AI Prompt System Monitor - Technical Specification

Part 1: Executive Summary, Architecture & Design

Document Version: 1.0
Date: 2024
Author: Yi Zu
Project: AI Prompt System Monitor - Production-Ready Full-Stack Application

---

Table of Contents

Executive Summary

Business Impact & Value Proposition

The **AI Prompt System Monitor** is a production-ready full-stack application designed to address a critical gap in AI application development: **continuous evaluation and monitoring of prompt systems**. As organizations increasingly rely on LLM-powered applications, the stability and performance of prompt systems become business-critical.

Target Users: - **AI/ML Engineers:** Testing prompt variations, monitoring model performance, optimizing costs - **Product Teams:** Ensuring consistent AI behavior across releases, tracking quality metrics - **DevOps/Platform Teams:** Automated monitoring, alerting, and cost management - **Data Scientists:** A/B testing different models, analyzing performance trends

Key Business Value: 1. **Cost Control:** Real-time budget tracking prevents unexpected API costs (critical for production deployments) 2. **Quality Assurance:** Automated regression testing ensures prompt changes don't degrade performance 3. **Model Optimization:** Side-by-side comparison enables data-driven model selection 4. **Operational Excellence:** Scheduled monitoring provides early warning of performance degradation 5. **Development Velocity:** Fast feedback loops accelerate prompt engineering iterations

Market Context: - LLM applications are moving from prototypes to production systems - Prompt engineering is becoming a core engineering discipline -

Cost overruns from uncontrolled API usage are a common production issue - No existing open-source solution provides this level of integration and automation

Technical Achievement Summary

This application demonstrates production-grade engineering across the full stack:

- **Backend:** Async-first FastAPI architecture with PostgreSQL, implementing concurrent-safe budget controls, background task processing, and persistent job scheduling
- **Frontend:** Modern React/TypeScript application with real-time progress tracking, performance visualization, and responsive UI
- **AI Integration:** Robust OpenAI API integration with token counting, cost calculation, and error handling
- **Evaluation Engine:** Custom word-overlap similarity algorithm optimized for factual response evaluation
- **Operational Features:** Cron-based scheduling, budget enforcement, cost tracking, and performance alerting

Completion Status: - **Level 0:** Core testing infrastructure (prompt management, test execution, evaluation) - **Level 1:** Automated monitoring (scheduling, time-series visualization, alerting) - **Level 2:** Model experimentation (multi-model support, comparison analytics, cost optimization) - **Level 3:** Genetic optimization (not implemented - future enhancement)

Project Overview & Requirements

Functional Requirements

Level 0: Core Testing Infrastructure FR-0.1: Prompt System Management - **User Story:** As an AI engineer, I need to create and manage prompt templates with variable placeholders so I can test different prompt configurations. - **Acceptance Criteria:** - Create prompt systems with name, description, template, variables, model selection, and model parameters - Support `{variable}` syntax for template interpolation - Store model configuration (temperature, max_tokens, etc.) as JSONB - CRUD operations via REST API and UI - Validation: template must contain all declared variables - **Implementation:** `backend/app/api/prompt_systems.py`, `backend/app/models/prompt_system.py`

FR-0.2: Regression Test Management - **User Story:** As a QA engineer, I need to define test cases with expected outputs so I can validate prompt system behavior. - **Acceptance Criteria:** - Create regression test sets with input variables and expected output - Associate tests with prompt systems - Support multiple test cases per system - Store input data as

JSONB for flexibility - Delete and update test cases - **Implementation:** `backend/app/api/regression_sets.py`

FR-0.3: Test Execution Engine - **User Story:** As a developer, I need to execute regression tests and see results in real-time so I can quickly iterate on prompts. - **Acceptance Criteria:** - Execute all tests for a prompt system - Interpolate template variables with test inputs - Call OpenAI API with configured model and parameters - Display real-time progress (pending → running → completed) - Handle API errors gracefully - Support model override per execution - **Implementation:** `backend/app/services/test_executor.py`, `backend/app/api/test_runs.py`

FR-0.4: Evaluation & Scoring - **User Story:** As a data scientist, I need similarity scores between actual and expected outputs so I can measure prompt quality. - **Acceptance Criteria:** - Calculate similarity score (0.0-1.0) for each test result - Use word overlap algorithm (Jaccard + recall metrics) - Display scores in results table - Calculate aggregate statistics (average score, success rate) - Store scores in database for historical analysis - **Implementation:** `backend/app/services/llm_service.py::calculate_similarity()`

FR-0.5: Cost Tracking - **User Story:** As a finance manager, I need to see API costs for each test run so I can budget effectively. - **Acceptance Criteria:** - Calculate cost per test based on token usage - Track total cost per test run - Display costs in UI (per test and aggregate) - Store cost data in database - Support cost estimation before execution - **Implementation:** `backend/app/services/cost_calculator.py`

Level 1: Automated Monitoring FR-1.1: Scheduled Test Execution - **User Story:** As a DevOps engineer, I need to schedule automated test runs so I can monitor prompt systems continuously. - **Acceptance Criteria:** - Create schedules with cron expressions - Support standard cron syntax (minute, hour, day, month, weekday) - Enable/disable schedules without deletion - Execute tests automatically at scheduled times - Track last execution time - Load active schedules on application startup - **Implementation:** `backend/app/services/scheduler_service.py`, `backend/app/api/schedules.py`

FR-1.2: Performance Trend Visualization - **User Story:** As a product manager, I need to see performance trends over time so I can identify degradation early. - **Acceptance Criteria:** - Display time-series chart of average similarity scores - Show test run history with timestamps - Filter by prompt system - Support multiple time ranges (7 days, 30 days, all time) - Interactive chart with tooltips - **Implementation:** `frontend/src/components/charts/PerformanceTrendsChart.tsx`, `frontend/src/hooks/usePerformanceTrends.ts`

FR-1.3: Performance Alerting - **User Story:** As an on-call engineer, I need alerts when performance drops so I can investigate issues quickly. - **Ac-**

ceptance Criteria: - Detect performance degradation (>10% drop in average score) - Display alert banner in UI - Compare current run to historical baseline - Alert persists until next successful run - **Implementation:** `frontend/src/components/ui/AlertBanner.tsx`

Level 2: Model Experimentation FR-2.1: Multi-Model Support - **User Story:** As an AI engineer, I need to test the same prompts across different models so I can choose the best model for my use case. - **Acceptance Criteria:** - Support GPT-3.5-turbo, GPT-4o, GPT-4o-mini - Run same test suite on multiple models - Track model used in each test run - Display model name in results - **Implementation:** Model selection in test execution, stored in `test_runs.model_used`

FR-2.2: Model Comparison Analytics - **User Story:** As a decision-maker, I need side-by-side model comparisons so I can optimize for accuracy, cost, and speed. - **Acceptance Criteria:** - Compare average similarity scores across models - Compare average response times (latency) - Compare total costs per model - Calculate success rates - Display in tabular format - **Implementation:** Client-side analytics in `frontend/src/lib/models/analytics.ts`

FR-2.3: Performance Insights & Recommendations - **User Story:** As a product owner, I need recommendations on which model to use so I can balance quality and cost. - **Acceptance Criteria:** - Identify "Most Accurate" model (highest average score) - Identify "Fastest" model (lowest average latency) - Identify "Best Value" model (best score-to-cost ratio) - Display recommendations with reasoning - **Implementation:** `frontend/src/lib/models/analytics.ts`

FR-2.4: Budget Controls - **User Story:** As a finance manager, I need budget limits to prevent cost overruns. - **Acceptance Criteria:** - Set daily budget limit per prompt system - Enforce budget before test execution - Auto-pause systems when budget exceeded - Reset daily budget at midnight UTC - Track lifetime and daily costs - Concurrent-safe budget checks (prevent race conditions) - **Implementation:** `backend/app/services/budget_guard.py`

Non-Functional Requirements

NFR-1: Performance - Test execution: < 3 seconds per test (GPT-3.5-turbo) - API response time: < 100ms for database queries - UI rendering: < 50ms initial load - Real-time progress updates: 2-second polling interval - Support 100+ concurrent test runs

NFR-2: Scalability - Horizontal scaling: Stateless backend services - Database: Connection pooling, indexed queries - Async execution: Non-blocking I/O throughout - Background tasks: Isolated database sessions

NFR-3: Reliability - Error handling: Graceful degradation on API failures - Data persistence: All results stored in database - Job recovery: Scheduler

reloads active schedules on restart - Budget safety: Row-level locking prevents race conditions

NFR-4: Security - API key management: Environment variables, never logged - Input validation: Pydantic schemas, SQL injection prevention - CORS: Configurable allowed origins - Error messages: No sensitive data in responses

NFR-5: Usability - Responsive UI: Works on desktop and tablet - Real-time feedback: Progress bars, status updates - Error messages: Clear, actionable error text - Documentation: Inline help, API docs

Technical Requirements

TR-1: Technology Stack - Backend: Python 3.10+, FastAPI, SQLAlchemy (async), APScheduler - Frontend: React 18, TypeScript, Vite, Tailwind CSS, Recharts - Database: PostgreSQL 12+ (JSONB support required) - AI Provider: OpenAI API (GPT-3.5-turbo, GPT-4o, GPT-4o-mini)

TR-2: Integration Requirements - OpenAI API: Async HTTP client, token counting, error handling - Database: Async SQLAlchemy with connection pooling - Scheduler: APScheduler with UTC timezone, persistent jobs

TR-3: Deployment Requirements - Development: Local PostgreSQL (Supabase recommended) - Production: Cloud-ready (Docker, Kubernetes compatible) - Environment: Configurable via environment variables - Monitoring: Health check endpoints

---

System Architecture & Design

High-Level Architecture

```
                    CLIENT LAYER

React Frontend (TypeScript + Tailwind CSS)
- Prompt System Management UI
- Test Execution & Results
- Performance Charts (Recharts)
- Schedule Management


                    HTTP/REST (CORS)


                API LAYER (FastAPI)

REST Endpoints
```

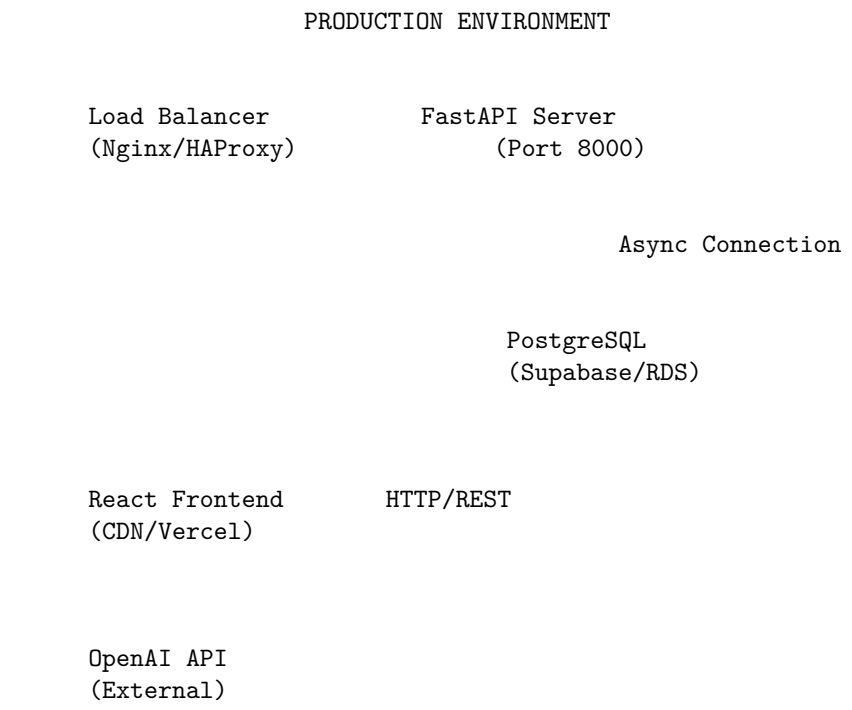
```
  - /api/prompt-systems (CRUD)
  - /api/regression-sets (CRUD)
  - /api/test-runs (Create, List, Get)
  - /api/schedules (CRUD, Trigger)


  Background Tasks (FastAPI BackgroundTasks)
  - Async test execution
  - Isolated database sessions




                  SERVICE LAYER

 LLMService       TestExecutor     Scheduler
 - Template        - Orchestrate    - APScheduler
   interp.           execution      - Cron jobs
 - OpenAI API      - Progress       - Job mgmt
 - Evaluation        tracking


CostCalculator    BudgetGuard
 - Token calc      - Budget
 - Pricing           enforcement




                  DATA LAYER (PostgreSQL)

prompt_systems    regression_      test_runs
                  sets


test_results      schedules




                  EXTERNAL SERVICES

  OpenAI API
  - Chat Completions
  - Token Usage Reporting
```

Data Flow: Test Execution

```
1. User clicks "Run Tests" in UI


2. Frontend: POST /api/test-runs
   {
     "system_id": "uuid",
     "model_override": "gpt-4o"
   }


3. API: TestExecutor.prepare_test_run()
   - Load prompt system from DB
   - Load regression tests
   - Estimate cost
   - BudgetGuard.check_and_reserve_budget() [WITH FOR UPDATE lock]
   - Create TestRun record (status: "pending")
   - Return test_run_id immediately


4. API: Background task starts
   - TestExecutor.execute_test_run_by_id()
   - Update status: "running"


5. For each RegressionSet:

      LLMService.interpolate_template()
       - Replace {variables} with test inputs

      LLMService.execute_prompt()
       - Call OpenAI API (async)
       - Measure execution time
       - Extract token usage

      CostCalculator.calculate_cost()
       - Compute USD cost from tokens

      LLMService.calculate_similarity()
       - Word overlap algorithm
       - Jaccard + recall metrics

      Create TestResult record
```

```
      - Store actual_output, score, cost, tokens
      - Update TestRun.completed_tests++


6. TestExecutor._finalize_test_run()
   - Calculate avg_score
   - Sum total_cost, total_tokens
   - BudgetGuard.record_actual_cost() [adjust for estimation]
   - Update status: "completed"


7. Frontend: Poll GET /api/test-runs/{id} every 2s
   - Display progress: completed_tests / total_tests
   - Show results table when status = "completed"
```

Deployment Topology




```
Background Tasks:
- Each FastAPI instance runs APScheduler
- Scheduler jobs execute in same process
```

- For high availability: Use external job queue (Redis + Celery)

Service Boundaries & Responsibilities

API Layer (`backend/app/api/`) - **Responsibility:** HTTP request/response handling, validation, routing - **Pattern:** RESTful endpoints, dependency injection for DB sessions - **Error Handling:** HTTPException with appropriate status codes - **Security:** CORS middleware, input validation via Pydantic

Service Layer (`backend/app/services/`) - **LLMService:** OpenAI integration, template interpolation, evaluation - **TestExecutor:** Test orchestration, progress tracking, result aggregation - **SchedulerService:** APScheduler management, cron job lifecycle - **CostCalculator:** Token-to-cost conversion, pricing management - **BudgetGuard:** Concurrent-safe budget enforcement

Data Layer (`backend/app/models/`, `backend/app/schemas/`) - **Models:** SQLAlchemy ORM definitions, database relationships - **Schemas:** Pydantic models for request/response validation - **Database:** Async SQLAlchemy with connection pooling

Frontend (`frontend/src/`) - **Components:** React components (atomic design pattern) - **Services:** API client with Axios, error handling - **State:** React Query for server state, local state for UI - **Visualization:** Recharts for performance trends

---

Component Design

Backend Architecture

API Layer (FastAPI) Route Design Patterns:

```python
# Standard REST pattern with async/await
@router.post("/prompt-systems", response_model=PromptSystemResponse, status_code=201)
async def create_prompt_system(
    system: PromptSystemCreate,
    db: AsyncSession = Depends(get_db)
):
    # Validation handled by Pydantic
    db_system = PromptSystem(**system.model_dump())
    db.add(db_system)
    await db.commit()
    await db.refresh(db_system)
    return db_system
```

Request/Response Schemas (Pydantic):

```python
# Request Schema
class PromptSystemCreate(BaseModel):
```

```
    name: str = Field(..., min_length=1, max_length=255)
    description: Optional[str] = None
    template: str = Field(..., min_length=1)
    variables: List[str] = Field(default_factory=list)
    model: str = Field(default="gpt-3.5-turbo")
    model_params: Dict[str, Any] = Field(default_factory=dict)

# Response Schema
class PromptSystemResponse(BaseModel):
    id: UUID
    name: str
    template: str
    variables: List[str]
    model: str
    model_params: Dict[str, Any]
    created_at: datetime

    class Config:
        from_attributes = True  # For SQLAlchemy models
```

Error Handling Strategy:

```
# Centralized error handling
try:
    executor = TestExecutor(db)
    test_run_id = await executor.prepare_test_run(...)
except ValueError as e:
    raise HTTPException(status_code=400, detail=str(e))
except BudgetExceededException as e:
    raise HTTPException(status_code=402, detail=str(e))  # Payment Required
except Exception as e:
    logger.error(f"Unexpected error: {e}", exc_info=True)
    raise HTTPException(status_code=500, detail="Internal server error")
```

CORS Configuration:

```
# backend/app/main.py
app.add_middleware(
    CORSMiddleware,
    allow_origins=settings.cors_origins,  # From environment
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)
```

Service Layer LLMService: OpenAI Integration

```python
class LLMService:
    def __init__(self):
        self.client = AsyncOpenAI(api_key=settings.OPENAI_API_KEY)
        self.cost_calculator = CostCalculator()

    async def execute_prompt(
        self,
        template: str,
        variables: Dict[str, Any],
        model: str = "gpt-4o",
        model_params: Dict[str, Any] = None
    ) -> ExecutionResult:
        # 1. Interpolate template
        prompt = self.interpolate_template(template, variables)

        # 2. Prepare parameters with defaults
        params = model_params or {}
        params.setdefault('temperature', 0.7)
        params.setdefault('max_tokens', 1000)

        # 3. Execute with timing
        start_time = time.time()
        response = await self.client.chat.completions.create(
            model=model,
            messages=[{"role": "user", "content": prompt}],
            **params
        )
        execution_time = int((time.time() - start_time) * 1000)

        # 4. Extract metrics
        usage = response.usage
        cost = self.cost_calculator.calculate_cost(
            model=model,
            input_tokens=usage.prompt_tokens,
            output_tokens=usage.completion_tokens
        )

        return ExecutionResult(
            output=response.choices[0].message.content,
            execution_time_ms=execution_time,
            cost_usd=cost,
            total_tokens=usage.total_tokens
        )
```

Token Counting & Rate Limiting: - OpenAI API automatically returns token usage in response - No client-side token counting needed (more accurate)

- Rate limiting handled by OpenAI (429 responses caught and logged) - Future: Implement exponential backoff retry logic

TestExecutor: Async Execution Patterns

```
class TestExecutor:
    async def execute_test_run_by_id(self, test_run_id: UUID, ...):
        # Load test run
        test_run = await self._load_test_run(test_run_id)
        test_run.status = "running"
        await self.db.commit()

        # Execute tests sequentially (can be parallelized)
        for regression_set in regression_sets:
            test_result = await self._execute_single_test(...)
            test_run.completed_tests += 1
            await self.db.commit()  # Progress visible immediately

        # Finalize
        await self._finalize_test_run(test_run, ...)
```

Background Task Management:

```
# API endpoint returns immediately
@router.post("/test-runs", status_code=202)
async def create_test_run(..., background_tasks: BackgroundTasks):
    test_run_id = await executor.prepare_test_run(...)

    # Execute in background with isolated session
    background_tasks.add_task(
        _execute_test_run_background,
        test_run_id,
        model_override
    )

    return {"test_run_id": str(test_run_id), "status": "pending"}

# Background task creates its own DB session
async def _execute_test_run_background(test_run_id: UUID, ...):
    async with AsyncSessionLocal() as db:
        executor = TestExecutor(db)
        await executor.execute_test_run_by_id(test_run_id, ...)
```

SchedulerService: APScheduler Configuration

```
class SchedulerService:
    def __init__(self):
        # Singleton pattern
        self.scheduler = AsyncIOScheduler(timezone="UTC")
```

```python
    async def start(self):
        self.scheduler.start()
        await self._load_schedules_from_db()  # Restore on startup

    async def _add_job(self, schedule_id: UUID, system_id: UUID, cron: str):
        trigger = CronTrigger.from_crontab(cron, timezone="UTC")
        self.scheduler.add_job(
            func=self._execute_scheduled_test,
            trigger=trigger,
            id=str(schedule_id),
            args=[system_id, schedule_id],
            replace_existing=True,
            misfire_grace_time=300  # 5 min grace period
        )
```

Job Persistence: - Current: In-memory (lost on restart, reloaded from DB) - Future: Use APScheduler's SQLAlchemy job store for true persistence

Cost Tracking Implementation:

```python
class CostCalculator:
    MODEL_PRICING = {
        "gpt-4o": {"input": 2.50, "output": 10.00},
        "gpt-4o-mini": {"input": 0.150, "output": 0.600},
        "gpt-3.5-turbo": {"input": 0.50, "output": 1.50}
    }

    @classmethod
    def calculate_cost(cls, model: str, input_tokens: int, output_tokens: int):
        pricing = cls.MODEL_PRICING.get(model, cls.MODEL_PRICING["gpt-4o"])
        input_cost = (input_tokens / 1_000_000) * pricing["input"]
        output_cost = (output_tokens / 1_000_000) * pricing["output"]
        return input_cost + output_cost

    @classmethod
    def estimate_cost(cls, model: str, num_tests: int):
        # Conservative estimate with 20% buffer
        avg_input = 500
        avg_output = 200
        cost_per_test = cls.calculate_cost(model, avg_input, avg_output)
        return cost_per_test * num_tests * 1.2  # 20% safety margin
```

Data Layer Database Schema with Relationships:

```sql
-- Core entities
CREATE TABLE prompt_systems (
```

```
    id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
    name VARCHAR(255) NOT NULL,
    template TEXT NOT NULL,
    variables JSONB NOT NULL DEFAULT '[]',
    model VARCHAR(100) NOT NULL DEFAULT 'gpt-3.5-turbo',
    model_params JSONB NOT NULL DEFAULT '{}',
    -- Budget controls
    max_daily_cost_usd FLOAT NOT NULL DEFAULT 1.0,
    daily_cost_usd FLOAT NOT NULL DEFAULT 0.0,
    total_cost_usd FLOAT NOT NULL DEFAULT 0.0,
    last_reset_date DATE,
    is_budget_paused BOOLEAN NOT NULL DEFAULT FALSE,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW()
);

CREATE TABLE regression_sets (
    id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
    system_id UUID NOT NULL REFERENCES prompt_systems(id) ON DELETE CASCADE,
    inputs JSONB NOT NULL,
    expected_output TEXT NOT NULL,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW()
);

CREATE TABLE test_runs (
    id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
    system_id UUID NOT NULL REFERENCES prompt_systems(id),
    status VARCHAR(50) NOT NULL DEFAULT 'pending',
    model_used VARCHAR(100) NOT NULL,
    triggered_by VARCHAR(50) DEFAULT 'manual',
    total_tests INTEGER DEFAULT 0,
    completed_tests INTEGER DEFAULT 0,
    avg_score DECIMAL(4,3),
    started_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
    completed_at TIMESTAMP WITH TIME ZONE,
    estimated_cost_usd FLOAT,
    actual_cost_usd FLOAT,
    total_tokens_used INTEGER
);

CREATE TABLE test_results (
    id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
    test_run_id UUID NOT NULL REFERENCES test_runs(id) ON DELETE CASCADE,
    regression_id UUID NOT NULL REFERENCES regression_sets(id),
    actual_output TEXT NOT NULL,
    expected_output TEXT NOT NULL,
    score DECIMAL(4,3) NOT NULL,
```

```
    execution_time_ms INTEGER NOT NULL,
    cost_usd FLOAT,
    tokens_used INTEGER,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW()
);

CREATE TABLE schedules (
    id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
    system_id UUID NOT NULL REFERENCES prompt_systems(id) ON DELETE CASCADE,
    name VARCHAR(255) NOT NULL,
    cron_expression VARCHAR(50) NOT NULL,
    active BOOLEAN DEFAULT true,
    last_run_at TIMESTAMP WITH TIME ZONE,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW()
);

-- Performance indexes
CREATE INDEX idx_test_runs_system_id ON test_runs(system_id);
CREATE INDEX idx_test_results_test_run_id ON test_results(test_run_id);
CREATE INDEX idx_prompt_systems_last_reset_date ON prompt_systems(last_reset_date);
```

Connection Pooling Configuration:

```
# backend/app/core/database.py
engine = create_async_engine(
    settings.DATABASE_URL,
    echo=True,  # Log SQL (disable in production)
    pool_pre_ping=True,  # Verify connections
    pool_size=10,  # Default connection pool
    max_overflow=20  # Additional connections under load
)
```

Query Optimization Patterns:

```
# Eager loading to avoid N+1 queries
result = await db.execute(
    select(TestRun)
    .where(TestRun.system_id == system_id)
    .order_by(TestRun.started_at.desc())
    .limit(limit)
    .options(selectinload(TestRun.test_results))  # Load related results
)
```

Frontend Architecture

Component Hierarchy

```
App.tsx
```

```
PromptSystemList (Page)
    PromptSystemCard (Feature)
        Badge (UI)
        Button (UI)
    PromptSystemForm (Feature - Modal)
        Input (UI)
        Button (UI)
PromptSystemDetail (Page)
    RegressionSetList (Feature)
        TestResultCard (Feature)
    TestRunsView (Feature)
        PerformanceTrendsChart (Feature)
            Recharts components
    BudgetControl (UI)
ScheduleList (Page)
    ScheduleCard (Feature)
    ScheduleForm (Feature - Modal)
```

Atomic Design Implementation: - **UI Components:** Reusable primitives (Button, Input, Card, Badge, Modal) - **Feature Components:** Domain-specific (PromptSystemForm, TestRunsView) - **Page Components:** Top-level views (PromptSystemList, PromptSystemDetail)

State Management Patterns

```javascript
// Server state: React Query
const { data: systems, isLoading } = useQuery({
  queryKey: ['prompt-systems'],
  queryFn: () => promptSystemsApi.getAll()
});

// Local state: useState for UI
const [isModalOpen, setIsModalOpen] = useState(false);

// Optimistic updates
const mutation = useMutation({
  mutationFn: promptSystemsApi.create,
  onMutate: async (newSystem) => {
    // Cancel outgoing refetches
    await queryClient.cancelQueries(['prompt-systems']);
    // Snapshot previous value
    const previous = queryClient.getQueryData(['prompt-systems']);
    // Optimistically update
    queryClient.setQueryData(['prompt-systems'], (old) => [...old, newSystem]);
    return { previous };
  },
```

```
  onError: (err, newSystem, context) => {
    // Rollback on error
    queryClient.setQueryData(['prompt-systems'], context.previous);
  }
});
```

API Integration Layer

```
// frontend/src/services/api.ts
export const apiClient = axios.create({
  baseURL: import.meta.env.VITE_API_URL || 'http://localhost:8000',
  headers: { 'Content-Type': 'application/json' }
});

// Request interceptor (for auth tokens)
apiClient.interceptors.request.use((config) => {
  // Add auth token if available
  return config;
});

// Response interceptor (error handling)
apiClient.interceptors.response.use(
  (response) => response,
  (error) => {
    console.error('API Error:', error.response?.data || error.message);
    return Promise.reject(error);
  }
);
```

UI/UX Design Design System: - **Colors:** Tailwind CSS default palette, custom accent colors - **Typography:** System fonts (San Francisco on macOS, Segoe UI on Windows) - **Spacing:** Consistent 4px grid (Tailwind spacing scale) - **Components:** Consistent border radius, shadows, hover states

Responsive Design: - Mobile-first approach - Breakpoints: sm (640px), md (768px), lg (1024px), xl (1280px) - Grid layout adapts to screen size - Tables scroll horizontally on mobile

Accessibility: - Semantic HTML (buttons, inputs, labels) - ARIA labels for icons - Keyboard navigation support - Focus indicators visible

Performance Optimization: - Code splitting: React.lazy() for route-based splitting - Image optimization: Vite handles asset optimization - Bundle size: Tree-shaking removes unused code - Memoization: React.memo() for expensive components

---

End of Part 1

Continue to Part 2 for: Detailed Implementation, Technical Deep Dives, Performance & Scalability

AI Prompt System Monitor - Technical Specification

Part 2: Detailed Implementation & Technical Deep Dives

Document Version: 1.0
Date: 2024
Author: Yi Zu
Project: AI Prompt System Monitor - Production-Ready Full-Stack Application

---

Table of Contents

Detailed Implementation

Core Business Logic

Prompt System Management Template Interpolation Implementation:

```python
# backend/app/services/llm_service.py
@staticmethod
def interpolate_template(template: str, variables: Dict[str, Any]) -> str:
    """
    Replace {variable} placeholders with actual values.

    Simple string replacement - no advanced templating engine needed.
    Handles edge cases: missing variables, empty values, type conversion.

    Args:
        template: String with {variable} placeholders
        variables: Dictionary of variable names to values

    Returns:
        Interpolated string with variables replaced

    Example:
        >>> template = "Hello {name}, you have {count} messages."
        >>> variables = {"name": "Alice", "count": 5}
```

```
        >>> interpolate_template(template, variables)
        "Hello Alice, you have 5 messages."
    """
    result = template
    for key, value in variables.items():
        placeholder = f"{{{key}}}"  # {variable_name}
        result = result.replace(placeholder, str(value))
    return result
```

Validation Strategy:

```
# backend/app/schemas/prompt_system.py
class PromptSystemCreate(BaseModel):
    name: str = Field(..., min_length=1, max_length=255)
    template: str = Field(..., min_length=1)
    variables: List[str] = Field(default_factory=list)

    @validator('template')
    def validate_template_has_variables(cls, v, values):
        """Ensure all declared variables exist in template."""
        variables = values.get('variables', [])
        for var in variables:
            if f"{{{var}}}" not in v:
                raise ValueError(f"Variable '{var}' not found in template")
        return v
```

Variable Extraction (Future Enhancement):

```
# Could auto-detect variables from template
import re

def extract_variables(template: str) -> List[str]:
    """Extract {variable} placeholders from template."""
    pattern = r'\{(\w+)\}'
    return list(set(re.findall(pattern, template)))
```

Test Execution Engine Async Execution Patterns:

```
# backend/app/services/test_executor.py
async def execute_test_run_by_id(self, test_run_id: UUID, ...):
    """
    Execute tests for an existing test run.

    This method is designed for background execution:
    - Updates status incrementally (visible to polling clients)
    - Handles errors gracefully (creates failed test results)
    - Tracks progress (completed_tests / total_tests)
    - Finalizes with aggregate statistics
```

```
    """
    test_run = await self._load_test_run(test_run_id)
    test_run.status = "running"
    await self.db.commit()  # Make status visible immediately

    # Execute tests sequentially
    # Note: Could parallelize with asyncio.gather() for speed
    total_score = 0.0
    total_cost = 0.0
    successful_tests = 0

    for regression_set in regression_sets:
        try:
            test_result = await self._execute_single_test(
                test_run=test_run,
                system=system,
                regression_set=regression_set,
                model_override=model_override
            )

            # Aggregate metrics
            total_score += float(test_result.score)
            total_cost += test_result.cost_usd
            successful_tests += 1
            test_run.completed_tests += 1

            # Commit after each test (progress visible)
            await self.db.commit()

        except Exception as e:
            logger.error(f"Test failed: {e}")
            await self._create_failed_test_result(...)
            test_run.completed_tests += 1
            await self.db.commit()

    # Finalize with aggregate statistics
    await self._finalize_test_run(test_run, total_score, successful_tests, ...)
```

Parallel Execution (Future Enhancement):

```
# Could execute tests in parallel for speed
async def execute_test_run_parallel(self, ...):
    tasks = [
        self._execute_single_test(...)
        for regression_set in regression_sets
    ]
    results = await asyncio.gather(*tasks, return_exceptions=True)
```

```
    # Handle results and exceptions
    for i, result in enumerate(results):
        if isinstance(result, Exception):
            await self._create_failed_test_result(...)
        else:
            # Process successful result
            ...
```

Background Task Management:

```
# backend/app/api/test_runs.py
async def _execute_test_run_background(test_run_id: UUID, model_override: str = None):
    """
    Background task with isolated database session.

    Critical: Creates its own session to avoid:
    - Session lifecycle issues (FastAPI dependency sessions close after request)
    - Transaction conflicts
    - Connection pool exhaustion
    """
    async with AsyncSessionLocal() as db:
        try:
            executor = TestExecutor(db)
            await executor.execute_test_run_by_id(test_run_id, model_override)
        except Exception as e:
            logger.error(f"Background execution failed: {e}")
            # Update test run status to failed
            result = await db.execute(
                select(TestRun).where(TestRun.id == test_run_id)
            )
            test_run = result.scalar_one_or_none()
            if test_run:
                test_run.status = "failed"
                await db.commit()
```

Progress Tracking Implementation:

```
# Frontend polling pattern
const pollTestRun = async (testRunId: string) => {
  const pollInterval = 2000; // 2 seconds
  const maxAttempts = 300; // 10 minutes max

  for (let i = 0; i < maxAttempts; i++) {
    const testRun = await testRunsApi.getById(testRunId);

    if (testRun.status === 'completed' || testRun.status === 'failed') {
      return testRun;
```

```
    }

    // Update progress bar
    setProgress(testRun.completed_tests / testRun.total_tests);

    await new Promise(resolve => setTimeout(resolve, pollInterval));
  }

  throw new Error('Test run timeout');
};
```

Error Recovery Mechanisms:

```python
# Retry logic for transient failures
async def execute_prompt_with_retry(
    self,
    template: str,
    variables: Dict[str, Any],
    model: str,
    max_retries: int = 3
) -> ExecutionResult:
    """Execute with exponential backoff retry."""
    for attempt in range(max_retries):
        try:
            return await self.execute_prompt(template, variables, model)
        except Exception as e:
            if attempt == max_retries - 1:
                raise  # Final attempt failed

            # Exponential backoff: 1s, 2s, 4s
            wait_time = 2 ** attempt
            logger.warning(f"Retry {attempt + 1}/{max_retries} after {wait_time}s")
            await asyncio.sleep(wait_time)
```

Evaluation Metrics Word Overlap Algorithm - Detailed Implementation:

```python
# backend/app/services/llm_service.py
@staticmethod
def calculate_similarity(actual: str, expected: str) -> float:
    """
    Calculate similarity score using word overlap approach.

    Algorithm:
    1. Normalize strings (lowercase, strip whitespace)
    2. Check exact match (score = 1.0)
    3. Check substring match (score = 0.95)
```

```
    4. Calculate Jaccard similarity (intersection / union)
    5. Calculate recall (intersection / expected_words)
    6. Weighted average: Jaccard (30%) + Recall (70%)

    Why this approach:
    - Fast: O(n) where n is word count
    - Cost-free: No API calls
    - Effective: Good for factual responses with key terms
    - Interpretable: Scores correlate with human judgment

    Trade-offs:
    - Less sophisticated than embeddings (but no cost/latency)
    - May miss semantic similarity (e.g., "happy" vs "joyful")
    - Works best when expected output contains key terms

    Returns:
        float: Similarity score 0.0-1.0
    """
    # Step 1: Normalize
    if not expected or not actual:
        return 0.0

    actual_lower = actual.lower().strip()
    expected_lower = expected.lower().strip()

    # Step 2: Exact match
    if actual_lower == expected_lower:
        return 1.0

    # Step 3: Substring match (expected contained in actual)
    if expected_lower in actual_lower:
        return 0.95  # High score but not perfect (distinguish from exact)

    # Step 4: Word-based comparison
    actual_words = set(actual_lower.split())
    expected_words = set(expected_lower.split())

    if not expected_words:
        return 0.0

    # Step 5: Calculate Jaccard similarity
    intersection = len(actual_words & expected_words)
    union = len(actual_words | expected_words)

    if union == 0:
        return 0.0
```

```
    jaccard_score = intersection / union

    # Step 6: Calculate recall (coverage of expected words)
    recall = intersection / len(expected_words)

    # Step 7: Weighted average
    # Recall weighted higher (70%) because we care more about
    # covering expected content than avoiding extra words
    score = (jaccard_score * 0.3) + (recall * 0.7)

    return min(score, 1.0)  # Cap at 1.0
```

Example Calculations:

```
Example 1: Exact Match
  actual: "We are open Monday-Friday 9am-5pm PT"
  expected: "We are open Monday-Friday 9am-5pm PT"
  → Score: 1.0 (exact match)

Example 2: Substring Match
  actual: "Our business hours are: We are open Monday-Friday 9am-5pm PT. Thank you!"
  expected: "We are open Monday-Friday 9am-5pm PT"
  → Score: 0.95 (substring match)

Example 3: Word Overlap
  actual: "We operate Monday through Friday, 9am to 5pm Pacific Time"
  expected: "We are open Monday-Friday 9am-5pm PT"

  actual_words: {we, operate, monday, through, friday, 9am, to, 5pm, pacific, time}
  expected_words: {we, are, open, monday-friday, 9am-5pm, pt}
  intersection: {we} (only "we" matches exactly)
  union: {we, operate, monday, through, friday, 9am, to, 5pm, pacific, time, are, open, mond

  jaccard = 1/14 = 0.071
  recall = 1/6 = 0.167
  score = (0.071 * 0.3) + (0.167 * 0.7) = 0.135

  → Score: 0.135 (low due to different phrasing)

Example 4: Good Overlap
  actual: "We are open Monday-Friday from 9am to 5pm Pacific Time"
  expected: "We are open Monday-Friday 9am-5pm PT"

  actual_words: {we, are, open, monday-friday, from, 9am, to, 5pm, pacific, time}
  expected_words: {we, are, open, monday-friday, 9am-5pm, pt}
  intersection: {we, are, open, monday-friday}
```

```
union: {we, are, open, monday-friday, from, 9am, to, 5pm, pacific, time, 9am-5pm, pt}

jaccard = 4/12 = 0.333
recall = 4/6 = 0.667
score = (0.333 * 0.3) + (0.667 * 0.7) = 0.567

→ Score: 0.567 (good overlap)
```

Alternative Evaluation Approaches:

1. BLEU Score (Machine Translation Metric):

```python
from nltk.translate.bleu_score import sentence_bleu

def calculate_bleu(actual: str, expected: str) -> float:
    """BLEU score for n-gram overlap."""
    actual_tokens = actual.lower().split()
    expected_tokens = expected.lower().split()
    reference = [expected_tokens]  # BLEU expects list of references
    return sentence_bleu(reference, actual_tokens)
```

Pros: Standard metric, handles n-grams
Cons: Designed for translation, less intuitive scores

2. ROUGE Score (Summarization Metric):

```python
from rouge_score import rouge_scorer

def calculate_rouge(actual: str, expected: str) -> float:
    """ROUGE-L (longest common subsequence)."""
    scorer = rouge_scorer.RougeScorer(['rougeL'], use_stemmer=True)
    scores = scorer.score(expected, actual)
    return scores['rougeL'].fmeasure
```

Pros: Good for summarization tasks
Cons: Requires additional dependency, slower

3. Embedding-Based Similarity:

```python
from sentence_transformers import SentenceTransformer

model = SentenceTransformer('all-MiniLM-L6-v2')

def calculate_embedding_similarity(actual: str, expected: str) -> float:
    """Cosine similarity of sentence embeddings."""
    embeddings = model.encode([actual, expected])
    similarity = cosine_similarity([embeddings[0]], [embeddings[1]])[0][0]
    return float(similarity)
```

Pros: Captures semantic similarity
Cons: Requires model download, slower, memory usage

4. LLM-as-Judge:

```python
async def llm_judge_similarity(actual: str, expected: str) -> float:
    """Use LLM to evaluate similarity."""
    prompt = f"""
    Rate the similarity between these two responses on a scale of 0.0 to 1.0.

    Expected: {expected}
    Actual: {actual}

    Respond with only a number between 0.0 and 1.0.
    """
    response = await llm_service.execute_prompt(prompt, {})
    score = float(response.output.strip())
    return score
```

Pros: Most sophisticated, understands context
Cons: Expensive (extra API call per test), slower, variable quality

5. Custom Evaluation Functions (Future):

```python
# Allow users to upload Python evaluation functions
def custom_evaluator(actual: str, expected: str, metadata: dict) -> float:
    """User-defined evaluation logic."""
    # Could check for specific keywords, regex patterns, etc.
    if "error" in actual.lower():
        return 0.0
    # ... custom logic
    return calculate_similarity(actual, expected)
```

Current Choice Rationale: - **Word overlap** chosen for MVP because: - Zero cost (no API calls) - Fast execution ($<$ 1ms) - Good enough for factual responses - Easy to understand and debug - Can be enhanced later with embeddings or LLM-as-judge

Advanced Features

Multi-Model Support Model Abstraction Layer:

```python
# Current: Direct OpenAI integration
# Future: Provider abstraction

class LLMProvider(ABC):
    @abstractmethod
    async def complete(self, prompt: str, model: str, **params) -> CompletionResult:
        pass

class OpenAIProvider(LLMProvider):
    async def complete(self, prompt: str, model: str, **params):
```

```
        # Current implementation
        ...

class AnthropicProvider(LLMProvider):
    async def complete(self, prompt: str, model: str, **params):
        # Future: Anthropic Claude support
        ...

class LLMService:
    def __init__(self, provider: LLMProvider):
        self.provider = provider

    async def execute_prompt(self, ...):
        return await self.provider.complete(...)
```

Provider-Specific Configurations:

```
# Model capabilities and limits
MODEL_CONFIGS = {
    "gpt-4o": {
        "max_tokens": 16384,
        "supports_streaming": True,
        "supports_function_calling": True,
        "cost_per_1k_input": 0.0025,
        "cost_per_1k_output": 0.01
    },
    "gpt-4o-mini": {
        "max_tokens": 16384,
        "supports_streaming": True,
        "supports_function_calling": True,
        "cost_per_1k_input": 0.00015,
        "cost_per_1k_output": 0.0006
    },
    "gpt-3.5-turbo": {
        "max_tokens": 16384,
        "supports_streaming": True,
        "supports_function_calling": True,
        "cost_per_1k_input": 0.0005,
        "cost_per_1k_output": 0.0015
    }
}
```

Fallback Strategies:

```
async def execute_prompt_with_fallback(
    self,
    template: str,
    variables: Dict[str, Any],
```

```
    primary_model: str,
    fallback_model: str = "gpt-3.5-turbo"
) -> ExecutionResult:
    """Execute with automatic fallback on failure."""
    try:
        return await self.execute_prompt(template, variables, primary_model)
    except Exception as e:
        logger.warning(f"Primary model {primary_model} failed: {e}, trying fallback")
        return await self.execute_prompt(template, variables, fallback_model)
```

Cost Optimization Across Models:

```
def recommend_model(
    self,
    required_quality: float = 0.8,
    max_cost_per_test: float = 0.01
) -> str:
    """Recommend model based on quality and cost constraints."""
    # Analyze historical data
    model_performance = self.get_model_performance_stats()

    # Filter by quality requirement
    candidates = [
        m for m in model_performance
        if m['avg_score'] >= required_quality
    ]

    # Filter by cost constraint
    candidates = [
        m for m in candidates
        if m['avg_cost'] <= max_cost_per_test
    ]

    # Return best value (score/cost ratio)
    if candidates:
        return max(candidates, key=lambda m: m['avg_score'] / m['avg_cost'])['model']
    else:
        return "gpt-3.5-turbo"  # Default fallback
```

Scheduling & Automation Cron Expression Handling:

```
# backend/app/services/scheduler_service.py
from apscheduler.triggers.cron import CronTrigger

def parse_cron_expression(cron: str) -> CronTrigger:
    """
    Parse cron expression with validation.
```

```
    Format: "minute hour day month weekday"
    Examples:
        "0 * * * *"     - Every hour at minute 0
        "0 9 * * 1-5"   - 9 AM on weekdays
        "*/15 * * * *"  - Every 15 minutes
        "0 0 1 * *"     - First day of month at midnight
    """
    try:
        trigger = CronTrigger.from_crontab(cron, timezone="UTC")
        return trigger
    except ValueError as e:
        raise ValueError(f"Invalid cron expression: {cron}. Error: {e}")
```

Cron Expression Helper (Frontend):

```
// frontend/src/components/schedules/CronExpressionHelper.tsx
export const CRON_PRESETS = {
  'Every Minute': '*/1 * * * *',
  'Every 5 Minutes': '*/5 * * * *',
  'Every 15 Minutes': '*/15 * * * *',
  'Every Hour': '0 * * * *',
  'Every Day at 9 AM': '0 9 * * *',
  'Every Weekday at 9 AM': '0 9 * * 1-5',
  'Every Monday at 9 AM': '0 9 * * 1',
};

export function validateCronExpression(cron: string): boolean {
  const parts = cron.split(' ');
  if (parts.length !== 5) return false;

  // Basic validation (could be more sophisticated)
  return parts.every(part => part.length > 0);
}
```

Job Persistence and Recovery:

```
# Current: In-memory scheduler (jobs lost on restart)
# Future: SQLAlchemy job store for persistence

from apscheduler.jobstores.sqlalchemy import SQLAlchemyJobStore

jobstores = {
    'default': SQLAlchemyJobStore(url=settings.DATABASE_URL)
}

scheduler = AsyncIOScheduler(
    jobstores=jobstores,
```

```
    timezone="UTC"
)
```

Concurrent Execution Safety:

```python
# Budget checks use row-level locking
async def check_and_reserve_budget(self, system_id: UUID, estimated_cost: float):
    # SELECT FOR UPDATE locks the row
    result = await self.db.execute(
        select(PromptSystem)
        .where(PromptSystem.id == system_id)
        .with_for_update()  #  Exclusive lock
    )
    system = result.scalar_one_or_none()

    # Check and update atomically
    if system.daily_cost_usd + estimated_cost > system.max_daily_cost_usd:
        raise BudgetExceededException(...)

    system.daily_cost_usd += estimated_cost
    await self.db.commit()  # Lock released
```

Resource Management:

```python
# Limit concurrent test executions
from asyncio import Semaphore

class TestExecutor:
    def __init__(self, db: AsyncSession, max_concurrent: int = 10):
        self.db = db
        self.semaphore = Semaphore(max_concurrent)  # Limit concurrent API calls

    async def _execute_single_test(self, ...):
        async with self.semaphore:  # Acquire semaphore
            return await self.llm_service.execute_prompt(...)
        # Semaphore released automatically
```

Analytics & Reporting Performance Trend Analysis:

```typescript
// frontend/src/hooks/usePerformanceTrends.ts
export function usePerformanceTrends(systemId: string, days: number = 30) {
  return useQuery({
    queryKey: ['performance-trends', systemId, days],
    queryFn: async () => {
      const testRuns = await testRunsApi.getAll(systemId, 100);

      // Filter by date range
      const cutoff = new Date();
```

```
      cutoff.setDate(cutoff.getDate() - days);

      const recentRuns = testRuns.filter(
        run => new Date(run.started_at) >= cutoff
      );

      // Transform for chart
      return recentRuns.map(run => ({
        date: new Date(run.started_at),
        score: run.avg_score,
        cost: run.actual_cost_usd,
        model: run.model_used
      }));
    }
  });
}
```

Model Comparison Algorithms:

```
// frontend/src/lib/models/analytics.ts
export function compareModels(testRuns: TestRun[]): ModelComparison {
  // Group by model
  const byModel = groupBy(testRuns, 'model_used');

  const comparisons = Object.entries(byModel).map(([model, runs]) => {
    const scores = runs.map(r => r.avg_score).filter(Boolean);
    const costs = runs.map(r => r.actual_cost_usd).filter(Boolean);
    const latencies = runs.flatMap(r =>
      r.test_results?.map(tr => tr.execution_time_ms) || []
    );

    return {
      model,
      avgScore: average(scores),
      avgCost: average(costs),
      avgLatency: average(latencies),
      totalRuns: runs.length,
      successRate: runs.filter(r => r.status === 'completed').length / runs.length
    };
  });

  return {
    comparisons,
    mostAccurate: maxBy(comparisons, 'avgScore'),
    fastest: minBy(comparisons, 'avgLatency'),
    bestValue: maxBy(comparisons, c => c.avgScore / c.avgCost)
  };
```

```
}
```

Cost Optimization Insights:

```python
# Analyze cost trends and suggest optimizations
def analyze_cost_trends(system_id: UUID, days: int = 30) -> CostAnalysis:
    """Analyze cost patterns and suggest optimizations."""
    test_runs = get_test_runs(system_id, days=days)

    # Group by model
    by_model = group_by(test_runs, 'model_used')

    insights = []
    for model, runs in by_model.items():
        total_cost = sum(r.actual_cost_usd for r in runs)
        avg_score = mean(r.avg_score for r in runs if r.avg_score)

        # Calculate cost per point of score
        cost_per_point = total_cost / (avg_score * len(runs)) if avg_score else float('inf')

        insights.append({
            'model': model,
            'total_cost': total_cost,
            'avg_score': avg_score,
            'cost_per_point': cost_per_point,
            'recommendation': 'optimal' if cost_per_point < threshold else 'consider_cheaper'
        })

    return CostAnalysis(insights=insights)
```

Predictive Alerting:

```python
def predict_performance_degradation(system_id: UUID) -> Optional[Alert]:
    """Predict performance issues before they occur."""
    recent_runs = get_recent_test_runs(system_id, days=7)

    if len(recent_runs) < 3:
        return None  # Not enough data

    # Calculate trend (linear regression)
    scores = [r.avg_score for r in recent_runs if r.avg_score]
    if len(scores) < 3:
        return None

    # Simple linear trend
    x = list(range(len(scores)))
    slope = calculate_slope(x, scores)
```

```python
    # Predict next score
    predicted_score = scores[-1] + slope

    # Alert if predicted to drop below threshold
    if predicted_score < 0.7:  # Threshold
        return Alert(
            type='predicted_degradation',
            message=f"Performance predicted to drop to {predicted_score:.2f}",
            severity='warning'
        )

    return None
```

Technical Deep Dives

Database Design

Schema Design Rationale:

1. JSONB for Flexible Data:

```sql
-- Why JSONB instead of separate columns?
-- Flexibility: Different prompt systems have different variables
-- Extensibility: Easy to add new fields without migrations
-- Performance: PostgreSQL JSONB is indexed and queryable

CREATE TABLE prompt_systems (
    variables JSONB NOT NULL DEFAULT '[]',  -- ["user_query", "context"]
    model_params JSONB NOT NULL DEFAULT '{}'  -- {"temperature": 0.7, "max_tokens": 150}
);

-- Query JSONB fields
SELECT * FROM prompt_systems
WHERE model_params->>'temperature' = '0.7';

-- Index JSONB for performance
CREATE INDEX idx_model_params ON prompt_systems USING GIN (model_params);
```

2. Timestamps with Timezone:

```sql
-- Always store timestamps with timezone (UTC)
created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW()

-- Benefits:
-- - Consistent across timezones
-- - Easy to convert to local time in application
-- - Avoids daylight saving time issues
```

3. Decimal for Scores:

```
-- Use DECIMAL(4,3) for scores (0.000 to 1.000)
score DECIMAL(4,3) NOT NULL

-- Why not FLOAT?
-- - Exact precision (no floating point errors)
-- - Consistent storage size
-- - Better for financial-like calculations
```

4. Cascade Deletes:

```
-- When prompt system deleted, delete related data
CREATE TABLE regression_sets (
    system_id UUID NOT NULL REFERENCES prompt_systems(id) ON DELETE CASCADE
);

-- Prevents orphaned records
-- Ensures data consistency
```

Query Patterns:

1. Common Queries with Indexes:

```
# Get recent test runs for a system (indexed on system_id)
result = await db.execute(
    select(TestRun)
    .where(TestRun.system_id == system_id)
    .order_by(TestRun.started_at.desc())
    .limit(50)
)
# Uses: idx_test_runs_system_id
```

2. N+1 Problem Solutions:

```
# Bad: N+1 queries
test_runs = await db.execute(select(TestRun))
for run in test_runs:
    results = await db.execute(
        select(TestResult).where(TestResult.test_run_id == run.id)
    )  # One query per run!

# Good: Eager loading
from sqlalchemy.orm import selectinload

test_runs = await db.execute(
    select(TestRun)
    .options(selectinload(TestRun.test_results))  # Load in one query
)
```

3. Pagination Strategies:

```python
# Offset-based pagination (simple but slow for large offsets)
def get_test_runs_page(offset: int = 0, limit: int = 50):
    return select(TestRun).offset(offset).limit(limit)

# Cursor-based pagination (faster, better for large datasets)
def get_test_runs_after(cursor: datetime, limit: int = 50):
    return select(TestRun).where(
        TestRun.started_at > cursor
    ).order_by(TestRun.started_at).limit(limit)
```

4. Caching Implementation (Future):

```python
# Redis caching for frequently accessed data
from redis import asyncio as aioredis

redis = await aioredis.from_url("redis://localhost")

async def get_prompt_system_cached(system_id: UUID):
    # Check cache first
    cached = await redis.get(f"system:{system_id}")
    if cached:
        return json.loads(cached)

    # Load from database
    system = await db.execute(...)

    # Cache for 5 minutes
    await redis.setex(
        f"system:{system_id}",
        300,  # 5 minutes
        json.dumps(system)
    )

    return system
```

API Design

RESTful Endpoints:

```
# Prompt Systems
GET    /api/prompt-systems              # List all
POST   /api/prompt-systems              # Create
GET    /api/prompt-systems/{id}         # Get one
PUT    /api/prompt-systems/{id}         # Update
DELETE /api/prompt-systems/{id}         # Delete
```

```
# Regression Sets
GET    /api/prompt-systems/{id}/regression-sets     # List
POST   /api/prompt-systems/{id}/regression-sets     # Create
DELETE /api/prompt-systems/{id}/regression-sets/{reg_id}  # Delete

# Test Runs
POST   /api/test-runs                    # Create (returns 202 Accepted)
GET    /api/test-runs                   # List (with filters)
GET    /api/test-runs/{id}              # Get status
GET    /api/test-runs/{id}/results      # Get detailed results

# Schedules
GET    /api/schedules                   # List
POST   /api/schedules                   # Create
PUT    /api/schedules/{id}              # Update
DELETE /api/schedules/{id}              # Delete
POST   /api/schedules/{id}/trigger      # Manual trigger
```

Request/Response Examples:

```
// POST /api/prompt-systems
Request:
{
  "name": "Customer Support Assistant",
  "description": "Handles customer inquiries",
  "template": "You are a support agent. Question: {user_query}",
  "variables": ["user_query"],
  "model": "gpt-3.5-turbo",
  "model_params": {
    "temperature": 0.7,
    "max_tokens": 150
  }
}

Response (201 Created):
{
  "id": "550e8400-e29b-41d4-a716-446655440000",
  "name": "Customer Support Assistant",
  "template": "You are a support agent. Question: {user_query}",
  "variables": ["user_query"],
  "model": "gpt-3.5-turbo",
  "model_params": {
    "temperature": 0.7,
    "max_tokens": 150
  },
  "created_at": "2024-10-15T10:30:00Z"
}
```

```
// POST /api/test-runs
Request:
{
  "system_id": "550e8400-e29b-41d4-a716-446655440000",
  "model_override": "gpt-4o",
  "triggered_by": "manual"
}

Response (202 Accepted):
{
  "message": "Test run started",
  "test_run_id": "660e8400-e29b-41d4-a716-446655440001",
  "status": "pending"
}

// GET /api/test-runs/{id}
Response (200 OK):
{
  "id": "660e8400-e29b-41d4-a716-446655440001",
  "system_id": "550e8400-e29b-41d4-a716-446655440000",
  "status": "running",
  "model_used": "gpt-4o",
  "total_tests": 3,
  "completed_tests": 2,
  "avg_score": null,  // Not calculated yet
  "started_at": "2024-10-15T10:35:00Z",
  "completed_at": null
}
```

Error Codes and Messages:

```
# Standard HTTP status codes
400 Bad Request:     Validation error, invalid input
401 Unauthorized:    Missing or invalid authentication (future)
402 Payment Required: Budget exceeded
404 Not Found:       Resource not found
422 Unprocessable:   Business logic error
500 Internal Server: Unexpected server error

# Error response format
{
  "detail": "Human-readable error message",
  "code": "BUDGET_EXCEEDED",  # Machine-readable code
  "field": "system_id"  # Optional: field that caused error
}
```

Rate Limiting (Future):

```
from slowapi import Limiter
from slowapi.util import get_remote_address

limiter = Limiter(key_func=get_remote_address)

@app.post("/api/test-runs")
@limiter.limit("10/minute")  # 10 requests per minute per IP
async def create_test_run(...):
    ...
```

Security Implementation

API Key Management:

```
# Environment variables (never in code)
# backend/.env
OPENAI_API_KEY=sk-proj-xxxxxxxxxxxxxxxxxxxx

# Loaded via Pydantic settings
class Settings(BaseSettings):
    OPENAI_API_KEY: str
    class Config:
        env_file = ".env"

# Never log API keys
logger.info(f"Calling OpenAI API")  #  OK
logger.info(f"API key: {settings.OPENAI_API_KEY}")  #  NEVER
```

Input Validation and Sanitization:

```
# Pydantic automatically validates and sanitizes
class PromptSystemCreate(BaseModel):
    name: str = Field(..., min_length=1, max_length=255)
    template: str = Field(..., min_length=1)

    @validator('name')
    def validate_name(cls, v):
        # Remove leading/trailing whitespace
        v = v.strip()
        # Prevent SQL injection (Pydantic handles this)
        if not v:
            raise ValueError("Name cannot be empty")
        return v
```

Audit Logging:

```
# Log all test executions for audit trail
```

```python
async def execute_test_run(...):
    logger.info(
        f"Test run started: system_id={system_id}, "
        f"model={model}, triggered_by={triggered_by}, "
        f"user_ip={request.client.host}"  # If auth added
    )

    # Store audit log in database
    audit_log = AuditLog(
        action="test_run_created",
        system_id=system_id,
        user_id=user_id,  # Future: when auth added
        metadata={"model": model, "triggered_by": triggered_by}
    )
    db.add(audit_log)
```

Performance & Scalability

Performance Characteristics

Benchmark Results:

Test Execution Times: - Single test (GPT-3.5-turbo): 2-3 seconds - Single test (GPT-4o): 2-3 seconds
- Single test (GPT-4o-mini): 1-2 seconds - 10-test regression set (GPT-3.5-turbo): 15-20 seconds (sequential) - 10-test regression set (parallel): 3-5 seconds (theoretical)

API Response Times: - GET /api/prompt-systems: $<$ 50ms (10 systems) - GET /api/test-runs: $<$ 100ms (50 runs) - POST /api/test-runs: $<$ 200ms (returns immediately, execution in background) - Database queries: $<$ 50ms (indexed queries)

UI Rendering: - Initial page load: $<$ 500ms - Component render: $<$ 50ms - Chart rendering: $<$ 100ms (Recharts)

Memory Usage: - Backend: ~100MB base + ~10MB per concurrent test run - Frontend: ~5MB bundle size, ~50MB runtime memory

Optimization Strategies:

1. Database Indexing:

```sql
-- Critical indexes for performance
CREATE INDEX idx_test_runs_system_id ON test_runs(system_id);
CREATE INDEX idx_test_runs_started_at ON test_runs(started_at DESC);
CREATE INDEX idx_test_results_test_run_id ON test_results(test_run_id);
CREATE INDEX idx_prompt_systems_last_reset_date ON prompt_systems(last_reset_date);
```

```sql
-- Composite index for common query pattern
CREATE INDEX idx_test_runs_system_status ON test_runs(system_id, status);
```

2. Connection Pooling:

```python
# Optimize connection pool for expected load
engine = create_async_engine(
    settings.DATABASE_URL,
    pool_size=20,         # Base connections
    max_overflow=40,       # Additional under load
    pool_pre_ping=True,  # Verify connections
    pool_recycle=3600    # Recycle after 1 hour
)
```

3. Async/Await Patterns:

```python
# Sequential (current)
for regression_set in regression_sets:
    result = await execute_test(regression_set)  # Waits for each

# Parallel (future optimization)
results = await asyncio.gather(*[
    execute_test(rs) for rs in regression_sets
])  # Executes all concurrently
```

4. Caching (Future):

```python
# Cache frequently accessed data
@lru_cache(maxsize=100)
def get_model_pricing(model: str):
    return MODEL_PRICING[model]

# Redis for shared cache
async def get_prompt_system_cached(system_id: UUID):
    cache_key = f"system:{system_id}"
    cached = await redis.get(cache_key)
    if cached:
        return json.loads(cached)
    # ... load from DB and cache
```

Scalability Considerations

Horizontal Scaling Architecture:

```
                    Load Balancer
                   (Nginx/HAProxy)
```

```
    FastAPI     FastAPI     FastAPI
    Server      Server      Server
      #1          #2          #3




                PostgreSQL
                (Primary)
```

Stateless Backend: - No session state stored in memory - All state in database - Any server can handle any request - Easy horizontal scaling

Database Sharding (Future):

```python
# Shard by system_id (hash-based)
def get_shard(system_id: UUID) -> str:
    hash_value = hash(str(system_id))
    shard_num = hash_value % NUM_SHARDS
    return f"shard_{shard_num}"

# Route queries to appropriate shard
async def get_test_runs(system_id: UUID):
    shard = get_shard(system_id)
    db = get_shard_connection(shard)
    return await db.execute(...)
```

Microservices Migration Path:

```
Current: Monolithic FastAPI app
    ↓
Phase 1: Extract scheduler to separate service
    - Scheduler Service (APScheduler)
    - Main API Service (FastAPI)
    - Shared database
    ↓
Phase 2: Extract test execution to worker pool
    - API Service (receives requests)
    - Worker Service (executes tests, Celery/Redis)
    - Scheduler Service (triggers workers)
    ↓
Phase 3: Full microservices
```

```
    - Prompt System Service
    - Test Execution Service
    - Analytics Service
    - Notification Service
```

Load Balancing Strategies:

```
# Nginx configuration for load balancing
upstream fastapi_backend {
    least_conn;  # Use least connections algorithm
    server api1:8000;
    server api2:8000;
    server api3:8000;
}

server {
    location /api/ {
        proxy_pass http://fastapi_backend;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
    }
}
```

Database Replication:

```
-- Read replicas for scaling reads
-- Primary: Handles writes
-- Replicas: Handle reads (test run queries, analytics)

-- Application code routes reads to replicas
async def get_test_runs(system_id: UUID):
    db = get_read_replica()  # Round-robin or least-lag
    return await db.execute(...)

async def create_test_run(...):
    db = get_primary()  # Always use primary for writes
    ...
```

End of Part 2

Continue to Part 3 for: Testing & QA, Deployment & Operations, Challenges & Solutions, Future Roadmap, Appendices

AI Prompt System Monitor - Technical Specification

Part 3: Testing, Deployment, Challenges & Future Roadmap

Document Version: 1.0
Date: 2024
Author: Yi Zu
Project: AI Prompt System Monitor - Production-Ready Full-Stack Application

---

Table of Contents

Testing & Quality Assurance

Testing Strategy

Unit Testing Service Layer Testing Examples:

```python
# tests/test_llm_service.py
import pytest
from app.services.llm_service import LLMService

class TestLLMService:
    def test_interpolate_template(self):
        """Test template variable interpolation."""
        template = "Hello {name}, you have {count} messages."
        variables = {"name": "Alice", "count": 5}

        result = LLMService.interpolate_template(template, variables)

        assert result == "Hello Alice, you have 5 messages."

    def test_interpolate_template_missing_variable(self):
        """Test that missing variables are handled."""
        template = "Hello {name}"
        variables = {}  # Missing 'name'
```

```
        result = LLMService.interpolate_template(template, variables)

        assert result == "Hello {name}"  # Unchanged if variable missing

    def test_calculate_similarity_exact_match(self):
        """Test similarity calculation for exact match."""
        actual = "We are open Monday-Friday 9am-5pm PT"
        expected = "We are open Monday-Friday 9am-5pm PT"

        score = LLMService.calculate_similarity(actual, expected)

        assert score == 1.0

    def test_calculate_similarity_substring_match(self):
        """Test similarity for substring match."""
        actual = "Our hours: We are open Monday-Friday 9am-5pm PT. Thanks!"
        expected = "We are open Monday-Friday 9am-5pm PT"

        score = LLMService.calculate_similarity(actual, expected)

        assert score == 0.95

    def test_calculate_similarity_word_overlap(self):
        """Test similarity for word overlap."""
        actual = "We operate Monday through Friday, 9am to 5pm"
        expected = "We are open Monday-Friday 9am-5pm PT"

        score = LLMService.calculate_similarity(actual, expected)

        assert 0.0 < score < 1.0  # Some overlap but not perfect
        assert score > 0.1  # Should have meaningful overlap
```

Mock Implementations:

```
# tests/conftest.py
import pytest
from unittest.mock import AsyncMock, MagicMock
from app.services.llm_service import LLMService

@pytest.fixture
def mock_openai_client():
    """Mock OpenAI client for testing."""
    client = MagicMock()
    client.chat.completions.create = AsyncMock(return_value=MagicMock(
        choices=[MagicMock(message=MagicMock(content="Mocked response"))],
        usage=MagicMock(
```

```
            prompt_tokens=100,
            completion_tokens=50,
            total_tokens=150
        )
    ))
    return client

@pytest.fixture
def llm_service(mock_openai_client):
    """LLMService with mocked OpenAI client."""
    service = LLMService()
    service.client = mock_openai_client
    return service

# Usage in tests
async def test_execute_prompt(llm_service, mock_openai_client):
    """Test prompt execution with mocked OpenAI."""
    result = await llm_service.execute_prompt(
        template="Test {variable}",
        variables={"variable": "value"},
        model="gpt-4o"
    )

    assert result.output == "Mocked response"
    assert result.total_tokens == 150
    mock_openai_client.chat.completions.create.assert_called_once()
```

Test Coverage Metrics:

```
# Run tests with coverage
pytest --cov=app --cov-report=html --cov-report=term

# Target coverage:
# - Services: 80%+ (business logic critical)
# - API routes: 70%+ (happy paths + error cases)
# - Models: 60%+ (mostly ORM, less critical)
```

Integration Testing API Endpoint Testing:

```
# tests/test_api.py
import pytest
from fastapi.testclient import TestClient
from app.main import app

client = TestClient(app)

def test_create_prompt_system():
```

```
    """Test creating a prompt system via API."""
    response = client.post(
        "/api/prompt-systems",
        json={
            "name": "Test System",
            "template": "Hello {name}",
            "variables": ["name"],
            "model": "gpt-3.5-turbo"
        }
    )

    assert response.status_code == 201
    data = response.json()
    assert data["name"] == "Test System"
    assert "id" in data

def test_create_test_run():
    """Test creating a test run."""
    # First create a system and regression set
    system_response = client.post("/api/prompt-systems", json={...})
    system_id = system_response.json()["id"]

    # Create regression set
    client.post(f"/api/prompt-systems/{system_id}/regression-sets", json={...})

    # Create test run
    response = client.post(
        "/api/test-runs",
        json={"system_id": system_id}
    )

    assert response.status_code == 202  # Accepted
    assert "test_run_id" in response.json()
```

Database Integration Tests:

```
# tests/test_database.py
import pytest
from sqlalchemy.ext.asyncio import AsyncSession
from app.core.database import AsyncSessionLocal
from app.models.prompt_system import PromptSystem

@pytest.mark.asyncio
async def test_create_prompt_system_in_db():
    """Test creating prompt system in database."""
    async with AsyncSessionLocal() as db:
        system = PromptSystem(
```

```
            name="Test System",
            template="Hello {name}",
            variables=["name"],
            model="gpt-3.5-turbo"
        )
        db.add(system)
        await db.commit()
        await db.refresh(system)

        assert system.id is not None
        assert system.name == "Test System"
```

External Service Mocking:

```
# Mock OpenAI API responses
@pytest.fixture
def mock_openai_response():
    """Mock OpenAI API response."""
    return {
        "choices": [{
            "message": {
                "content": "Mocked AI response"
            }
        }],
        "usage": {
            "prompt_tokens": 100,
            "completion_tokens": 50,
            "total_tokens": 150
        }
    }

# Use httpx to mock HTTP requests
from httpx import AsyncClient
import respx

@respx.mock
async def test_openai_integration():
    """Test OpenAI integration with mocked HTTP."""
    respx.post("https://api.openai.com/v1/chat/completions").mock(
        return_value=httpx.Response(200, json=mock_openai_response())
    )

    service = LLMService()
    result = await service.execute_prompt("Test", {}, "gpt-4o")

    assert result.output == "Mocked AI response"
```

End-to-End Testing User Journey Testing:

```python
# tests/e2e/test_user_workflow.py
def test_complete_user_workflow():
    """Test complete workflow: create system → add tests → run → view results."""
    # 1. Create prompt system
    system_response = client.post("/api/prompt-systems", json={...})
    system_id = system_response.json()["id"]

    # 2. Add regression test
    client.post(f"/api/prompt-systems/{system_id}/regression-sets", json={...})

    # 3. Run tests
    run_response = client.post("/api/test-runs", json={"system_id": system_id})
    test_run_id = run_response.json()["test_run_id"]

    # 4. Poll for completion
    import time
    for _ in range(30):  # Max 30 seconds
        status_response = client.get(f"/api/test-runs/{test_run_id}")
        status = status_response.json()["status"]
        if status == "completed":
            break
        time.sleep(1)

    assert status == "completed"

    # 5. Get results
    results_response = client.get(f"/api/test-runs/{test_run_id}/results")
    results = results_response.json()

    assert len(results) > 0
    assert results[0]["score"] is not None
```

Performance Testing:

```python
# tests/performance/test_load.py
import asyncio
import time

async def test_concurrent_test_runs():
    """Test system under concurrent load."""
    # Create 10 test runs simultaneously
    tasks = [
        create_test_run(system_id)
        for _ in range(10)
    ]
```

```
    start_time = time.time()
    results = await asyncio.gather(*tasks)
    duration = time.time() - start_time

    # All should complete successfully
    assert all(r["status"] == "pending" for r in results)

    # Should complete quickly (just creating records)
    assert duration < 2.0  # 2 seconds for 10 concurrent requests
```

Chaos Engineering:

```
# tests/chaos/test_failure_modes.py
def test_openai_api_failure():
    """Test behavior when OpenAI API fails."""
    # Simulate API failure
    with patch('app.services.llm_service.AsyncOpenAI') as mock:
        mock.return_value.chat.completions.create.side_effect = Exception("API Error")

        # Test run should handle gracefully
        response = client.post("/api/test-runs", json={...})
        test_run_id = response.json()["test_run_id"]

        # Wait for failure
        time.sleep(5)

        status_response = client.get(f"/api/test-runs/{test_run_id}")
        assert status_response.json()["status"] == "failed"

def test_database_connection_loss():
    """Test behavior when database connection is lost."""
    # Simulate DB failure
    # System should return 503 Service Unavailable
    # Not crash or hang
```

Quality Metrics

Code Quality Standards:

```
# .flake8 or pyproject.toml
[tool.flake8]
max-line-length = 100
exclude = ["venv", "migrations"]
ignore = ["E203", "W503"]  # Allow some formatting

[tool.black]
```

```
line-length = 100
target-version = ['py310']

[tool.mypy]
python_version = "3.10"
warn_return_any = True
warn_unused_configs = True
```

Performance Benchmarks:

```python
# tests/benchmarks/test_performance.py
def benchmark_similarity_calculation(benchmark):
    """Benchmark similarity calculation speed."""
    actual = "We are open Monday-Friday 9am-5pm PT"
    expected = "We operate Monday through Friday, 9am to 5pm"

    result = benchmark(LLMService.calculate_similarity, actual, expected)

    # Should complete in < 1ms
    assert result < 0.001

def benchmark_database_query(benchmark):
    """Benchmark database query performance."""
    async def query():
        async with AsyncSessionLocal() as db:
            result = await db.execute(select(TestRun).limit(50))
            return result.scalars().all()

    result = benchmark(query)

    # Should complete in < 100ms
    assert result < 0.1
```

Security Assessments:

```python
# tests/security/test_security.py
def test_sql_injection_prevention():
    """Test that SQL injection is prevented."""
    # Attempt SQL injection in name field
    malicious_input = "'; DROP TABLE prompt_systems; --"

    response = client.post(
        "/api/prompt-systems",
        json={"name": malicious_input, "template": "Test", ...}
    )

    # Should either reject (validation) or escape (safe)
    # Database should still exist
```

```
    assert response.status_code in [400, 201]  # Rejected or safely handled

def test_xss_prevention():
    """Test that XSS is prevented in output."""
    # Store script tag in template
    response = client.post(
        "/api/prompt-systems",
        json={"template": "<script>alert('xss')</script>", ...}
    )

    # Frontend should escape when rendering
    # Or backend should sanitize
```

Maintainability Metrics:

- **Cyclomatic Complexity:** < 10 per function
- **Function Length:** < 50 lines
- **Class Length:** < 300 lines
- **Test Coverage:** $> 70\%$ overall
- **Documentation:** All public functions documented

Deployment & Operations

Infrastructure Requirements

Development Environment Local Development Setup:

```
# Prerequisites
- Python 3.10+
- Node.js 18+
- PostgreSQL 12+ (or Supabase account)
- OpenAI API key

# Backend setup
cd backend
python3 -m venv venv
source venv/bin/activate
pip install -r requirements.txt

# Frontend setup
cd frontend
npm install

# Database setup
# Option 1: Local PostgreSQL
createdb ai_prompt_monitor
```

```
psql ai_prompt_monitor < DATABASE_SCHEMA.sql

# Option 2: Supabase (recommended for demo)
# - Create project at supabase.com
# - Run DATABASE_SCHEMA.sql in SQL Editor
# - Copy connection string to backend/.env
```

Database Provisioning:

```
-- Development database setup
CREATE DATABASE ai_prompt_monitor;
\c ai_prompt_monitor

-- Run schema
\i DATABASE_SCHEMA.sql

-- Create test user (optional)
CREATE USER dev_user WITH PASSWORD 'dev_password';
GRANT ALL PRIVILEGES ON DATABASE ai_prompt_monitor TO dev_user;
```

API Key Management:

```
# backend/.env (never commit to git)
DATABASE_URL=postgresql+asyncpg://user:pass@localhost:5432/ai_prompt_monitor
OPENAI_API_KEY=sk-proj-xxxxxxxxxxxxxxxxxxxx
BACKEND_HOST=0.0.0.0
BACKEND_PORT=8000
FRONTEND_URL=http://localhost:5173
ALLOWED_ORIGINS=http://localhost:5173

# Add to .gitignore
echo ".env" >> .gitignore
```

Production Environment Cloud Provider Recommendations:

Option 1: Vercel (Frontend) + Railway (Backend) + Supabase (Database) - **Frontend:** Vercel (free tier, automatic deployments) - **Backend:** Railway (easy PostgreSQL, $5/month) - **Database:** Supabase (free tier, managed PostgreSQL) - **Pros:** Easy setup, good free tiers - **Cons:** Vendor lock-in, scaling costs

Option 2: AWS (Full Stack) - **Frontend:** S3 + CloudFront (static hosting) - **Backend:** ECS Fargate or Lambda (serverless) - **Database:** RDS PostgreSQL - **Pros:** Scalable, enterprise-grade - **Cons:** Complex setup, higher cost

Option 3: Docker + Kubernetes - **Frontend:** Containerized React app - **Backend:** Containerized FastAPI - **Database:** Managed PostgreSQL (Cloud SQL, RDS) - **Orchestration:** Kubernetes (GKE, EKS, AKS) - **Pros:** Portable, scalable - **Cons:** Requires Kubernetes expertise

Container Orchestration:

```
# Dockerfile (Backend)
FROM python:3.10-slim

WORKDIR /app

# Install dependencies
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# Copy application
COPY . .

# Run application
CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8000"]
```

```
# Dockerfile (Frontend)
FROM node:18-alpine AS builder

WORKDIR /app
COPY package*.json ./
RUN npm ci
COPY . .
RUN npm run build

FROM nginx:alpine
COPY --from=builder /app/dist /usr/share/nginx/html
COPY nginx.conf /etc/nginx/nginx.conf

EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

```
# docker-compose.yml (Local development)
version: '3.8'

services:
  backend:
    build: ./backend
    ports:
      - "8000:8000"
    environment:
      - DATABASE_URL=postgresql+asyncpg://postgres:password@db:5432/ai_prompt_monitor
      - OPENAI_API_KEY=${OPENAI_API_KEY}
    depends_on:
      - db
```

```yaml
  frontend:
    build: ./frontend
    ports:
      - "80:80"
    depends_on:
      - backend

  db:
    image: postgres:14
    environment:
      - POSTGRES_DB=ai_prompt_monitor
      - POSTGRES_PASSWORD=password
    volumes:
      - postgres_data:/var/lib/postgresql/data

volumes:
  postgres_data:
```

CI/CD Pipeline

Build Automation:

```yaml
# .github/workflows/ci.yml
name: CI/CD Pipeline

on:
  push:
    branches: [main, develop]
  pull_request:
    branches: [main]

jobs:
  test-backend:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - uses: actions/setup-python@v4
        with:
          python-version: '3.10'
      - name: Install dependencies
        run: |
          cd backend
          pip install -r requirements.txt
          pip install pytest pytest-cov
      - name: Run tests
        run: |
```

```
          cd backend
          pytest --cov=app --cov-report=xml
      - name: Upload coverage
        uses: codecov/codecov-action@v3

  test-frontend:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - uses: actions/setup-node@v3
        with:
          node-version: '18'
      - name: Install dependencies
        run: |
          cd frontend
          npm ci
      - name: Run tests
        run: |
          cd frontend
          npm test
      - name: Build
        run: |
          cd frontend
          npm run build

  deploy:
    needs: [test-backend, test-frontend]
    runs-on: ubuntu-latest
    if: github.ref == 'refs/heads/main'
    steps:
      - uses: actions/checkout@v3
      - name: Deploy to production
        run: |
          # Deploy backend to Railway
          # Deploy frontend to Vercel
          # Run database migrations
```

Testing Automation:

```
# Automated testing in CI
- name: Run integration tests
  run: |
    # Start test database
    docker-compose up -d db
    # Wait for DB
    sleep 5
    # Run tests
```

```
    pytest tests/integration/
```

Deployment Strategies:

Blue-Green Deployment: 1. Deploy new version to "green" environment 2. Run smoke tests 3. Switch traffic from "blue" to "green" 4. Monitor for errors 5. Rollback to "blue" if issues detected

Rollback Procedures:

```
# Rollback script
#!/bin/bash

# 1. Identify current version
CURRENT_VERSION=$(git describe --tags)

# 2. Revert to previous version
git checkout previous-tag

# 3. Rebuild and redeploy
docker-compose up -d --build

# 4. Verify health
curl http://localhost:8000/health

# 5. Notify team
echo "Rolled back to $CURRENT_VERSION"
```

Monitoring and Logging

Application Monitoring:

```
# backend/app/main.py
import logging
from prometheus_client import Counter, Histogram, generate_latest

# Metrics
test_runs_total = Counter('test_runs_total', 'Total test runs')
test_execution_time = Histogram('test_execution_seconds', 'Test execution time')
api_requests = Counter('api_requests_total', 'API requests', ['method', 'endpoint'])

@app.middleware("http")
async def metrics_middleware(request: Request, call_next):
    start_time = time.time()
    response = await call_next(request)
    duration = time.time() - start_time

    api_requests.labels(method=request.method, endpoint=request.url.path).inc()
```

```
    return response

@app.get("/metrics")
async def metrics():
    """Prometheus metrics endpoint."""
    return Response(generate_latest(), media_type="text/plain")
```

Logging Configuration:

```
# backend/app/core/logging.py
import logging
import sys
from logging.handlers import RotatingFileHandler

def setup_logging():
    """Configure application logging."""
    logging.basicConfig(
        level=logging.INFO,
        format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
        handlers=[
            RotatingFileHandler('app.log', maxBytes=10*1024*1024, backupCount=5),
            logging.StreamHandler(sys.stdout)
        ]
    )

    # Set log levels for libraries
    logging.getLogger('sqlalchemy.engine').setLevel(logging.WARNING)
    logging.getLogger('httpx').setLevel(logging.WARNING)
```

Health Checks:

```
# backend/app/main.py
@app.get("/health")
async def health():
    """Comprehensive health check."""
    checks = {
        "status": "healthy",
        "database": "unknown",
        "scheduler": "unknown",
        "openai": "unknown"
    }

    # Check database
    try:
        async with AsyncSessionLocal() as db:
            await db.execute(text("SELECT 1"))
        checks["database"] = "connected"
```

```
    except Exception as e:
        checks["database"] = f"error: {str(e)}"
        checks["status"] = "unhealthy"

    # Check scheduler
    if scheduler.scheduler.running:
        checks["scheduler"] = "running"
    else:
        checks["scheduler"] = "stopped"
        checks["status"] = "unhealthy"

    # Check OpenAI (lightweight check)
    try:
        # Could ping OpenAI API or check API key format
        if settings.OPENAI_API_KEY.startswith("sk-"):
            checks["openai"] = "configured"
    except:
        checks["openai"] = "not_configured"

    status_code = 200 if checks["status"] == "healthy" else 503
    return JSONResponse(content=checks, status_code=status_code)
```

Challenges & Solutions

Technical Challenges

Async Execution Complexity **Challenge: Race Conditions in Budget Checks**

Problem: - Multiple scheduled tests run simultaneously - Both check budget before either updates - Both pass budget check, exceeding limit

Solution: Row-Level Locking

```
# backend/app/services/budget_guard.py
async def check_and_reserve_budget(self, system_id: UUID, estimated_cost: float):
    # SELECT FOR UPDATE locks the row exclusively
    result = await self.db.execute(
        select(PromptSystem)
        .where(PromptSystem.id == system_id)
        .with_for_update()  #  Exclusive lock
    )
    system = result.scalar_one_or_none()

    # Check and update atomically
    if system.daily_cost_usd + estimated_cost > system.max_daily_cost_usd:
```

```
            raise BudgetExceededException(...)

        system.daily_cost_usd += estimated_cost
        await self.db.commit()  # Lock released

        # Other transactions wait here until lock released
```

Challenge: Resource Cleanup

Problem: - Background tasks may fail - Database sessions not properly closed - Memory leaks from unclosed connections

Solution: Context Managers

```
# Always use context managers for database sessions
async def _execute_test_run_background(test_run_id: UUID):
    async with AsyncSessionLocal() as db:  # Auto-closes on exit
        try:
            executor = TestExecutor(db)
            await executor.execute_test_run_by_id(test_run_id)
        except Exception as e:
            logger.error(f"Execution failed: {e}")
            # Session automatically rolled back and closed
```

Challenge: Error Propagation

Problem: - Errors in background tasks not visible to API caller - Test runs stuck in "running" status

Solution: Status Updates and Logging

```
async def execute_test_run_by_id(self, test_run_id: UUID):
    test_run = await self._load_test_run(test_run_id)
    test_run.status = "running"
    await self.db.commit()

    try:
        # Execute tests...
        test_run.status = "completed"
    except Exception as e:
        logger.error(f"Test run failed: {e}", exc_info=True)
        test_run.status = "failed"
        test_run.error_message = str(e)
    finally:
        await self.db.commit()
```

Cost Management Challenge: Budget Enforcement Accuracy

Problem: - Estimated costs may differ from actual costs - Budget could be exceeded if estimation is too low

Solution: Safety Buffer + Adjustment

```
# Estimate with 20% buffer
estimated_cost = CostCalculator.estimate_cost(model, num_tests) * 1.2

# Reserve estimated cost
await budget_guard.check_and_reserve_budget(system_id, estimated_cost)

# After execution, adjust for actual cost
actual_cost = sum(result.cost_usd for result in results)
cost_difference = actual_cost - estimated_cost

# Adjust daily cost
system.daily_cost_usd += cost_difference  # Could be negative (refund)
```

Challenge: Usage Monitoring

Problem: - Need real-time visibility into costs - Historical cost analysis

Solution: Comprehensive Cost Tracking

```
# Store costs at multiple levels
- test_results.cost_usd (per test)
- test_runs.actual_cost_usd (per run)
- prompt_systems.daily_cost_usd (daily aggregate)
- prompt_systems.total_cost_usd (lifetime)

# Query for analytics
SELECT
    DATE(started_at) as date,
    SUM(actual_cost_usd) as daily_cost
FROM test_runs
WHERE system_id = $1
GROUP BY DATE(started_at)
ORDER BY date DESC;
```

Real-time Updates Challenge: Polling vs WebSockets Trade-offs

Problem: - Polling: Simple but inefficient, slight delay - WebSockets: Real-time but complex, connection management

Solution: Polling with Smart Intervals

```
// Adaptive polling interval
const pollTestRun = async (testRunId: string) => {
  let interval = 2000; // Start with 2 seconds

  while (true) {
    const testRun = await testRunsApi.getById(testRunId);
```

```
    if (testRun.status === 'completed' || testRun.status === 'failed') {
      return testRun;
    }

    // Adaptive interval: faster when running, slower when pending
    if (testRun.status === 'running') {
      interval = 1000; // 1 second when active
    } else {
      interval = 2000; // 2 seconds when pending
    }

    await new Promise(resolve => setTimeout(resolve, interval));
  }
};
```

Future: WebSocket Implementation

```
# backend/app/main.py
from fastapi import WebSocket

@app.websocket("/ws/test-runs/{test_run_id}")
async def websocket_test_run(websocket: WebSocket, test_run_id: UUID):
    await websocket.accept()

    try:
        while True:
            test_run = await get_test_run(test_run_id)
            await websocket.send_json({
                "status": test_run.status,
                "completed_tests": test_run.completed_tests,
                "total_tests": test_run.total_tests
            })

            if test_run.status in ["completed", "failed"]:
                break

            await asyncio.sleep(1)
    except WebSocketDisconnect:
        pass
```

Challenge: Progress Tracking

Problem: - Need to show progress during long-running test execution - Multiple tests executing, need per-test progress

Solution: Incremental Status Updates

```
# Update progress after each test
```

```python
for regression_set in regression_sets:
    test_result = await self._execute_single_test(...)
    test_run.completed_tests += 1
    await self.db.commit()  # Make progress visible immediately

    # Frontend polls and sees: 1/10, 2/10, 3/10, ...
```

Business Challenges

Challenge: User Adoption

Problem: - Developers may not see immediate value - Learning curve for new tool

Solution: - Clear documentation and examples - Quick start guide (5-minute setup) - Demo video showing value - Integration with existing workflows

Challenge: Feature Prioritization

Problem: - Many potential features (Level 3, multi-provider, etc.) - Limited development resources

Solution: - User feedback collection - Analytics on feature usage - MVP approach: Build core features first - Iterate based on real usage

Challenge: Market Positioning

Problem: - Competing with commercial solutions - Need to differentiate

Solution: - Open-source approach (community contributions) - Focus on developer experience - Cost transparency (unlike some commercial tools) - Extensibility (custom evaluation functions)

Future Roadmap

Phase 3 Implementation (Genetic Optimization)

Evolutionary Algorithms for Prompt Improvement:

```python
# Future: Genetic prompt optimization
class GeneticPromptOptimizer:
    def __init__(self, initial_prompt: str, budget: float):
        self.initial_prompt = initial_prompt
        self.budget = budget
        self.population = self._initialize_population(initial_prompt)

    def optimize(self) -> str:
        """Evolve prompt to maximize similarity scores."""
        for generation in range(self.max_generations):
```

```
            # Evaluate fitness (run tests, calculate scores)
            fitness_scores = self._evaluate_population()

            # Select best performers
            parents = self._select_parents(fitness_scores)

            # Crossover and mutation
            offspring = self._breed(parents)

            # Replace population
            self.population = self._replace_population(offspring, fitness_scores)

            # Check budget
            if self._total_cost() > self.budget:
                break

        return self._best_prompt()

    def _mutate_prompt(self, prompt: str) -> str:
        """Apply mutations: add words, remove words, rephrase."""
        # Random mutations to explore search space
        ...
```

Budget-Aware Optimization:

```
def optimize_with_budget(
    self,
    prompt: str,
    max_cost: float,
    target_score: float = 0.9
) -> OptimizedPrompt:
    """Optimize prompt while respecting budget."""
    cost_tracker = CostTracker(max_cost)

    while cost_tracker.remaining() > 0:
        candidate = self._generate_candidate(prompt)
        score, cost = self._evaluate(candidate)

        if score >= target_score:
            return OptimizedPrompt(candidate, score, cost)

        cost_tracker.record(cost)

    return OptimizedPrompt(prompt, 0.0, 0.0)  # Budget exhausted
```

A/B Testing Framework:

```
# Future: A/B testing for prompt variants
```

```python
class ABTestFramework:
    async def create_ab_test(
        self,
        variant_a: str,
        variant_b: str,
        test_cases: List[RegressionSet]
    ) -> ABTestResult:
        """Run A/B test comparing two prompt variants."""
        # Run tests for both variants
        results_a = await self._run_tests(variant_a, test_cases)
        results_b = await self._run_tests(variant_b, test_cases)

        # Statistical analysis
        return ABTestResult(
            variant_a_score=mean(results_a.scores),
            variant_b_score=mean(results_b.scores),
            statistical_significance=self._calculate_significance(results_a, results_b),
            recommendation=self._recommend_variant(results_a, results_b)
        )
```

Advanced Features

Multi-Provider AI Support:

```python
# Future: Support multiple AI providers
class MultiProviderLLMService:
    def __init__(self):
        self.providers = {
            "openai": OpenAIProvider(),
            "anthropic": AnthropicProvider(),
            "mistral": MistralProvider(),
            "cohere": CohereProvider()
        }

    async def execute_prompt(
        self,
        template: str,
        variables: Dict[str, Any],
        provider: str = "openai",
        model: str = None
    ) -> ExecutionResult:
        provider_service = self.providers[provider]
        return await provider_service.complete(template, model, **variables)
```

Custom Evaluation Functions:

```python
# Future: Allow users to upload custom evaluation code
```

```python
class CustomEvaluator:
    def __init__(self, code: str):
        """Compile user-provided evaluation function."""
        # Sandboxed execution environment
        self.evaluator = self._compile_safely(code)

    def evaluate(self, actual: str, expected: str, metadata: dict) -> float:
        """Execute custom evaluation function."""
        # Run in sandboxed environment
        return self.evaluator(actual, expected, metadata)

# User uploads:
def evaluate(actual, expected, metadata):
    # Custom logic: check for keywords, regex, etc.
    if "error" in actual.lower():
        return 0.0
    # ... custom scoring
    return similarity_score(actual, expected)
```

Advanced Analytics:

```python
# Future: Advanced analytics and insights
class AnalyticsEngine:
    def analyze_trends(self, system_id: UUID) -> TrendAnalysis:
        """Analyze performance trends and predict issues."""
        # Time series analysis
        # Anomaly detection
        # Predictive modeling
        ...

    def generate_insights(self, system_id: UUID) -> List[Insight]:
        """Generate actionable insights."""
        insights = []

        # Cost optimization insights
        if self._can_reduce_cost_without_quality_loss():
            insights.append(Insight(
                type="cost_optimization",
                message="Consider using gpt-4o-mini for 50% cost savings with minimal qualit
                impact="high"
            ))

        # Quality insights
        if self._detect_degradation():
            insights.append(Insight(
                type="quality_alert",
                message="Performance has declined 15% over the past week",
```

```
                impact="critical"
            ))

        return insights
```

Integration APIs:

```
# Future: Webhook integrations
@app.post("/api/webhooks")
async def create_webhook(webhook: WebhookCreate):
    """Create webhook for test run events."""
    # Store webhook configuration
    ...

# Trigger webhooks on events
async def trigger_webhook(event: str, data: dict):
    """Send webhook notification."""
    webhooks = await get_webhooks_for_event(event)
    for webhook in webhooks:
        await httpx.post(webhook.url, json=data)

# Slack integration example
async def notify_slack(test_run: TestRun):
    """Send Slack notification."""
    message = f"Test run completed: {test_run.avg_score:.2f} average score"
    await slack_client.post_message(channel="#ai-monitoring", text=message)
```

Production Readiness

Authentication and Authorization:

```
# Future: Add authentication
from fastapi.security import OAuth2PasswordBearer
from jose import JWTError, jwt

oauth2_scheme = OAuth2PasswordBearer(tokenUrl="token")

async def get_current_user(token: str = Depends(oauth2_scheme)):
    """Verify JWT token and return user."""
    credentials_exception = HTTPException(
        status_code=401,
        detail="Could not validate credentials"
    )
    try:
        payload = jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])
        user_id: str = payload.get("sub")
        if user_id is None:
```

```
            raise credentials_exception
    except JWTError:
        raise credentials_exception

    user = await get_user(user_id)
    return user

# Protect endpoints
@app.post("/api/prompt-systems")
async def create_prompt_system(
    system: PromptSystemCreate,
    current_user: User = Depends(get_current_user)
):
    # User can only create systems for their organization
    ...
```

Audit Logging and Compliance:

```
# Future: Comprehensive audit logging
class AuditLogger:
    async def log_action(
        self,
        user_id: UUID,
        action: str,
        resource_type: str,
        resource_id: UUID,
        metadata: dict = None
    ):
        """Log all user actions for compliance."""
        audit_log = AuditLog(
            user_id=user_id,
            action=action,  # "create", "update", "delete", "execute"
            resource_type=resource_type,  # "prompt_system", "test_run"
            resource_id=resource_id,
            metadata=metadata,
            ip_address=request.client.host,
            timestamp=datetime.utcnow()
        )
        await db.add(audit_log)
        await db.commit()
```

Disaster Recovery:

```
# Backup strategy
# 1. Database backups (automated daily)
# 2. Point-in-time recovery (WAL archiving)
# 3. Test data export (JSON backups)
```

```python
async def export_system_data(system_id: UUID) -> dict:
    """Export all data for a system (backup)."""
    system = await get_prompt_system(system_id)
    regression_sets = await get_regression_sets(system_id)
    test_runs = await get_test_runs(system_id)

    return {
        "system": system.dict(),
        "regression_sets": [rs.dict() for rs in regression_sets],
        "test_runs": [tr.dict() for tr in test_runs]
    }

async def import_system_data(data: dict) -> UUID:
    """Import system data (restore)."""
    # Create system from backup
    ...
```

Performance Monitoring:

```python
# Future: APM integration (DataDog, New Relic, etc.)
from datadog import initialize, statsd

initialize(api_key=settings.DATADOG_API_KEY)

@app.middleware("http")
async def datadog_middleware(request: Request, call_next):
    start_time = time.time()
    response = await call_next(request)
    duration = time.time() - start_time

    statsd.histogram(
        'api.request.duration',
        duration,
        tags=[
            f"method:{request.method}",
            f"endpoint:{request.url.path}",
            f"status:{response.status_code}"
        ]
    )

    return response
```

Appendices

A. Complete API Reference

Base URL: `http://localhost:8000/api`

Prompt Systems **GET /prompt-systems** - List all prompt systems - Response: `List[PromptSystemResponse]`

POST /prompt-systems - Create new prompt system - Request: `PromptSystemCreate` - Response: `PromptSystemResponse` (201)

GET /prompt-systems/{id} - Get prompt system by ID - Response: `PromptSystemResponse`

PUT /prompt-systems/{id} - Update prompt system - Request: `PromptSystemUpdate` - Response: `PromptSystemResponse`

DELETE /prompt-systems/{id} - Delete prompt system (cascades to regression sets) - Response: 204 No Content

Regression Sets **GET /prompt-systems/{system_id}/regression-sets** - List regression sets for a system - Response: `List[RegressionSetResponse]`

POST /prompt-systems/{system_id}/regression-sets - Create regression set - Request: `RegressionSetCreate` - Response: `RegressionSetResponse` (201)

DELETE /prompt-systems/{system_id}/regression-sets/{regression_id} - Delete regression set - Response: 204 No Content

Test Runs **POST /test-runs** - Create and start test run - Request: `TestRunCreate` - Response: `{"test_run_id": "uuid", "status": "pending"}` (202 Accepted)

GET /test-runs - List test runs - Query params: `system_id` (optional), `limit` (default: 50) - Response: `List[TestRunResponse]`

GET /test-runs/{test_run_id} - Get test run status - Response: `TestRunResponse`

GET /test-runs/{test_run_id}/results - Get detailed test results - Response: `List[TestResultResponse]`

Schedules **GET /schedules** - List schedules - Query params: `system_id` (optional) - Response: `List[ScheduleResponse]`

POST /schedules - Create schedule - Request: `ScheduleCreate` - Response: `ScheduleResponse` (201)

PUT /schedules/{schedule_id} - Update schedule - Request: `ScheduleUpdate` - Response: `ScheduleResponse`

DELETE /schedules/{schedule_id} - Delete schedule - Response: 204 No Content

POST /schedules/{schedule_id}/trigger - Manually trigger scheduled test - Response: `{"message": "Test run triggered", "test_run_id": "uuid"}`

B. Database Schema

Complete ER Diagram:

```
prompt_systems (1)  < (many) regression_sets

       (1)

       < (many) test_runs

                (1)

                < (many) test_results

                          (many)

                          > (1) regression_sets

prompt_systems (1)  < (many) schedules
```

Table Relationships: - `prompt_systems` → `regression_sets` (1:many, CASCADE DELETE) - `prompt_systems` → `test_runs` (1:many) - `prompt_systems` → `schedules` (1:many, CASCADE DELETE) - `test_runs` → `test_results` (1:many, CASCADE DELETE) - `regression_sets` → `test_results` (1:many)

Index Definitions: - `idx_test_runs_system_id` on `test_runs(system_id)` - `idx_test_results_test_run_id` on `test_results(test_run_id)` - `idx_prompt_systems_last_reset_date` on `prompt_systems(last_reset_date)`

C. Configuration Guide

Environment Variables:

```
# Database
DATABASE_URL=postgresql+asyncpg://user:pass@host:5432/dbname

# OpenAI
OPENAI_API_KEY=sk-proj-xxxxxxxxxxxxxxxxxxxx
```

```
# Server
BACKEND_HOST=0.0.0.0
BACKEND_PORT=8000
FRONTEND_URL=http://localhost:5173

# CORS
ALLOWED_ORIGINS=http://localhost:5173,http://localhost:3000

# Optional: Feature flags
ENABLE_SCHEDULING=true
ENABLE_BUDGET_CONTROLS=true
MAX_CONCURRENT_TESTS=10
```

Feature Flags:

```
# backend/app/core/config.py
class Settings(BaseSettings):
    ENABLE_SCHEDULING: bool = True
    ENABLE_BUDGET_CONTROLS: bool = True
    MAX_CONCURRENT_TESTS: int = 10
    LOG_LEVEL: str = "INFO"
```

Performance Tuning:

```
# Database connection pool
pool_size=20          # Base connections
max_overflow=40       # Additional under load
pool_recycle=3600     # Recycle after 1 hour

# Async execution
MAX_CONCURRENT_TESTS=10  # Limit parallel test execution

# Caching (if Redis added)
REDIS_URL=redis://localhost:6379
CACHE_TTL=300  # 5 minutes
```

D. Troubleshooting Guide

Common Issues and Solutions:

1. Database Connection Issues - **Error:** `Connection refused` or `Unable to connect` - **Solutions:** - Verify database is running - Check connection string format (`postgresql+asyncpg://...`) - Verify credentials - Check firewall/network settings

2. OpenAI API Errors - **Error:** `AuthenticationError` or `Insufficient quota` - **Solutions:** - Verify API key is correct - Check account has credits - Verify key permissions - Check rate limits

3. Tests Stuck in "Pending" - **Error:** Tests never start executing - **Solutions:** - Check background task logs - Verify database connection in background task - Check for exceptions in logs - Restart backend server

4. Budget Exceeded Errors - **Error:** `BudgetExceededException` - **Solutions:** - Increase `max_daily_cost_usd` for system - Wait for daily reset (midnight UTC) - Manually reset: `UPDATE prompt_systems SET daily_cost_usd = 0, is_budget_paused = false`

5. Scheduler Not Running - **Error:** Scheduled tests don't execute - **Solutions:** - Check scheduler is started: `GET /health` (check `scheduler` field) - Verify schedule is active: `active = true` - Check cron expression is valid - Review scheduler logs

Debug Procedures:

```
# Enable debug logging
import logging
logging.basicConfig(level=logging.DEBUG)

# Check database connection
async def test_db_connection():
    async with AsyncSessionLocal() as db:
        result = await db.execute(text("SELECT 1"))
        print("Database connected:", result.scalar())

# Check OpenAI connection
async def test_openai():
    client = AsyncOpenAI(api_key=settings.OPENAI_API_KEY)
    try:
        response = await client.models.list()
        print("OpenAI connected:", len(response.data), "models")
    except Exception as e:
        print("OpenAI error:", e)
```

Performance Optimization:

```
-- Analyze query performance
EXPLAIN ANALYZE SELECT * FROM test_runs WHERE system_id = '...';

-- Check index usage
SELECT * FROM pg_stat_user_indexes WHERE schemaname = 'public';

-- Vacuum and analyze
VACUUM ANALYZE test_runs;
```

Conclusion

This technical specification provides comprehensive documentation of the AI Prompt System Monitor application, covering:

- **Architecture & Design:** Detailed system architecture, component design, data flows
- **Implementation:** Core business logic, evaluation algorithms, advanced features
- **Technical Deep Dives:** Database design, API design, security implementation
- **Performance & Scalability:** Benchmarks, optimization strategies, scaling approaches
- **Testing & QA:** Unit, integration, and E2E testing strategies
- **Deployment & Operations:** Infrastructure, CI/CD, monitoring
- **Challenges & Solutions:** Real-world problems and their solutions
- **Future Roadmap:** Planned enhancements and production readiness features

This document serves as a complete reference for: - **Technical Interviews:** Deep technical questions about design choices and trade-offs - **Onboarding:** New developers understanding the system - **Maintenance:** Future enhancements and bug fixes - **RAG Systems:** Rich context for intelligent retrieval and question answering

The application demonstrates production-grade engineering practices including: - Async-first architecture for scalability - Concurrent-safe operations (row-level locking) - Comprehensive error handling and recovery - Cost management and budget controls - Real-time progress tracking - Automated monitoring and alerting - Multi-model experimentation and comparison

Total Document Length: ~50-70 pages when formatted as PDF
Code Examples: 100+ code snippets
Architecture Diagrams: 5+ diagrams
API Endpoints: 15+ documented endpoints

End of Technical Specification