🎯 50 Interview Questions with Answers

AgenticTestGenerator - Architecture, Patterns & Technical Concepts

🎯 Summary

These 50 questions cover:

Architecture (Q1-Q15):

- Multi-layered agentic system
- 9-layer guardrails (95% security)
- RAG retrieval and context assembly
- Multi-agent collaboration
- Persistence and data flow

Agentic AI Patterns (Q16-Q30):

- ReAct (Reason + Act)
- Plan-and-Execute
- Constitutional AI
- Tool calling
- HITL (Human-in-the-Loop)
- State management
- Critic pattern
- Multi-agent collaboration
- Structured output
- Goal-oriented agents
- Chain-of-Thought

Technical Concepts (Q31-Q50):

- AST parsing
- Vector embeddings and semantic search
- Reciprocal Rank Fusion (RRF)
- Cross-encoder reranking
- Docker sandboxing
- Pydantic validation
- SQLite persistence
- Performance optimization
- Git change detection
- Token budget tracking
- Prometheus metrics
- OpenTelemetry tracing
- ChromaDB
- Symbol indexing

- Error handling
- Security best practices
- Scaling strategies

Key Takeaways:

1. System achieves 90% coverage + 90% pass rate through goal-driven design
2. 95% security coverage via 9-layer defense-in-depth
3. Enterprise observability with metrics, traces, and logs
4. LangChain 1.0/LangGraph for modern agent patterns
5. Context assembly forces comprehensive information gathering
6. Hybrid search combines exact + semantic + keyword
7. Multi-agent collaboration for specialized expertise
8. Production-ready with error handling, retries, and fallbacks

---

End of Interview Questions Document

Total Questions: 50 Total Lines: ~4500 Sections: 3 (Architecture, Agentic AI Patterns, Technical Concepts)

📐 SECTION 1: Architecture (Questions 1-15)

Q1: What is the overall architecture pattern used in AgenticTestGenerator?

Answer: AgenticTestGenerator follows a **Multi-Layered Agentic Architecture** with:

- **Presentation Layer**: CLI interface (main.py), future REST API
- **Application Layer**: Multi-agent orchestration (Planner, Coder, Critic)
- **Service Layer**: RAG retrieval, Git integration, AST analysis, Code embeddings
- **Data Layer**: ChromaDB (vector store), SQLite (tracking DB), File system

The architecture implements **LangChain 1.0/LangGraph patterns** for agent coordination and uses **defense-in-depth** with 9-layer guardrails for security.

Key Design Principles:

1. **Goal-Driven**: All components target 90% coverage + 90% pass rate
2. **Security-First**: 95% security coverage with comprehensive guardrails
3. **Observability-by-Default**: Full instrumentation (metrics, traces, logs)
4. **Modular & Extensible**: Plugin architecture for custom components

---

Q2: Explain the multi-agent system and agent roles

Answer: The system implements a **3-agent collaborative architecture**:

1. Planner Agent (`src/planner.py`):

- **Role**: Task decomposition and planning
- **Responsibilities**:
  - Breaks goals into executable subtasks

    - Builds dependency graphs
    - Estimates effort and confidence
    - Dynamic replanning on failures
- **Output**: `ExecutionPlan` with task graph

2. Coder Agent (Orchestrator):

- **Role**: Test code generation
- **Responsibilities**:
    - Generates comprehensive test code
    - Uses tools (search_code, get_context, execute_tests)
    - Follows framework conventions (pytest, JUnit, Jest)
    - Achieves 90% coverage goal
- **Implementation**: LangGraph ReAct agent with tool calling

3. Critic Agent (`src/critic.py`):

- **Role**: Quality assurance and review
- **Responsibilities**:
    - Reviews generated tests
    - Checks correctness, completeness, determinism
    - Provides actionable feedback
    - Enforces quality standards
- **Output**: `ReviewResult` with scores and recommendations

Agent Communication:

- Shared state via `AgentState` (TypedDict)
- Message passing with `add_messages` reducer
- JSON-based structured communication (schemas.py)
- Guard Manager coordinates all security checks

Q3: What is the 9-Layer Guardrails Model?

Answer: The system implements **defense-in-depth** with 9 layers of security guardrails (95% coverage):

Layer 1-3: Core Policy & Validation (60% coverage)

1. **Policy Engine** (`policy_engine.py`)
    - ALLOW/DENY/REVIEW decisions based on rules
    - Context-aware policy evaluation
    - Tool-specific constraints
2. **Schema Validator** (`schema_validator.py`)
    - Pydantic-based parameter validation
    - Auto-correction of invalid inputs
    - Type safety enforcement

3. **Audit Logger** (`audit_logger.py`)
    - SQLite-based event logging
    - Full audit trail (who, what, when, why)
    - Security event tracking

Layer 4-9: Advanced Guardrails (35% additional → 95% total) 4. **Input Guardrails** (`input_guardrails.py`)

- PII detection (emails, phones, SSNs)
- Prompt injection prevention
- Toxic content filtering

5. **Output Guardrails** (`output_guardrails.py`)
    - Code safety validation (no `eval`, `exec`)
    - License compliance checking
    - Citation validation
6. **Constitutional AI** (`constitutional_ai.py`)
    - Self-verification loops
    - Principle-based constraints
    - Automated correction
7. **Budget Tracker** (`budget_tracker.py`)
    - Token usage limits
    - Cost tracking
    - Time constraints
    - Rate limiting
8. **HITL Manager** (`hitl_manager.py`)
    - Human-in-the-loop approvals
    - Risk-based escalation
    - Interactive review
9. **File Boundary & Determinism**
    - Write restrictions (only tests/ directory)
    - Determinism enforcement (no random/time without mocks)
    - Secret detection (API keys, passwords)

Orchestration: All layers managed by `GuardManager` which:

- Coordinates pre/post execution checks
- Enforces policies consistently
- Logs all security events
- Provides unified API for all guardrails

Q4: How does RAG (Retrieval-Augmented Generation) work in this system?

Answer: RAG provides intelligent context for test generation through multi-source retrieval:

Components:

1. **RAGRetriever** (`src/rag_retrieval.py`):
    - Orchestrates context gathering
    - Combines multiple data sources
    - Ranks relevance
2. **Code Embeddings** (`src/code_embeddings.py`):
    - **Vector Store**: ChromaDB for semantic search
    - **Chunking Strategy**: Function/class-level granularity
    - **Embeddings**: Provider-agnostic (Ollama/OpenAI/Gemini)
    - **Search**: Cosine similarity + reranking
3. **Git Integration** (`src/git_integration.py`):
    - Detects changed files
    - Extracts new/modified functions
    - Provides diff context
4. **Symbol Index** (`src/symbol_index.py`):
    - O(1) exact lookups for functions/classes
    - Call graph tracking (callers/callees)
    - Import dependency graph
5. **Hybrid Search** (`src/hybrid_search.py`):
    - Combines exact + semantic + keyword search
    - **RRF (Reciprocal Rank Fusion)** for score aggregation
    - Cross-encoder reranking

Retrieval Flow:

```
1. Git detects changes → file_path, function_name
2. Symbol Index → exact function location, callers, callees
3. Embeddings → semantic similar code (top 5)
4. Hybrid Search → combines all results with RRF
5. Context Assembler → aggregates into structured context
6. LLM → receives comprehensive context for test generation
```

Context Types Assembled:

- Target code (the function to test)
- Related functions (semantic similarity)
- Dependencies (imports, external packages)

- Callers (who uses this function)
- Callees (what this function calls)
- Usage examples (from codebase)
- Git history (why this changed)
- Existing tests (for consistency)

Q5: Explain the Incremental Indexing strategy

Answer: The system uses **smart incremental indexing** to avoid redundant work:

Components:

1. **IncrementalIndexer** (`src/indexer.py`):
   - Tracks file hashes (SHA256)
   - Monitors modification times
   - Stores metadata persistently
2. **Metadata Storage**:
   - **SQLite**: `test_tracking.db` → `index_metadata` table
   - Fields: file_path, file_hash, last_indexed, chunk_count
3. **Indexing Process**:

```python
def index(self, source_dir: Path):
    current_files = scan_directory(source_dir)

    for file in current_files:
        current_hash = sha256(file.read_bytes())
        metadata = db.get_index_metadata(file)

        if metadata and metadata.file_hash == current_hash:
            # SKIP - file unchanged
            continue

        # INDEX - file is new or modified
        chunks = parse_file(file)  # AST parsing
        embedding_store.add_chunks(chunks)  # Vector embeddings
        symbol_index.index_file(file)  # Symbol extraction
        db.update_metadata(file, current_hash)  # Update tracking
```

Benefits:

- ⚡ **10-100x faster** on subsequent runs
- 💾 Reduced storage (no duplicate embeddings)
- 🔄 Handles file deletions (cleanup orphaned data)
- 📊 Tracks indexing history

Dual Indexing:

- **Embeddings** (ChromaDB): Semantic search
- **Symbols** (JSON): Exact lookups, call graphs

Q6: What is the Test Tracking Database and why is it needed?

Answer: `TestTrackingDB` (`src/test_tracking_db.py`) provides **function-level test tracking** for intelligent test lifecycle management.

Schema:

```sql
-- Source functions
CREATE TABLE source_functions (
    id INTEGER PRIMARY KEY,
    file_path TEXT,
    function_name TEXT,
    function_hash TEXT,  -- SHA256 of function body
    start_line, end_line,
    last_modified TIMESTAMP,
    has_test BOOLEAN,
    test_file_path TEXT,
    test_count INTEGER,
    UNIQUE(file_path, function_name)
);

-- Test cases
CREATE TABLE test_cases (
    id INTEGER PRIMARY KEY,
    test_file TEXT,
    test_function TEXT,
    source_function_id INTEGER,  -- FK to source_functions
    test_type TEXT,  -- 'unit', 'integration'
    is_passing BOOLEAN,
    FOREIGN KEY (source_function_id)
);

-- Coverage history
CREATE TABLE coverage_history (
    timestamp TIMESTAMP,
    total_functions INTEGER,
    functions_with_tests INTEGER,
    coverage_percentage REAL
);

-- Index metadata (consolidates .index_metadata.json)
CREATE TABLE index_metadata (
    file_path TEXT PRIMARY KEY,
    file_hash TEXT,
    last_indexed TIMESTAMP,
    chunk_count INTEGER
);
```

```
-- File relationships (consolidates .test_relationships.json)
CREATE TABLE file_relationships (
    source_file TEXT,
    test_file TEXT,
    source_hash TEXT,
    test_hash TEXT,
    last_synced TIMESTAMP
);
```

Key Features:

1. **Function Discovery**:
   - AST parsing to extract all functions
   - Hashing for change detection
   - Line number tracking
2. **Test Detection**:
   - Scans test files for `test_*` functions
   - Maps tests to source functions (naming convention)
   - Tracks test counts per function
3. **Intelligent Test Generation**:

```
untested = db.get_functions_needing_tests()
# Returns functions with:
# - has_test = False, OR
# - function_hash changed (source modified), OR
# - test_file deleted

for func in untested:
    generate_test(func)
    db.update_function(func.id, has_test=True)
```

4. **Coverage Metrics**:
   - Per-function coverage
   - Per-file coverage
   - Overall project coverage
   - Trend analysis (coverage_history)

Why Needed:

- ✅ **Prevents redundant generation** (don't regenerate existing tests)
- ✅ **Detects test gaps** (which functions lack tests)
- ✅ **Change detection** (function modified → update test)
- ✅ **Cleanup** (source deleted → delete orphaned tests)
- ✅ **Metrics** (track coverage over time)

Q7: Describe the Context Assembler and its role

Answer: `ContextAssembler` (`src/context_assembler.py`) **forces comprehensive context gathering before LLM calls** - a critical pattern identified in Phase 1 architecture review.

Problem Solved: Previously, the LLM was called without pre-gathering context, leading to:

- ❌ Incomplete tests (missing edge cases)
- ❌ Poor coverage (not aware of related code)
- ❌ Hallucinations (no grounding in codebase)

Solution - 9 Context Types:

```
@dataclass
class AssembledContext:
    target_code: str           # 1. The function to test
    related_functions: List    # 2. Semantically similar (embeddings)
    dependencies: List         # 3. Imports, external packages
    callers: List             # 4. Who calls this function (symbol index)
    callees: List             # 5. What this function calls
    usage_examples: List      # 6. Real usage from codebase
    git_history: List         # 7. Recent changes (Git)
    existing_tests: List      # 8. Similar tests (for consistency)
    metadata: Dict            # 9. File info, line numbers, etc.

    quality_score: float      # How complete is this context?
```

Assembly Process:

```
def assemble(self, source_code, file_path, function_name, max_related=5):
    context = AssembledContext()

    # 1. Target code (always present)
    context.target_code = source_code

    # 2. Related functions (semantic search via embeddings)
    if self.hybrid_search and function_name:
        results = self.hybrid_search.search(function_name,
top_k=max_related)
        context.related_functions = results
    else:
        results = self.embedding_store.search_similar_code(source_code,
max_related)
        context.related_functions = results

    # 3. Dependencies (AST parsing)
    tree = ast.parse(source_code)
    imports = extract_imports(tree)
    context.dependencies = imports
    # 4. Callers (who uses this function)
    if self.symbol_index and function_name:
        callers = self.symbol_index.get_callers(function_name, file_path)
        context.callers = callers

    # 5. Callees (what this calls)
    calls = extract_function_calls(tree)
    context.callees = calls

    # 6. Usage examples (search codebase)
    if function_name:
        usage = grep_codebase(function_name)
        context.usage_examples = usage[:3]

    # 7. Git history
    if self.git_integration:
        history = self.git_integration.get_file_history(file_path,
max_commits=5)
        context.git_history = history

    # 8. Existing tests (check test directory)
    test_file = get_corresponding_test_file(file_path)
    if test_file.exists():
        context.existing_tests = test_file.read_text()

    # 9. Metadata
    context.metadata = {
        "file_path": file_path,
        "function_name": function_name,
        "language": "python"
    }

    # Calculate quality score (0.0-1.0)
    context.quality_score = calculate_quality(context)

    return context
```

LLM Prompt Integration:

```
def to_llm_prompt_section(self) -> str:
    prompt = f"""
=== CODE TO TEST ===
{self.target_code}

=== RELATED CODE (for context) ===
{format_related(self.related_functions)}

=== DEPENDENCIES ===
{format_imports(self.dependencies)}

=== USAGE EXAMPLES ===
{format_examples(self.usage_examples)}
```

```
=== CALLERS (who uses this) ===
{format_callers(self.callers)}

=== CALLEES (what this calls) ===
{format_callees(self.callees)}

=== GIT HISTORY (recent changes) ===
{format_history(self.git_history)}

=== EXISTING TESTS (for consistency) ===
{self.existing_tests or "None"}

Quality Score: {self.quality_score:.2f} (0.0-1.0)
"""
    return prompt
```

Quality Score Calculation:

```python
def calculate_quality(context) -> float:
    score = 0.0
    weights = {
        "target_code": 0.20,       # Must have
        "related_functions": 0.15,
        "dependencies": 0.15,
        "callers": 0.10,
        "callees": 0.10,
        "usage_examples": 0.10,
        "git_history": 0.10,
        "existing_tests": 0.10
    }

    for field, weight in weights.items():
        if has_content(getattr(context, field)):
            score += weight

    return score
```

Integration Points:

- Called by `TestGenerator._create_test()`
- Called by `TestGenerator._update_test()`
- Ensures **ALL** context is gathered **BEFORE** LLM call
- Prevents "blind" test generation

Benefits:

- ✅ **Comprehensive tests** (aware of full context)
- ✅ **Better coverage** (knows callers/callees)
- ✅ **Consistent style** (follows existing tests)

- ✅ **Grounded in reality** (uses actual codebase examples)

Q8: How does the system handle multiple LLM providers?

Answer: The system uses a **Provider Abstraction Pattern** for LLM flexibility:

Architecture:

1. **Base Interface** (`src/llm_providers.py`):

```
class BaseLLMProvider(ABC):
    @abstractmethod
    def generate(self, prompt: str, **kwargs) -> LLMResponse:
        """Generate text from prompt"""
        pass

    @abstractmethod
    def get_langchain_model(self):
        """Get LangChain-compatible model"""
        pass
```

2. **Provider Implementations:**
   - **OllamaProvider**: Local inference (qwen3-coder:30b)
   - **OpenAIProvider**: OpenAI API (gpt-4, gpt-3.5-turbo)
   - **GeminiProvider**: Google Gemini (gemini-1.5-pro/flash)

3. **Factory Pattern:**

```
def get_llm_provider() -> BaseLLMProvider:
    provider = settings.llm_provider  # from .env

    if provider == "ollama":
        return OllamaProvider(
            base_url=settings.ollama_base_url,
            model=settings.ollama_model
        )
    elif provider == "openai":
        return OpenAIProvider(
            api_key=settings.openai_api_key,
            model=settings.openai_model
        )
    elif provider == "gemini":
        return GeminiProvider(
            api_key=settings.google_api_key,
            model=settings.google_model
        )
    else:
        raise ValueError(f"Unknown provider: {provider}")
```

4. **LangChain Integration:** Each provider returns a LangChain-compatible model:

```python
# OllamaProvider
def get_langchain_model(self):
    from langchain_ollama import ChatOllama
    return ChatOllama(
        base_url=self.base_url,
        model=self.model,
        temperature=0.2
    )

# OpenAIProvider
def get_langchain_model(self):
    from langchain_openai import ChatOpenAI
    return ChatOpenAI(
        api_key=self.api_key,
        model=self.model
    )

# GeminiProvider
def get_langchain_model(self):
    from langchain_google_genai import ChatGoogleGenerativeAI
    return ChatGoogleGenerativeAI(
        google_api_key=self.api_key,
        model=self.model
    )
```

Configuration (`.env`):

```bash
# Choose provider
LLM_PROVIDER=ollama  # or 'openai', 'gemini'

# Ollama settings
OLLAMA_BASE_URL=http://localhost:11434
OLLAMA_MODEL=qwen3-coder:30b

# OpenAI settings
OPENAI_API_KEY=sk-...
OPENAI_MODEL=gpt-4

# Gemini settings
GOOGLE_API_KEY=AIza...
GOOGLE_MODEL=gemini-1.5-pro
```

Embedding Provider Abstraction: Similarly, embeddings are provider-agnostic:

```python
def get_chroma_embedding_function(provider: str):
    if provider == "ollama":
        return OllamaEmbeddingFunction(
            url="http://localhost:11434/api/embeddings",
            model_name="qwen3-embedding:8b"
        )
    elif provider == "openai":
        return OpenAIEmbeddingFunction(
            api_key=settings.openai_api_key,
            model_name="text-embedding-3-small"
        )
    elif provider == "gemini":
        return GoogleGenerativeAIEmbeddingFunction(
            api_key=settings.google_api_key,
            model_name="models/embedding-001"
        )
```

Benefits:

- ✅ **Provider agnostic** (switch via config)
- ✅ **Local + Cloud** (Ollama for privacy, OpenAI/Gemini for power)
- ✅ **Cost optimization** (use cheaper models for some tasks)
- ✅ **Fallback** (if one provider fails, try another)
- ✅ **A/B testing** (compare provider quality)

Q9: Explain the Observability Architecture

Answer: The system implements **enterprise-grade observability** with 3 pillars:

1. Metrics (`src/observability/metrics.py`):

- **Counter**: Incremental counts (tests_generated, errors_total)
- **Gauge**: Point-in-time values (active_sessions, queue_length)
- **Histogram**: Distribution of values (latency, token_count)

```python
# Metrics examples
TESTS_GENERATED = Counter('tests_generated_total', 'Total tests
generated')
TEST_GENERATION_LATENCY = Histogram('test_generation_seconds', 'Test
generation latency')
ACTIVE_SESSIONS = Gauge('active_sessions', 'Number of active sessions')
LLM_TOKENS_USED = Counter('llm_tokens_total', 'Total LLM tokens used',
['provider'])
```

2. Distributed Tracing (`src/observability/tracer.py`):

- Uses **OpenTelemetry** for trace instrumentation
- Tracks request flow across components

- Parent-child span relationships

```
from opentelemetry import trace

tracer = trace.get_tracer(__name__)

@tracer.start_as_current_span("generate_test")
def generate_test(file_path: str):
    with tracer.start_as_current_span("assemble_context") as span:
        span.set_attribute("file_path", file_path)
        context = assembler.assemble(...)

    with tracer.start_as_current_span("llm_call") as span:
        span.set_attribute("provider", settings.llm_provider)
        span.set_attribute("model", settings.ollama_model)
        tests = orchestrator.generate_tests(...)
        span.set_attribute("response_length", len(tests))

    return tests
```

Trace Example:

```
generate_test (duration: 5.2s)
├── assemble_context (1.2s)
│   ├── symbol_search (0.3s)
│   ├── embedding_search (0.7s)
│   └── git_history (0.2s)
├── llm_call (3.8s)
│   ├── input_guardrails (0.1s)
│   ├── llm_inference (3.5s)
│   └── output_guardrails (0.2s)
└── save_test (0.2s)
```

3. Structured Logging (`src/observability/logger.py`):

- JSON-formatted logs for machine parsing
- Correlation IDs for request tracking
- Log levels (DEBUG, INFO, WARNING, ERROR, CRITICAL)

```
logger.info(
    "Test generated",
    extra={
        "session_id": session_id,
        "file_path": file_path,
        "function_count": 3,
        "test_count": 9,
        "coverage": 0.92,
        "duration_ms": 5200
    }
)
```

Prometheus Exporter (`src/observability/prometheus_exporter.py`):

- HTTP server on port 8000
- `/metrics` endpoint for Prometheus scraping
- Exports all counters, gauges, histograms

Console Tracker (`src/console_tracker.py`):

- Rich, colorful console output
- Component-specific sections
- Real-time progress tracking

```
tracker = get_tracker()

tracker.section_header("Orchestrator", "Test Generation")
tracker.llm_call("qwen3-coder:30b", 1250, 3.5)
tracker.tool_call("search_code", "factorial", 0.3, {"results": 5})
tracker.guardrail_check("input", "PASSED", "No PII detected")
```

Benefits:

- 📊 **Real-time monitoring** (Grafana dashboards)
- 🔍 **Root cause analysis** (distributed tracing)
- 📈 **Performance optimization** (identify bottlenecks)
- 🚨 **Alerting** (Prometheus AlertManager)
- 📝 **Audit trail** (structured logs)

Q10: What is the Evaluation (Evals) System?

Answer: The Evals system (`src/evals/`) is an **enterprise-grade, multi-level evaluation framework** that validates test generation quality.

5 Evaluation Levels:

1. **UNIT**: Function-level micro-benchmarks
2. **COMPONENT**: Module-level test quality
3. **AGENT**: Planner/Coder/Critic performance
4. **SYSTEM**: Safety guardrails, end-to-end
5. **BUSINESS**: ROI, 90/90 goals, cost

6 Test Quality Metrics (`src/evals/metrics/test_quality.py`):

Metric	Weight	Measures
Correctness	30%	Syntax valid + tests execute

Metric	Weight	Measures
Coverage	25%	% of source code covered (goal: 90%)
Completeness	20%	Edge cases, exceptions, parametrized
Determinism	10%	No random/time without mocks
Assertions	10%	3+ assertions per test
Mocking	5%	External deps properly mocked

Metric Calculators:

```python
# Correctness
class CorrectnessCalculator(BaseMetricCalculator):
    def calculate(self, test_code, source_code) -> float:
        score = 0.0

        # Syntax check (0.5 points)
        try:
            ast.parse(test_code)
            score += 0.5
        except SyntaxError:
            return 0.0

        # Execution check (0.5 points)
        result = run_pytest(test_code)
        if result.returncode in [0, 1]:  # 0=pass, 1=fail
            score += 0.5

        return score

# Coverage
class CoverageCalculator(BaseMetricCalculator):
    def calculate(self, test_code, source_code) -> float:
        # Run pytest with coverage
        result = run_coverage(test_code, source_code)

        # Parse output for coverage percentage
        coverage = parse_coverage_output(result.stdout)
        return coverage  # 0.0-1.0

# Completeness
class CompletenessCalculator(BaseMetricCalculator):
    def calculate(self, test_code, source_code) -> float:
        score = 0.0

        # Multiple test cases (0.3)
        test_count = count_test_functions(test_code)
        if test_count >= 3:
            score += 0.3
        # Exception testing (0.3)
        if "pytest.raises" in test_code or "raises" in test_code:
            score += 0.3

        # Edge cases (0.2)
        if has_edge_cases(test_code):  # None, [], {}, 0, -1
            score += 0.2

        # Parametrized (0.2)
        if "@pytest.mark.parametrize" in test_code:
            score += 0.2

        return min(score, 1.0)
```

Agent Evaluations (`src/evals/agents/agent_evals.py`):

- **PlannerEvaluator**: Task decomposition quality
    - Completeness (30%): Has all steps
    - Tool Accuracy (30%): Correct tools selected
    - Efficiency (20%): Optimal step count (5-8)
    - Goal Alignment (20%): Targets 90/90 goals
- **CoderEvaluator**: Test generation quality
    - Syntax Correctness (25%): Valid code
    - Framework Usage (15%): Proper pytest/JUnit/Jest
    - Coverage Goal (25%): Achieves 90%
    - Pass Rate Goal (25%): 90% tests pass
    - Code Quality (10%): Documentation, naming
- **CriticEvaluator**: Review effectiveness
    - Detection Accuracy (35%): Catches issues
    - False Positive Rate (25%): Low false alarms
    - Completeness (20%): Reviews all dimensions
    - Actionable Feedback (20%): Specific suggestions

Safety Evaluations (`src/evals/metrics/safety_evals.py`):

- PII detection (emails, phones, SSNs)
- Secret protection (API keys, passwords)
- Prompt injection blocking
- File boundary enforcement
- Determinism enforcement

Regression Detection (`src/evals/reporters/result_storage.py`):

```
class RegressionDetector:
    def check_regression(self, current: EvalResult) -> Dict:
        baseline = storage.get_baseline(current.eval_name)

        if not baseline:
            return {"has_regression": False}

        regressions = []
        for metric_name, metric in current.metrics.items():
            baseline_value = baseline.metrics[metric_name].value
            current_value = metric.value
            delta = current_value - baseline_value

            # Regression if drop > 5%
            if delta < -0.05:
                regressions.append({
                    "metric": metric_name,
                    "current": current_value,
                    "baseline": baseline_value,
                    "delta": delta,
                    "delta_percent": (delta / baseline_value) * 100
                })

        return {
            "has_regression": len(regressions) > 0,
            "regressions": regressions
        }
```

90/90 Goal Tracking:

```
class GoalAchievementCalculator:
    @staticmethod
    def calculate_goal_score(coverage: float, pass_rate: float) -> Dict:
        return {
            "coverage_met": coverage >= 0.90,
            "pass_rate_met": pass_rate >= 0.90,
            "both_goals_met": coverage >= 0.90 and pass_rate >= 0.90,
            "coverage_gap": max(0.0, 0.90 - coverage),
            "pass_rate_gap": max(0.0, 0.90 - pass_rate),
            "goal_score": (coverage + pass_rate) / 2.0
        }
```

Report Generation:

- **Console**: Colorful terminal output
- **Markdown**: For CI/CD, GitHub PRs
- **JSON**: Programmatic access
- **HTML**: Dashboards, web viewing

Usage:

```
from src.evals.runner import EvalRunner

runner = EvalRunner(workspace_dir=Path("evals"))

# Evaluate generated tests
results = runner.evaluate_generated_tests(
    test_code=generated_tests,
    source_code=source_code,
    language="python",
    check_goals=True
)

# Output:
# Test Quality: 89.5%
# Coverage: 87.0% (Goal: 90%) ⚠ Gap: 3.0%
# Pass Rate: 93.0% (Goal: 90%) ✅
```

Q11: How does the system handle test updates vs. test creation?

Answer: The system has **intelligent test lifecycle management** through `TestGenerator` (`src/test_generator.py`):

Test Actions:

1. **CREATE**: New function, no test exists
2. **UPDATE**: Function modified, test needs updating
3. **DELETE**: Function removed, orphan test cleanup
4. **SKIP**: Test exists and function unchanged

Create Test Flow:

```
def _create_test(self, analysis, orchestrator):
    # 1. Extract metadata
    module_name = analysis.source_file.stem  # e.g., "main"
    functions = extract_functions_via_ast(source_code)

    # 2. Build comprehensive context
    assembled_context = self.context_assembler.assemble(
        source_code=source_code,
        file_path=str(analysis.source_file),
        function_name=None,
        max_related=5
    )

    # 3. Add explicit requirements to context
    context_section = assembled_context.to_llm_prompt_section()
    context_section += f"""
=== IMPORTANT TEST FILE REQUIREMENTS ===
1. DO NOT copy/paste the source code into the test file
```

```
2. Use proper imports at the top:
   from {module_name} import {', '.join(functions)}
3. Generate comprehensive tests for ALL functions:
   - {'\n   - '.join(functions)}
4. Use pytest framework with proper assertions
5. Include edge cases, error cases, and normal cases
"""

    # 4. Generate tests
    tests = orchestrator.generate_tests(
        target_code=source_code,
        file_path=str(analysis.source_file),
        function_name=None,
        context=context_section
    )

    # 5. Save and update DB
    analysis.test_file.parent.mkdir(parents=True, exist_ok=True)
    analysis.test_file.write_text(tests, encoding='utf-8')
    self.tracking_db.sync_from_codebase(...)
```

Update Test Flow:

```
def _update_test(self, analysis, orchestrator):
    # 1. Read existing test file
    existing_tests = ""
    if analysis.test_file.exists():
        existing_tests = analysis.test_file.read_text(encoding='utf-8')

    # 2. Build context (same as create)
    assembled_context = self.context_assembler.assemble(...)
    context_section = assembled_context.to_llm_prompt_section()

    # 3. Add existing tests and merge instructions
    context_section += f"""
=== EXISTING TEST FILE ===
{existing_tests}

=== UPDATE INSTRUCTIONS ===
You are UPDATING existing tests. Follow these rules:
1. DO NOT duplicate the source code in the test file
2. Use proper imports: from {module_name} import {', '.join(functions)}
3. Keep all existing test functions that are still valid
4. Add new test functions for any new functions in source
5. Update test functions if source function signature changed
6. Remove test functions if source function was deleted
7. Fix any import issues (don't redefine, import them)
8. Maintain the same test structure and style

Current source file has these functions:
- {'\n- '.join(functions)}
```

```
Make sure there are tests for ALL of these functions.
"""

    # 4. Generate updated tests
    tests = orchestrator.generate_tests(...)

    # 5. Overwrite test file
    analysis.test_file.write_text(tests, encoding='utf-8')
    self.tracking_db.sync_from_codebase(...)
```

Key Differences CREATE vs UPDATE:

Aspect	CREATE	UPDATE
Reads existing test	No	Yes
Instructions	"Generate comprehensive tests"	"Merge with existing, preserve structure"
Context	New code only	Existing tests + new code
Risk	None (new file)	Overwrite (must preserve valid tests)

Change Detection:

```
def _analyze_changes_from_files(self, modified_files: List[str]) ->
List[TestAction]:
    actions = []

    for source_file in modified_files:
        test_file = get_corresponding_test_file(source_file)

        if not test_file.exists():
            # No test file → CREATE
            actions.append(TestAction(
                action="create",
                source_file=source_file,
                test_file=test_file
            ))
        else:
            # Test exists → check if source changed
            source_hash = sha256(source_file.read_bytes())
            cached_hash = self.tracking_db.get_file_hash(source_file)

            if source_hash != cached_hash:
                # Source modified → UPDATE
                actions.append(TestAction(
                    action="update",
                    source_file=source_file,
                    test_file=test_file
                ))
            else:
                # No change → SKIP
                actions.append(TestAction(
                    action="skip",
                    source_file=source_file,
                    test_file=test_file
                ))

    return actions
```

Benefits:

- ✅ **Preserves manual edits** (UPDATE doesn't blindly overwrite)
- ✅ **Consistent structure** (follows existing test style)
- ✅ **Incremental** (only updates what changed)
- ✅ **Safe** (explicit instructions to LLM prevent data loss)

Q12: Explain the Hybrid Search Engine

Answer: `HybridSearchEngine` (`src/hybrid_search.py`) combines **3 search methods** for optimal code retrieval:

1. Exact Search (Symbol Index):

```
# O(1) lookup for exact matches
results = symbol_index.search("factorial")
# Returns: {
#   "file_path": "src/math_utils.py",
#   "start_line": 10,
#   "end_line": 15,
#   "type": "function"
# }
```

2. Semantic Search (Embeddings):

```
# Cosine similarity search
results = embedding_store.search_similar_code(
    query="calculate fibonacci sequence",
    n_results=10
)
# Returns semantically similar code even if names differ
```

3. Keyword Search (BM25-style):

```
# Full-text search with TF-IDF weighting
results = keyword_search(
    query="fibonacci",
    corpus=all_code_chunks
)
```

Reciprocal Rank Fusion (RRF):

```python
def rrf_fusion(
    exact_results: List,
    semantic_results: List,
    keyword_results: List,
    k: int = 60  # RRF constant
) -> List:
    """
    Combine multiple ranked lists using RRF.

    RRF Score = Σ 1 / (k + rank_i)
    """
    scores = defaultdict(float)

    # Score exact matches (highest weight)
    for rank, result in enumerate(exact_results, start=1):
        scores[result.id] += 3.0 / (k + rank)  # 3x weight

    # Score semantic matches
    for rank, result in enumerate(semantic_results, start=1):
        scores[result.id] += 1.0 / (k + rank)

    # Score keyword matches
    for rank, result in enumerate(keyword_results, start=1):
        scores[result.id] += 0.5 / (k + rank)  # 0.5x weight

    # Sort by score
    ranked = sorted(scores.items(), key=lambda x: x[1], reverse=True)
    return ranked
```

Cross-Encoder Reranking:

```python
def rerank(self, query: str, candidates: List, top_k: int) -> List:
    """
    Use cross-encoder model to rerank candidates.

    Cross-encoder computes similarity score for (query, candidate) pairs.
    More accurate than dot-product of embeddings.
    """
    if not self.reranker:
        return candidates[:top_k]

    # Compute cross-encoder scores
    pairs = [(query, candidate.content) for candidate in candidates]
    scores = self.reranker.predict(pairs)
    # Combine with RRF scores
    for candidate, score in zip(candidates, scores):
        candidate.rerank_score = score
        candidate.final_score = (
            0.7 * candidate.rrf_score +
            0.3 * candidate.rerank_score
        )

    # Sort by final score
    reranked = sorted(candidates, key=lambda x: x.final_score,
reverse=True)
    return reranked[:top_k]
```

Complete Hybrid Search:

```
class HybridSearchEngine:
    def search(
        self,
        query: str,
        top_k: int = 10,
        use_reranking: bool = True
    ) -> List[SearchResult]:
        # 1. Exact search
        exact = self.symbol_index.search(query) if self.symbol_index else
[]

        # 2. Semantic search
        semantic = self.embedding_store.search_similar_code(query,
n_results=50)

        # 3. Keyword search
        keyword = self._keyword_search(query, top_k=50)

        # 4. Reciprocal Rank Fusion
        fused = self._rrf_fusion(exact, semantic, keyword, k=60)

        # 5. Cross-encoder reranking (optional)
        if use_reranking and self.reranker:
            reranked = self.rerank(query, fused[:top_k * 3], top_k)
            return reranked

        return fused[:top_k]
```

Benefits:

- ✅ **Recall**: Exact finds precise matches, semantic finds similar concepts
- ✅ **Precision**: Reranking boosts most relevant results
- ✅ **Robustness**: Multiple methods reduce risk of missing relevant code
- ✅ **Performance**: RRF is fast, reranking only on top candidates

Example:

```
query = "function that calculates fibonacci"

# Exact: Matches "fibonacci" function name
# Semantic: Matches recursive number sequences
# Keyword: Matches "fibonacci" in comments/docstrings
# RRF: Combines all three
# Reranking: Boosts exact match + semantic similarity

Final results (sorted by relevance):
1. fibonacci(n) - direct match
2. fibonacci_memoized(n) - variant
3. fibonacci_iterative(n) - alternative implementation
4. lucas_numbers(n) - similar recursive sequence
5. factorial(n) - different but related recursive pattern
```

Q13: How does the Git Integration work?

Answer: `GitIntegration` (`src/git_integration.py`) provides **comprehensive Git change detection**:

Key Features:

1. **Change Detection:**

```
def get_changed_files_since_last_commit(self) -> List[FileChange]:
    """Get files changed since last commit (staged + unstaged)."""
    try:
        # Get diff from HEAD
        diff = self.repo.head.commit.diff(None)

        changes = []
        for diff_item in diff:
            if diff_item.a_path.endswith('.py'):  # Filter by extension
                changes.append(FileChange(
                    file_path=diff_item.a_path,
                    change_type=diff_item.change_type,  # A=added,
M=modified, D=deleted
                    lines_added=diff_item.diff.count('+'),
                    lines_removed=diff_item.diff.count('-')
                ))

        # Also get untracked files
        untracked = self.repo.untracked_files
        for file_path in untracked:
            if file_path.endswith('.py'):
                changes.append(FileChange(
                    file_path=file_path,
                    change_type='A',  # Treat as added
                    lines_added=count_lines(file_path),
                    lines_removed=0
                ))

        return changes

    except ValueError:  # No commits yet
        # Return all files as new
        return get_all_files_as_new()
```

2. **Function-Level Diff:**

```python
def get_new_functions_since_commit(
    self,
    file_path: str,
    base_commit: str = "HEAD"
) -> List[FunctionInfo]:
    """Extract new/modified functions in a file."""

    # Get current version
    current_code = Path(file_path).read_text()
    current_functions = extract_functions(current_code)

    try:
        # Get previous version from git
        previous_code = self.repo.git.show(f"{base_commit}:{file_path}")
        previous_functions = extract_functions(previous_code)
    except:
        # File didn't exist before
        return current_functions

    # Compare function signatures and bodies
    new_functions = []
    for func in current_functions:
        prev_func = find_function(previous_functions, func.name)

        if not prev_func:
            # Brand new function
            new_functions.append(func)
        elif func.hash != prev_func.hash:
            # Function modified
            new_functions.append(func)

    return new_functions
```

3. **Git Status Summary:**

```
def get_status(self) -> Dict:
    """Get comprehensive git status."""
    try:
        # Modified files
        modified = [
            {
                "path": item.a_path,
                "lines_added": count_additions(item),
                "lines_removed": count_deletions(item),
                "status": "modified"
            }
            for item in self.repo.index.diff(None)
        ]

        # Staged files
        staged = [
            {
                "path": item.a_path,
                "status": "staged"
            }
            for item in self.repo.index.diff("HEAD")
        ]

        # Untracked files
        untracked = [
            {
                "path": path,
                "size": Path(path).stat().st_size,
                "status": "untracked"
            }
            for path in self.repo.untracked_files
        ]

        return {
            "modified": modified,
            "staged": staged,
            "untracked": untracked,
            "total_files": len(modified) + len(staged) + len(untracked),
            "clean": len(modified) == 0 and len(untracked) == 0
        }

    except ValueError:  # No commits
        return {
            "modified": [],
            "staged": [],
            "untracked": get_all_files(),
            "no_commits": True
        }
```

4. **File History:**

```python
def get_file_history(
    self,
    file_path: str,
    max_commits: int = 10
) -> List[CommitInfo]:
    """Get commit history for a file."""
    commits = list(self.repo.iter_commits(
        paths=file_path,
        max_count=max_commits
    ))

    return [
        CommitInfo(
            sha=commit.hexsha[:8],
            message=commit.message.strip(),
            author=commit.author.name,
            date=commit.committed_datetime,
            changes=get_file_diff(commit, file_path)
        )
        for commit in commits
    ]
```

Integration Points:

- **TestGenerator**: Detects which files need test generation
- **RAGRetriever**: Provides git history as context
- **ContextAssembler**: Includes recent changes in context

Benefits:

- ✅ **Selective generation** (only changed code)
- ✅ **Incremental updates** (not full regeneration)
- ✅ **Historical context** (why code changed)
- ✅ **Handles edge cases** (repos without commits, untracked files)

Q14: What are the data persistence strategies?

Answer: The system uses **multiple storage backends** for different data types:

1. SQLite Databases:

A. Test Tracking DB (`tests/.test_tracking.db`):

- **Purpose**: Function-level test tracking
- **Tables**: source_functions, test_cases, coverage_history, index_metadata, file_relationships
- **Size**: Grows with codebase (typically < 10MB)
- **Backup**: Committed to git (small size)

B. Audit Logs DB (`data/audit_logs.db`):

- **Purpose**: Security event logging
- **Tables**: events, sessions, policy_decisions
- **Size**: Grows with usage (rotate periodically)
- **Backup**: Not committed (contains sensitive data)

C. Evals DB (`evals/evals.db`):

- **Purpose**: Evaluation history
- **Tables**: eval_results, eval_suites, baselines
- **Size**: Moderate (KB per eval run)
- **Backup**: Optional (for trend analysis)

2. ChromaDB (`chroma_db/`):

- **Purpose**: Vector embeddings for semantic search
- **Storage**: Persistent on-disk database
- **Size**: ~100MB per 10K code chunks
- **Backup**: Can rebuild from source (takes time)
- **Format**: SQLite + numpy arrays

3. JSON Files (deprecated, being migrated):

- `.index_metadata.json` → Moved to test_tracking.db
- `.test_relationships.json` → Moved to test_tracking.db
- `.symbol_index.json` → Still used (fast O(1) lookups)

4. File System:

- **Source code**: Read-only from SOURCE_CODE_DIR
- **Test code**: Written to TEST_OUTPUT_DIR
- **Logs**: `logs/` directory (rotating file handler)
- **Metrics**: In-memory (exported via /metrics endpoint)

Data Flow:

```
Source Code
    ↓
Incremental Indexer
    ↓
    ├─→ ChromaDB (embeddings)
    ├─→ Symbol Index (exact lookups)
    └─→ Test Tracking DB (metadata)

Git Changes
    ↓
Test Generator
    ↓
    ├─→ Test Files (filesystem)
    └─→ Test Tracking DB (update status)

LLM Calls
```

```
     ↓
Guard Manager
     ↓
Audit Logger
     ↓
Audit DB (events)
```

Persistence Strategy:

Data Type	Storage	Persistence	Backup
Source code	Filesystem	Git	Yes
Test code	Filesystem	Git	Yes
Embeddings	ChromaDB	Disk	Rebuild
Symbols	JSON	Disk	Git
Tracking	SQLite	Disk	Git
Audit logs	SQLite	Disk	No (sensitive)
Evals	SQLite	Disk	Optional
Metrics	Prometheus	Time-series	External
Traces	OpenTelemetry	External	External

Data Lifecycle:

1. **Initialization** (`make init`): Create empty DBs
2. **Indexing** (`make index`): Populate embeddings + symbols
3. **Generation** (`make generate`): Update tracking DB, write tests
4. **Cleanup** (`make clean-data`): Remove all persistent data
5. **Migration** (`migrate_to_db.py`): Move JSON → SQLite

Q15: Describe the complete test generation workflow

Answer: The **end-to-end workflow** from code change to test generation:

Phase 1: Initialization

```
$ make init
# 1. Create virtual environment
# 2. Install dependencies (LangChain 1.0, LangGraph, ChromaDB)
# 3. Pull Ollama models (if LLM_PROVIDER=ollama)
# 4. Verify installation
```

Phase 2: Configuration

```
$ cat .env
SOURCE_CODE_DIR=/Users/naren/dev/repos/MyProject/src
TEST_OUTPUT_DIR=/Users/naren/dev/repos/MyProject/tests
LLM_PROVIDER=ollama
OLLAMA_MODEL=qwen3-coder:30b
```

Phase 3: Initial Indexing

```
$ make index
# → main.py index_codebase()
#     ├─ IncrementalIndexer.index()
#     │  ├─ Scan source directory
#     │  ├─ Parse files with AST
#     │  ├─ Generate embeddings (ChromaDB)
#     │  ├─ Extract symbols (SymbolIndex)
#     │  └─ Update metadata (TestTrackingDB)
#     └─ Output: "Indexed 150 files, 1200 chunks"
```

Phase 4: Test Generation Trigger

```
$ make generate
# → main.py generate_smart()
```

Phase 5: Change Detection

```
# TestGenerator.generate()
1. Git Integration → get_changed_files_since_last_commit()
2. TestTrackingDB → get_functions_needing_tests()
3. Combine results → List[source_file, test_file, action]

Actions:
- CREATE: New file, no test
- UPDATE: File modified, test exists
- SKIP: No changes
- DELETE: File removed, orphan test
```

Phase 6: Context Assembly (per file)

```
# ContextAssembler.assemble()
1. Symbol Index → get_function(name) # Exact location
2. Embeddings → search_similar_code(code) # Top 5 related
3. Hybrid Search → combine exact + semantic + keyword
4. Git History → get_file_history(path, max=5)
5. Dependencies → extract_imports(ast)
```

```
6. Callers → symbol_index.get_callers(function)
7. Callees → extract_function_calls(ast)
8. Usage Examples → grep_codebase(function_name)
9. Existing Tests → read_test_file(test_path)

Output: AssembledContext (quality_score: 0.85)
```

Phase 7: Guardrails Check (Input)

```
# GuardManager.check_input()
1. Policy Engine → ALLOW/DENY/REVIEW
2. Input Guardrails:
   - PII detection → redact if found
   - Prompt injection → block if detected
   - Toxic content → filter
3. Schema Validator → validate params, auto-correct
4. Budget Tracker → check token/cost limits
5. Audit Logger → log input event

If REVIEW or budget exceeded → HITL prompt
If DENY → raise SecurityError
```

Phase 8: LLM Test Generation

```
# Orchestrator.generate_tests()
1. Format prompt with assembled context
2. Add system instructions:
   - "Generate comprehensive tests"
   - "Use pytest framework"
   - "Include edge cases, errors, normal cases"
   - "Achieve 90% coverage"
   - "DO NOT duplicate source code"
   - "USE TOOLS: search_code, execute_tests"
3. Create LangGraph ReAct agent
4. Invoke agent with config {recursion_limit: 50}

Agent Loop:
├─ Thought: "I need to search for related tests"
├─ Action: search_code("factorial")
├─ Observation: [found 3 related functions]
├─ Thought: "I need to understand usage"
├─ Action: get_code_context("factorial")
├─ Observation: [context with callers/callees]
├─ Thought: "Now I can generate tests"
└─ Final Answer: [test code in JSON or Python]

5. Parse response:
   - Try JSON parsing (structured response)
   - Fallback to markdown code block extraction
```

```
6. Post-process:
   - Extract from ```python blocks
   - Remove markdown artifacts
   - Ensure pytest import
   - Add file header
```

Phase 9: Guardrails Check (Output)

```
# GuardManager.check_output()
1. Output Guardrails:
   - Code safety → no eval, exec, __import__
   - License compliance → check for violations
   - Determinism → flag random/time without mocks
2. File Boundary → ensure writing to tests/ only
3. Constitutional AI → self-verify against principles
4. Audit Logger → log output event
```

Phase 10: Test Execution (Optional)

```
# SandboxExecutor.run_tests()
1. Create Docker container (isolated environment)
2. Copy source + test code
3. Install dependencies
4. Run pytest with coverage
5. Parse results:
   - Tests passed/failed
   - Coverage percentage
   - Errors/warnings
6. Return results
```

Phase 11: Critic Review

```
# Critic.review_tests()
1. Check correctness (syntax, execution)
2. Check coverage (goal: 90%)
3. Check completeness (edge cases, exceptions)
4. Check determinism (no flaky tests)
5. Check assertions (3+ per test)
6. Generate recommendations
7. Calculate quality score (0.0-1.0)

If score < 0.7 → trigger refinement loop
```

Phase 12: Persistence

```
# TestGenerator._create_test() or _update_test()
1. Write test file to TEST_OUTPUT_DIR
2. Update TestTrackingDB:
   - sync_from_codebase()
   - Update function.has_test = True
   - Update function.test_count
   - Record in coverage_history
3. Console output:
   ✓ Created test_factorial.py (3 functions, 9 tests)
```

Phase 13: Metrics & Observability

```
# Automatic instrumentation
1. Metrics:
   - TESTS_GENERATED.inc()
   - TEST_GENERATION_LATENCY.observe(duration)
   - LLM_TOKENS_USED.inc(token_count)
2. Traces:
   - Span: generate_test (5.2s)
   - Span: assemble_context (1.2s)
   - Span: llm_call (3.8s)
3. Logs:
   logger.info("Test generated", extra={...})
```

Complete Flow Diagram:

```
User: make generate
     ↓
TestGenerator.generate()
     ↓
Git.get_changed_files() + DB.get_untested_functions()
     ↓
For each file:
     ├─ ContextAssembler.assemble()
     │    ├─ SymbolIndex.search()
     │    ├─ Embeddings.search()
     │    ├─ HybridSearch.combine()
     │    └─ Git.get_history()
     ↓
     ├─ GuardManager.check_input()
     │    ├─ PolicyEngine.decide()
     │    ├─ InputGuardrails.validate()
     │    └─ BudgetTracker.check()
     ↓
     ├─ Orchestrator.generate_tests()
     │    ├─ LangGraph.ReActAgent.invoke()
     │    │    ├─ Tool: search_code()
     │    │    ├─ Tool: get_context()
     │    │    └─ Final: test code
     │    │
```

```
      │   └─ PostProcess.clean()
      ↓
      ├─ GuardManager.check_output()
      │   ├─ OutputGuardrails.validate()
      │   ├─ ConstitutionalAI.verify()
      │   └─ AuditLogger.log()
      ↓
      ├─ (Optional) SandboxExecutor.run_tests()
      ├─ (Optional) Critic.review_tests()
      ↓
      └─ Save test file + Update DB
          ├─ Filesystem: write test_*.py
          └─ TestTrackingDB: update tracking

Output:
✅ 5 files processed
✅ 3 tests created
✅ 2 tests updated
✅ 0 tests deleted
✅ Overall coverage: 87.5%
```

Time Breakdown (typical):

- Git detection: 0.1s
- Context assembly: 1-2s
- Guardrails (input): 0.2s
- LLM generation: 3-5s
- Guardrails (output): 0.2s
- File write + DB update: 0.1s
- **Total per file: 5-8s**

🤖 SECTION 2: Agentic AI Patterns (Questions 16-30)

Q16: What is the ReAct (Reason + Act) pattern and how is it implemented?

Answer: ReAct is an agent pattern where the model **reasons** about what to do and then **acts** by calling tools, iterating until the task is complete.

Pattern:

```
Thought → Action → Observation → Thought → Action → ... → Answer
```

Implementation (LangGraph 1.0):

```python
from langgraph.prebuilt import create_react_agent

# 1. Define tools
```

```
tools = [
    search_code_tool,
    get_code_context_tool,
    execute_tests_tool,
    check_git_changes_tool
]

# 2. Create agent
agent = create_react_agent(
    model=llm,  # ChatOllama, ChatOpenAI, etc.
    tools=tools,
    prompt=system_prompt
)

# 3. Invoke agent
response = agent.invoke(
    {"messages": [HumanMessage(content="Generate tests for factorial")]},
    config={"recursion_limit": 50}
)
```

Agent Loop (under the hood):

```
Iteration 1:
  Thought: "I need to find the factorial function"
  Action: search_code("factorial")
  Observation: Found in src/math_utils.py

Iteration 2:
  Thought: "I need to understand how it's used"
  Action: get_code_context("factorial")
  Observation: Called by fibonacci(), used in 3 places

Iteration 3:
  Thought: "I should check existing tests"
  Action: check_git_changes("src/math_utils.py")
  Observation: No test file exists yet

Iteration 4:
  Thought: "Now I can generate comprehensive tests"
  Action: FINISH with test code
  Final Answer: [Python test code]
```

System Prompt (key for ReAct):

```
system_prompt = """
You are a test generation agent using the ReAct (Reason + Act) pattern.

Available Tools:
1. search_code(query) - Search codebase semantically
```

```
2. get_code_context(function_name) - Get related code
3. check_git_changes(file_path) - Check recent changes
4. execute_tests(test_code) - Run tests in sandbox

Process:
1. THINK about what information you need
2. USE TOOLS to gather that information
3. REASON about the observations
4. DECIDE on next action or generate tests
5. RESPOND with PURE PYTHON CODE (no markdown, no explanations)

Example:
Thought: "I need to find the function first"
Action: search_code("factorial")
Observation: Found function at src/math.py:10-15
Thought: "Now I need to understand its usage"
Action: get_code_context("factorial")
Observation: Called by 3 functions, accepts int, returns int
Thought: "I have enough context, generate tests"
Final Answer: [test code]

Remember:
- Use tools BEFORE generating tests
- Generate COMPLETE tests (don't truncate)
- Include edge cases, errors, normal cases
- Target 90% coverage
"""
```

Benefits of ReAct:

- ✅ **Transparency**: See agent's reasoning
- ✅ **Tool use**: Agent decides when to use tools
- ✅ **Iterative refinement**: Can gather more info if needed
- ✅ **Self-correction**: Can verify results with execute_tests

Comparison with other patterns:

Pattern	When to Use	Pros	Cons
ReAct	Complex tasks, need reasoning	Transparent, flexible	Token-heavy
Plan-and-Execute	Multi-step with planning	Structured, efficient	Less flexible
Zero-shot	Simple, direct tasks	Fast, cheap	No tool use
Few-shot	Need examples	Better quality	Static examples

Q17: Explain the Plan-and-Execute pattern

Answer: Plan-and-Execute separates planning from execution, creating a structured multi-agent workflow.

Components:

1. **Planner Agent** (`src/planner.py`):
   - Decomposes goal into tasks
   - Builds dependency graph
   - Estimates effort
2. **Executor Agent** (Orchestrator):
   - Executes tasks in order
   - Handles dependencies
   - Reports results

Pattern:

```
Goal → [Planner] → ExecutionPlan → [Executor] → Results
                        ↓
                    Task Graph
                    ├─ Task 1 (ready)
                    ├─ Task 2 (depends on 1)
                    └─ Task 3 (depends on 2)
```

Implementation:

```python
# 1. Planner creates structured plan
planner = PlannerWithDecomposition()
plan = planner.create_plan(
    goal="Generate tests for all changed functions",
    context={"changed_files": ["src/math.py", "src/utils.py"]}
)

# Plan structure:
# {
#   "goal": "Generate tests...",
#   "tasks": [
#     {
#       "id": "task_1",
#       "name": "Check git changes",
#       "tool": "check_git_changes",
#       "dependencies": [],
#       "status": "pending"
#     },
#     {
#       "id": "task_2",
#       "name": "Search for related code",
#       "tool": "search_code",
#       "dependencies": ["task_1"],
#       "status": "pending"
#     },
#     {
#       "id": "task_3",
#       "name": "Generate test for src/math.py",
#       "tool": "generate_tests",
#       "dependencies": ["task_2"],
#       "status": "pending"
#     },
#     {
#       "id": "task_4",
#       "name": "Execute tests",
#       "tool": "execute_tests",
#       "dependencies": ["task_3"],
#       "status": "pending"
#     }
#   ],
#   "task_graph": {
#     "task_1": [],
#     "task_2": ["task_1"],
#     "task_3": ["task_2"],
#     "task_4": ["task_3"]
#   },
#   "estimated_steps": 4,
#   "confidence": 0.85
# }

# 2. Executor runs plan
executor = PlanExecutor(plan)
results = executor.execute()

# Execution flow:
# task_1 (ready) → execute → mark complete
#   ↓
# task_2 (now ready) → execute → mark complete
#   ↓
# task_3 (now ready) → execute → mark complete
#   ↓
# task_4 (now ready) → execute → mark complete
```

Task Dependency Graph:

```
def _build_dependency_graph(tasks: List[Task]) -> Dict[str, List[str]]:
    """Build graph where edge A→B means B depends on A."""
    graph = {task.id: task.dependencies for task in tasks}
    return graph

def _get_ready_tasks(tasks: List[Task], graph: Dict) -> List[Task]:
    """Get tasks with all dependencies completed."""
    completed = {t.id for t in tasks if t.status == TaskStatus.COMPLETED}

    ready = []
    for task in tasks:
        if task.status == TaskStatus.PENDING:
```

```python
            deps_met = all(dep in completed for dep in task.dependencies)
            if deps_met:
                ready.append(task)

    return ready
```

Replanning on Failure:

```python
def execute_with_replanning(self) -> ExecutionResults:
    while not self.plan.is_complete():
        ready_tasks = self._get_ready_tasks()

        for task in ready_tasks:
            try:
                result = self._execute_task(task)
                task.status = TaskStatus.COMPLETED
                task.result = result
            except Exception as e:
                task.status = TaskStatus.FAILED

                # Replan
                console.print(f"[red]Task {task.name} failed: {e}[/red]")
                console.print("[yellow]Replanning...[/yellow]")

                new_tasks = self.planner.replan(
                    failed_task=task,
                    error=e,
                    current_state=self.get_state()
                )

                # Update plan
                self.plan.tasks.extend(new_tasks)
                self._update_dependency_graph()

    return self.plan.results
```

Benefits:

- ✅ **Structured execution**: Clear task breakdown
- ✅ **Parallel execution**: Independent tasks can run concurrently
- ✅ **Progress tracking**: Know which step is executing
- ✅ **Failure recovery**: Replan on errors
- ✅ **Auditability**: Full execution trace

Trade-offs:

- ➖ More complex than ReAct
- ➖ Upfront planning overhead
- ➕ Better for multi-step workflows
- ➕ Easier to debug (structured plan)

Q18: What is Constitutional AI and how is it used for safety?

Answer: Constitutional AI is a self-correcting pattern where the AI system validates its outputs against predefined principles.

Implementation (`src/guardrails/constitutional_ai.py`):

```
class ConstitutionalPrinciple(BaseModel):
    name: str
    rule: str           # The principle/rule
    critique_prompt: str  # How to evaluate violation
    revision_prompt: str  # How to correct violation

class ConstitutionalAI:
    """Self-verification loop for AI outputs."""

    principles = [
        ConstitutionalPrinciple(
            name="no_harmful_code",
            rule="Generated code must not contain eval(), exec(), or
__import__()",
            critique_prompt="Does this code contain eval, exec, or
__import__?",
            revision_prompt="Remove all eval, exec, and __import__ calls"
        ),
        ConstitutionalPrinciple(
            name="no_hardcoded_secrets",
            rule="Code must not contain API keys, passwords, or tokens",
            critique_prompt="Does this code contain hardcoded secrets?",
            revision_prompt="Replace all secrets with environment
variables"
        ),
        ConstitutionalPrinciple(
            name="deterministic_tests",
            rule="Tests must be deterministic (no random, time without
mocks)",
            critique_prompt="Does this test use random or time without
mocking?",
            revision_prompt="Add mocking for all non-deterministic
operations"
        )
    ]

    def verify_and_correct(
        self,
        output: str,
        max_iterations: int = 3
    ) -> Tuple[str, List[str]]:
        """
        Self-verification loop.
        Returns:
            (corrected_output, violations_found)
        """
        violations = []
        corrected = output

        for iteration in range(max_iterations):
            # Check each principle
            current_violations = []

            for principle in self.principles:
                # Critique: Does output violate this principle?
                is_violation = self._critique(corrected, principle)

                if is_violation:
                    current_violations.append(principle.name)
                    violations.append(principle.name)

                    # Revise: Auto-correct the violation
                    corrected = self._revise(corrected, principle)

            # If no violations found, we're done
            if not current_violations:
                break

        return corrected, violations

    def _critique(self, output: str, principle: ConstitutionalPrinciple) -
> bool:
        """Check if output violates principle."""
        # Simple pattern matching (can use LLM for complex checks)
        if principle.name == "no_harmful_code":
            return any(pattern in output for pattern in ["eval(", "exec(",
"__import__"])

        elif principle.name == "no_hardcoded_secrets":
            import re
            # Check for common secret patterns
            patterns = [
                r'api[_-]key\s*=\s*["\'][^"\']+["\']',
                r'password\s*=\s*["\'][^"\']+["\']',
                r'token\s*=\s*["\'][^"\']+["\']',
                r'["\'][A-Za-z0-9]{32,}["\']'  # Long random strings
            ]
            return any(re.search(pat, output, re.IGNORECASE) for pat in
patterns)

        elif principle.name == "deterministic_tests":
            return any(pattern in output for pattern in [
                "random.random()", "datetime.now()", "time.time()",
                "uuid.uuid4()"
            ]) and not any(mock in output for mock in [
                "@patch", "monkeypatch", "mock", "freezegun"
            ])
        return False

    def _revise(self, output: str, principle: ConstitutionalPrinciple) ->
str:
        """Auto-correct violation."""
        if principle.name == "no_harmful_code":
            # Remove dangerous functions
            output = output.replace("eval(", "# REMOVED: eval(")
            output = output.replace("exec(", "# REMOVED: exec(")
            output = output.replace("__import__(", "# REMOVED:
__import__(")

        elif principle.name == "no_hardcoded_secrets":
            import re
            # Replace secrets with env vars
            output = re.sub(
                r'api[_-]key\s*=\s*["\']([^"\']+)["\']',
                'api_key = os.environ.get("API_KEY")',
                output,
                flags=re.IGNORECASE
            )

        elif principle.name == "deterministic_tests":
            # Add comment suggesting mocking
            if "random" in output:
                output = "# TODO: Mock random for determinism\n" + output
            if "datetime.now" in output:
                output = "# TODO: Mock datetime.now for determinism\n" +
output

        return output
```

Integration:

```
# GuardManager uses Constitutional AI
def check_output(self, output: str) -> GuardResult:
    # ... other checks ...

    if self.constitutional_ai:
        corrected, violations =
self.constitutional_ai.verify_and_correct(output)

        if violations:
            console.print(f"[yellow]⚠  Constitutional AI corrected
{len(violations)} violations[/yellow]")
            for violation in violations:
                console.print(f"  - {violation}")

        # Use corrected output
        output = corrected
    # ... continue with output validation ...
```

Benefits:

- ✅ **Self-correcting**: Automatically fixes violations
- ✅ **Principled**: Based on explicit rules
- ✅ **Transparent**: Reports what was corrected
- ✅ **Layered**: Multiple principles checked sequentially
- ✅ **Iterative**: Runs multiple correction loops if needed

Example Flow:

```
Input: Generated test code with eval()

Iteration 1:
  Check no_harmful_code → VIOLATION (found eval)
  Correct: Replace eval() with # REMOVED: eval()

Iteration 2:
  Check no_harmful_code → PASS
  Check no_hardcoded_secrets → PASS
  Check deterministic_tests → PASS

Output: Corrected code without eval()
```

Q19: How does the Tool Calling pattern work?

Answer: Tool Calling allows LLMs to invoke external functions to gather information or perform actions.

Tool Definition (`src/tools.py`):

```python
from langchain_core.tools import BaseTool
from pydantic import BaseModel, Field

class SearchCodeInput(BaseModel):
    """Input for search_code tool."""
    query: str = Field(..., description="Search query for code")
    max_results: int = Field(default=5, description="Maximum results to return")

class SearchCodeTool(BaseTool):
    """Tool for semantic code search."""

    name: str = "search_code"
    description: str = """
    Search the codebase semantically using embeddings.
    Use this to find functions, classes, or code similar to a query.
```

~~~
    Args:
        query: What to search for (e.g., "factorial function", "error
handling")
        max_results: Number of results (default 5)

    Returns:
        List of code chunks with file paths and line numbers
    """
    args_schema: Type[BaseModel] = SearchCodeInput

    # Dependencies
    embedding_store: CodeEmbeddingStore

    def _run(self, query: str, max_results: int = 5) -> str:
        """Execute the tool."""
        results = self.embedding_store.search_similar_code(
            query=query,
            n_results=max_results
        )

        # Format results for LLM
        formatted = []
        for i, result in enumerate(results, 1):
            formatted.append(f"""
Result {i}:
File: {result['file_path']}
Lines: {result['start_line']}-{result['end_line']}
Code:
```python
{result['content']}
~~~

""")

```
    return "\n".join(formatted)
```

Tool Registry:
```python
def get_all_tools() -> List[BaseTool]:
    """Get all available tools for the agent."""
    embedding_store = CodeEmbeddingStore()
    git_integration = GitIntegration()
    rag_retriever = RAGRetriever()
    sandbox_executor = SandboxExecutor()

    tools = [
        SearchCodeTool(embedding_store=embedding_store),
        GetCodeContextTool(rag_retriever=rag_retriever),
```

```
        CheckGitChangesTool(git_integration=git_integration),
        ExecuteTestsTool(sandbox_executor=sandbox_executor),
    ]

    return tools
```

Agent Tool Usage:

```
# LangGraph automatically handles tool calling
agent = create_react_agent(
    model=llm,
    tools=tools
)

response = agent.invoke({
    "messages": [HumanMessage(content="Generate tests for factorial")]
})

# Agent execution trace:
# 1. LLM decides: "I need to find the factorial function"
# 2. LLM calls: search_code(query="factorial")
# 3. Tool executes: SearchCodeTool._run()
# 4. Tool returns: List of matching code
# 5. LLM receives: Tool result as ToolMessage
# 6. LLM decides: "I found it, now get context"
# 7. LLM calls: get_code_context(function_name="factorial")
# 8. ... continues until task complete
```

Tool Call Format (OpenAI Function Calling):

```
{
  "role": "assistant",
  "content": null,
  "tool_calls": [
    {
      "id": "call_abc123",
      "type": "function",
      "function": {
        "name": "search_code",
        "arguments": "{\"query\": \"factorial\", \"max_results\": 5}"
      }
    }
  ]
}
```

Tool Response:

```
{
  "role": "tool",
  "tool_call_id": "call_abc123",
  "content": "Result 1:\nFile: src/math.py\nLines: 10-15\nCode:\ndef
factorial(n):\n  ..."
}
```

Guardrails Integration:

```python
# Before tool call
def call_tool(tool_name: str, params: Dict) -> Any:
    # 1. Pre-execution check
    guard_result = guard_manager.check_tool_call(
        tool=tool_name,
        params=params,
        context={"session_id": session_id}
    )

    if not guard_result.allowed:
        raise SecurityError(guard_result.reason)

    # 2. Use corrected params if provided
    params = guard_result.corrected_params or params

    # 3. Execute tool
    start = time.time()
    result = tools[tool_name].run(**params)
    duration = time.time() - start

    # 4. Post-execution logging
    guard_manager.log_tool_result(
        tool=tool_name,
        success=True,
        duration_ms=duration * 1000
    )

    return result
```

Benefits:

- ✅ **Extensibility**: Easy to add new tools
- ✅ **Type Safety**: Pydantic schemas for validation
- ✅ **Composability**: Tools can call other tools
- ✅ **Observability**: All calls logged and traced
- ✅ **Security**: Guardrails check every call

Q20: What is Human-in-the-Loop (HITL) and when is it used?

Answer: HITL adds human review checkpoints for high-risk operations.

Implementation (`src/guardrails/hitl_manager.py`):

```python
class RiskLevel(str, Enum):
    LOW = "low"
    MEDIUM = "medium"
    HIGH = "high"
    CRITICAL = "critical"

class ApprovalRequest(BaseModel):
    request_id: str
    action: str
    risk_level: RiskLevel
    reason: str
    details: Dict
    proposed_changes: Optional[str] = None

class HITLManager:
    """Manages human-in-the-loop approvals."""

    def __init__(self, interactive: bool = True):
        self.interactive = interactive
        self.approval_queue = []

    def request_approval(
        self,
        action: str,
        risk_level: RiskLevel,
        reason: str,
        details: Dict
    ) -> bool:
        """
        Request human approval.

        Returns:
            True if approved, False if denied
        """
        if not self.interactive:
            # Non-interactive mode: auto-approve LOW/MEDIUM
            return risk_level in [RiskLevel.LOW, RiskLevel.MEDIUM]

        request = ApprovalRequest(
            request_id=str(uuid.uuid4())[:8],
            action=action,
            risk_level=risk_level,
            reason=reason,
            details=details
        )

        # Display request
        self._display_approval_request(request)
        # Get user input
        response = input("\nApprove? (y/n/details): ").strip().lower()

        if response == 'details':
            self._display_details(request)
            response = input("\nApprove after reviewing details? (y/n):
").strip().lower()

        approved = response in ['y', 'yes']

        # Log decision
        self._log_decision(request, approved)

        return approved

    def _display_approval_request(self, request: ApprovalRequest):
        risk_color = {
            RiskLevel.LOW: "green",
            RiskLevel.MEDIUM: "yellow",
            RiskLevel.HIGH: "red",
            RiskLevel.CRITICAL: "red bold"
        }[request.risk_level]

        console.print(f"\n[{risk_color}]🔔 APPROVAL
REQUIRED[/{risk_color}]")
        console.print(f"Request ID: {request.request_id}")
        console.print(f"Action: {request.action}")
        console.print(f"Risk Level: {request.risk_level.value.upper()}")
        console.print(f"Reason: {request.reason}")
```

Risk Assessment:

```
def assess_risk(action: str, params: Dict) -> RiskLevel:
    """Determine risk level of an action."""

    # CRITICAL: System modifications
    if action in ["delete_file", "modify_source", "execute_shell"]:
        return RiskLevel.CRITICAL

    # HIGH: Expensive operations
    if action == "generate_tests":
        if params.get("max_iterations", 0) > 20:
            return RiskLevel.HIGH

    # MEDIUM: External API calls
    if action in ["llm_call", "external_api"]:
        return RiskLevel.MEDIUM

    # LOW: Read-only operations
    return RiskLevel.LOW
```

Integration with GuardManager:

```python
class GuardManager:
    def check_tool_call(self, tool: str, params: Dict, context: Dict) -> 
GuardResult:
        # 1. Policy decision
        decision = self.policy_engine.evaluate(tool, params, context)

        # 2. If REVIEW decision → trigger HITL
        if decision.decision == "REVIEW":
            risk = assess_risk(tool, params)

            approved = self.hitl_manager.request_approval(
                action=tool,
                risk_level=risk,
                reason=decision.reason,
                details={
                    "tool": tool,
                    "params": params,
                    "context": context
                }
            )

            if not approved:
                return GuardResult(
                    allowed=False,
                    reason="Approval denied by human reviewer",
                    policy_decision=decision
                )

        # 3. Continue with normal checks
        # ...
```

Example Scenarios:

Scenario 1: High Token Usage

```
🔔 APPROVAL REQUIRED
Request ID: abc12345
Action: generate_tests
Risk Level: HIGH
Reason: Requested iterations (50) exceeds threshold (20)

Details:
  file_path: src/complex_module.py
  max_iterations: 50
  estimated_cost: $2.50
  estimated_tokens: 100,000

Approve? (y/n/details):
```

Scenario 2: Deleting Test Files

```
🔔 APPROVAL REQUIRED
Request ID: xyz78901
Action: delete_test
Risk Level: CRITICAL
Reason: Attempting to delete test file with manual modifications

Details:
  test_file: tests/test_critical.py
  last_modified: 2024-01-20 (manual edit)
  reason_for_deletion: Source file deleted

Approve? (y/n/details):
```

Benefits:

- ✅ **Risk mitigation**: Human oversight for dangerous operations
- ✅ **Compliance**: Audit trail of approvals
- ✅ **Learning**: System learns from human decisions
- ✅ **Flexibility**: Can be disabled for automation
- ✅ **Context-aware**: Shows relevant details for decision

Non-Interactive Mode (CI/CD):

```
# .env
HITL_INTERACTIVE=false

# Behavior:
# - LOW/MEDIUM: Auto-approve
# - HIGH/CRITICAL: Auto-deny (log for review)
```

Q21: How does the State Management work in LangGraph?

Answer: State Management in LangGraph uses **TypedDict** with **reducers** for message aggregation.

AgentState Definition:

```python
from typing import Annotated, TypedDict, List
from langgraph.graph.message import add_messages
from langchain_core.messages import BaseMessage

class AgentState(TypedDict):
    """
    State for test generation agent.
    Attributes:
        messages: Conversation history (with add_messages reducer)
        task: Current task description
        iteration: Current iteration number
        max_iterations: Maximum iterations allowed
        generated_tests: Final test code output
        completed: Whether task is complete
        session_id: Persistent session ID
        context_summary: Summarized context for long conversations
    """

    # Messages with add_messages reducer (appends new messages)
    messages: Annotated[List[BaseMessage], add_messages]

    # Other state fields (last write wins)
    task: str
    iteration: int
    max_iterations: int
    generated_tests: str
    completed: bool
    session_id: str
    context_summary: str
```

Reducer Function (`add_messages`):

```python
def add_messages(existing: List[BaseMessage], new: List[BaseMessage]) ->
List[BaseMessage]:
    """
    Reducer that appends new messages to existing messages.

    Handles deduplication and ordering.
    """
    # Create lookup by message ID
    existing_ids = {msg.id: msg for msg in existing if hasattr(msg, 'id')}

    result = existing.copy()

    for msg in new:
        if hasattr(msg, 'id') and msg.id in existing_ids:
            # Update existing message
            idx = next(i for i, m in enumerate(result) if getattr(m, 'id',
None) == msg.id)
            result[idx] = msg
        else:
            # Append new message
            result.append(msg)

    return result
```

State Updates:

```python
# Initialize state
initial_state = {
    "messages": [HumanMessage(content="Generate tests for factorial")],
    "task": "Generate tests for factorial",
    "iteration": 0,
    "max_iterations": 10,
    "generated_tests": "",
    "completed": False,
    "session_id": "session_123",
    "context_summary": ""
}

# Agent adds messages (tool calls, responses)
agent_output = {
    "messages": [
        AIMessage(content="Let me search for the factorial function"),
        ToolMessage(tool_call_id="call_1", content="Found factorial in
src/math.py")
    ]
}

# State after update (messages appended)
# state["messages"] = [
#   HumanMessage(...),           # Original
#   AIMessage("Let me search"),  # Added
#   ToolMessage(...)             # Added
# ]

# Other fields updated (last write wins)
# state["iteration"] = 1
# state["completed"] = False
```

Checkpointing (Persistence):

```python
from langgraph.checkpoint.sqlite import SqliteSaver

# Create checkpointer
checkpointer = SqliteSaver.from_conn_string("checkpoints.db")

# Create agent with checkpointing
agent = create_react_agent(
    model=llm,
    tools=tools,
    checkpointer=checkpointer
)

# Invoke with thread ID (enables resume)
response = agent.invoke(
    initial_state,
    config={
        "configurable": {"thread_id": "thread_123"},
        "recursion_limit": 50
    }
)

# State is automatically saved to SQLite after each step
# Can resume from any checkpoint:
resumed_state = agent.get_state(config={"configurable": {"thread_id":
"thread_123"}})
```

State Inspection:

```
# Get current state
current_state = agent.get_state(config={"configurable": {"thread_id":
"thread_123"}})

print(f"Iteration: {current_state['iteration']}")
print(f"Messages: {len(current_state['messages'])}")
print(f"Completed: {current_state['completed']}")

# Get state history
history = agent.get_state_history(config={"configurable": {"thread_id":
"thread_123"}})
for state in history:
    print(f"Step {state.step}: {state.values['iteration']} iterations")
```

Benefits:

- ✅ **Type Safety**: TypedDict provides IDE autocomplete
- ✅ **Reducers**: Flexible message aggregation
- ✅ **Persistence**: SQLite checkpointing for resume
- ✅ **Versioning**: State history for debugging
- ✅ **Isolation**: Thread IDs isolate concurrent sessions

Q22: What is the Critic Pattern for quality assurance?

Answer: The **Critic Pattern** adds a review step after generation to ensure quality.

Implementation (`src/critic.py`):

```
class ReviewResult(BaseModel):
    """Result of critic review."""

    overall_score: float = Field(..., description="Overall quality (0-1)")
    issues: List[str] = Field(default_factory=list)
    suggestions: List[str] = Field(default_factory=list)
    style_score: float = Field(..., description="Code style score")
    correctness_score: float = Field(..., description="Correctness score")
    completeness_score: float = Field(..., description="Completeness 
score")
    determinism_score: float = Field(..., description="Determinism score")
    passed: bool = Field(..., description="Meets quality threshold")

class TestCritic:
    """Critic agent that reviews generated tests."""

    def __init__(self, quality_threshold: float = 0.7):
        self.quality_threshold = quality_threshold
        self.llm_provider = get_llm_provider()

    def review_tests(
        self,
        test_code: str,
        source_code: str,
        context: Optional[Dict] = None
    ) -> ReviewResult:
        """
        Review generated tests for quality.

        Checks:
        1. Style (PEP8, formatting)
        2. Correctness (syntax, logic)
        3. Completeness (edge cases, coverage)
        4. Determinism (no flaky tests)

        Returns:
            ReviewResult with scores and recommendations
        """

        # 1. Check style
        style_score, style_issues = self._check_style(test_code)

        # 2. Check correctness
        correctness_score, correctness_issues = self._check_correctness(
            test_code, source_code
        )

        # 3. Check completeness
        completeness_score, completeness_issues = 
self._check_completeness(
            test_code, source_code
        )

        # 4. Check determinism
        determinism_score, determinism_issues = 
self._check_determinism(test_code)

        # Aggregate
        all_issues = (
            style_issues +
            correctness_issues +
            completeness_issues +
            determinism_issues
        )

        # Calculate overall score (weighted)
        overall_score = (
            0.20 * style_score +
            0.40 * correctness_score +
            0.30 * completeness_score +
            0.10 * determinism_score
        )

        # Generate suggestions
        suggestions = self._generate_suggestions(
            test_code, source_code, all_issues
        )

        return ReviewResult(
            overall_score=overall_score,
            issues=all_issues,
            suggestions=suggestions,
            style_score=style_score,
            correctness_score=correctness_score,
            completeness_score=completeness_score,
            determinism_score=determinism_score,
            passed=overall_score >= self.quality_threshold
        )

    def _check_style(self, test_code: str) -> Tuple[float, List[str]]:
        """Check PEP8 compliance."""
        issues = []
        score = 1.0

        # Check line length
        lines = test_code.split('\n')
        long_lines = [i for i, line in enumerate(lines, 1) if len(line) >
100]
        if long_lines:
            issues.append(f"Lines {long_lines} exceed 100 chars")
            score -= 0.1

        # Check indentation
        if '\t' in test_code:
            issues.append("Uses tabs instead of spaces")
            score -= 0.2

        # Check naming
        if not re.search(r'def test_\w+', test_code):
            issues.append("Test functions should start with 'test_'")
            score -= 0.3

        return max(0.0, score), issues

    def _check_correctness(
        self,
        test_code: str,
        source_code: str
    ) -> Tuple[float, List[str]]:
        """Check syntax and logical correctness."""
        issues = []
        score = 1.0

        # Syntax check
        try:
            ast.parse(test_code)
        except SyntaxError as e:
            issues.append(f"Syntax error: {e}")
            return 0.0, issues

        # Check imports
        if "import pytest" not in test_code and "from pytest" not in
test_code:
            issues.append("Missing pytest import")
            score -= 0.2

        # Check assertions
        assert_count = test_code.count("assert ")
        if assert_count == 0:
            issues.append("No assertions found")
            score -= 0.5

        # Check for source code duplication
        if source_code[:100] in test_code:
            issues.append("Test file duplicates source code")
            score -= 0.3

        return max(0.0, score), issues

    def _check_completeness(
        self,
        test_code: str,
        source_code: str
    ) -> Tuple[float, List[str]]:
        """Check test coverage and completeness."""
        issues = []
        score = 1.0

        # Extract source functions
        source_functions = self._extract_functions(source_code)
        test_functions = self._extract_test_functions(test_code)

        # Check if all source functions have tests
        for func in source_functions:
            test_name = f"test_{func}"
            if not any(test_name in tf for tf in test_functions):
                issues.append(f"Missing test for function '{func}'")
                score -= 0.2

        # Check for edge cases
        edge_case_indicators = ["None", "[]", "0", "''", "empty"]
        if not any(indicator in test_code for indicator in
edge_case_indicators):
            issues.append("No edge case tests detected")
            score -= 0.2

        # Check for exception testing
        if "pytest.raises" not in test_code and "with raises" not in
test_code:
            issues.append("No exception tests detected")
            score -= 0.2

        return max(0.0, score), issues

    def _check_determinism(self, test_code: str) -> Tuple[float,
List[str]]:
        """Check for non-deterministic patterns."""
        issues = []
        score = 1.0

        non_deterministic = [
            ("time.sleep", "time.sleep() makes tests slow and flaky"),
            ("random.random", "random.random() without seed is non-
deterministic"),
            ("datetime.now", "datetime.now() is non-deterministic"),
            ("uuid.uuid4", "uuid.uuid4() is non-deterministic")
        ]

        mocking_indicators = ["@patch", "monkeypatch", "mock", "Mock()"]
        has_mocking = any(mock in test_code for mock in
mocking_indicators)

        for pattern, message in non_deterministic:
            if pattern in test_code and not has_mocking:
                issues.append(message)
                score -= 0.3

        return max(0.0, score), issues

    def _generate_suggestions(
        self,
        test_code: str,
        source_code: str,
        issues: List[str]
    ) -> List[str]:
        """Generate actionable suggestions."""
        suggestions = []

        if "Missing test for function" in str(issues):
            suggestions.append("Add test cases for all source functions")

        if "No edge case tests" in str(issues):
            suggestions.append("Add tests for None, empty lists, zero
values")
```

```
        if "No exception tests" in str(issues):
            suggestions.append("Add tests for error conditions using
pytest.raises")

        if "non-deterministic" in str(issues):
            suggestions.append("Use @patch or monkeypatch for time/random
operations")

        if "exceeds 100 chars" in str(issues):
            suggestions.append("Break long lines for better readability")

        return suggestions
```

Integration with Test Generation:

```
def generate_with_review(file_path: str, max_refinements: int = 3) -> str:
    """Generate tests with critic review loop."""

    source_code = Path(file_path).read_text()

    for refinement in range(max_refinements):
        # Generate tests
        tests = orchestrator.generate_tests(source_code, file_path)

        # Critic review
        review = critic.review_tests(tests, source_code)

        console.print(f"\n[cyan]Critic Review (Refinement {refinement +
1})[/cyan]")
        console.print(f"Overall Score: {review.overall_score:.2f}")
        console.print(f"Passed: {review.passed}")

        if review.passed:
            console.print("[green]✓ Tests passed review[/green]")
            return tests

        # Show issues
        console.print(f"[yellow]Issues found:[/yellow]")
        for issue in review.issues:
            console.print(f"  - {issue}")

        # Show suggestions
        console.print(f"[cyan]Suggestions:[/cyan]")
        for suggestion in review.suggestions:
            console.print(f"  - {suggestion}")

        # Refine prompt with feedback
        refinement_prompt = f"""
Previous attempt had quality score {review.overall_score:.2f}.

Issues:
```

```
{chr(10).join(f"- {issue}" for issue in review.issues)}

Suggestions:
{chr(10).join(f"- {suggestion}" for suggestion in review.suggestions)}

Please regenerate tests addressing all issues.
"""

            # Regenerate with feedback
            tests = orchestrator.generate_tests(
                source_code,
                file_path,
                context=refinement_prompt
            )

        # Max refinements reached
        console.print("[red]⚠  Max refinements reached[/red]")
        return tests
```

Benefits:

- ✅ **Quality gates**: Prevents bad code from being saved
- ✅ **Iterative refinement**: Improves quality through feedback
- ✅ **Automated QA**: No manual review needed
- ✅ **Learning**: LLM learns from critic feedback
- ✅ **Metrics**: Quantitative quality scores

Final Questions (Q23-Q50)

Continuing Agentic AI Patterns...

Q23: Explain Memory and Context Management in long conversations

Answer: For long test generation sessions, the system uses **context summarization** and **checkpointing**:

1. Context Window Management:

```
MAX_CONTEXT_TOKENS = 100000  # 100K tokens for Qwen3-Coder

def manage_context(messages: List[BaseMessage]) -> List[BaseMessage]:
    """Trim messages if context too large."""
    total_tokens = estimate_tokens(messages)

    if total_tokens > MAX_CONTEXT_TOKENS:
        # Summarize old messages
        summary = summarize_conversation(messages[:-10])
        recent = messages[-10:]  # Keep last 10

        return [HumanMessage(content=f"[Context Summary] {summary}")] +
recent

    return messages
```

2. Persistent State (Checkpointing):

```
# State persisted to SQLite
checkpointer = SqliteSaver.from_conn_string("checkpoints.db")

# Resume from checkpoint
agent.invoke(
    state,
    config={"configurable": {"thread_id": "session_123"}}
)
```

Q24: What is the Multi-Agent Collaboration pattern?

Answer: Multiple specialized agents work together, each with expertise:

```
# Planner Agent: Plans tasks
plan = planner_agent.create_plan(goal)

# Coder Agent: Generates tests
tests = coder_agent.generate(plan.tasks[0])

# Critic Agent: Reviews quality
review = critic_agent.review(tests)

# Refiner Agent: Improves based on feedback
if not review.passed:
    tests = refiner_agent.refine(tests, review.suggestions)
```

Benefits: Division of labor, specialized expertise, checks & balances

---

Q25: How does the system handle Structured Output from LLMs?

Answer: Uses **Pydantic schemas** for validated, type-safe responses:

```
class TestGenerationResponse(BaseModel):
    reasoning: str
    tool_calls_summary: List[str]
    test_code: TestCode
    coverage: Dict[str, int]

# LLM instructed to respond in JSON
```

```
system_prompt = """
Respond with JSON matching this schema:
{
  "reasoning": "Your thought process",
  "tool_calls_summary": ["Searched for X", "Retrieved Y"],
  "test_code": {
    "code": "import pytest\\n...",
    "imports": ["pytest", "mock"],
    "test_functions": ["test_add", "test_subtract"],
    "test_classes": ["TestCalculator"],
    "dependencies": ["pytest", "mock"]
  },
  "coverage": {
    "positive_cases": 3,
    "negative_cases": 2,
    "edge_cases": 2
  }
}
"""

# Parse and validate
response_json = json.loads(llm_response)
validated = TestGenerationResponse(**response_json)
```

Benefits: Type safety, validation, easy parsing, structured data

---

Q26: What is the Goal-Oriented Agent pattern?

Answer: Agent explicitly tracks goals and measures progress toward them:

```python
class GoalTracker:
    def __init__(self):
        self.goals = {
            "coverage": {"target": 0.90, "current": 0.0},
            "pass_rate": {"target": 0.90, "current": 0.0}
        }

    def update(self, metric: str, value: float):
        self.goals[metric]["current"] = value

    def is_goal_met(self) -> bool:
        return all(
            goal["current"] >= goal["target"]
            for goal in self.goals.values()
        )

    def get_gap(self) -> Dict:
        return {
            metric: goal["target"] - goal["current"]
            for metric, goal in self.goals.items()
        }
```

```
# Agent loop
while not goal_tracker.is_goal_met() and iterations < max_iterations:
    tests = generate_tests()
    coverage = measure_coverage(tests)
    pass_rate = run_tests(tests)

    goal_tracker.update("coverage", coverage)
    goal_tracker.update("pass_rate", pass_rate)

    if not goal_tracker.is_goal_met():
        gap = goal_tracker.get_gap()
        refine_prompt = f"Current coverage: {coverage:.1%}, need
{gap['coverage']:.1%} more"
        tests = refine_tests(tests, refine_prompt)
```

Q27: How does Multi-Modal reasoning work (if applicable)?

Answer: While the current system focuses on code (text), the architecture supports multi-modal:

```
# Future: Handle diagrams, screenshots, error logs
class MultiModalInput(BaseModel):
    text: Optional[str]
    image: Optional[bytes]  # Screenshot of UI
    diagram: Optional[str]  # Mermaid/PlantUML
    error_log: Optional[str]

# Agent can reason across modalities
# e.g., Generate UI tests from screenshot + code
```

Q28: What is the Self-Reflection pattern?

Answer: Agent evaluates its own outputs before finalizing:

```
def generate_with_reflection(source_code: str) -> str:
    # 1. Generate initial tests
    tests_v1 = llm.generate(source_code)

    # 2. Self-reflect
    reflection_prompt = f"""
You generated these tests:
{tests_v1}

Reflect on the quality:
- Are all edge cases covered?
- Are there any redundant tests?
- Is the naming clear?
- Would these tests catch bugs?

Provide honest critique.
"""

    critique = llm.generate(reflection_prompt)

    # 3. Refine based on self-critique
    refinement_prompt = f"""
Based on your critique:
{critique}

Improve the tests.
"""

    tests_v2 = llm.generate(refinement_prompt)

    return tests_v2
```

Q29: How does the Agent handle Ambiguity and Clarification?

Answer: Through **HITL** and **prompt engineering**:

```
# Detect ambiguity
if is_ambiguous(task):
    clarification = hitl_manager.request_clarification(
        question="Should I test private methods or only public API?",
        options=["Public only", "All methods", "User decides per file"]
    )

    # Update context with clarification
    context += f"\nClarification: {clarification}"
```

Q30: What is the Chain-of-Thought (CoT) prompting pattern?

Answer: Explicitly ask LLM to show reasoning steps:

```
cot_prompt = """
Generate tests for this function step by step:

1. **Analyze**: What does the function do?
2. **Identify inputs**: What are all possible input types?
3. **Identify outputs**: What should be returned?
4. **Edge cases**: What are the boundary conditions?
5. **Error cases**: What inputs cause errors?
6. **Generate**: Write comprehensive tests
```

```
Think through each step, then generate tests.
"""
```

Benefits: Better reasoning, fewer hallucinations, explainable results

---

🔧 SECTION 3: Technical Concepts (Questions 31-50)

Q31: How does the AST (Abstract Syntax Tree) parsing work?

Answer: Python's `ast` module extracts code structure:

```python
import ast

def extract_functions(code: str) -> List[FunctionInfo]:
    """Extract all function definitions."""
    tree = ast.parse(code)
    functions = []

    for node in ast.walk(tree):
        if isinstance(node, ast.FunctionDef):
            functions.append(FunctionInfo(
                name=node.name,
                args=[arg.arg for arg in node.args.args],
                decorators=[d.id for d in node.decorator_list if
isinstance(d, ast.Name)],
                docstring=ast.get_docstring(node),
                start_line=node.lineno,
                end_line=node.end_lineno,
                is_async=isinstance(node, ast.AsyncFunctionDef)
            ))

    return functions

def extract_imports(code: str) -> List[str]:
    """Extract all imports."""
    tree = ast.parse(code)
    imports = []

    for node in ast.walk(tree):
        if isinstance(node, ast.Import):
            imports.extend(alias.name for alias in node.names)
        elif isinstance(node, ast.ImportFrom):
            module = node.module or ""
            imports.extend(f"{module}.{alias.name}" for alias in
node.names)

    return imports

def get_function_calls(code: str) -> List[str]:
    """Extract all function calls."""
    tree = ast.parse(code)
    calls = []

    for node in ast.walk(tree):
        if isinstance(node, ast.Call):
            if isinstance(node.func, ast.Name):
                calls.append(node.func.id)
            elif isinstance(node.func, ast.Attribute):
                calls.append(node.func.attr)

    return calls
```

Use Cases:

- Extract functions for indexing
- Build call graphs
- Detect dependencies
- Calculate cyclomatic complexity
- Find test functions (start with `test_`)

Q32: Explain Vector Embeddings and Semantic Search

Answer: Embeddings convert text to high-dimensional vectors for similarity:

```
# Text → Vector
text = "def factorial(n): return 1 if n <= 1 else n * factorial(n-1)"
embedding = embedding_model.encode(text)  # → [0.12, -0.34, 0.56, ...,
0.21] (768 dims)

# Storage in ChromaDB
collection.add(
    documents=[text],
    embeddings=[embedding],
    ids=["chunk_1"]
)

# Search by similarity (cosine distance)
query = "recursive function that calculates factorials"
query_embedding = embedding_model.encode(query)

results = collection.query(
    query_embeddings=[query_embedding],
    n_results=5
)
# Returns top 5 most similar code chunks
```

Why it works: Similar concepts cluster in vector space, even with different wording.

Q33: What is Reciprocal Rank Fusion (RRF)?

Answer: RRF combines multiple ranked lists into one:

```python
def rrf_fusion(
    lists: List[List[Result]],
    k: int = 60
) -> List[Result]:
    """
    RRF formula: score = Σ 1/(k + rank_i)

    where:
    - k = constant (typically 60)
    - rank_i = rank in list i (1-indexed)
    """
    scores = defaultdict(float)

    for result_list in lists:
        for rank, result in enumerate(result_list, start=1):
            scores[result.id] += 1.0 / (k + rank)

    # Sort by score descending
    ranked = sorted(scores.items(), key=lambda x: x[1], reverse=True)
    return ranked

# Example:
# List A: [doc1, doc2, doc3]
# List B: [doc2, doc1, doc4]
#
# Scores:
# doc1: 1/(60+1) + 1/(60+2) = 0.0164 + 0.0161 = 0.0325
# doc2: 1/(60+1) + 1/(60+1) = 0.0164 + 0.0164 = 0.0328
# doc3: 1/(60+3) = 0.0159
# doc4: 1/(60+3) = 0.0159
#
# Final ranking: doc2, doc1, doc3, doc4
```

Benefits: Simple, robust, no tuning needed, works across different scoring systems

---

Q34: How does Cross-Encoder Reranking work?

Answer: Cross-encoder computes similarity for (query, document) pairs:

```python
from sentence_transformers import CrossEncoder

# Initialize model
reranker = CrossEncoder('cross-encoder/ms-marco-MiniLM-L-12-v2')

# Candidate documents from initial search
```

```
candidates = [
    "def factorial(n): ...",
    "def fibonacci(n): ...",
    "def prime_check(n): ..."
]

# Query
query = "function to calculate factorials"

# Create pairs
pairs = [(query, doc) for doc in candidates]

# Compute similarity scores
scores = reranker.predict(pairs)
# scores = [0.92, 0.15, 0.08]  # factorial is most similar

# Rerank by score
reranked = sorted(
    zip(candidates, scores),
    key=lambda x: x[1],
    reverse=True
)
```

Difference from Bi-Encoder:

- **Bi-Encoder** (embeddings): Encodes query and docs separately, computes dot product
- **Cross-Encoder** (reranker): Encodes query+doc together, more accurate but slower

---

Q35: What is Docker Sandboxing and why is it important?

Answer: Sandboxing isolates test execution for security:

```
class SandboxExecutor:
    def run_tests(self, test_code: str, source_code: str) -> TestResult:
        """Run tests in isolated Docker container."""

        # 1. Create temporary directory
        with tempfile.TemporaryDirectory() as tmpdir:
            # 2. Write files
            Path(tmpdir, "source.py").write_text(source_code)
            Path(tmpdir, "test_source.py").write_text(test_code)
            Path(tmpdir, "requirements.txt").write_text("pytest\npytest-
cov")

            # 3. Create Dockerfile
            dockerfile = """
FROM python:3.11-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install -r requirements.txt
```

```
COPY . .
CMD ["pytest", "test_source.py", "--cov=source", "-v"]
"""
            Path(tmpdir, "Dockerfile").write_text(dockerfile)

            # 4. Build image
            client = docker.from_env()
            image, logs = client.images.build(path=tmpdir, tag="test-
sandbox")

            # 5. Run container with limits
            container = client.containers.run(
                image="test-sandbox",
                mem_limit="512m",     # Memory limit
                cpu_quota=50000,      # CPU limit (50%)
                network_mode="none", # No network access
                timeout=30,           # Max 30 seconds
                remove=True,
                stdout=True,
                stderr=True
            )

            # 6. Parse results
            output = container.decode()
            return self._parse_pytest_output(output)
```

Why Important:

- ✅ **Security**: Prevents malicious code from harming host
- ✅ **Isolation**: Clean environment for each test run
- ✅ **Resource limits**: Prevents DoS (infinite loops, memory leaks)
- ✅ **Reproducibility**: Same environment every time

Q36: Explain the Pydantic validation strategy

Answer: Pydantic provides runtime type validation:

```
from pydantic import BaseModel, Field, validator

class FunctionInfo(BaseModel):
    """Function metadata with validation."""

    name: str = Field(..., min_length=1, max_length=100)
    file_path: str
    start_line: int = Field(..., gt=0)
    end_line: int = Field(..., gt=0)
    function_hash: str = Field(..., regex=r'^[a-f0-9]{64}$')  # SHA256

    @validator('end_line')
    def end_after_start(cls, v, values):
        """Ensure end_line > start_line."""
        if 'start_line' in values and v <= values['start_line']:
            raise ValueError('end_line must be greater than start_line')
        return v

    @validator('file_path')
    def valid_file_path(cls, v):
        """Ensure file exists."""
        if not Path(v).exists():
            raise ValueError(f"File does not exist: {v}")
        return v

# Usage
try:
    func = FunctionInfo(
        name="factorial",
        file_path="/path/to/file.py",
        start_line=10,
        end_line=15,
        function_hash="abc...123"
    )
except ValidationError as e:
    print(e.json())
```

Benefits: Type safety, validation, auto-documentation, IDE support

---

Q37: How does the SQLite persistence layer work?

Answer: SQLite provides lightweight, serverless persistence:

```
import sqlite3
from contextlib import contextmanager

class TestTrackingDB:
    def __init__(self, db_path: Path):
        self.conn = sqlite3.connect(str(db_path))
        self.conn.row_factory = sqlite3.Row  # Dict-like access
        self._create_tables()

    @contextmanager
    def _get_cursor(self):
        """Context manager for transactions."""
        cursor = self.conn.cursor()
        try:
            yield cursor
            self.conn.commit()
        except Exception as e:
            self.conn.rollback()
            raise e
        finally:
            cursor.close()
    def add_function(self, func: FunctionInfo) -> int:
        """Add function to database."""
        with self._get_cursor() as cursor:
            cursor.execute("""
                INSERT INTO source_functions
                (file_path, function_name, function_hash, start_line, 
end_line, last_modified)
                VALUES (?, ?, ?, ?, ?, ?)
                ON CONFLICT(file_path, function_name) DO UPDATE SET
                    function_hash=excluded.function_hash,
                    last_modified=excluded.last_modified
            """, (
                func.file_path,
                func.function_name,
                func.function_hash,
                func.start_line,
                func.end_line,
                func.last_modified
            ))
            return cursor.lastrowid
```

Features Used:

- **UNIQUE constraints**: Prevent duplicates
- **FOREIGN KEY**: Maintain relationships
- **ON CONFLICT**: Upsert pattern
- **Transactions**: ACID guarantees
- **Indexes**: Fast lookups

Q38: What are the performance optimization strategies?

Answer:

1. Incremental Indexing:

- Only reindex changed files (hash-based)
- 10-100x faster on subsequent runs

2. Caching:

```
from functools import lru_cache

@lru_cache(maxsize=1000)
def get_function_context(function_name: str) -> Dict:
    """Cache function context lookups."""
    return expensive_context_retrieval(function_name)
```

3. Batch Operations:

```
# Bad: N queries
for chunk in chunks:
    db.insert(chunk)  # N round-trips

# Good: 1 batch query
db.insert_many(chunks)  # 1 round-trip
```

4. Async Operations:

```
import asyncio

async def index_files(files: List[Path]):
    """Index files concurrently."""
    tasks = [index_file(f) for f in files]
    await asyncio.gather(*tasks)
```

5. Connection Pooling:

```
# Reuse embedding model
embedding_model = load_model_once()  # Expensive
embeddings = [embedding_model.encode(chunk) for chunk in chunks]
```

Q39: Explain the Git change detection algorithm

Answer:

```
def get_changed_files_since_last_commit(self) -> List[FileChange]:
    """
    Detects changes at 4 levels:
    1. Committed (HEAD)
    2. Staged (git add)
    3. Modified (unstaged)
    4. Untracked (new files)
    """
    changes = []

    try:
        # 1. Modified files (staged + unstaged)
        diff = self.repo.head.commit.diff(None)  # HEAD vs working tree
        for item in diff:
            changes.append(FileChange(
                file_path=item.a_path,
                change_type=item.change_type,  # M=modified, A=added,
D=deleted
                status="modified"
            ))

        # 2. Staged files
        staged_diff = self.repo.index.diff("HEAD")
        for item in staged_diff:
            changes.append(FileChange(
                file_path=item.a_path,
                change_type=item.change_type,
                status="staged"
            ))

        # 3. Untracked files
        for file_path in self.repo.untracked_files:
            changes.append(FileChange(
                file_path=file_path,
                change_type="A",
                status="untracked"
            ))

    except ValueError:
        # No commits yet - treat all files as new
        all_files = list(Path(self.repo_path).rglob("*.py"))
        changes = [FileChange(file_path=str(f), change_type="A",
status="new") for f in all_files]

    return changes
```

Q40: How does the hashing strategy work for change detection?

Answer: SHA256 hashing detects file/function changes:

```
import hashlib

def hash_file(file_path: Path) -> str:
    """Hash entire file."""
    return hashlib.sha256(file_path.read_bytes()).hexdigest()

def hash_function(function_code: str) -> str:
    """Hash function body (ignores whitespace/comments)."""
    # Normalize code
    normalized = normalize_code(function_code)
    return hashlib.sha256(normalized.encode()).hexdigest()

def normalize_code(code: str) -> str:
    """Remove whitespace and comments for stable hashing."""
    # Parse and unparse to normalize
    tree = ast.parse(code)
    # Remove docstrings
    for node in ast.walk(tree):
        if isinstance(node, ast.Expr) and isinstance(node.value, ast.Str):
            node.value.s = ""  # Clear docstring

    return ast.unparse(tree)

# Change detection
current_hash = hash_function(current_code)
stored_hash = db.get_function_hash(function_name)

if current_hash != stored_hash:
    # Function changed → regenerate tests
    generate_tests(function_name)
    db.update_hash(function_name, current_hash)
```

Benefits: Deterministic, fast (O(n) in file size), ignores irrelevant changes

---

Q41: What is the Token Budget Tracking system?

Answer: Prevents runaway costs and API abuse:

```
class BudgetTracker:
    def __init__(self, session_id: str):
        self.session_id = session_id
        self.limits = {
            BudgetType.TOKEN: {"limit": 1_000_000, "used": 0, "period":
"day"},
            BudgetType.COST: {"limit": 100.0, "used": 0.0, "period":
"month"},
            BudgetType.TIME: {"limit": 900, "used": 0, "period":
"session"}
        }

    def check_limit(self, budget_type: BudgetType, amount: float) -> bool:
        """Check if adding amount would exceed limit."""
        budget = self.limits[budget_type]
        return (budget["used"] + amount) <= budget["limit"]

    def consume(self, budget_type: BudgetType, amount: float):
        """Consume budget."""
        if not self.check_limit(budget_type, amount):
            raise BudgetExceededError(
                f"{budget_type} budget exceeded: "
                f"{self.limits[budget_type]['used'] + amount} > "
                f"{self.limits[budget_type]['limit']}"
            )

        self.limits[budget_type]["used"] += amount

    def estimate_cost(self, token_count: int, model: str) -> float:
        """Estimate API cost."""
        # OpenAI pricing (example)
        pricing = {
            "gpt-4": {"input": 0.03 / 1000, "output": 0.06 / 1000},
            "gpt-3.5-turbo": {"input": 0.001 / 1000, "output": 0.002 /
1000}
        }

        rate = pricing.get(model, {"input": 0, "output": 0})
        return token_count * rate["input"]

# Usage
budget_tracker.consume(BudgetType.TOKEN, 5000)  # 5K tokens
budget_tracker.consume(BudgetType.COST, 0.25)   # $0.25
budget_tracker.consume(BudgetType.TIME, 15)     # 15 seconds
```

Q42: How does the Prometheus metrics export work?

Answer: Exports metrics for Prometheus scraping:

```
from prometheus_client import Counter, Histogram, Gauge, start_http_server

# Define metrics
TESTS_GENERATED = Counter(
    'tests_generated_total',
    'Total number of tests generated',
    ['provider', 'language']
)

TEST_GENERATION_LATENCY = Histogram(
    'test_generation_seconds',
    'Time to generate tests',
    buckets=[1, 2, 5, 10, 30, 60, 120]
)

ACTIVE_SESSIONS = Gauge(
    'active_sessions',
    'Number of active generation sessions'
)

# Instrument code
@TEST_GENERATION_LATENCY.time()
def generate_tests(source_code: str) -> str:
    ACTIVE_SESSIONS.inc()
    try:
        tests = orchestrator.generate(source_code)
        TESTS_GENERATED.labels(provider='ollama', language='python').inc()
        return tests
    finally:
        ACTIVE_SESSIONS.dec()

# Start HTTP server for Prometheus
```

```
start_http_server(8000)
# Metrics available at http://localhost:8000/metrics
```

Prometheus Config:

```
scrape_configs:
  - job_name: 'agentic_test_gen'
    static_configs:
      - targets: ['localhost:8000']
    scrape_interval: 15s
```

Q43: What is the OpenTelemetry tracing strategy?

Answer: Distributed tracing for request flows:

```
from opentelemetry import trace
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.sdk.trace.export import ConsoleSpanExporter,
BatchSpanProcessor

# Setup
provider = TracerProvider()
processor = BatchSpanProcessor(ConsoleSpanExporter())
provider.add_span_processor(processor)
trace.set_tracer_provider(provider)

tracer = trace.get_tracer(__name__)

# Instrument
@tracer.start_as_current_span("generate_test")
def generate_test(file_path: str):
    span = trace.get_current_span()
    span.set_attribute("file_path", file_path)

    with tracer.start_as_current_span("context_assembly") as ctx_span:
        ctx_span.set_attribute("context_types", 9)
        context = assemble_context(file_path)
        ctx_span.set_attribute("quality_score", context.quality_score)

    with tracer.start_as_current_span("llm_call") as llm_span:
        llm_span.set_attribute("provider", "ollama")
        llm_span.set_attribute("model", "qwen3-coder:30b")
        tests = llm_generate(context)
        llm_span.set_attribute("output_length", len(tests))

    return tests
```

Trace Visualization (Jaeger):

```
generate_test [5.2s]
├── context_assembly [1.2s]
│   ├── symbol_search [0.3s]
│   ├── embedding_search [0.7s]
│   └── git_history [0.2s]
└── llm_call [3.8s]
    ├── input_guardrails [0.1s]
    ├── llm_inference [3.5s]
    └── output_guardrails [0.2s]
```

Q44: How does the Rich console output work?

Answer: Uses `rich` library for beautiful console output:

```python
from rich.console import Console
from rich.progress import Progress, SpinnerColumn, TextColumn
from rich.table import Table
from rich.tree import Tree

console = Console()

# Colored output
console.print("[green]✓[/green] Test generation complete")
console.print("[red]✗[/red] Error: File not found")

# Progress bars
with Progress(
    SpinnerColumn(),
    TextColumn("[progress.description]{task.description}"),
    console=console
) as progress:
    task = progress.add_task("[cyan]Indexing files...", total=100)
    for i in range(100):
        progress.update(task, advance=1)

# Tables
table = Table(title="Test Coverage")
table.add_column("File", style="cyan")
table.add_column("Functions", justify="right")
table.add_column("Tested", justify="right", style="green")
table.add_column("Coverage", justify="right")

table.add_row("math.py", "10", "9", "90%")
table.add_row("utils.py", "5", "3", "60%")
console.print(table)

# Trees
```

```
tree = Tree("📁 Test Plan")
tree.add("1. Check git changes")
task2 = tree.add("2. Search codebase")
task2.add("→ Found 5 functions")
tree.add("3. Generate tests")
console.print(tree)
```

Q45: What is the ChromaDB persistence strategy?

Answer: ChromaDB stores embeddings on disk:

```python
import chromadb
from chromadb.config import Settings

# Persistent client
client = chromadb.PersistentClient(
    path="./chroma_db",
    settings=Settings(
        anonymized_telemetry=False,
        allow_reset=True
    )
)

# Collections
collection = client.get_or_create_collection(
    name="code_embeddings",
    metadata={"description": "Code chunks for RAG"}
)

# Add with embeddings
collection.add(
    documents=["def factorial(n): ..."],
    embeddings=[[0.1, 0.2, ..., 0.8]],  # 768-dim vector
    metadatas=[{"file": "math.py", "line": 10}],
    ids=["chunk_1"]
)

# Query
results = collection.query(
    query_embeddings=[[0.1, 0.2, ..., 0.8]],
    n_results=5,
    where={"file": "math.py"}  # Optional filter
)
```

Storage:

- SQLite database: Metadata
- Numpy arrays: Embeddings
- HNSW index: Fast similarity search

Q46: How does the Symbol Index provide O(1) lookups?

Answer: Dictionary-based exact matching:

```
class SymbolIndex:
    def __init__(self):
        self.symbols = {
            # symbol_name → [FileLocation, FileLocation, ...]
            "factorial": [
                {"file": "src/math.py", "line": 10, "type": "function"},
                {"file": "src/utils.py", "line": 50, "type": "import"}
            ]
        }
        self.call_graph = {
            # caller → [callee, callee, ...]
            "fibonacci": ["factorial"],
            "main": ["fibonacci", "factorial"]
        }

    def search(self, symbol: str) -> List[FileLocation]:
        """O(1) lookup by symbol name."""
        return self.symbols.get(symbol, [])

    def get_callers(self, symbol: str) -> List[str]:
        """O(1) lookup of who calls this symbol."""
        return [
            caller for caller, callees in self.call_graph.items()
            if symbol in callees
        ]
```

Benefits: Instant lookups, no embedding computation, complements semantic search

---

Q47: What are the concurrency and threading strategies?

Answer: Uses thread-safe patterns and locks:

```
import threading
from threading import RLock

class ThreadSafeDB:
    def __init__(self):
        self.lock = RLock()  # Reentrant lock
        self.conn = sqlite3.connect("db.sqlite", check_same_thread=False)

    def add_function(self, func: FunctionInfo):
        with self.lock:  # Acquire lock
            cursor = self.conn.cursor()
            cursor.execute("INSERT INTO ...", (func,))
```

```python
            self.conn.commit()
        # Lock automatically released

# Async operations (asyncio)
import asyncio

async def index_files_concurrently(files: List[Path]):
    """Index files in parallel."""
    tasks = [index_file(file) for file in files]
    results = await asyncio.gather(*tasks, return_exceptions=True)
    return results
```

Q48: How does the error handling and recovery work?

Answer: Multi-level error handling with graceful degradation:

```python
def generate_test_with_fallback(file_path: str) -> str:
    """Generate tests with multiple fallback strategies."""

    try:
        # 1. Try primary method (with full context)
        context = context_assembler.assemble(file_path, max_related=10)
        return orchestrator.generate(context)

    except TokenLimitError as e:
        # 2. Fallback: Reduce context
        logger.warning(f"Token limit exceeded, reducing context: {e}")
        context = context_assembler.assemble(file_path, max_related=3)
        return orchestrator.generate(context)

    except LLMTimeoutError as e:
        # 3. Fallback: Try different provider
        logger.error(f"LLM timeout, switching provider: {e}")
        fallback_orchestrator = create_orchestrator(provider="openai")
        return fallback_orchestrator.generate(context)

    except Exception as e:
        # 4. Last resort: Template-based generation
        logger.critical(f"All methods failed, using template: {e}")
        return generate_from_template(file_path)

# Retry decorator
from tenacity import retry, stop_after_attempt, wait_exponential

@retry(
    stop=stop_after_attempt(3),
    wait=wait_exponential(multiplier=1, min=2, max=10)
)
def call_llm_with_retry(prompt: str) -> str:
    """Retry LLM call on transient failures."""
    return llm.generate(prompt)
```

Q49: What are the security best practices implemented?

Answer:

1. Input Validation:

- Pydantic schemas for all inputs
- File path sanitization (prevent directory traversal)
- SQL parameterization (prevent injection)

2. Secrets Management:

- Environment variables (no hardcoded keys)
- Secret detection in generated code
- Audit logging of secret access

3. Sandboxing:

- Docker isolation for test execution
- Resource limits (CPU, memory, time)
- No network access in containers

4. Least Privilege:

- Read-only access to source code
- Write access only to tests/ directory
- File boundary enforcement

5. Audit Trail:

- All operations logged to SQLite
- Timestamps, actors, decisions recorded
- Immutable audit log

6. Defense in Depth:

- 9 layers of guardrails
- Multiple validation stages
- Constitutional AI self-verification

Q50: How would you scale this system to handle 1000+ developers?

Answer:

Architecture Changes:

1. Distributed Queue (Celery + Redis):

```
# Producer
from celery import Celery

app = Celery('test_gen', broker='redis://localhost:6379')

@app.task
def generate_test_task(file_path: str) -> str:
    return TestGenerator().generate(file_path)

# Trigger
result = generate_test_task.delay(file_path)

# Consumer (worker)
celery -A tasks worker --concurrency=10
```

2. Distributed Vector Store (Weaviate/Milvus):

- Replace ChromaDB with distributed vector DB
- Horizontal scaling of embedding search
- Sharding by repository

3. Caching Layer (Redis):

```
import redis

cache = redis.Redis(host='localhost', port=6379)

def get_context_cached(function_name: str) -> Dict:
    cached = cache.get(f"context:{function_name}")
    if cached:
        return json.loads(cached)

    context = expensive_context_retrieval(function_name)
    cache.setex(f"context:{function_name}", 3600, json.dumps(context))
    return context
```

4. Load Balancing:

- Multiple LLM endpoints behind load balancer
- Round-robin or least-connections
- Health checks and failover

5. Database Sharding:

- Shard by repository or team
- PostgreSQL instead of SQLite
- Read replicas for analytics

6. Observability at Scale:

- Centralized logging (ELK stack)
- Distributed tracing (Jaeger)
- Metrics aggregation (Prometheus + Grafana)

7. API Rate Limiting:

```
from slowapi import Limiter

limiter = Limiter(key_func=get_user_id)

@limiter.limit("100/hour")
def generate_endpoint(file_path: str):
    return generate_test(file_path)
```

8. Cost Optimization:

- Model quantization (smaller LLMs)
- Batch processing (group requests)
- Smart caching (deduplicate similar requests)
- Provider selection (Ollama for simple, GPT-4 for complex)

Expected Performance:

- **Throughput**: 1000+ tests/hour
- **Latency**: < 10s per test (p95)
- **Availability**: 99.9% uptime
- **Cost**: < $0.10 per test