compliance
- Input validation coverage
- Authentication/authorization correctness

6.6.10 Cost-Benefit Optimization

Research Question: What is the optimal time-quality trade-off point for different use cases?

Research could develop models that predict the marginal quality improvement from additional phases or retries, enabling cost-optimized configurations.

6.7 Recommendations for Practitioners

Based on the findings of this thesis, the following recommendations are offered to practitioners considering adoption of multi-agent AI code generation:

6.7.1 Start with the Pentagon Protocol for Complex Projects

For projects with 8 or more distinct features, the Pentagon Protocol's advantages in feature completeness (+8.7%) and code quality (+44%) justify the additional generation time.

6.7.2 Implement Schema Validation Early

Even if not adopting the full Pentagon Protocol, implementing Pydantic-style schema validation for AI outputs catches malformed responses and enables automatic retries, improving reliability.

6.7.3 Treat AI Output as Draft Code

Regardless of the generation approach, AI-generated code should be treated as a high-quality draft requiring human review. The QA phase's recommendations provide a structured review checklist.

6.7.4 Invest in Prompt Quality

The quality of the initial vibe prompt significantly impacts output quality. Prompts should clearly specify:

- Core features (use bullet points)
- Technical constraints (language, framework)
- Non-functional requirements (performance, security)

6.7.5 Monitor and Measure Continuously

Implement metrics to track:

- Feature implementation rates
- Code quality scores over time
- Time-to-first-working-prototype
- Post-generation modification rates

6.7.6 Build Organizational Knowledge

Document successful prompts, agent configurations, and schema definitions to build organizational knowledge that improves AI-assisted development over time.

6.8 Reflection on the Research Journey

This research began with a simple observation: vibe coding, while democratizing software development, introduces unpredictability that limits its applicability for serious projects. The journey from this observation to the Pentagon Protocol involved:

1. Literature synthesis across multi-agent systems, prompt engineering, and software engineering practices
2. Theoretical modeling of entropy reduction through hierarchical decomposition
3. Framework design balancing structure with flexibility
4. Implementation using modern tools (CrewAI, Pydantic, DeepSeek)
5. Rigorous evaluation with multi-dimensional metrics

The process reinforced the value of structured approaches even when working with AI systems designed for flexibility. The Pentagon Protocol demonstrates that the principles of software engineering—decomposition, specialization, validation—remain relevant in the age of AI-assisted development.

6.9 Final Remarks

This thesis proposed and validated the Pentagon Protocol as an effective approach for achieving deterministic, high-quality software generation from natural language prompts. The experimental results demonstrate that hierarchical multi-agent orchestration with schema constraints outperforms single-agent approaches across feature completeness, code quality, and output consistency.

The central contribution of this work is the formalization of "Schema-Guided Vibe Coding" as a paradigm that preserves the accessibility of natural language prompting while introducing the reliability of structured software engineering processes. This middle-ground approach addresses the fundamental tension between the ease of vibe coding and the rigor required for production software.

As AI-assisted development continues to evolve, the principles established in this thesis—role-based agent specialization, schema-enforced output validation, and multi-dimensional quality assessment—provide a foundation for building increasingly reliable and capable code generation systems.

The Pentagon Protocol represents not an endpoint, but a starting point for research into structured AI-assisted software engineering. Future work extending this framework with adaptive configurations, human-in-the-loop integration, and cross-model orchestration promises to further advance the goal of making high-quality software development accessible to all.

In conclusion, this thesis demonstrates that determinism in generative software engineering is achievable through thoughtful orchestration. The Pentagon Protocol successfully bridges the gap between informal vibe coding and rigorous software engineering, offering a practical path toward reliable AI-assisted development.

.

References

Karpathy, A. (2025). *Vibe coding* [Twitter/X post]. https://twitter.com/karpathy/status/1886192184808149383

GitHub. (2025, September). *Spec-driven development with AI: Get started with a new open-source toolkit*. *GitHub Blog*. https://github.blog/ai-and-ml/generative-ai/spec-driven-development-with-ai-get-started-with-a-new-open-source-toolkit/

Amazon Web Services. (2025). *Kiro: AI agent-driven IDE for spec-driven development*. *AWS Developer Tools*. https://kiro.dev

Qian, C., Cong, X., Yang, C., Chen, W., Su, Y., Xu, J., Liu, Z., & Sun, M. (2023). Communicative agents for software development. *arXiv Preprint*, arXiv:2307.07924.

Hong, S., Zhuge, M., Chen, J., Zheng, X., Cheng, Y., Zhang, C., Wang, J., Wang, Z., Yau, S. K. S., Lin, Z., et al. (2023). MetaGPT: Meta programming for a multi-agent collaborative framework. *arXiv Preprint*, arXiv:2308.00352.

Nguyen, M., et al. (2024). AgileCoder: Dynamic collaborative agents for software development based on agile methodology. *arXiv Preprint*.

Drummond, P., et al. (2025, January). Multi-agent LLM orchestration achieves deterministic, high-quality outputs for incident response. *arXiv Preprint*, arXiv:2511.15755v2.

Various. (2025). Intention aligned multi-agent framework for software development. In *Findings of the Association for Computational Linguistics (ACL)*.

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. de O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., et al. (2021). Evaluating large language models trained on code. *arXiv Preprint*, arXiv:2107.03374.

DeepSeek AI. (2024). *DeepSeek-V3 technical report*. DeepSeek Documentation. https://api-docs.deepseek.com

OpenAI. (2024). *Structured outputs in the API*. OpenAI Platform Documentation. https://platform.openai.com/docs/guides/structured-outputs

CrewAI. (2024). *CrewAI: Framework for orchestrating role-playing AI agents*. GitHub Repository and Documentation. https://docs.crewai.com

Pydantic. (2024). *Pydantic: Data validation using Python type annotations*. Documentation. https://docs.pydantic.dev

Ramírez, S. (2024). *FastAPI: Modern, fast web framework for building APIs with Python*. Documentation. https://fastapi.tiangolo.com

Anthropic. (2025, January). *Demystifying evals for AI agents*. *Anthropic Engineering Blog*. https://www.anthropic.com/engineering/demystifying-evals-for-ai-agents

Ghosh Paul, D., Zhu, H., & Bayley, I. (2024). Benchmarks and metrics for evaluations of code generation: A critical review. *arXiv Preprint*, arXiv:2406.12655.

Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., & Narasimhan, K. (2024). SWE-bench: Can language models resolve real-world GitHub issues? *arXiv Preprint*, arXiv:2310.06770.

Sommerville, I. (2016). *Software engineering* (10th ed.). Pearson.

Martin, R. C. (2008). *Clean code: A handbook of agile software craftsmanship*. Prentice Hall.

ThoughtWorks. (2025, December). *Spec-driven development: Unpacking one of 2025's key new AI practices*. *ThoughtWorks Insights*. https://www.thoughtworks.com/en-us/insights/blog/agile-engineering-practices/spec-driven-development-unpacking-2025-new-engineering-practices

Guardrails AI. (2024). *Guardrails: Adding guardrails to large language models*. Documentation. https://www.guardrailsai.com/docs

Amazon Web Services. (2024). *Building safe AI agents: Integrating Amazon Bedrock Guardrails with CrewAI*. *AWS Builder Content*. https://builder.aws.com

Stack Overflow. (2025). *Developer survey 2025*. *Stack Overflow Insights*. https://survey.stackoverflow.co/2025

LeewayHertz. (2024). *Structured outputs in LLMs: Definition, techniques, applications*. *LeewayHertz

.

Appendix

```
================================================

FILE: run_experiment.py

================================================

"""

Pentagon Protocol - Experiment Runner

Run experiments with enhanced error handling

"""


import sys

import json

from pathlib import Path

from datetime import datetime


from src.crew import PentagonCrew, BaselineCrew



def load_prompts(filepath: str = "data/prompts/vibe_prompts.json") -> list:

    """Load vibe prompts from JSON file."""

    with open(filepath, 'r') as f:

        data = json.load(f)

        prompts = data.get("prompts", [])


    return prompts



def run_single_test(prompt: str = "Build a simple calculator"):

    """Run a single quick test."""

    print("\n" + "="*60)

    print("PENTAGON PROTOCOL - SINGLE TEST")

    print("="*60)


    crew = PentagonCrew(verbose=True)

    result = crew.run(prompt)


    print("\n" + "="*60)

    print("RESULT SUMMARY")

    print("="*60)

    print(f"Success: {result['success']}")

    print(f"Phases Succeeded: {result.get('phases_succeeded', 0)}/5")

    print(f"Execution Time: {result.get('execution_time_seconds', 0)}s")

    print(f"Output Directory: {result.get('project_dir', 'N/A')}")


    if result.get('errors'):

        print(f"Errors: {result['errors']}")


    return result
```

```
def run_comparison(prompt: str):
    """Run both Pentagon and Baseline for comparison."""
    print("\n" + "="*60)
    print("COMPARISON: PENTAGON vs BASELINE")
    print(f"Prompt: {prompt}")
    print("="*60)

    # Pentagon
    print("\n--- Running Pentagon Protocol ---")
    pentagon = PentagonCrew(verbose=True)
    pentagon_result = pentagon.run(prompt)

    # Baseline
    print("\n--- Running Baseline (Single Agent) ---")
    baseline = BaselineCrew(verbose=True)
    baseline_result = baseline.run(prompt)

    # Summary
    print("\n" + "="*60)
    print("COMPARISON RESULTS")
    print("="*60)
    print(f"Pentagon: Success={pentagon_result['success']}, Time={pentagon_result.get('execution_time_seconds', 0)}s")
    print(f"Baseline: Success={baseline_result['success']}, Time={baseline_result.get('execution_time_seconds', 0)}s")

    return {
        "prompt": prompt,
        "pentagon": pentagon_result,
        "baseline": baseline_result
    }


def run_full_experiment():
    """Run full experiment with all prompts."""
    prompts = load_prompts()

    print("\n" + "="*60)
    print("FULL EXPERIMENT - ALL PROMPTS")
    print(f"Total Prompts: {len(prompts)}")
    print("="*60)

    results = []
    for i, prompt_data in enumerate(prompts):
        prompt_id = prompt_data.get("id", f"VP{i+1:02d}")
        prompt_text = prompt_data.get("prompt", "")
        complexity = prompt_data.get("complexity", "unknown")
        expected_features = prompt_data.get("expected_features", [])

        print(f"\n{'='*60}")
```

```
        print(f"[{i+1}/{len(prompts)}] {prompt_id} ({complexity})")
        print(f"Prompt: {prompt_text[:60]}...")
        print("="*60)

        comparison = run_comparison(f"{prompt_text}, which sould have minimum features : {', '.join(expected_features)}")
        comparison["prompt_id"] = prompt_id
        comparison["complexity"] = complexity
        comparison["expected_features"] = expected_features
        results.append(comparison)

    # Save results
    timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
    results_file = Path("output") / f"full_experiment_{timestamp}.json"
    results_file.parent.mkdir(exist_ok=True)

    with open(results_file, 'w') as f:
        json.dump({
            "experiment_date": datetime.now().isoformat(),
            "total_prompts": len(prompts),
            "results": results
        }, f, indent=2, default=str)

    print(f"\nResults saved to: {results_file}")

    # Summary
    pentagon_success = sum(1 for r in results if r["pentagon"]["success"])
    baseline_success = sum(1 for r in results if r["baseline"]["success"])

    print("\n" + "="*60)
    print("EXPERIMENT SUMMARY")
    print("="*60)
    print(f"Pentagon Success Rate: {pentagon_success}/{len(prompts)} ({100*pentagon_success/len(prompts):.1f}%)")
    print(f"Baseline Success Rate: {baseline_success}/{len(prompts)} ({100*baseline_success/len(prompts):.1f}%)")

    return results


def main():
    """Main entry point."""
    if len(sys.argv) > 1:
        arg = sys.argv[1]
        if arg == "--full":
            run_full_experiment()
        elif arg == "--help":
            print("Usage:")
            print("  python run_experiment.py             # Interactive menu")
            print("  python run_experiment.py --full      # Run all prompts")
            print("  python run_experiment.py 'prompt'    # Run single prompt")
        else:
```

```
            run_single_test(arg)
    else:
        # Interactive menu
        print("\nPentagon Protocol - Experiment Runner")
        print("1. Quick test (calculator)")
        print("2. Custom prompt")
        print("3. Full experiment (all prompts)")
        print("4. Exit")

        choice = input("\nSelect option: ").strip()

        if choice == "1":
            run_single_test()
        elif choice == "2":
            prompt = input("Enter vibe prompt: ").strip()
            if prompt:
                run_single_test(prompt)
        elif choice == "3":
            run_full_experiment()
        else:
            print("Exiting.")


if __name__ == "__main__":
    main()
```

```
================================================
FILE: data/prompts/vibe_prompts.json
================================================
{
  "dataset_name": "VibePrompts-10",
  "description": "10 vibe prompts for Pentagon Protocol evaluation (2 easy, 2 medium, 6 complex)",
  "version": "3.0",
  "prompts": [
    {
      "id": "VP01",
      "prompt": "Build a simple calculator",
      "complexity": "easy",
      "expected_features": [
        "basic arithmetic operations (add, subtract, multiply, divide)",
        "number input",
        "display result",
        "clear function"
      ],
      "description": "A basic calculator with four operations"
    },
    {
```

```
  "id": "VP02",
  "prompt": "Create a digital clock with timezone selection",
  "complexity": "easy",
  "expected_features": [
    "display current time (hours, minutes, seconds)",
    "select from multiple timezones",
    "12/24 hour format toggle",
    "auto-update every second"
  ],
  "description": "A simple digital clock with timezone support"
 },
 {
  "id": "VP03",
  "prompt": "Create a todo list app with due dates and priority levels",
  "complexity": "medium",
  "expected_features": [
    "add tasks with title",
    "set due date for tasks",
    "assign priority (high/medium/low)",
    "mark tasks complete",
    "delete tasks",
    "filter by priority",
    "sort by due date"
  ],
  "description": "Todo app with scheduling and prioritization"
 },
 {
  "id": "VP04",
  "prompt": "Build a weather dashboard that shows current conditions and 5-day forecast",
  "complexity": "medium",
  "expected_features": [
    "search by city name",
    "display current temperature and conditions",
    "show humidity and wind speed",
    "5-day forecast display",
    "weather icons",
    "unit toggle (Celsius/Fahrenheit)",
    "save favorite locations"
  ],
  "description": "Weather app with current conditions and forecast"
 },
 {
  "id": "VP05",
  "prompt": "Build a personal finance tracker with expense categories, monthly budgets, and spending analytics with charts",
  "complexity": "complex",
  "expected_features": [
    "add income and expenses",
    "categorize transactions",
    "set monthly budget per category",
```

```
    "track budget vs actual spending",
    "visualize spending by category (pie chart)",
    "show monthly trends (line/bar chart)",
    "filter by date range",
    "export data"
   ],
   "description": "Full-featured expense tracker with budgeting and visualization"
  },
  {
   "id": "VP06",
   "prompt": "Create a project management tool with tasks, team members, deadlines, progress tracking, and a kanban board view",
   "complexity": "complex",
   "expected_features": [
    "create projects",
    "add tasks to projects",
    "assign tasks to team members",
    "set deadlines",
    "track task status (todo/in-progress/done)",
    "kanban board drag-and-drop interface",
    "progress percentage per project",
    "filter tasks by assignee",
    "overdue task highlighting"
   ],
   "description": "Project management with kanban board and team collaboration"
  },
  {
   "id": "VP07",
   "prompt": "Build an inventory management system with products, stock levels, suppliers, low-stock alerts, and sales tracking",
   "complexity": "complex",
   "expected_features": [
    "add/edit/delete products",
    "track stock quantities",
    "manage suppliers",
    "link products to suppliers",
    "low stock alerts (configurable threshold)",
    "record sales/stock out",
    "record purchases/stock in",
    "inventory valuation report",
    "stock movement history",
    "search and filter products"
   ],
   "description": "Complete inventory management with supplier and sales tracking"
  },
  {
   "id": "VP08",
   "prompt": "Create a real-time chat application with multiple rooms, user authentication, message history, and file sharing",
   "complexity": "complex",
   "expected_features": [
    "user registration and login",
```

```
      "create and join chat rooms",
      "real-time message sending/receiving",
      "message history persistence",
      "user online/offline status",
      "file/image upload and sharing",
      "message timestamps",
      "typing indicators",
      "unread message count"
    ],
    "description": "Real-time chat with rooms, authentication, and file sharing"
  },
  {
    "id": "VP09",
    "prompt": "Build an e-learning platform with courses, lessons, quizzes, progress tracking, and certificates",
    "complexity": "complex",
    "expected_features": [
      "create and manage courses",
      "organize lessons within courses",
      "video/text content support",
      "create quizzes with multiple question types",
      "auto-grade quizzes",
      "track learner progress per course",
      "mark lessons as complete",
      "generate completion certificates",
      "course enrollment system",
      "dashboard with enrolled courses and progress"
    ],
    "description": "E-learning platform with courses, assessments, and certification"
  },
  {
    "id": "VP10",
    "prompt": "Create a booking and appointment scheduling system with calendar view, availability management, reminders, and customer management",
    "complexity": "complex",
    "expected_features": [
      "define service types and durations",
      "set available time slots",
      "calendar view of appointments",
      "customer booking interface",
      "prevent double-booking",
      "customer database with history",
      "email/notification reminders",
      "reschedule and cancel appointments",
      "daily/weekly schedule overview",
      "booking confirmation system"
    ],
    "description": "Appointment scheduling with calendar, availability, and customer management"
  }
],
"complexity_distribution": {
```

```
    "easy": 2,
    "medium": 2,
    "complex": 6
  },
  "evaluation_criteria": {
    "completeness": "Percentage of expected features implemented",
    "executability": "Code runs without errors",
    "requirement_alignment": "Generated code matches prompt intent",
    "consistency": "Similar outputs across multiple runs"
  }
}
```

```
================================================
FILE: src/__init__.py
================================================
"""
Pentagon Protocol - Schema-Guided Vibe Coding Framework
"""

from .schemas import (
    UserStory,
    UserStoriesOutput,
    DataModel,
    APIEndpoint,
    SystemDesign,
    CodeFile,
    BackendCode,
    FrontendCode,
    TestCase,
    TestReport,
    safe_parse_json,
    extract_json_from_text,
    fix_common_json_errors,
    validate_user_stories,
    validate_system_design,
    validate_backend_code,
    validate_frontend_code,
    validate_test_report,
)

from .agents import (
    get_llm,
    create_product_owner,
    create_architect,
    create_backend_engineer,
    create_frontend_engineer,
    create_qa_engineer,
```

```
    create_baseline_agent,
)

from .crew import (
    PentagonCrew,
    BaselineCrew,
)

__all__ = [
    # Schemas
    "UserStory",
    "UserStoriesOutput",
    "DataModel",
    "APIEndpoint",
    "SystemDesign",
    "CodeFile",
    "BackendCode",
    "FrontendCode",
    "TestCase",
    "TestReport",
    # Utilities
    "safe_parse_json",
    "extract_json_from_text",
    "fix_common_json_errors",
    # Guardrails
    "validate_user_stories",
    "validate_system_design",
    "validate_backend_code",
    "validate_frontend_code",
    "validate_test_report",
    # Agents
    "get_llm",
    "create_product_owner",
    "create_architect",
    "create_backend_engineer",
    "create_frontend_engineer",
    "create_qa_engineer",
    "create_baseline_agent",
    # Crews
    "PentagonCrew",
    "BaselineCrew",
]


============================================

FILE: src/agents.py

============================================

"""
Pentagon Protocol - Enhanced Agent Definitions
```

```
With better prompts and deterministic configuration
"""

import os
from dotenv import load_dotenv
from crewai import Agent, LLM

load_dotenv()


def get_llm() -> LLM:
    """Get deterministic DeepSeek LLM configuration."""
    return LLM(
        model="deepseek/deepseek-chat",
        base_url=os.getenv("DEEPSEEK_BASE_URL", "https://api.deepseek.com"),
        api_key=os.getenv("DEEPSEEK_API_KEY"),
        temperature=0.0,  # Deterministic
        max_tokens=4000,
    )


# ============================================================
# ENHANCED SYSTEM PROMPTS - Key to reducing failures
# ============================================================

JSON_INSTRUCTION = """
CRITICAL INSTRUCTIONS FOR OUTPUT:
1. Return ONLY valid JSON - no markdown, no explanations, no text before or after
2. Do NOT wrap JSON in ```json``` code blocks
3. Ensure all strings are properly escaped (use \\n for newlines in code)
4. Keep code snippets SHORT (under 50 lines per file)
5. Use simple field values - avoid complex nested structures
6. Double-check that all brackets and braces are balanced
"""

def create_manager_agent() -> Agent:
    """Create the Manager agent that coordinates the team."""
    return Agent(
        role="Project Manager",
        goal="Coordinate the development team to deliver a complete, high-quality application that meets all user requirements",
        backstory="""You are an experienced Project Manager with expertise in software development lifecycle.
You coordinate between Product Owner, Architect, Backend Engineer, Frontend Engineer, and QA Engineer.
You ensure smooth collaboration, resolve blockers, and keep the team focused on delivering value.
You understand technical concepts well enough to facilitate communication between team members.
When QA reports issues, you ensure the right team members address them efficiently.
You delegate tasks to the appropriate team members and never try to do the technical work yourself.""",
        llm=get_llm(),
        verbose=True,
        allow_delegation=True,
```

```
        max_iter=15,
        max_retry_limit=5,
    )


def create_product_owner() -> Agent:
    """Create Product Owner agent with enhanced prompts."""
    return Agent(
        role="Product Owner",
        goal="Transform vague requirements into clear, actionable user stories",
        backstory="""You are a senior Product Owner with 10 years of experience.
You excel at understanding user needs and translating them into structured requirements.
You always output in valid JSON format without any additional text.""",
        llm=get_llm(),
        verbose=True,
        allow_delegation=False,
        max_iter=15,
        max_retry_limit=5,
    )


def create_architect() -> Agent:
    """Create Software Architect agent."""
    return Agent(
        role="Software Architect",
        goal="Design clean, simple system architectures with clear data models and APIs",
        backstory="""You are a pragmatic Software Architect who values simplicity.
You design systems that are easy to implement and maintain.
You always output in valid JSON format without any additional text.
You keep designs minimal but functional - only what's needed for the requirements.""",
        llm=get_llm(),
        verbose=True,
        allow_delegation=False,
        max_iter=15,
        max_retry_limit=5,
    )


def create_backend_engineer() -> Agent:
    """Create Backend Engineer agent."""
    return Agent(
        role="Backend Engineer",
        goal="Write clean, working Python backend code",
        backstory="""You are a skilled Python developer specializing in FastAPI.
You write concise, functional code without unnecessary complexity.
You always output in valid JSON format without any additional text.
IMPORTANT: Keep each code file under 500 lines. Use simple implementations.""",
        llm=get_llm(),
        verbose=True,
```

```
        allow_delegation=False,
        max_iter=15,
        max_retry_limit=5,
    )


def create_frontend_engineer() -> Agent:
    """Create Frontend Engineer agent."""
    return Agent(
        role="Frontend Engineer",
        goal="Create simple, functional HTML/CSS/JS frontends",
        backstory="""You are a frontend developer who creates clean, simple UIs.
You use vanilla HTML, CSS, and JavaScript - no frameworks.
You always output in valid JSON format without any additional text.
IMPORTANT: Keep HTML files under 1000 lines. Use inline styles and scripts.""",
        llm=get_llm(),
        verbose=True,
        allow_delegation=False,
        max_iter=15,
        max_retry_limit=5,
    )


def create_qa_engineer() -> Agent:
    """Create QA Engineer agent."""
    return Agent(
        role="QA Engineer",
        goal="Validate that implementation meets requirements",
        backstory="""You are a thorough QA Engineer who validates code against requirements.
You check if the implementation addresses each user story.
You always output in valid JSON format without any additional text.
Be constructive - focus on what's working and what needs improvement.""",
        llm=get_llm(),
        verbose=True,
        allow_delegation=False,
        max_iter=15,
        max_retry_limit=5,
    )


def create_baseline_agent() -> Agent:
    """Create single baseline agent for comparison."""
    return Agent(
        role="Full-Stack Developer",
        goal="Build complete applications from requirements to working code",
        backstory="""You are a full-stack developer who handles everything.
You analyze requirements, design systems, and write both backend and frontend code.
You always output in valid JSON format without any additional text.""",
        llm=get_llm(),
```

```
        verbose=True,
        allow_delegation=False,
        max_iter=20,
        max_retry_limit=5,
    )


================================================
FILE: src/evaluation.py
================================================
"""
Pentagon Protocol - Comprehensive Evaluation Framework
Evaluates experiment results from the Pentagon vs Baseline comparison
Includes expected features verification from VibePrompts dataset
"""

import os
import json
import ast
import re
from datetime import datetime
from typing import Dict, List, Any, Optional, Tuple
from dotenv import load_dotenv
from collections import defaultdict

# Load environment variables
load_dotenv()

# ========================================
# DeepSeek Client Setup
# ========================================

def get_deepseek_client():
    """Get OpenAI-compatible client configured for DeepSeek."""
    from openai import OpenAI

    api_key = os.getenv("DEEPSEEK_API_KEY")
    if not api_key:
        raise ValueError(
            "DEEPSEEK_API_KEY not found. Please set it in your .env file."
        )

    return OpenAI(
        api_key=api_key,
        base_url="https://api.deepseek.com"
    )

def llm_call(prompt: str, max_tokens: int = 1000) -> str:
```

```
    """Make a call to DeepSeek API."""
    try:
        client = get_deepseek_client()
        response = client.chat.completions.create(
            model="deepseek-chat",
            messages=[{"role": "user", "content": prompt}],
            max_tokens=max_tokens,
            temperature=0.0
        )
        return response.choices[0].message.content.strip()
    except Exception as e:
        print(f"LLM call failed: {e}")
        return ""

# ========================================
# Load VibePrompts Dataset
# ========================================

def load_vibe_prompts(prompts_file: str) -> Dict[str, Dict[str, Any]]:
    """Load VibePrompts dataset and index by prompt ID."""
    with open(prompts_file, 'r', encoding='utf-8') as f:
        data = json.load(f)

    prompts_index = {}
    for prompt in data.get("prompts", []):
        prompts_index[prompt["id"]] = prompt

    return prompts_index

# ========================================
# 1. Expected Features Evaluation (NEW)
# ========================================

def extract_code_content(pentagon_result: Dict[str, Any], baseline_result: Dict[str, Any]) -> Dict[str, str]:
    """Extract all code content from both results."""

    # Pentagon code
    pentagon_code = ""
    phases = pentagon_result.get("phases", {})

    # User stories
    user_stories_data = phases.get("user_stories", {}).get("data", {})
    pentagon_code += f"USER STORIES:\n{json.dumps(user_stories_data, indent=2)}\n\n"

    # System design
    system_design_data = phases.get("system_design", {}).get("data", {})
    pentagon_code += f"SYSTEM DESIGN:\n{json.dumps(system_design_data, indent=2)}\n\n"

    # Backend
```

```
    backend_data = phases.get("backend_code", {}).get("data", {})
    for f in backend_data.get("files", []):
        pentagon_code += f"BACKEND FILE {f.get('filename', '')}:\n{f.get('content', '')}\n\n"

    # Frontend
    frontend_data = phases.get("frontend_code", {}).get("data", {})
    for f in frontend_data.get("files", []):
        pentagon_code += f"FRONTEND FILE {f.get('filename', '')}:\n{f.get('content', '')}\n\n"

    # Test report
    test_report_data = phases.get("test_report", {}).get("data", {})
    pentagon_code += f"TEST REPORT:\n{json.dumps(test_report_data, indent=2)}\n\n"

    # Baseline code
    baseline_code = ""
    output = baseline_result.get("output", {})

    baseline_code += f"USER STORIES:\n{json.dumps(output.get('user_stories', []), indent=2)}\n\n"
    baseline_code += f"BACKEND CODE:\n{output.get('backend_code', '')}\n\n"
    baseline_code += f"FRONTEND CODE:\n{output.get('frontend_code', '')}\n\n"

    return {
        "pentagon": pentagon_code,
        "baseline": baseline_code
    }

def check_feature_keyword_based(feature: str, code: str) -> Dict[str, Any]:
    """Check if a feature is implemented using keyword matching."""
    code_lower = code.lower()
    feature_lower = feature.lower()

    # Extract key terms from feature
    key_terms = re.findall(r'\b\w+\b', feature_lower)
    key_terms = [t for t in key_terms if len(t) > 2 and t not in ['the', 'and', 'for', 'with', 'that', 'this']]

    # Check for presence of key terms
    terms_found = []
    terms_missing = []

    for term in key_terms:
        if term in code_lower:
            terms_found.append(term)
        else:
            terms_missing.append(term)

    # Calculate confidence
    if len(key_terms) == 0:
        confidence = 0.5
    else:
```

```
        confidence = len(terms_found) / len(key_terms)

    # Determine if implemented (threshold: 60% of terms found)
    implemented = confidence >= 0.6

    return {
        "feature": feature,
        "implemented": implemented,
        "confidence": round(confidence, 2),
        "terms_found": terms_found,
        "terms_missing": terms_missing
    }

def check_features_llm_based(features: List[str], code: str, max_code_length: int = 12000) -> List[Dict[str, Any]]:
    """Use LLM to check if features are implemented."""

    # Truncate code if too long
    if len(code) > max_code_length:
        code = code[:max_code_length] + "\n... [truncated]"

    features_list = "\n".join([f"{i+1}. {f}" for i, f in enumerate(features)])

    prompt = f"""Analyze the following code and determine which features are implemented.

EXPECTED FEATURES:
{features_list}

CODE TO ANALYZE:
```
{code}
```

For each feature, determine if it is:
- "implemented": fully or partially implemented in the code
- "not_implemented": not found in the code
- "partial": some aspects present but incomplete

Respond in this exact JSON format:
{{
    "features": [
        {{"feature": "feature text", "status": "implemented/not_implemented/partial", "evidence": "brief explanation"}}
    ]
}}"""

    response = llm_call(prompt, max_tokens=2000)

    try:
        result = json.loads(response)
        return result.get("features", [])
```

```
    except json.JSONDecodeError:
        # Try to extract JSON
        match = re.search(r'\{.*\}', response, re.DOTALL)
        if match:
            try:
                result = json.loads(match.group())
                return result.get("features", [])
            except:
                pass

        # Fallback to keyword-based
        return []


def evaluate_expected_features(
    prompt_id: str,
    pentagon_result: Dict[str, Any],
    baseline_result: Dict[str, Any],
    vibe_prompts: Dict[str, Dict[str, Any]],
    use_llm: bool = True
) -> Dict[str, Any]:
    """
    Evaluate expected features implementation for both Pentagon and Baseline.
    """

    # Get expected features from VibePrompts
    prompt_info = vibe_prompts.get(prompt_id, {})
    expected_features = prompt_info.get("expected_features", [])

    if not expected_features:
        return {
            "error": f"No expected features found for prompt {prompt_id}",
            "pentagon": {"implemented": 0, "total": 0, "percentage": 0},
            "baseline": {"implemented": 0, "total": 0, "percentage": 0}
        }

    # Extract code content
    code_content = extract_code_content(pentagon_result, baseline_result)

    # Evaluate Pentagon
    pentagon_features = []
    if use_llm:
        llm_results = check_features_llm_based(expected_features, code_content["pentagon"])
        if llm_results:
            for llm_result in llm_results:
                pentagon_features.append({
                    "feature": llm_result.get("feature", ""),
                    "implemented": llm_result.get("status") in ["implemented", "partial"],
                    "status": llm_result.get("status", "unknown"),
                    "evidence": llm_result.get("evidence", ""),
```

```
                "method": "llm"
            })

    # Fallback or supplement with keyword-based
    if not pentagon_features:
        for feature in expected_features:
            result = check_feature_keyword_based(feature, code_content["pentagon"])
            result["method"] = "keyword"
            result["status"] = "implemented" if result["implemented"] else "not_implemented"
            pentagon_features.append(result)

    # Evaluate Baseline
    baseline_features = []
    if use_llm:
        llm_results = check_features_llm_based(expected_features, code_content["baseline"])
        if llm_results:
            for llm_result in llm_results:
                baseline_features.append({
                    "feature": llm_result.get("feature", ""),
                    "implemented": llm_result.get("status") in ["implemented", "partial"],
                    "status": llm_result.get("status", "unknown"),
                    "evidence": llm_result.get("evidence", ""),
                    "method": "llm"
                })

    # Fallback or supplement with keyword-based
    if not baseline_features:
        for feature in expected_features:
            result = check_feature_keyword_based(feature, code_content["baseline"])
            result["method"] = "keyword"
            result["status"] = "implemented" if result["implemented"] else "not_implemented"
            baseline_features.append(result)

    # Calculate statistics
    pentagon_implemented = sum(1 for f in pentagon_features if f.get("implemented", False))
    baseline_implemented = sum(1 for f in baseline_features if f.get("implemented", False))
    total_features = len(expected_features)

    return {
        "prompt_id": prompt_id,
        "prompt_description": prompt_info.get("description", ""),
        "complexity": prompt_info.get("complexity", "unknown"),
        "total_expected_features": total_features,
        "expected_features_list": expected_features,
        "pentagon": {
            "implemented_count": pentagon_implemented,
            "total": total_features,
            "percentage": round((pentagon_implemented / total_features) * 100, 1) if total_features > 0 else 0,
            "features_detail": pentagon_features
```

```
        },
        "baseline": {
            "implemented_count": baseline_implemented,
            "total": total_features,
            "percentage": round((baseline_implemented / total_features) * 100, 1) if total_features > 0 else 0,
            "features_detail": baseline_features
        },
        "comparison": {
            "pentagon_advantage": pentagon_implemented - baseline_implemented,
            "pentagon_percentage_advantage": round(
                ((pentagon_implemented / total_features) - (baseline_implemented / total_features)) * 100, 1
            ) if total_features > 0 else 0,
            "winner": "Pentagon" if pentagon_implemented > baseline_implemented else (
                "Baseline" if baseline_implemented > pentagon_implemented else "Tie"
            )
        }
    }

# ========================================
# 2. Pipeline Success Evaluation
# ========================================

def evaluate_pentagon_pipeline(pentagon_result: Dict[str, Any]) -> Dict[str, Any]:
    """Evaluate Pentagon pipeline success."""
    phases = ["user_stories", "system_design", "backend_code", "frontend_code", "test_report"]
    phase_details = {}
    succeeded = 0

    phases_data = pentagon_result.get("phases", {})

    for phase in phases:
        phase_info = phases_data.get(phase, {})
        phase_success = phase_info.get("success", False)
        has_data = bool(phase_info.get("data"))
        has_error = bool(phase_info.get("error"))

        phase_details[phase] = {
            "success": phase_success,
            "has_data": has_data,
            "has_error": has_error
        }

        if phase_success and has_data:
            succeeded += 1

    return {
        "phases_succeeded": succeeded,
        "total_phases": len(phases),
        "success_rate": succeeded / len(phases),
```

```
        "phase_details": phase_details,
        "overall_success": pentagon_result.get("success", False)
    }

def evaluate_baseline_pipeline(baseline_result: Dict[str, Any]) -> Dict[str, Any]:
    """Evaluate Baseline pipeline success."""
    output = baseline_result.get("output", {})

    has_user_stories = bool(output.get("user_stories"))
    has_backend = bool(output.get("backend_code"))
    has_frontend = bool(output.get("frontend_code"))
    has_test = bool(output.get("test_summary"))

    components = {
        "user_stories": has_user_stories,
        "backend_code": has_backend,
        "frontend_code": has_frontend,
        "test_summary": has_test
    }

    succeeded = sum(1 for v in components.values() if v)

    return {
        "components_succeeded": succeeded,
        "total_components": len(components),
        "success_rate": succeeded / len(components),
        "component_details": components,
        "overall_success": baseline_result.get("success", False)
    }

# ========================================
# 3. Code Executability Evaluation
# ========================================

def check_python_syntax(code: str) -> Dict[str, Any]:
    """Check Python code for syntax errors."""
    if not code or not code.strip():
        return {"valid": False, "error": "Empty code"}
    try:
        ast.parse(code)
        return {"valid": True, "error": None}
    except SyntaxError as e:
        return {"valid": False, "error": str(e)}

def check_html_structure(code: str) -> Dict[str, Any]:
    """Check HTML code for basic structure."""
    if not code or not code.strip():
        return {"valid": False, "issues": ["Empty code"]}
```

```
    issues = []
    code_lower = code.lower()

    if "<html" not in code_lower:
        issues.append("Missing <html> tag")
    if "<head" not in code_lower:
        issues.append("Missing <head> tag")
    if "<body" not in code_lower:
        issues.append("Missing <body> tag")

    script_opens = len(re.findall(r'<script[^>]*>', code, re.IGNORECASE))
    script_closes = len(re.findall(r'</script>', code, re.IGNORECASE))
    if script_opens != script_closes:
        issues.append(f"Mismatched script tags: {script_opens} opens, {script_closes} closes")

    return {
        "valid": len(issues) == 0,
        "issues": issues
    }

def evaluate_pentagon_executability(pentagon_result: Dict[str, Any]) -> Dict[str, Any]:
    """Evaluate code executability for Pentagon output."""
    phases = pentagon_result.get("phases", {})

    backend_data = phases.get("backend_code", {}).get("data", {})
    backend_files = backend_data.get("files", [])

    backend_results = []
    for f in backend_files:
        content = f.get("content", "")
        filename = f.get("filename", "unknown")
        check = check_python_syntax(content)
        backend_results.append({
            "filename": filename,
            "valid": check["valid"],
            "error": check.get("error")
        })

    backend_valid = all(r["valid"] for r in backend_results) if backend_results else False

    frontend_data = phases.get("frontend_code", {}).get("data", {})
    frontend_files = frontend_data.get("files", [])

    frontend_results = []
    for f in frontend_files:
        content = f.get("content", "")
        filename = f.get("filename", "unknown")
        check = check_html_structure(content)
        frontend_results.append({
```

```
            "filename": filename,
            "valid": check["valid"],
            "issues": check.get("issues", [])
        })

    frontend_valid = all(r["valid"] for r in frontend_results) if frontend_results else False

    return {
        "backend": {
            "files_checked": len(backend_results),
            "all_valid": backend_valid,
            "details": backend_results
        },
        "frontend": {
            "files_checked": len(frontend_results),
            "all_valid": frontend_valid,
            "details": frontend_results
        },
        "overall_score": (1.0 if backend_valid else 0.0) * 0.5 + (1.0 if frontend_valid else 0.0) * 0.5
    }

def evaluate_baseline_executability(baseline_result: Dict[str, Any]) -> Dict[str, Any]:
    """Evaluate code executability for Baseline output."""
    output = baseline_result.get("output", {})

    backend_code = output.get("backend_code", "")
    backend_check = check_python_syntax(backend_code)

    frontend_code = output.get("frontend_code", "")
    frontend_check = check_html_structure(frontend_code)

    return {
        "backend": {
            "valid": backend_check["valid"],
            "error": backend_check.get("error")
        },
        "frontend": {
            "valid": frontend_check["valid"],
            "issues": frontend_check.get("issues", [])
        },
        "overall_score": (1.0 if backend_check["valid"] else 0.0) * 0.5 + (1.0 if frontend_check["valid"] else 0.0) * 0.5
    }

# ========================================
# 4. QA Test Results Evaluation
# ========================================

def evaluate_qa_results(pentagon_result: Dict[str, Any]) -> Dict[str, Any]:
    """Evaluate QA test results from Pentagon."""
```

```
    phases = pentagon_result.get("phases", {})
    test_report = phases.get("test_report", {}).get("data", {})

    if not test_report:
        return {
            "has_qa": False,
            "overall_status": "missing",
            "test_cases": 0,
            "passed": 0,
            "failed": 0,
            "skipped": 0,
            "pass_rate": 0.0
        }

    overall_status = test_report.get("overall_status", "unknown")
    test_cases = test_report.get("test_cases", [])

    passed = sum(1 for tc in test_cases if tc.get("status") == "pass")
    failed = sum(1 for tc in test_cases if tc.get("status") == "fail")
    skipped = sum(1 for tc in test_cases if tc.get("status") == "skip")
    total = len(test_cases)

    return {
        "has_qa": True,
        "overall_status": overall_status,
        "test_cases": total,
        "passed": passed,
        "failed": failed,
        "skipped": skipped,
        "pass_rate": passed / total if total > 0 else 0.0,
        "recommendations_count": len(test_report.get("recommendations", []))
    }

# ========================================
# 5. User Stories Quality Evaluation
# ========================================

def evaluate_user_stories(pentagon_result: Dict[str, Any], baseline_result: Dict[str, Any]) -> Dict[str, Any]:
    """Compare user stories between Pentagon and Baseline."""

    pentagon_phases = pentagon_result.get("phases", {})
    pentagon_stories_data = pentagon_phases.get("user_stories", {}).get("data", {})
    pentagon_stories = pentagon_stories_data.get("stories", [])

    baseline_output = baseline_result.get("output", {})
    baseline_stories = baseline_output.get("user_stories", [])

    def analyze_stories(stories: List[Dict]) -> Dict[str, Any]:
        if not stories:
```

```
        return {"count": 0, "has_priorities": False, "has_descriptions": False}

    has_priorities = all("priority" in s for s in stories)
    has_descriptions = all("description" in s and len(s.get("description", "")) > 10 for s in stories)
    has_ids = all("id" in s for s in stories)

    priority_counts = defaultdict(int)
    for s in stories:
        priority_counts[s.get("priority", "unknown")] += 1

    return {
        "count": len(stories),
        "has_priorities": has_priorities,
        "has_descriptions": has_descriptions,
        "has_ids": has_ids,
        "priority_distribution": dict(priority_counts)
    }

  return {
    "pentagon": analyze_stories(pentagon_stories),
    "baseline": analyze_stories(baseline_stories)
  }

# ========================================
# 6. System Design Evaluation (Pentagon Only)
# ========================================

def evaluate_system_design(pentagon_result: Dict[str, Any]) -> Dict[str, Any]:
  """Evaluate system design quality (Pentagon only)."""
  phases = pentagon_result.get("phases", {})
  design_data = phases.get("system_design", {}).get("data", {})

  if not design_data:
    return {
      "has_design": False,
      "models_count": 0,
      "endpoints_count": 0,
      "has_architecture_notes": False
    }

  models = design_data.get("models", [])
  endpoints = design_data.get("endpoints", [])
  architecture_notes = design_data.get("architecture_notes", "")

  model_details = []
  for model in models:
    model_details.append({
      "name": model.get("name", "unknown"),
      "fields_count": len(model.get("fields", []))
```

```
        })

    endpoint_methods = defaultdict(int)
    for ep in endpoints:
        endpoint_methods[ep.get("method", "UNKNOWN")] += 1

    return {
        "has_design": True,
        "models_count": len(models),
        "models": model_details,
        "endpoints_count": len(endpoints),
        "endpoint_methods": dict(endpoint_methods),
        "has_architecture_notes": bool(architecture_notes),
        "architecture_notes_length": len(architecture_notes)
    }

# ========================================
# 7. Execution Efficiency Evaluation
# ========================================

def evaluate_efficiency(pentagon_result: Dict[str, Any], baseline_result: Dict[str, Any]) -> Dict[str, Any]:
    """Compare execution efficiency between Pentagon and Baseline."""

    pentagon_time = pentagon_result.get("execution_time_seconds", 0)
    baseline_time = baseline_result.get("execution_time_seconds", 0)

    pentagon_phases = pentagon_result.get("phases_succeeded", 0)

    return {
        "pentagon": {
            "execution_time_seconds": round(pentagon_time, 2),
            "phases_succeeded": pentagon_phases,
            "time_per_phase": round(pentagon_time / pentagon_phases, 2) if pentagon_phases > 0 else 0
        },
        "baseline": {
            "execution_time_seconds": round(baseline_time, 2)
        },
        "comparison": {
            "pentagon_slower_by_seconds": round(pentagon_time - baseline_time, 2),
            "pentagon_slower_by_factor": round(pentagon_time / baseline_time, 2) if baseline_time > 0 else 0
        }
    }

# ========================================
# 8. Code Quality Evaluation (LLM-based)
# ========================================

def evaluate_code_quality_llm(prompt: str, pentagon_result: Dict[str, Any], baseline_result: Dict[str, Any]) -> Dict[str, Any]:
    """Use LLM to evaluate code quality."""
```

```
    pentagon_phases = pentagon_result.get("phases", {})
    pentagon_backend = ""
    pentagon_frontend = ""

    backend_data = pentagon_phases.get("backend_code", {}).get("data", {})
    for f in backend_data.get("files", []):
        pentagon_backend += f.get("content", "") + "\n"

    frontend_data = pentagon_phases.get("frontend_code", {}).get("data", {})
    for f in frontend_data.get("files", []):
        pentagon_frontend += f.get("content", "") + "\n"

    baseline_output = baseline_result.get("output", {})
    baseline_backend = baseline_output.get("backend_code", "")
    baseline_frontend = baseline_output.get("frontend_code", "")

    evaluation_prompt = f"""Evaluate code quality for both implementations.

REQUIREMENT: {prompt}

PENTAGON BACKEND (first 3000 chars):
{pentagon_backend[:3000]}

PENTAGON FRONTEND (first 3000 chars):
{pentagon_frontend[:3000]}

BASELINE BACKEND (first 3000 chars):
{baseline_backend[:3000]}

BASELINE FRONTEND (first 3000 chars):
{baseline_frontend[:3000]}

Rate each on a scale of 1-10:
1. Code structure (organization, modularity)
2. Readability (naming, comments)
3. API design (RESTful, clear endpoints)
4. Error handling (validation, edge cases)

Respond in this exact JSON format:
{{
  "pentagon": {{
    "code_structure": <score>,
    "readability": <score>,
    "api_design": <score>,
    "error_handling": <score>,
    "notes": "<brief notes>"
  }},
  "baseline": {{
```

```
        "code_structure": <score>,
        "readability": <score>,
        "api_design": <score>,
        "error_handling": <score>,
        "notes": "<brief notes>"
    }}
}}"""

    response = llm_call(evaluation_prompt, max_tokens=800)

    try:
        result = json.loads(response)
        for key in ["pentagon", "baseline"]:
            if key in result:
                scores = [result[key].get(m, 5) for m in ["code_structure", "readability", "api_design", "error_handling"]]
                result[key]["average"] = round(sum(scores) / len(scores), 2)
        return result
    except json.JSONDecodeError:
        match = re.search(r'\{.*\}', response, re.DOTALL)
        if match:
            try:
                result = json.loads(match.group())
                for key in ["pentagon", "baseline"]:
                    if key in result:
                        scores = [result[key].get(m, 5) for m in ["code_structure", "readability", "api_design", "error_handling"]]
                        result[key]["average"] = round(sum(scores) / len(scores), 2)
                return result
            except:
                pass
        return {
            "pentagon": {"average": 5, "notes": "Could not parse evaluation"},
            "baseline": {"average": 5, "notes": "Could not parse evaluation"}
        }

# ============================================
# 9. Single Prompt Comprehensive Evaluation
# ============================================

def evaluate_single_prompt(
    prompt_result: Dict[str, Any],
    vibe_prompts: Dict[str, Dict[str, Any]],
    use_llm: bool = True
) -> Dict[str, Any]:
    """Comprehensive evaluation for a single prompt result."""

    prompt = prompt_result.get("prompt", "")
    prompt_id = prompt_result.get("prompt_id", "unknown")
    complexity = prompt_result.get("complexity", "unknown")
```

```
pentagon = prompt_result.get("pentagon", {})
baseline = prompt_result.get("baseline", {})

evaluation = {
    "prompt_id": prompt_id,
    "prompt": prompt[:100] + "..." if len(prompt) > 100 else prompt,
    "complexity": complexity,
    "timestamp": datetime.now().isoformat(),

    # 1. Expected Features (NEW - Primary metric)
    "expected_features": evaluate_expected_features(
        prompt_id, pentagon, baseline, vibe_prompts, use_llm
    ),

    # 2. Pipeline Success
    "pipeline": {
        "pentagon": evaluate_pentagon_pipeline(pentagon),
        "baseline": evaluate_baseline_pipeline(baseline)
    },

    # 3. Code Executability
    "executability": {
        "pentagon": evaluate_pentagon_executability(pentagon),
        "baseline": evaluate_baseline_executability(baseline)
    },

    # 4. QA Results (Pentagon only)
    "qa_results": evaluate_qa_results(pentagon),

    # 5. User Stories
    "user_stories": evaluate_user_stories(pentagon, baseline),

    # 6. System Design (Pentagon only)
    "system_design": evaluate_system_design(pentagon),

    # 7. Efficiency
    "efficiency": evaluate_efficiency(pentagon, baseline)
}

# 8. LLM-based code quality (optional)
if use_llm:
    try:
        evaluation["code_quality_llm"] = evaluate_code_quality_llm(prompt, pentagon, baseline)
    except Exception as e:
        evaluation["code_quality_llm"] = {"error": str(e)}

# Calculate summary scores
evaluation["summary"] = calculate_summary_scores(evaluation)
```

```
    return evaluation


# ============================================
# 10. Summary Score Calculation
# ============================================

def calculate_summary_scores(evaluation: Dict[str, Any]) -> Dict[str, Any]:
    """Calculate summary scores from evaluation results."""

    # Expected Features scores (NEW - weighted heavily)
    expected_features = evaluation.get("expected_features", {})
    pentagon_features_pct = expected_features.get("pentagon", {}).get("percentage", 0) / 100
    baseline_features_pct = expected_features.get("baseline", {}).get("percentage", 0) / 100

    # Pipeline scores
    pentagon_pipeline = evaluation["pipeline"]["pentagon"]["success_rate"]
    baseline_pipeline = evaluation["pipeline"]["baseline"]["success_rate"]

    # Executability scores
    pentagon_exec = evaluation["executability"]["pentagon"]["overall_score"]
    baseline_exec = evaluation["executability"]["baseline"]["overall_score"]

    # QA score (Pentagon only)
    pentagon_qa = evaluation["qa_results"]["pass_rate"]

    # LLM quality scores if available
    quality_eval = evaluation.get("code_quality_llm", {})
    pentagon_quality = quality_eval.get("pentagon", {}).get("average", 5) / 10
    baseline_quality = quality_eval.get("baseline", {}).get("average", 5) / 10

    # Weighted composite scores
    # Pentagon: Features (30%) + Pipeline (15%) + Exec (15%) + QA (20%) + Quality (20%)
    pentagon_composite = (
        pentagon_features_pct * 0.30 +
        pentagon_pipeline * 0.15 +
        pentagon_exec * 0.15 +
        pentagon_qa * 0.20 +
        pentagon_quality * 0.20
    )

    # Baseline: Features (40%) + Pipeline (20%) + Exec (20%) + Quality (20%)
    baseline_composite = (
        baseline_features_pct * 0.40 +
        baseline_pipeline * 0.20 +
        baseline_exec * 0.20 +
        baseline_quality * 0.20
    )

    return {
```

```
        "pentagon": {
            "features_score": round(pentagon_features_pct, 3),
            "pipeline_score": round(pentagon_pipeline, 3),
            "executability_score": round(pentagon_exec, 3),
            "qa_score": round(pentagon_qa, 3),
            "quality_score": round(pentagon_quality, 3),
            "composite_score": round(pentagon_composite, 3)
        },
        "baseline": {
            "features_score": round(baseline_features_pct, 3),
            "pipeline_score": round(baseline_pipeline, 3),
            "executability_score": round(baseline_exec, 3),
            "quality_score": round(baseline_quality, 3),
            "composite_score": round(baseline_composite, 3)
        },
        "comparison": {
            "features_advantage": round(pentagon_features_pct - baseline_features_pct, 3),
            "composite_advantage": round(pentagon_composite - baseline_composite, 3),
            "pentagon_wins": pentagon_composite > baseline_composite
        }
    }

# ============================================
# 11. Full Experiment Evaluation
# ============================================

def evaluate_full_experiment(
    experiment_data: Dict[str, Any],
    vibe_prompts: Dict[str, Dict[str, Any]],
    use_llm: bool = True
) -> Dict[str, Any]:
    """Evaluate complete experiment results."""

    print("=" * 60)
    print("PENTAGON PROTOCOL - COMPREHENSIVE EVALUATION")
    print("=" * 60)

    results = experiment_data.get("results", [])
    total_prompts = len(results)

    print(f"\nEvaluating {total_prompts} prompts...")
    print(f"Expected features loaded: {len(vibe_prompts)} prompts")

    prompt_evaluations = []

    for i, prompt_result in enumerate(results):
        prompt_id = prompt_result.get("prompt_id", f"P{i+1}")
        print(f"\n[{i+1}/{total_prompts}] Evaluating {prompt_id}...")
```

```
        evaluation = evaluate_single_prompt(prompt_result, vibe_prompts, use_llm=use_llm)
        prompt_evaluations.append(evaluation)

        # Print quick summary
        summary = evaluation["summary"]
        features = evaluation["expected_features"]

        print(f"  Expected Features: Pentagon {features['pentagon']['percentage']:.1f}% vs Baseline {features['baseline']['percentage']:.1f}%")
        print(f"  Composite Score:   Pentagon {summary['pentagon']['composite_score']:.3f} vs Baseline {summary['baseline']['composite_score']:.3f}")
        print(f"  Winner: {'Pentagon' if summary['comparison']['pentagon_wins'] else 'Baseline'}")

    # Aggregate results
    aggregate = calculate_aggregate_results(prompt_evaluations)

    # Generate final report
    final_report = {
        "experiment_date": experiment_data.get("experiment_date"),
        "evaluation_date": datetime.now().isoformat(),
        "total_prompts": total_prompts,
        "vibe_prompts_version": "2.0",
        "prompt_evaluations": prompt_evaluations,
        "aggregate": aggregate,
        "conclusions": generate_conclusions(aggregate)
    }

    print("\n" + "=" * 60)
    print("EVALUATION COMPLETE")
    print("=" * 60)

    return final_report

# ============================================
# 12. Aggregate Results Calculation
# ============================================

def calculate_aggregate_results(evaluations: List[Dict[str, Any]]) -> Dict[str, Any]:
    """Calculate aggregate statistics across all prompts."""

    if not evaluations:
        return {}

    # Collect scores
    pentagon_scores = {
        "features": [],
        "pipeline": [],
        "executability": [],
        "qa": [],
        "quality": [],
        "composite": []
```

```
}

baseline_scores = {
    "features": [],
    "pipeline": [],
    "executability": [],
    "quality": [],
    "composite": []
}

pentagon_wins = 0
features_wins = 0

# By complexity
by_complexity = defaultdict(lambda: {
    "pentagon_features": [],
    "baseline_features": [],
    "pentagon_composite": [],
    "baseline_composite": []
})

for eval_result in evaluations:
    summary = eval_result["summary"]
    complexity = eval_result.get("complexity", "unknown")

    # Pentagon scores
    pentagon_scores["features"].append(summary["pentagon"]["features_score"])
    pentagon_scores["pipeline"].append(summary["pentagon"]["pipeline_score"])
    pentagon_scores["executability"].append(summary["pentagon"]["executability_score"])
    pentagon_scores["qa"].append(summary["pentagon"]["qa_score"])
    pentagon_scores["quality"].append(summary["pentagon"]["quality_score"])
    pentagon_scores["composite"].append(summary["pentagon"]["composite_score"])

    # Baseline scores
    baseline_scores["features"].append(summary["baseline"]["features_score"])
    baseline_scores["pipeline"].append(summary["baseline"]["pipeline_score"])
    baseline_scores["executability"].append(summary["baseline"]["executability_score"])
    baseline_scores["quality"].append(summary["baseline"]["quality_score"])
    baseline_scores["composite"].append(summary["baseline"]["composite_score"])

    # Wins
    if summary["comparison"]["pentagon_wins"]:
        pentagon_wins += 1

    if summary["comparison"]["features_advantage"] > 0:
        features_wins += 1

    # By complexity
    by_complexity[complexity]["pentagon_features"].append(summary["pentagon"]["features_score"])
```

```
    by_complexity[complexity]["baseline_features"].append(summary["baseline"]["features_score"])
    by_complexity[complexity]["pentagon_composite"].append(summary["pentagon"]["composite_score"])
    by_complexity[complexity]["baseline_composite"].append(summary["baseline"]["composite_score"])

def calc_stats(scores: List[float]) -> Dict[str, float]:
    if not scores:
        return {"mean": 0, "min": 0, "max": 0, "std": 0}
    mean = sum(scores) / len(scores)
    variance = sum((x - mean) ** 2 for x in scores) / len(scores)
    return {
        "mean": round(mean, 3),
        "min": round(min(scores), 3),
        "max": round(max(scores), 3),
        "std": round(variance ** 0.5, 3)
    }

pentagon_stats = {k: calc_stats(v) for k, v in pentagon_scores.items()}
baseline_stats = {k: calc_stats(v) for k, v in baseline_scores.items()}

# Complexity breakdown
complexity_stats = {}
for complexity, scores in by_complexity.items():
    complexity_stats[complexity] = {
        "count": len(scores["pentagon_composite"]),
        "pentagon_features_mean": round(sum(scores["pentagon_features"]) / len(scores["pentagon_features"]), 3) if scores["pentagon_features"] else 0,
        "baseline_features_mean": round(sum(scores["baseline_features"]) / len(scores["baseline_features"]), 3) if scores["baseline_features"] else 0,
        "pentagon_composite_mean": round(sum(scores["pentagon_composite"]) / len(scores["pentagon_composite"]), 3) if scores["pentagon_composite"] else 0,
        "baseline_composite_mean": round(sum(scores["baseline_composite"]) / len(scores["baseline_composite"]), 3) if scores["baseline_composite"] else 0
    }

return {
    "pentagon": pentagon_stats,
    "baseline": baseline_stats,
    "comparison": {
        "pentagon_wins": pentagon_wins,
        "baseline_wins": len(evaluations) - pentagon_wins,
        "pentagon_win_rate": round(pentagon_wins / len(evaluations), 3),
        "features_win_rate": round(features_wins / len(evaluations), 3),
        "average_features_advantage": round(
            pentagon_stats["features"]["mean"] - baseline_stats["features"]["mean"], 3
        ),
        "average_composite_advantage": round(
            pentagon_stats["composite"]["mean"] - baseline_stats["composite"]["mean"], 3
        )
    },
    "by_complexity": complexity_stats
}

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

```
# 13. Conclusions Generation
# ============================================

def generate_conclusions(aggregate: Dict[str, Any]) -> Dict[str, Any]:
    """Generate conclusions from aggregate results."""

    pentagon = aggregate.get("pentagon", {})
    baseline = aggregate.get("baseline", {})
    comparison = aggregate.get("comparison", {})
    by_complexity = aggregate.get("by_complexity", {})

    conclusions = {
        "overall_winner": "Pentagon" if comparison.get("pentagon_win_rate", 0) > 0.5 else "Baseline",
        "composite_win_rate": comparison.get("pentagon_win_rate", 0),
        "features_win_rate": comparison.get("features_win_rate", 0),
        "key_findings": []
    }

    # Key findings
    features_adv = comparison.get("average_features_advantage", 0)
    if features_adv > 0.1:
        conclusions["key_findings"].append(
            f"Pentagon implements {features_adv*100:.1f}% more expected features on average"
        )
    elif features_adv < -0.1:
        conclusions["key_findings"].append(
            f"Baseline implements {abs(features_adv)*100:.1f}% more expected features on average"
        )

    if comparison.get("pentagon_win_rate", 0) >= 0.8:
        conclusions["key_findings"].append(
            "Pentagon Protocol significantly outperforms Baseline across most prompts"
        )
    elif comparison.get("pentagon_win_rate", 0) >= 0.6:
        conclusions["key_findings"].append(
            "Pentagon Protocol shows consistent improvement over Baseline"
        )

    # QA advantage
    if pentagon.get("qa", {}).get("mean", 0) > 0.7:
        conclusions["key_findings"].append(
            f"Pentagon's built-in QA achieves {pentagon['qa']['mean']*100:.1f}% test pass rate"
        )

    # Composite scores
    conclusions["key_findings"].append(
        f"Average composite: Pentagon {pentagon.get('composite', {}).get('mean', 0):.3f} vs Baseline {baseline.get('composite', {}).get('mean', 0):.3f}"
    )
```

```
    # Complexity analysis
    for complexity, stats in by_complexity.items():
        features_diff = stats["pentagon_features_mean"] - stats["baseline_features_mean"]
        if features_diff > 0.05:
            conclusions["key_findings"].append(
                f"Pentagon excels in {complexity} prompts: {stats['pentagon_features_mean']*100:.1f}% vs {stats['baseline_features_mean']*100:.1f}% features"
            )

    return conclusions

# ============================================
# 14. Report Generation for Thesis
# ============================================

def generate_thesis_tables(evaluation_report: Dict[str, Any]) -> str:
    """Generate markdown tables for thesis Chapter 5."""

    md = "# Chapter 5: Results and Analysis\n\n"

    # Table 1: Expected Features Results (NEW - Primary table)
    md += "## Table 5.1: Expected Features Implementation Rate\n\n"
    md += "| Prompt ID | Complexity | Expected Features | Pentagon | Baseline | Advantage |\n"
    md += "|----------|------------|-------------------|----------|----------|----------|\n"

    for eval_result in evaluation_report.get("prompt_evaluations", []):
        prompt_id = eval_result.get("prompt_id", "")
        complexity = eval_result.get("complexity", "")
        features = eval_result.get("expected_features", {})

        total = features.get("total_expected_features", 0)
        p_pct = features.get("pentagon", {}).get("percentage", 0)
        b_pct = features.get("baseline", {}).get("percentage", 0)
        adv = p_pct - b_pct

        md += f"| {prompt_id} | {complexity} | {total} | {p_pct:.1f}% | {b_pct:.1f}% | {adv:+.1f}% |\n"

    # Add averages row
    aggregate = evaluation_report.get("aggregate", {})
    p_avg = aggregate.get("pentagon", {}).get("features", {}).get("mean", 0) * 100
    b_avg = aggregate.get("baseline", {}).get("features", {}).get("mean", 0) * 100
    md += f"| **Average** | - | - | **{p_avg:.1f}%** | **{b_avg:.1f}%** | **{p_avg-b_avg:+.1f}%** |\n"

    # Table 2: Overall Metric Comparison
    md += "\n## Table 5.2: Overall Experimental Results\n\n"
    md += "| Metric | Pentagon | Baseline | Advantage |\n"
    md += "|--------|----------|----------|----------|\n"

    pentagon = aggregate.get("pentagon", {})
    baseline = aggregate.get("baseline", {})
```

```
    metrics = [
        ("Expected Features (%)", "features", 100),
        ("Pipeline Success Rate", "pipeline", 1),
        ("Code Executability", "executability", 1),
        ("QA Pass Rate", "qa", 1),
        ("Code Quality (LLM)", "quality", 1),
        ("Composite Score", "composite", 1)
    ]

    for label, key, multiplier in metrics:
        p_val = pentagon.get(key, {}).get("mean", 0) * multiplier
        b_val = baseline.get(key, {}).get("mean", 0) * multiplier if key != "qa" else "N/A"

        if b_val != "N/A":
            adv = p_val - b_val
            adv_str = f"+{adv:.1f}" if adv > 0 else f"{adv:.1f}"
            if multiplier == 100:
                md += f"| {label} | {p_val:.1f}% | {b_val:.1f}% | {adv_str}% |\n"
            else:
                md += f"| {label} | {p_val:.3f} | {b_val:.3f} | {adv_str} |\n"
        else:
            md += f"| {label} | {p_val:.3f} | N/A | N/A |\n"

    # Table 3: Results by Complexity
    md += "\n## Table 5.3: Results by Complexity Level\n\n"
    md += "| Complexity | Count | Pentagon Features | Baseline Features | Pentagon Composite | Baseline Composite |\n"
    md += "|------------|-------|-------------------|-------------------|--------------------|--------------------|\n"

    for complexity, stats in aggregate.get("by_complexity", {}).items():
        md += f"| {complexity} | {stats['count']} | {stats['pentagon_features_mean']*100:.1f}% | {stats['baseline_features_mean']*100:.1f}% | {stats['pentagon_composite_mean']:.3f} |
{stats['baseline_composite_mean']:.3f} |\n"

    # Table 4: Detailed Feature Analysis (sample)
    md += "\n## Table 5.4: Feature Implementation Details (Sample - VP01)\n\n"

    # Find VP01
    for eval_result in evaluation_report.get("prompt_evaluations", []):
        if eval_result.get("prompt_id") == "VP01":
            features = eval_result.get("expected_features", {})
            md += "| Feature | Pentagon | Baseline |\n"
            md += "|---------|----------|----------|\n"

            p_details = features.get("pentagon", {}).get("features_detail", [])
            b_details = features.get("baseline", {}).get("features_detail", [])

            expected = features.get("expected_features_list", [])
            for i, feat in enumerate(expected):
                p_status = "✓" if i < len(p_details) and p_details[i].get("implemented") else "✗"
```

```
            b_status = "✓" if i < len(b_details) and b_details[i].get("implemented") else "✗"
            md += f"| {feat} | {p_status} | {b_status} |\n"
        break

    # Conclusions
    md += "\n## Key Findings\n\n"
    conclusions = evaluation_report.get("conclusions", {})
    for finding in conclusions.get("key_findings", []):
        md += f"- {finding}\n"

    md += f"\n**Overall Winner: {conclusions.get('overall_winner', 'Unknown')}**\n"
    md += f"- Composite Win Rate: {conclusions.get('composite_win_rate', 0)*100:.1f}%\n"
    md += f"- Features Win Rate: {conclusions.get('features_win_rate', 0)*100:.1f}%\n"

    return md

# ============================================
# Main Entry Point
# ============================================

def run_evaluation(
    experiment_json_path: str,
    prompts_json_path: str,
    output_dir: str = "evaluation_output",
    use_llm: bool = True
) -> Dict[str, Any]:
    """
    Main function to run evaluation on experiment results.

    Args:
        experiment_json_path: Path to experiment_results.json
        prompts_json_path: Path to vibe_prompts.json (with expected features)
        output_dir: Directory to save evaluation outputs
        use_llm: Whether to use LLM for quality evaluation

    Returns:
        Complete evaluation report
    """

    # Load experiment data
    print(f"Loading experiment data from: {experiment_json_path}")
    with open(experiment_json_path, 'r', encoding='utf-8') as f:
        experiment_data = json.load(f)

    # Load VibePrompts
    print(f"Loading VibePrompts from: {prompts_json_path}")
    vibe_prompts = load_vibe_prompts(prompts_json_path)
    print(f"Loaded {len(vibe_prompts)} prompts with expected features")
```

```
    # Run evaluation
    evaluation_report = evaluate_full_experiment(experiment_data, vibe_prompts, use_llm=use_llm)

    # Create output directory
    os.makedirs(output_dir, exist_ok=True)

    # Save evaluation report
    report_path = os.path.join(output_dir, "evaluation_report.json")
    with open(report_path, 'w', encoding='utf-8') as f:
        json.dump(evaluation_report, f, indent=2, default=str)
    print(f"\nEvaluation report saved to: {report_path}")

    # Generate thesis tables
    thesis_md = generate_thesis_tables(evaluation_report)
    tables_path = os.path.join(output_dir, "thesis_tables.md")
    with open(tables_path, 'w', encoding='utf-8') as f:
        f.write(thesis_md)
    print(f"Thesis tables saved to: {tables_path}")

    # Print summary
    print("\n" + "=" * 60)
    print("EVALUATION SUMMARY")
    print("=" * 60)

    aggregate = evaluation_report.get("aggregate", {})
    comparison = aggregate.get("comparison", {})
    conclusions = evaluation_report.get("conclusions", {})

    print(f"\n📊 Expected Features Implementation:")
    print(f"   Pentagon: {aggregate.get('pentagon', {}).get('features', {}).get('mean', 0)*100:.1f}%")
    print(f"   Baseline: {aggregate.get('baseline', {}).get('features', {}).get('mean', 0)*100:.1f}%")
    print(f"   Advantage: {comparison.get('average_features_advantage', 0)*100:+.1f}%")

    print(f"\n🏆 Overall Results:")
    print(f"   Winner: {conclusions.get('overall_winner', 'Unknown')}")
    print(f"   Pentagon Win Rate: {comparison.get('pentagon_win_rate', 0)*100:.1f}%")

    print("\n📝 Key Findings:")
    for finding in conclusions.get("key_findings", []):
        print(f"   • {finding}")

    return evaluation_report


if __name__ == "__main__":
    import sys

    if len(sys.argv) >= 3:
        experiment_path = sys.argv[1]
```

```
    prompts_path = sys.argv[2]
else:
    experiment_path = r"output\full_experiment_20260119_004837.json"
    prompts_path = "data/prompts/vibe_prompts.json"

# use_llm = "--no-llm" not in sys.argv

run_evaluation(experiment_path, prompts_path, use_llm=True)
```

```
================================================
FILE: src/schemas.py
================================================
"""
Pentagon Protocol - Robust Pydantic Schemas
Simplified schemas with better defaults, validation, and truncation recovery
"""

import re
import json
from pydantic import BaseModel, Field, field_validator
from typing import List, Optional, Any, Tuple, Dict
from datetime import datetime


# ================================================
# UTILITY FUNCTIONS FOR ROBUST JSON PARSING
# ================================================

def extract_json_from_text(text: str) -> Optional[str]:
    """Extract JSON from text with multiple strategies."""
    if not text:
        return None

    # Strategy 1: Text is already valid JSON
    text = text.strip()
    if text.startswith('{') and text.endswith('}'):
        return text
    if text.startswith('[') and text.endswith(']'):
        return text

    # Strategy 2: Extract from markdown code blocks
    patterns = [
        r'```json\s*([\s\S]*?)\s*```',
        r'```\s*([\s\S]*?)\s*```',
        r'`([\s\S]*?)`',
    ]
    for pattern in patterns:
```

```
        matches = re.findall(pattern, text, re.MULTILINE)
        for match in matches:
            match = match.strip()
            if match.startswith('{') or match.startswith('['):
                return match

    # Strategy 3: Find JSON object in text
    brace_start = text.find('{')
    brace_end = text.rfind('}')
    if brace_start != -1 and brace_end > brace_start:
        return text[brace_start:brace_end + 1]

    # Strategy 4: Find JSON array
    bracket_start = text.find('[')
    bracket_end = text.rfind(']')
    if bracket_start != -1 and bracket_end > bracket_start:
        return text[bracket_start:bracket_end + 1]

    return None


def fix_common_json_errors(json_str: str) -> str:
    """Fix common JSON formatting errors from LLMs."""
    if not json_str:
        return json_str

    # Remove control characters
    json_str = re.sub(r'[\x00-\x1f\x7f-\x9f]', ' ', json_str)

    # Fix trailing commas
    json_str = re.sub(r',\s*}', '}', json_str)
    json_str = re.sub(r',\s*]', ']', json_str)

    # Fix unquoted keys (simple cases)
    json_str = re.sub(r'(\{|\,)\s*([a-zA-Z_][a-zA-Z0-9_]*)\s*:', r'\1"\2":', json_str)

    # Fix single quotes to double quotes (careful with apostrophes)
    if re.search(r"'\w+':", json_str):
        json_str = re.sub(r"'([^']+)':", r'"\1":', json_str)

    return json_str


def repair_truncated_json(json_str: str) -> str:
    """Attempt to repair truncated JSON by closing open structures."""
    if not json_str:
        return json_str

    # Count open brackets and braces
```

```
    open_braces = json_str.count('{') - json_str.count('}')
    open_brackets = json_str.count('[') - json_str.count(']')

    # Check if we're inside a string (unmatched quotes)
    in_string = False
    escaped = False
    for char in json_str:
        if escaped:
            escaped = False
            continue
        if char == '\\':
            escaped = True
            continue
        if char == '"':
            in_string = not in_string

    # If inside a string, close it
    if in_string:
        json_str += '"'

    # Remove trailing comma if present
    json_str = re.sub(r',\s*$', '', json_str)

    # Close open brackets and braces
    json_str += ']' * open_brackets
    json_str += '}' * open_braces

    return json_str


def extract_complete_array_items(json_str: str, array_key: str, item_pattern: str) -> List[Dict]:
    """Extract complete items from a potentially truncated array."""
    items = []

    # Find the array
    array_match = re.search(rf'"{array_key}"\s*:\s*\[', json_str)
    if not array_match:
        return items

    # Extract items using the pattern
    for match in re.finditer(item_pattern, json_str):
        try:
            item_str = match.group(0)
            # Try to parse as JSON
            item = json.loads(item_str)
            items.append(item)
        except json.JSONDecodeError:
            continue
```

```
    return items


def safe_parse_json(text: str, model_class: Optional[type] = None) -> Tuple[bool, Any]:
    """
    Safely parse JSON with multiple fallback strategies including truncation recovery.
    Returns (success, result_or_error)
    """
    if not text:
        return False, "Empty input"

    # Step 1: Extract JSON from text
    json_str = extract_json_from_text(text)
    if not json_str:
        return False, "No JSON found in text"

    # Step 2: Try parsing as-is
    try:
        data = json.loads(json_str)
        if model_class:
            return True, model_class.model_validate(data)
        return True, data
    except (json.JSONDecodeError, Exception):
        pass

    # Step 3: Fix common errors and retry
    fixed_json = fix_common_json_errors(json_str)
    try:
        data = json.loads(fixed_json)
        if model_class:
            return True, model_class.model_validate(data)
        return True, data
    except (json.JSONDecodeError, Exception):
        pass

    # Step 4: Repair truncated JSON and retry
    repaired_json = repair_truncated_json(fixed_json)
    try:
        data = json.loads(repaired_json)
        if model_class:
            return True, model_class.model_validate(data)
        return True, data
    except (json.JSONDecodeError, Exception):
        pass

    # Step 5: Try model-specific recovery
    if model_class:
        recovered = recover_truncated_output(text, model_class)
        if recovered:
```

```
            return True, recovered

    return False, f"Failed to parse JSON after all attempts"


def recover_truncated_output(text: str, model_class: type) -> Optional[Any]:
    """
    Attempt to recover valid data from truncated output based on model type.
    """
    try:
        if model_class == UserStoriesOutput:
            return _recover_user_stories(text)
        elif model_class == SystemDesign:
            return _recover_system_design(text)
        elif model_class == BackendCode:
            return _recover_backend_code(text)
        elif model_class == FrontendCode:
            return _recover_frontend_code(text)
        elif model_class == TestReport:
            return _recover_test_report(text)
    except Exception:
        pass
    return None


def _recover_user_stories(text: str) -> Optional['UserStoriesOutput']:
    """Recover user stories from truncated output."""
    # Pattern for complete user story objects
    story_pattern = r'\{\s*"id"\s*:\s*"([^"]+)"\s*,\s*"title"\s*:\s*"([^"]+)"\s*,\s*"description"\s*:\s*"((?:[^"\\]|\\.)*)"\s*,\s*"priority"\s*:\s*"([^"]+)"\s*\}'

    stories = []
    for match in re.finditer(story_pattern, text):
        stories.append({
            "id": match.group(1),
            "title": match.group(2),
            "description": match.group(3).replace('\\"', '"').replace('\\n', '\n'),
            "priority": match.group(4)
        })

    if stories:
        # Extract summary if available
        summary_match = re.search(r'"summary"\s*:\s*"((?:[^"\\]|\\.)*)"', text)
        summary = summary_match.group(1) if summary_match else ""

        return UserStoriesOutput(
            stories=[UserStory(**s) for s in stories],
            summary=summary
        )
    return None
```

```
def _recover_system_design(text: str) -> Optional['SystemDesign']:
    """Recover system design from truncated output."""
    # Pattern for complete model objects
    model_pattern = r'\{\s*"name"\s*:\s*"([^"]+)"\s*,\s*"fields"\s*:\s*\[((?:[^\]]*)?)\]\s*\}'

    models = []
    for match in re.finditer(model_pattern, text):
        fields_str = match.group(2)
        fields = re.findall(r'"([^"]+)"', fields_str)
        models.append({
            "name": match.group(1),
            "fields": fields
        })

    # Pattern for complete endpoint objects
    endpoint_pattern = r'\{\s*"method"\s*:\s*"([^"]+)"\s*,\s*"path"\s*:\s*"([^"]+)"\s*,\s*"description"\s*:\s*"((?:[^"\\]|\\.)*)"\s*\}'

    endpoints = []
    for match in re.finditer(endpoint_pattern, text):
        endpoints.append({
            "method": match.group(1),
            "path": match.group(2),
            "description": match.group(3).replace('\\"', '"')
        })

    if models or endpoints:
        # Extract architecture_notes if available
        notes_match = re.search(r'"architecture_notes"\s*:\s*"((?:[^"\\]|\\.)*)"', text)
        notes = notes_match.group(1) if notes_match else ""

        return SystemDesign(
            models=[DataModel(**m) for m in models] if models else [DataModel(name="Item", fields=["id: int", "name: str"])],
            endpoints=[APIEndpoint(**e) for e in endpoints] if endpoints else [APIEndpoint(method="GET", path="/api/items", description="Get all items")],
            architecture_notes=notes
        )
    return None


def _recover_backend_code(text: str) -> Optional['BackendCode']:
    """Recover backend code from truncated output."""
    # Pattern for complete file objects - more flexible
    file_pattern = r'\{\s*"filename"\s*:\s*"([^"]+)"\s*,\s*"content"\s*:\s*"((?:[^"\\]|\\.)*)"\s*,\s*"description"\s*:\s*"((?:[^"\\]|\\.)*)"\s*\}'

    files = []
    for match in re.finditer(file_pattern, text, re.DOTALL):
        content = match.group(2)
        # Unescape the content
```

```
        content = content.replace('\\n', '\n').replace('\\t', '\t').replace('\\"', '"').replace('\\\\', '\\')
        files.append({
            "filename": match.group(1),
            "content": content,
            "description": match.group(3)
        })

    # If no files found with description, try without description
    if not files:
        file_pattern_simple = r'\{\s*"filename"\s*:\s*"([^"]+)"\s*,\s*"content"\s*:\s*"((?:[^"\\]|\\.)*)"'
        for match in re.finditer(file_pattern_simple, text, re.DOTALL):
            content = match.group(2)
            content = content.replace('\\n', '\n').replace('\\t', '\t').replace('\\"', '"').replace('\\\\', '\\')
            files.append({
                "filename": match.group(1),
                "content": content,
                "description": ""
            })

    if files:
        # Check if we have main.py
        has_main = any('main' in f['filename'].lower() for f in files)
        if has_main:
            # Extract setup_instructions if available
            setup_match = re.search(r'"setup_instructions"\s*:\s*"((?:[^"\\]|\\.)*)"', text)
            setup = setup_match.group(1) if setup_match else "pip install fastapi uvicorn && uvicorn main:app --reload"

            return BackendCode(
                files=[CodeFile(**f) for f in files],
                setup_instructions=setup
            )
    return None


def _recover_frontend_code(text: str) -> Optional['FrontendCode']:
    """Recover frontend code from truncated output."""
    # Pattern for complete file objects
    file_pattern = r'\{\s*"filename"\s*:\s*"([^"]+)"\s*,\s*"content"\s*:\s*"((?:[^"\\]|\\.)*)"\s*,\s*"description"\s*:\s*"((?:[^"\\]|\\.)*)"\s*\}'

    files = []
    for match in re.finditer(file_pattern, text, re.DOTALL):
        content = match.group(2)
        content = content.replace('\\n', '\n').replace('\\t', '\t').replace('\\"', '"').replace('\\\\', '\\')
        files.append({
            "filename": match.group(1),
            "content": content,
            "description": match.group(3)
        })
```

```
    # If no files found with description, try without description
    if not files:
        file_pattern_simple = r'\{\s*"filename"\s*:\s*"([^"]+)"\s*,\s*"content"\s*:\s*"((?:[^"\\]|\\.)*)"'
        for match in re.finditer(file_pattern_simple, text, re.DOTALL):
            content = match.group(2)
            content = content.replace('\\n', '\n').replace('\\t', '\t').replace('\\"', '"').replace('\\\\', '\\')
            files.append({
                "filename": match.group(1),
                "content": content,
                "description": ""
            })

    if files:
        # Check if we have index.html
        has_index = any('index' in f['filename'].lower() for f in files)
        if has_index:
            setup_match = re.search(r'"setup_instructions"\s*:\s*"((?:[^"\\]|\\.)*)"', text)
            setup = setup_match.group(1) if setup_match else "Open index.html in browser"

            return FrontendCode(
                files=[CodeFile(**f) for f in files],
                setup_instructions=setup
            )
    return None


def _recover_test_report(text: str) -> Optional['TestReport']:
    """Recover test report from truncated output."""
    # Pattern for complete test case objects
    tc_pattern = r'\{\s*"id"\s*:\s*"([^"]+)"\s*,\s*"description"\s*:\s*"((?:[^"\\]|\\.)*)"\s*,\s*"status"\s*:\s*"([^"]+)"\s*,\s*"notes"\s*:\s*"((?:[^"\\]|\\.)*)"\s*,\s*"responsible_agent"\s*:\s*"([^"]*)"\s*\}'

    test_cases = []
    for match in re.finditer(tc_pattern, text):
        test_cases.append({
            "id": match.group(1),
            "description": match.group(2).replace('\\"', '"'),
            "status": match.group(3),
            "notes": match.group(4).replace('\\"', '"'),
            "responsible_agent": match.group(5)
        })

    # Try simpler pattern if no matches
    if not test_cases:
        tc_pattern_simple = r'\{\s*"id"\s*:\s*"([^"]+)"\s*,\s*"description"\s*:\s*"((?:[^"\\]|\\.)*)"\s*,\s*"status"\s*:\s*"([^"]+)"'
        for match in re.finditer(tc_pattern_simple, text):
            test_cases.append({
                "id": match.group(1),
                "description": match.group(2).replace('\\"', '"'),
                "status": match.group(3),
```

```
                "notes": "",
                "responsible_agent": ""
            })

    if test_cases:
        # Extract overall_status
        status_match = re.search(r'"overall_status"\s*:\s*"([^"]+)"', text)
        overall_status = status_match.group(1) if status_match else "needs_review"

        # Extract summary
        summary_match = re.search(r'"summary"\s*:\s*"((?:[^"\\]|\\.)*)"', text)
        summary = summary_match.group(1) if summary_match else ""

        # Extract recommendations
        recommendations = []
        rec_match = re.search(r'"recommendations"\s*:\s*\[((?:[^\]]*)?)\]', text)
        if rec_match:
            recommendations = re.findall(r'"([^"]+)"', rec_match.group(1))

        # Extract issues_by_agent
        issues_by_agent = {
            "product_owner": [],
            "architect": [],
            "backend_engineer": [],
            "frontend_engineer": []
        }

        return TestReport(
            overall_status=overall_status,
            test_cases=[TestCase(**tc) for tc in test_cases],
            summary=summary,
            recommendations=recommendations,
            issues_by_agent=issues_by_agent
        )
    return None


# ============================================================
# SIMPLIFIED SCHEMAS - Easier for LLMs to generate
# ============================================================

class UserStory(BaseModel):
    """Single user story - simplified"""
    id: str = Field(description="Unique ID like US001")
    title: str = Field(description="Short title")
    description: str = Field(description="As a user, I want...")
    priority: str = Field(default="medium", description="high/medium/low")

    @field_validator('priority')
```

```
    @classmethod
    def validate_priority(cls, v):
        v = v.lower().strip()
        if v not in ['high', 'medium', 'low']:
            return 'medium'
        return v


class UserStoriesOutput(BaseModel):
    """Output from Product Owner"""
    stories: List[UserStory] = Field(description="List of user stories")
    summary: str = Field(default="", description="Brief summary")


class APIEndpoint(BaseModel):
    """API endpoint definition - simplified"""
    method: str = Field(description="GET/POST/PUT/DELETE")
    path: str = Field(description="URL path like /api/items")
    description: str = Field(description="What this endpoint does")

    @field_validator('method')
    @classmethod
    def validate_method(cls, v):
        v = v.upper().strip()
        if v not in ['GET', 'POST', 'PUT', 'DELETE', 'PATCH']:
            return 'GET'
        return v


class DataModel(BaseModel):
    """Data model definition - simplified"""
    name: str = Field(description="Model name like Task, User")
    fields: List[str] = Field(description="List of field names with types, e.g., 'id: int', 'name: str'")


class SystemDesign(BaseModel):
    """Output from Architect"""
    models: List[DataModel] = Field(description="Data models")
    endpoints: List[APIEndpoint] = Field(description="API endpoints")
    architecture_notes: str = Field(default="", description="Additional notes")


class CodeFile(BaseModel):
    """Single code file - simplified"""
    filename: str = Field(description="File name like main.py")
    content: str = Field(description="The actual code")
    description: str = Field(default="", description="What this file does")
```

```
class BackendCode(BaseModel):
    """Output from Backend Engineer"""
    files: List[CodeFile] = Field(description="List of code files")
    setup_instructions: str = Field(default="pip install fastapi uvicorn", description="How to run")


class FrontendCode(BaseModel):
    """Output from Frontend Engineer"""
    files: List[CodeFile] = Field(description="List of frontend files")
    setup_instructions: str = Field(default="Open index.html in browser", description="How to run")


class TestCase(BaseModel):
    """Single test case - simplified"""
    id: str = Field(description="Test ID like TC001")
    description: str = Field(description="What is being tested")
    status: str = Field(description="pass/fail/skip")
    notes: str = Field(default="", description="Additional notes")
    responsible_agent: str = Field(
        default="",
        description="Agent responsible for this test case: product_owner, architect, backend_engineer, or frontend_engineer"
    )

    @field_validator('status')
    @classmethod
    def validate_status(cls, v):
        v = v.lower().strip()
        if v not in ['pass', 'fail', 'skip', 'passed', 'failed', 'skipped']:
            return 'skip'
        if v == 'passed':
            return 'pass'
        if v in ['failed', 'skipped']:
            return v.replace('ed', '')
        return v

    @field_validator('responsible_agent')
    @classmethod
    def validate_responsible_agent(cls, v):
        v = v.lower().strip()
        valid_agents = ['product_owner', 'architect', 'backend_engineer', 'frontend_engineer', '']
        if v not in valid_agents:
            return ''
        return v


class TestReport(BaseModel):
    """Output from QA Engineer"""
    overall_status: str = Field(description="pass/fail/needs_review")
    test_cases: List[TestCase] = Field(description="List of test results")
```

```
    summary: str = Field(default="", description="Overall summary")
    recommendations: List[str] = Field(default_factory=list, description="Improvement suggestions")
    issues_by_agent: Dict[str, List[str]] = Field(
        default_factory=lambda: {
            "product_owner": [],
            "architect": [],
            "backend_engineer": [],
            "frontend_engineer": []
        },
        description="Issues categorized by responsible agent"
    )

    @field_validator('overall_status')
    @classmethod
    def validate_overall_status(cls, v):
        v = v.lower().strip()
        if v not in ['pass', 'fail', 'needs_review', 'passed', 'failed']:
            return 'needs_review'
        if v == 'passed':
            return 'pass'
        if v == 'failed':
            return 'fail'
        return v


# ============================================================
# GUARDRAIL FUNCTIONS
# ============================================================

def validate_user_stories(result) -> Tuple[bool, Any]:
    """Guardrail for user stories output."""
    try:
        raw = result.raw if hasattr(result, 'raw') else str(result)
        success, parsed = safe_parse_json(raw, UserStoriesOutput)

        if not success:
            return (False, f"Invalid JSON format: {parsed}. Please return valid JSON matching the schema.")

        if not parsed.stories or len(parsed.stories) == 0:
            return (False, "No user stories found. Please include at least 1 user story.")

        # Validate each story has required fields
        for story in parsed.stories:
            if not story.id or not story.title or not story.description:
                return (False, f"User story {story.id or 'unknown'} is missing required fields (id, title, description).")

        return (True, parsed.model_dump_json())
    except Exception as e:
        return (False, f"Validation error: {str(e)}. Please return valid JSON.")
```

```
def validate_system_design(result) -> Tuple[bool, Any]:
    """Guardrail for system design output."""
    try:
        raw = result.raw if hasattr(result, 'raw') else str(result)
        success, parsed = safe_parse_json(raw, SystemDesign)

        if not success:
            return (False, f"Invalid JSON format: {parsed}. Please return valid JSON matching the schema.")

        if not parsed.models:
            return (False, "No data models defined. Please include at least 1 data model.")

        if not parsed.endpoints:
            return (False, "No API endpoints defined. Please include at least 1 endpoint.")

        # Validate each model has required fields
        for model in parsed.models:
            if not model.name or not model.fields:
                return (False, f"Data model {model.name or 'unknown'} is missing required fields (name, fields).")

        # Validate each endpoint has required fields
        for endpoint in parsed.endpoints:
            if not endpoint.method or not endpoint.path or not endpoint.description:
                return (False, f"Endpoint {endpoint.path or 'unknown'} is missing required fields (method, path, description).")

        return (True, parsed.model_dump_json())
    except Exception as e:
        return (False, f"Validation error: {str(e)}. Please return valid JSON.")


def validate_backend_code(result) -> Tuple[bool, Any]:
    """Guardrail for backend code output."""
    try:
        raw = result.raw if hasattr(result, 'raw') else str(result)
        success, parsed = safe_parse_json(raw, BackendCode)

        if not success:
            return (False, f"Invalid JSON format: {parsed}. Please return valid JSON with smaller files (under 50 lines each).")

        if not parsed.files:
            return (False, "No code files provided. Please include at least main.py.")

        # Check that main.py exists
        has_main = any('main' in f.filename.lower() for f in parsed.files)
        if not has_main:
            return (False, "Missing main.py file. Please include a main.py file.")
```

```
        # Validate each file has required fields
        for file in parsed.files:
            if not file.filename or not file.content:
                return (False, f"Code file {file.filename or 'unknown'} is missing required fields (filename, content).")

        return (True, parsed.model_dump_json())
    except Exception as e:
        return (False, f"Validation error: {str(e)}. Please return valid JSON with smaller code files.")


def validate_frontend_code(result) -> Tuple[bool, Any]:
    """Guardrail for frontend code output."""
    try:
        raw = result.raw if hasattr(result, 'raw') else str(result)
        success, parsed = safe_parse_json(raw, FrontendCode)

        if not success:
            return (False, f"Invalid JSON format: {parsed}. Please return valid JSON with smaller files.")

        if not parsed.files:
            return (False, "No frontend files provided. Please include at least index.html.")

        # Check that index.html exists
        has_index = any('index' in f.filename.lower() for f in parsed.files)
        if not has_index:
            return (False, "Missing index.html file. Please include an index.html file.")

        # Validate each file has required fields
        for file in parsed.files:
            if not file.filename or not file.content:
                return (False, f"Frontend file {file.filename or 'unknown'} is missing required fields (filename, content).")

        return (True, parsed.model_dump_json())
    except Exception as e:
        return (False, f"Validation error: {str(e)}. Please return valid JSON with smaller files.")


def validate_test_report(result) -> Tuple[bool, Any]:
    """Guardrail for test report output."""
    try:
        raw = result.raw if hasattr(result, 'raw') else str(result)
        success, parsed = safe_parse_json(raw, TestReport)

        if not success:
            return (False, f"Invalid JSON format: {parsed}. Please return valid JSON matching the schema.")

        if not parsed.test_cases:
            return (False, "No test cases provided. Please include at least 1 test case.")
```

```
        # Validate each test case has required fields
        for tc in parsed.test_cases:
            if not tc.id or not tc.description or not tc.status:
                return (False, f"Test case {tc.id or 'unknown'} is missing required fields (id, description, status).")

        # Validate that failed test cases have responsible_agent
        failed_without_agent = [
            tc for tc in parsed.test_cases
            if tc.status == 'fail' and not tc.responsible_agent
        ]
        if failed_without_agent:
            tc_ids = ", ".join(tc.id for tc in failed_without_agent)
            return (False, f"Failed test cases missing responsible_agent: {tc_ids}. Please specify which agent (product_owner, architect, backend_engineer, frontend_engineer) is responsible for each failed test.")

        # Validate responsible_agent values are valid
        valid_agents = ['product_owner', 'architect', 'backend_engineer', 'frontend_engineer', '']
        for tc in parsed.test_cases:
            if tc.responsible_agent and tc.responsible_agent not in valid_agents:
                return (False, f"Test case {tc.id} has invalid responsible_agent: {tc.responsible_agent}. Must be one of: product_owner, architect, backend_engineer, frontend_engineer.")

        # Validate consistency: if overall_status is 'pass', there should be no failed test cases
        if parsed.overall_status == 'pass':
            failed_tests = [tc for tc in parsed.test_cases if tc.status == 'fail']
            if failed_tests:
                return (False, f"overall_status is 'pass' but there are {len(failed_tests)} failed test cases. Please set overall_status to 'fail' or fix the test case statuses.")

        return (True, parsed.model_dump_json())
    except Exception as e:
        return (False, f"Validation error: {str(e)}. Please return valid JSON.")




================================================
FILE: src/tasks.py
================================================
"""
Pentagon Protocol - Enhanced Task Definitions
With guardrails and structured output enforcement
"""
from typing import List
from crewai import Task, Agent
from .schemas import (
    UserStoriesOutput, SystemDesign, BackendCode, FrontendCode, TestReport,
    validate_user_stories, validate_system_design, validate_backend_code,
    validate_frontend_code, validate_test_report
)



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

```
# TASK TEMPLATES WITH STRICT JSON REQUIREMENTS
# ============================================================


def create_user_stories_task(agent: Agent, vibe_prompt: str) -> Task:
    """Create Product Owner task for generating user stories."""
    return Task(
        description=f"""You are the Product Owner. Analyze this vibe prompt and create user stories.

Vibe Prompt: {vibe_prompt}

Return ONLY this JSON structure (no markdown, no extra text):
{{
    "stories": [
        {{
            "id": "US001",
            "title": "Short title",
            "description": "As a user, I want X so that Y",
            "priority": "high"
        }}
    ],
    "summary": "Brief project summary"
}}

CRITICAL RULES:
- Return ONLY valid JSON
- Create 5-10 user stories MAXIMUM (focus on core features)
- Keep descriptions under 100 characters
- Priority must be: high, medium, or low
- No markdown, no code blocks, no extra text""",
        expected_output="""Valid JSON with stories array and summary""",
        agent=agent,
        output_json=UserStoriesOutput,
    )



def create_system_design_task(agent: Agent, context_tasks: List[Task]) -> Task:
    """Create Architect task for system design."""
    return Task(
        description="""You are the Architect. Design a simple system based on the user stories.

Return ONLY this JSON structure (no markdown, no extra text):
{
    "models": [
        {
            "name": "Item",
            "fields": ["id: int", "name: str", "created_at: datetime"]
        }
    ],
    "endpoints": [
```

```
        {
          "method": "GET",
          "path": "/api/items",
          "description": "Get all items"
        }
      ],
      "architecture_notes": "Brief notes"
    }

    CRITICAL RULES:
    - Return ONLY valid JSON
    - Create 2-4 data models MAXIMUM
    - Create 5-8 API endpoints MAXIMUM (core CRUD operations only)
    - Keep descriptions under 50 characters
    - No markdown, no code blocks, no extra text""",
        expected_output="""Valid JSON with models, endpoints, and architecture_notes""",
        agent=agent,
        context=context_tasks,
        output_json=SystemDesign,
      )


    def create_backend_task(agent: Agent, context_tasks: List[Task]) -> Task:
      """Create Backend Engineer task for writing backend code."""
      return Task(
        description="""You are the Backend Engineer. Write a MINIMAL FastAPI backend.

    Return ONLY this JSON structure (no markdown, no extra text):
    {
      "files": [
        {
          "filename": "main.py",
          "content": "from fastapi import FastAPI\\napp = FastAPI()\\n...",
          "description": "Main FastAPI app"
        }
      ],
      "setup_instructions": "pip install fastapi uvicorn && uvicorn main:app --reload"
    }

    CRITICAL RULES:
    - Return ONLY valid JSON
    - ESCAPE all newlines as \\n
    - ESCAPE all quotes as \\"
    - ESCAPE all backslashes as \\\\
    - Create ONLY main.py (single file)
    - Keep code under 40 lines
    - Use simple in-memory dict storage
    - Include CORS middleware
    - Implement only 3-4 core endpoints
```

```
- No complex validation, no database
- No markdown, no code blocks""",
        expected_output="""Valid JSON with files array and setup_instructions""",
        agent=agent,
        context=context_tasks,
        output_json=BackendCode,
    )


def create_frontend_task(agent: Agent, context_tasks: List[Task]) -> Task:
    """Create Frontend Engineer task for writing frontend code."""
    return Task(
        description="""You are the Frontend Engineer. Write a MINIMAL HTML/JS frontend.

Return ONLY this JSON structure (no markdown, no extra text):
{
  "files": [
    {
      "filename": "index.html",
      "content": "<!DOCTYPE html>\\n<html>\\n...",
      "description": "Main HTML page"
    }
  ],
  "setup_instructions": "Open index.html in browser"
}

CRITICAL RULES:
- Return ONLY valid JSON
- ESCAPE all newlines as \\n
- ESCAPE all quotes as \\"
- ESCAPE all backslashes as \\\\
- Create ONLY index.html (single file with inline CSS/JS)
- Keep code under 50 lines
- Use fetch() for API calls
- Simple, functional UI only
- No frameworks, no external dependencies
- No markdown, no code blocks""",
        expected_output="""Valid JSON with files array and setup_instructions""",
        agent=agent,
        context=context_tasks,
        output_json=FrontendCode,
    )


def create_qa_task(agent: Agent, context_tasks: List[Task]) -> Task:
    """Create QA Engineer task for validation."""
    return Task(
        description="""You are the QA Engineer. Validate the implementation.
```

```
Return ONLY this JSON structure (no markdown, no extra text):

{
  "overall_status": "pass",
  "test_cases": [
    {
      "id": "TC001",
      "description": "Test description",
      "status": "pass",
      "notes": "",
      "responsible_agent": ""
    }
  ],
  "summary": "Brief summary",
  "recommendations": [],
  "issues_by_agent": {
    "product_owner": [],
    "architect": [],
    "backend_engineer": [],
    "frontend_engineer": []
  }
}

CRITICAL RULES:
- Return ONLY valid JSON
- Create 3-5 test cases MAXIMUM
- For FAILED tests, set responsible_agent to one of: product_owner, architect, backend_engineer, frontend_engineer
- overall_status: "pass" if all tests pass, "fail" if any fail
- Keep descriptions under 50 characters
- No markdown, no code blocks""",
        expected_output="""Valid JSON with overall_status, test_cases, summary, recommendations, and issues_by_agent""",
        agent=agent,
        context=context_tasks,
        output_json=TestReport,
    )


def create_baseline_task(agent: Agent, vibe_prompt: str) -> Task:
    """Create single baseline task for comparison."""
    return Task(
        description=f"""Build a complete application for this requirement:

VIBE PROMPT: "{vibe_prompt}"

Return ONLY this JSON structure:
{{
  "user_stories": [
    {{"id": "US001", "title": "Feature", "description": "As a user...", "priority": "high"}}
  ],
  "backend_code": "from fastapi import FastAPI\\napp = FastAPI()\\n...",
```

```
    "frontend_code": "<!DOCTYPE html>\\n<html>...</html>",
    "test_summary": "pass"
}}

RULES:
- Return ONLY valid JSON
- ESCAPE all newlines as \\n
- Keep code under 50 lines each
- Use simple implementations
""",
        expected_output="A valid JSON object with user_stories, backend_code, frontend_code, and test_summary",
        agent=agent,
        guardrail_max_retries=5,
    )
```

الخلاصة

تنسيق الحتمية في هندسة البرمجيات التوليدية: إطار عمل هرمي متعدد الوكلاء للبرمجة بالـ Vibe الموجّهة بالمخططات

أدى ظهور "البرمجة بالـ – "Vibe أي تطوير البرمجيات من خلال أوامر لغوية طبيعية موجهة لنماذج اللغة الكبيرة – (LLMs) إلى ديمقراطية البرمجة عبر تمكين غير المتخصصين من إنشاء تطبيقات وظيفية. إلا أن اللاحتمية الجوهرية في مخرجات نماذج اللغة الكبيرة تُدخل قدراً من عدم التنبؤ في جودة المخرجات، واكتمال الخصائص، وبنية الشفرة، مما يحد من قابلية البرمجة بالـ Vibe للاستخدام في برمجيات الإنتاج. تعالج هذه الرسالة السؤال الجوهري: هل يمكن لتنظيم متعدد الوكلاء، منظم ومقيّد ببنى محددة، أن يحقق توليداً حتمياً عالي الجودة للبرمجيات مع الحفاظ على سهولة الوصول التي يوفرها التوجيه باللغة الطبيعية؟

تقترح هذه الدراسة "بروتوكول البنتاغون(Pentagon Protocol) "، وهو إطار عمل هرمي متعدد الوكلاء مكوّن من خمسة وكلاء متخصصين: مالك المنتج(Product Owner) ، المهندس المعماري للنظام (System Architect)، مهندس الواجهة الخلفية(Backend Engineer) ، مهندس الواجهة الأمامية (Frontend Engineer)، ومهندس ضمان الجودة(QA Engineer) ، بما يعكس البنى المعتمدة في فرق تطوير البرمجيات التقليدية. يقدّم الإطار مفهوم "البرمجة بالـ Vibe الموجّهة بالمخططات(Schema-Guided Vibe Coding) "، وهو نموذج يُطبّق قيود مخططات قائمة على Pydantic في كل مرحلة من مراحل التوليد، فيقلّص تدريجياً عشوائية المخرجات، منتقلاً من أوامر لغوية طبيعية مبهمة إلى مصنوعات برمجية مُهيكلة ومُتحقَّق من صحتها.

تم تقييم بروتوكول البنتاغون مقارنةً بخط أساس (Baseline) ذو وكيل واحد على مجموعة بيانات VibePrompts-10، التي تضم 10 أوامر تغطي سيناريوهات تطبيقات سهلة، متوسطة، ومعقدة، بإجمالي 78 خاصية متوقَّعة. استخدم النهجان نموذج DeepSeek V3.2 بإعدادات حتمية (درجة حرارة 0.0). شمل التقييم ستة أبعاد: اكتمال الخصائص، نجاح خط الأنابيب(pipeline success) ، قابليّة تنفيذ الشفرة، معدل اجتياز اختبارات الجودة(QA pass rate) ، جودة الشفرة، وكفاءة التنفيذ.

تُظهر النتائج التجريبية أن بروتوكول البنتاغون يتفوّق significativamente على نهج خط الأساس. فقد حقق البنتاغون نسبة تنفيذ خصائص بلغت 97.8% مقابل 92.5% لخط الأساس (+5.3%)، مع زيادة الفارق في الأوامر المعقدة (+8.7%). كما تحسّنت جودة الشفرة بنسبة 44% (70.8% مقابل 49.2%)، مع أكبر مكاسب في التعامل مع الأخطاء (+74%) وتصميم واجهات البرمجة .(+50%) (API design) وأظهر البنتاغون انخفاضاً في تباين الدرجات المركّبة بنسبة 30.6%، بما يعكس اتساقاً أكبر في جودة المخرجات. فاز البنتاغون في 100% من مقارنات الدرجات المركبة (10/10 أوامر)، مما يؤكّد فعالية التنظيم متعدد الوكلاء الموجّه بالمخططات.

أظهرت دراسة المفاضلة أن بروتوكول البنتاغون يتطلّب ما يقارب 5 أضعاف زمن التنفيذ (255 ثانية مقابل 50 ثانية)، لكنه ينتج مخرجات ذات جودة أعلى بشكل ملموس تقلّل من تكاليف الصيانة والتنقيح اللاحقة. حققت مرحلة ضمان الجودة (QA) معدل اجتياز اختبارات بلغ 100% عبر جميع الأوامر، مع توليد توصيات عملية للتحسين.

تقدّم هذه الرسالة ثلاث مساهمات رئيسية

1. الإطار النظري لـ "البرمجة بالـ Vibe الموجّهة بالمخططات" بوصفه نموذجاً يجسر الفجوة بين التوجيه غير الرسمي وبين منهجيات هندسة البرمجيات الصارمة؛
2. (معمارية بروتوكول البنتاغون مع تعريفات وكلاء قابلة لإعادة الاستخدام، ومواصفات مخططات، وأنماط تنظيم(orchestration) ؛
3. أدلة تجريبية كمية تبيّن مزايا التنظيم متعدد الوكلاء عبر أبعاد متعددة للجودة.

تدعم النتائج الفرضية المركزية للرسالة، وهي أن التنظيم الهرمي متعدد الوكلاء المقيّد بالمخططات ينجح في "تنسيق الحتمية" ضمن هندسة البرمجيات التوليدية. يقدّم بروتوكول البنتاغون مقاربة عملية لتحقيق تطوير برمجيات بمساعدة الذكاء الاصطناعي تكون موثوقة وعالية الجودة، مع الحفاظ على سهولة الاستخدام التي تجعل البرمجة بالـ Vibe جذّابة. تشمل اتجاهات البحث المستقبلية تنظيم نماذج متعددة (multi-model orchestration)، وتكوينات مرحلية تكيفية (adaptive phase configurations)، ودمج آليات تغذية راجعة بشرية في الحلقة (human-in-the-loop).

Cairo University
Faculty of Graduate Studies
for Statistical Research

تنسيق الحتمية في هندسة البرمجيات التوليدية: إطار عمل هرمي متعدد الوكلاء للبرمجة بالـ Vibe الموجّهة بالمخططات

إعداد الطالب

نوران درويش

تحت إشراف

د محمد صبري

استاذ دكتور

تم تقديمه كجزء من متطلبات السنة التمهيدية للماجستير في علوم البيانات

بكلية الدراسات العليا للبحوث الإحصائية

Cairo, Egypt

2026