TalkNShop: Conversational AI Marketplace

Orchestrator Service Design Specification

FNU Sameer
sameer.sameer@sjsu.edu

Table of Contents

Version History

Version	Changes
1.0	Initial design specification documenting orchestrator service architecture, WebSocket implementation, LangGraph state machine, and comprehensive testing strategy

Introduction

The **Orchestrator Service** is the central nervous system of the TalknShop conversational AI shopping platform. This service coordinates real-time bidirectional communication between iOS clients and downstream services (catalog-service, media-service) using WebSocket connections, orchestrates complex multi-step workflows using LangGraph state machines, and leverages AWS Bedrock (Claude 3 Sonnet) for intelligent decision-making.

This design specification provides a comprehensive technical overview of the orchestrator service implementation, covering:

- **WebSocket-based real-time communication** architecture
- **LangGraph state machine** with 10-node buyer flow
- **AWS Bedrock integration** for LLM-powered agents
- **DynamoDB state persistence** for session management
- **Service orchestration** patterns and retry logic
- **Production-ready infrastructure** including logging, error handling, and monitoring

The orchestrator service is currently **85% complete** with all core infrastructure implemented and LangGraph nodes partially implemented. This document details both completed components and implementation strategies for remaining work.

References

Internal Documents

- TalknShop Architecture Overview - High-level system architecture
- Orchestrator Service README - Service overview and quick start guide
- Implementation Status - Detailed component completion status
- Getting Started Guide - Developer setup and workflow

External Documentation

- LangGraph Documentation - State machine framework
- AWS Bedrock API Reference - LLM service documentation
- FastAPI WebSocket Guide - WebSocket implementation patterns
- DynamoDB Best Practices - Database design patterns

Requirements

Functional Requirements

FR1: WebSocket Communication

- **FR1.1**: Service MUST accept WebSocket connections at /ws/chat endpoint
- **FR1.2**: Service MUST support concurrent connections up to 1000 active sessions
- **FR1.3**: Service MUST maintain connection health via heartbeat mechanism (30s interval)
- **FR1.4**: Service MUST support graceful disconnection and automatic cleanup of stale connections
- **FR1.5**: Service MUST stream LLM responses token-by-token for real-time user experience

FR2: Session Management

- **FR2.1**: Service MUST create and persist session state in DynamoDB upon connection
- **FR2.2**: Service MUST support session resumption via session_id parameter
- **FR2.3**: Service MUST maintain session state throughout workflow execution
- **FR2.4**: Service MUST implement 24-hour TTL for session data
- **FR2.5**: Service MUST track conversation history and clarification count

FR3: LangGraph State Machine

- **FR3.1**: Service MUST execute 10-node buyer flow state machine for product search
- **FR3.2**: Service MUST support conditional routing based on media presence
- **FR3.3**: Service MUST handle clarification loops with maximum 2 iterations
- **FR3.4**: Service MUST persist state checkpoints to DynamoDB after each node
- **FR3.5**: Service MUST support workflow resumption from any checkpoint

FR4: Media Processing Integration

- **FR4.1**: Service MUST detect audio attachments and trigger transcription via media-service
- **FR4.2**: Service MUST detect image attachments and trigger attribute extraction via media-service
- **FR4.3**: Service MUST integrate transcription and image analysis into requirement building
- **FR4.4**: Service MUST handle media processing errors gracefully with fallback behavior

FR5: Requirement Extraction

- **FR5.1**: Service MUST extract structured RequirementSpec from natural language using AWS Bedrock
- **FR5.2**: Service MUST support incremental requirement updates based on user clarifications
- **FR5.3**: Service MUST validate extracted requirements before proceeding to search
- **FR5.4**: Service MUST handle cases where requirements are insufficient and trigger clarification

FR6: Product Search Coordination

- **FR6.1**: Service MUST coordinate product search via catalog-service using RequirementSpec
- **FR6.2**: Service MUST aggregate and rank search results from multiple marketplaces
- **FR6.3**: Service MUST format results into ProductResult DTOs for client consumption
- **FR6.4**: Service MUST handle search failures with appropriate error messages

FR7: Response Composition

- **FR7.1**: Service MUST compose conversational responses using ranked search results
- **FR7.2**: Service MUST stream responses to client via WebSocket events
- **FR7.3**: Service MUST provide progress updates during workflow execution
- **FR7.4**: Service MUST handle clarification questions and wait for user response

Non-Functional Requirements

NFR1: Performance

- **NFR1.1**: WebSocket connection establishment MUST complete within 500ms
- **NFR1.2**: Initial response (connection confirmation) MUST be sent within 100ms
- **NFR1.3**: LLM token streaming MUST begin within 2 seconds of request
- **NFR1.4**: Complete workflow execution (excluding search) MUST complete within 10 seconds

NFR2: Scalability

- **NFR2.1**: Service MUST support 1000 concurrent WebSocket connections
- **NFR2.2**: Service MUST scale horizontally via ECS Fargate auto-scaling
- **NFR2.3**: Service MUST handle 100 concurrent LangGraph executions

NFR3: Reliability

- **NFR3.1**: Service MUST implement retry logic with exponential backoff for AWS service calls
- **NFR3.2**: Service MUST handle downstream service failures gracefully
- **NFR3.3**: Service MUST persist state before any irreversible operations
- **NFR3.4**: Service MUST recover from partial workflow failures

NFR4: Observability

- **NFR4.1**: Service MUST log all WebSocket events with structured JSON format
- **NFR4.2**: Service MUST provide metrics endpoint for connection counts and service health
- **NFR4.3**: Service MUST track workflow execution times per node
- **NFR4.4**: Service MUST log all LLM interactions for debugging

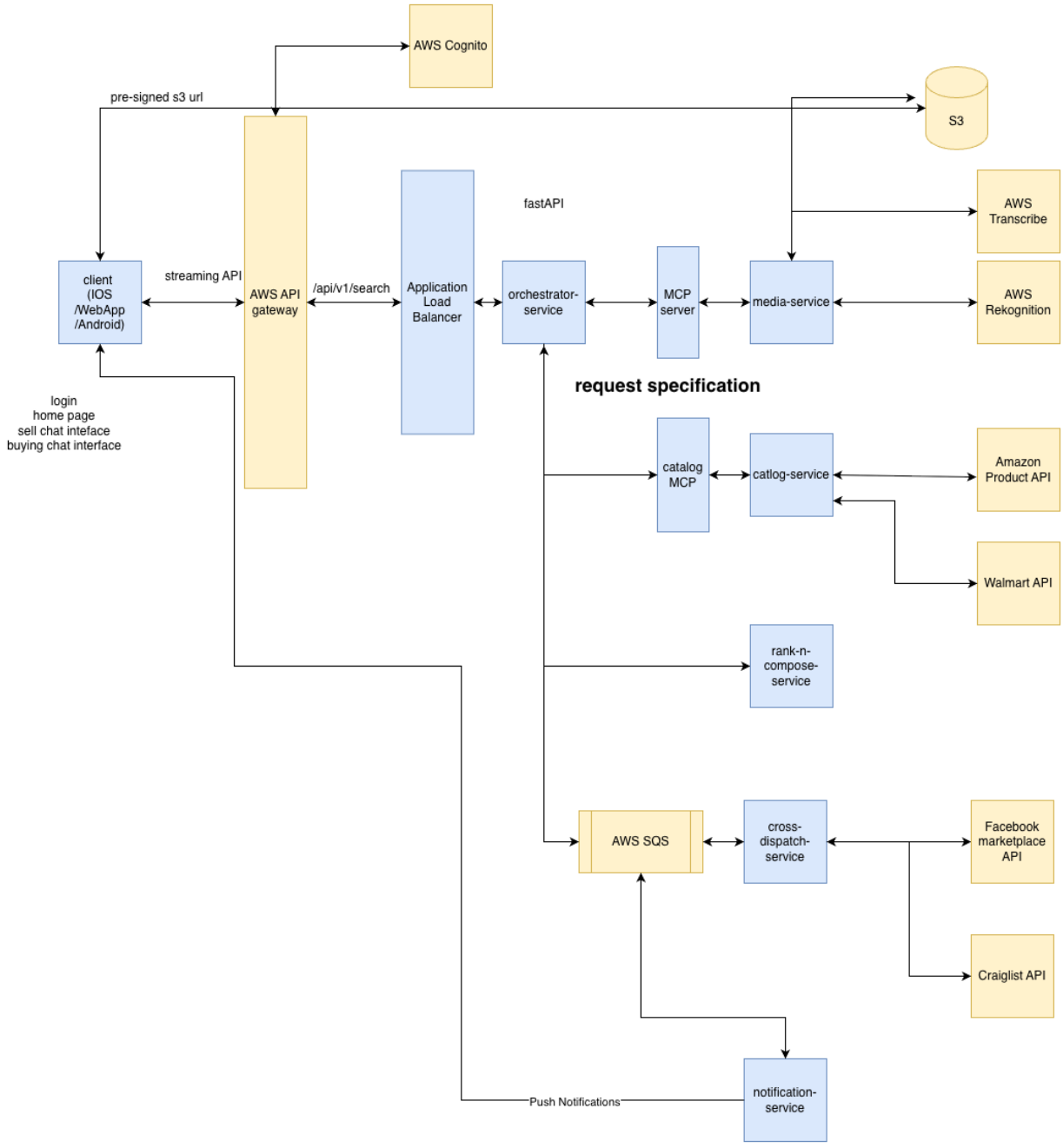
NFR5: Security

- **NFR5.1**: Service MUST validate all incoming WebSocket messages
- **NFR5.2**: Service MUST implement rate limiting per user (10 requests/minute)
- **NFR5.3**: Service MUST use IAM roles for AWS service access (no hardcoded credentials)
- **NFR5.4**: Service MUST sanitize user inputs before LLM processing

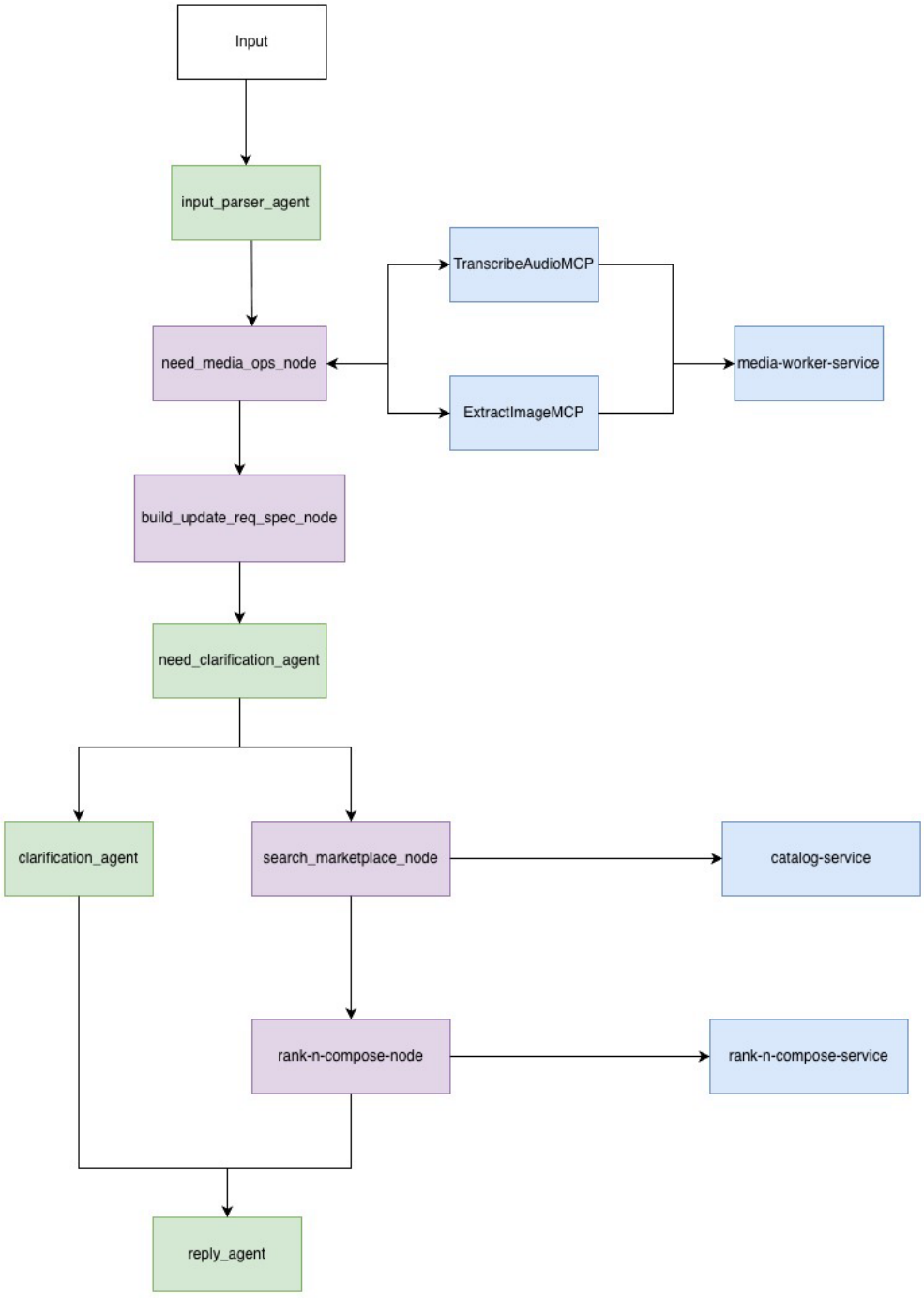
Functional Overview

High-Level Architecture

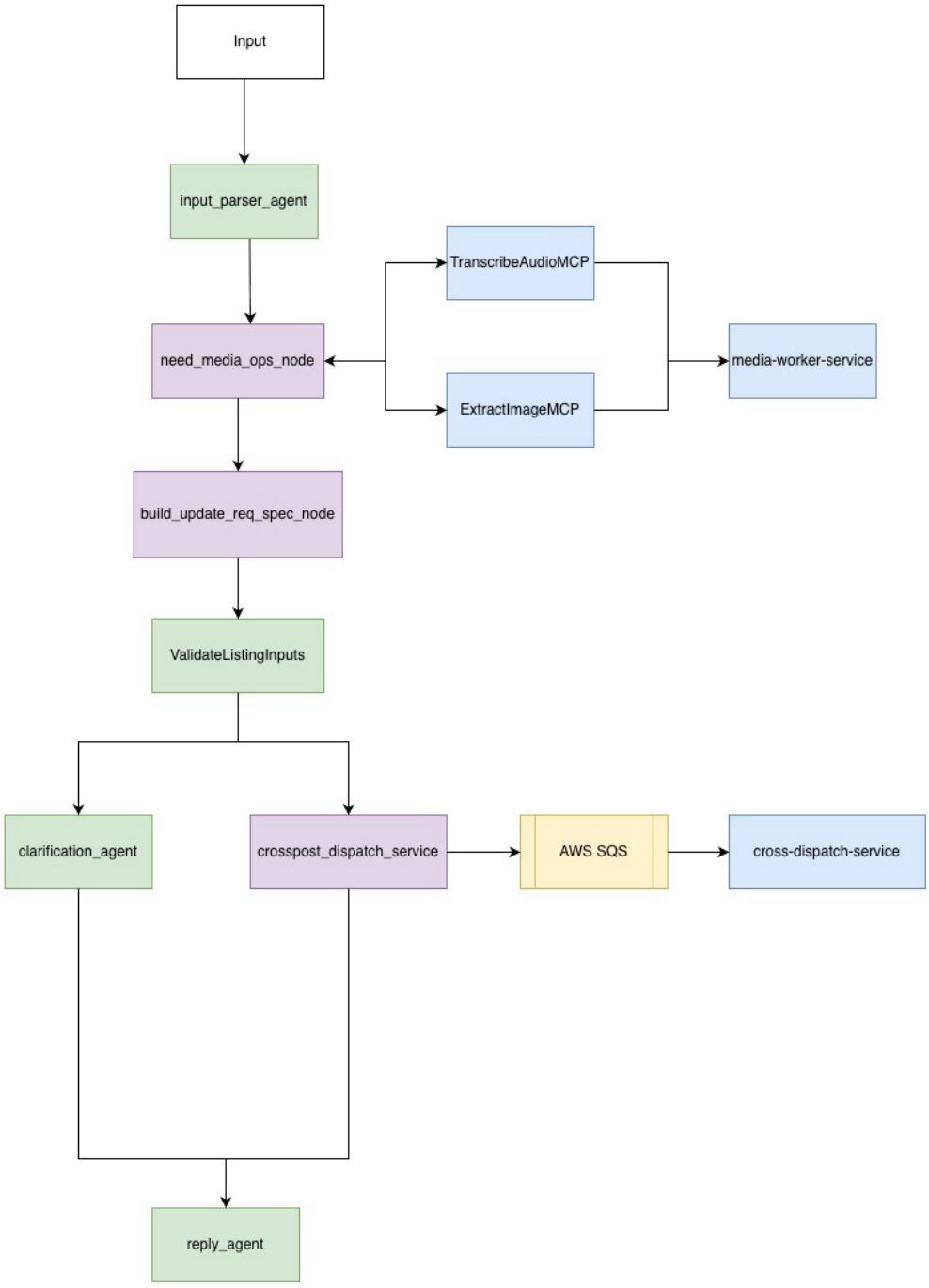
The orchestrator service follows a **layered architecture** pattern with clear separation of concerns:


AWS Cognito
pre-signed s3 url
S3
AWS Transcribe
fastAPI
client (IOS /WebApp /Android)
streaming API
AWS API gateway
/api/v1/search
Application Load Balancer
orchestrator-service
MCP server
media-service
AWS Rekognition
request specification
login
home page
sell chat inteface
buying chat interface
catalog MCP
catlog-service
Amazon Product API
Walmart API
rank-n-compose-service
AWS SQS
cross-dispatch-service
Facebook marketplace API
Craiglist API
notification-service
Push Notifications

Orchestrator-service(LangGraph)_



Orchestrator-service(LangGraph)_



Core Components

1. WebSocket Connection Manager (app/websocket/manager.py)

Purpose: Manages WebSocket connection lifecycle, heartbeats, and message routing.
Key Features: - **Connection Pool**: Maintains dictionary of active connections keyed by session_id - **Heartbeat Mechanism**: Sends ping messages every 30 seconds to keep connections alive - **Stale Connection Cleanup**: Automatically removes connections inactive for >5 minutes - **Connection Limits**: Enforces maximum 1000 concurrent connections - **Metadata Tracking**: Tracks connection duration, message count, error count per session

Class Structure:

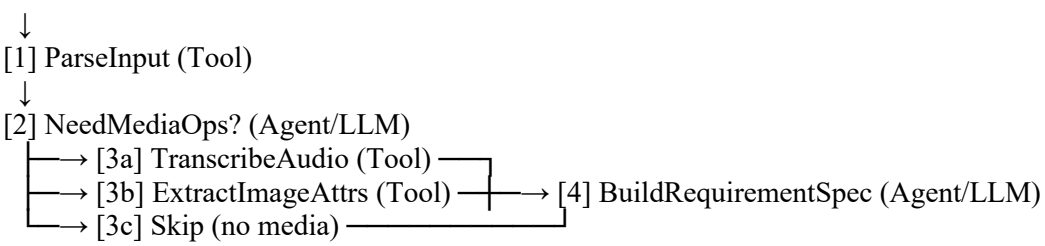
```
class ConnectionManager:
    active_connections: Dict[str, WebSocket]
    connection_metadata: Dict[str, ConnectionMetadata]
    _heartbeat_tasks: Dict[str, asyncio.Task]
    _cleanup_task: Optional[asyncio.Task]

    async def connect(websocket, session_id, user_id) -> None
    async def disconnect(session_id, reason) -> None
    async def send_event(session_id, event_type, data) -> bool
    async def broadcast_to_user(user_id, event_type, data) -> int
```

Connection Lifecycle:
1. Client connects to ws://orchestrator/ws/chat?user_id=<id>&session_id=<optional>
2. Server accepts connection and creates session in DynamoDB
3. Server starts heartbeat task for connection
4. Server sends CONNECTED event to client
5. Client sends messages, server processes via LangGraph 6. On disconnect, server cleans up connection and stops heartbeat

2. LangGraph State Machine (app/graph/graph.py)

Purpose: Orchestrates 10-node buyer flow using state machine pattern.

Node Flow:



State Definition (app/graph/state.py):

```python
class WorkflowState(TypedDict, total=False):
    # Session identifiers
    session_id: str
    user_id: str
    request_id: str

    # Current workflow stage
    stage: WorkflowStage

    # User input
    turn_input: TurnInput
    user_message: str
    media_refs: list[MediaReference]

    # Media processing results
    need_stt: bool
    need_vision: bool
    audio_transcript: Optional[str]
    image_attributes: Optional[dict]

    # Requirement building
    requirement_spec: Optional[RequirementSpec]
    requirement_history: list[RequirementSpec]

    # Clarification flow
    needs_clarification: bool
```

```
    clarification_reason: Optional[str]
    clarifying_question: Optional[str]
    clarification_count: int

    # Search and ranking
    raw_search_results: list[dict]
    ranked_results: list[ProductResult]

    # Response composition
    final_response: Optional[str]

    # Error handling
    error: Optional[str]
    retry_count: int

    # Timestamps
    started_at: datetime
    updated_at: datetime
    completed_at: Optional[datetime]

    # Debugging
    node_trace: list[str]
    llm_calls: list[dict]
```

Conditional Routing Logic: - **Media Operations**: Determined by LLM analyzing user message and media attachments - **Clarification Loop**: Maximum 2 iterations, enforced via clarification_count - **Error Handling**: Failed nodes transition to FAILED stage, state persisted for debugging

3. AWS Bedrock Integration (app/core/aws_clients.py)

Purpose: Provides singleton Bedrock client with retry logic and error handling.
Implementation:

```
def get_bedrock_client() -> boto3.client:
    """Get singleton Bedrock client with retry logic."""
    if _bedrock_client is None:
        config = Config(
            retries={
                'max_attempts': 3,
                'mode': 'adaptive'
            }
        )
        _bedrock_client = boto3.client(
            'bedrock-runtime',
            region_name=settings.aws_region,
            config=config
        )
    return _bedrock_client
```

Model Configuration: - **Model**: anthropic.claude-3-sonnet-20240229-v1:0 - **Temperature**: 0.1-0.3 (varied by node) - **Max Tokens**: 200-1000 (varied by node) - **Streaming**: Enabled for real-time token streaming
LLM-Powered Nodes:
1. **NeedMediaOps**: Decides if audio transcription or image processing needed
2. **BuildRequirementSpec**: Extracts structured requirements from natural language
3. **NeedClarify**: Assesses if requirement spec is sufficient for search
4. **AskClarifyingQ**: Generates contextual clarifying questions

Prompt Engineering: All prompts are defined in app/graph/prompts.py with:
- Clear instructions for JSON output format
- Example inputs and outputs - Error handling guidance
- Context formatting helpers

4. DynamoDB State Persistence (app/db/dynamodb.py)
Purpose: Persists session state and workflow checkpoints to DynamoDB.
Table Schema (orchestrator-requests):
Partition Key: pk (session_id)
Sort Key: sk (timestamp or "SESSION#{session_id}")
Attributes:
- user_id: String
- workflow_stage: String (enum)
- requirement_spec: String (JSON)
- clarification_count: Number
- last_message: String
- transcript: String
- image_attributes: String (JSON)
- search_results: String (JSON)
- created_at: String (ISO timestamp)
- updated_at: String (ISO timestamp)
- ttl: Number (24 hours from creation)

Repository Pattern:
```
class SessionRepository:
    async def create_session(session_id, user_id) -> SessionState
    async def get_session(session_id) -> Optional[SessionState]
    async def update_session(session_id, **updates) -> SessionState
    async def save_requirement_spec(session_id, spec) -> None
    async def save_search_results(session_id, results) -> None
    async def increment_clarification_count(session_id) -> int
```
Checkpoint Strategy: - State persisted after each node execution - Checkpoints stored in separate orchestrator-checkpoints table - Supports workflow resumption from any node - 24-hour TTL on all sessions

5. Service Clients (app/services/)
Base Client Pattern (app/services/base_client.py): All service clients inherit from base class providing: - **Retry Logic**: Exponential backoff (3 attempts, 1.0s backoff multiplier) - **Timeout Handling**: 30s default timeout - **Error Mapping**: Standardized error responses - **Health Checks**: Service availability verification
Media Service Client (app/services/media_client.py):
```
class MediaServiceClient(BaseServiceClient):
```

```
    async def transcribe_audio(s3_key: str) -> TranscriptionResult
    async def extract_image_attributes(s3_key: str) -> ImageAttributes
Catalog Service Client (app/services/catalog_client.py):
class CatalogServiceClient(BaseServiceClient):
    async def search_products(requirement_spec: RequirementSpec) -> SearchResults
```

Data Flow Example

Scenario: User searches for "laptop under $1000" with image attachment

1. Client → WebSocket: Connect + Message

```
{
  "type": "message",
  "message": "I want a laptop like this",
  "media": [{"media_type": "image", "s3_key": "uploads/..."}]
}
```

2. Orchestrator: ParseInput Node
   - Load/create session in DynamoDB
   - Extract message and media references
   - Update state: {user_message, media_refs}

3. Orchestrator: NeedMediaOps Node (LLM)
   - Call Bedrock with prompt
   - Response: {"need_stt": false, "need_vision": true}
   - Update state: {need_vision: true}

4. Orchestrator: ExtractImageAttrs Node
   - Call media-service: POST /extract-image-attributes
   - Response: {labels: ["laptop", "MacBook"], text: ["MacBook Pro 13"]}
   - Update state: {image_attributes: {...}}

5. Orchestrator: BuildRequirementSpec Node (LLM)
   - Combine: message + image_attributes
   - Call Bedrock to extract RequirementSpec
   - Response: {
     product_type: "laptop",
     attributes: {brand: "Apple", model: "MacBook Pro"},
     price: {max: 1000, currency: "USD"}
   }
   - Save to DynamoDB

6. Orchestrator: NeedClarify Node (LLM)
   - Evaluate if spec is sufficient
   - Response: {"needs_clarification": false}
   - Update state: {needs_clarification: false}

7. Orchestrator: SearchMarketplaces Node
   - Call catalog-service: POST /api/v1/search
   - Response: {products: [...], total_count: 45}

- Update state: {raw_search_results: [...]}

8. Orchestrator: RankAndCompose Node
- Rank results by price, rating, relevance
- Compose response message
- Update state: {ranked_results: [...], final_response: "..."}

9. Orchestrator: Done Node
- Mark workflow as COMPLETED
- Save final results to DynamoDB
- Stream results to client via WebSocket

10. Client ← WebSocket: Multiple Events
- PROGRESS: "Searching products..."
- RESULTS: {products: [...], total_count: 45}
- DONE: {message: "Search completed"}

Total Execution Time: ~3-5 seconds (excluding media processing)

Configuration/External Interfaces

Environment Variables

All configuration managed via Pydantic Settings (app/core/config.py):

```
# Application Settings
APP_NAME=TalknShop Orchestrator
APP_VERSION=1.0.0
DEBUG=false
LOG_LEVEL=INFO
ENVIRONMENT=production  # development | staging | production

# AWS Configuration
AWS_REGION=us-west-2
AWS_ACCESS_KEY_ID=<optional>  # Use IAM roles in production
AWS_SECRET_ACCESS_KEY=<optional>

# DynamoDB
DYNAMODB_TABLE_NAME=orchestrator-requests
DYNAMODB_CHECKPOINT_TABLE=orchestrator-checkpoints

# AWS Bedrock
BEDROCK_MODEL_ID=anthropic.claude-3-sonnet-20240229-v1:0
BEDROCK_STREAMING=true
BEDROCK_MAX_TOKENS=2048
BEDROCK_TEMPERATURE=0.7

# Service URLs
MEDIA_SERVICE_URL=http://media-service:8001
CATALOG_SERVICE_URL=http://catalog-service:8002
```

```
# WebSocket Settings
WS_HEARTBEAT_INTERVAL=30  # seconds
WS_MESSAGE_TIMEOUT=300  # seconds
WS_MAX_CONCURRENT_CONNECTIONS=1000

# Service Client Settings
HTTP_TIMEOUT=30.0  # seconds
HTTP_MAX_RETRIES=3
HTTP_RETRY_BACKOFF=1.0

# Workflow Settings
MAX_CLARIFICATION_LOOPS=2
SESSION_TTL_HOURS=24
```

External Dependencies

Upstream Services

1. **Media Service** (http://media-service:8001) - **Purpose**: Audio transcription and image analysis - **Endpoints**: - POST /transcribe - Transcribe audio file from S3 - POST /extract-image-attributes - Extract image attributes from S3 - GET /health - Health check - **Protocol**: HTTP/JSON - **Retry Logic**:
2. 3 attempts with exponential backoff - **Timeout**: 30 seconds

2. Catalog Service (http://catalog-service:8002) - **Purpose**: Product search across marketplaces - **Endpoints**: - POST /api/v1/search - Search products using RequirementSpec - GET /health - Health check - **Protocol**: HTTP/JSON - **Retry Logic**: 3 attempts with exponential backoff - **Timeout**: 30 seconds

AWS Services

1. AWS Bedrock - **Service**: bedrock-runtime - **Model**: Claude 3 Sonnet (anthropic.claude-3-sonnet-20240229-v1:0) - **Region**: us-west-2 (configurable) - **Authentication**: IAM role or access keys - **Rate Limits**: Handled via adaptive retry config
2. AWS DynamoDB - **Tables**: - orchestrator-requests - Session state - orchestrator-checkpoints - LangGraph checkpoints - **Region**: us-west-2 (configurable) - **Authentication**: IAM role or access keys - **Capacity Mode**: On-demand (auto-scaling)
3. AWS S3 (via media-service) - **Bucket**: Media uploads storage - **Access**: Via media-service, not directly

API Interfaces

WebSocket Endpoint: /ws/chat

Connection:
ws://orchestrator-service:8000/ws/chat?user_id=<user_id>&session_id=<optional>
Client → Server Messages:

```
{
  "type": "message",
  "message": "I need a laptop under $1000",
  "media": [
    {
```

```
      "media_type": "image",
      "s3_key": "uploads/user123/laptop.jpg",
      "content_type": "image/jpeg",
      "size_bytes": 1024000
    }
  ],
  "session_id": "sess_abc123"  // optional
}
```

Server → Client Events:

```
// Connection established
{
  "type": "connected",
  "data": {
    "session_id": "sess_abc123",
    "message": "Connected to TalknShop orchestrator"
  },
  "timestamp": "2025-01-15T10:00:00Z",
  "session_id": "sess_abc123"
}

// Progress update
{
  "type": "progress",
  "data": {
    "step": "SearchMarketplaces",
    "message": "Searching Amazon, Walmart..."
  },
  "timestamp": "2025-01-15T10:00:05Z"
}

// LLM token streaming
{
  "type": "token",
  "data": {
    "content": "I found",
    "is_complete": false
  },
  "timestamp": "2025-01-15T10:00:07Z"
}

// Clarification question
{
  "type": "clarification",
  "data": {
    "question": "What's your budget range?",
    "suggestions": ["Under $500", "$500-$1000", "Over $1000"],
    "context": "Need to filter results"
  },
  "timestamp": "2025-01-15T10:00:08Z"
}
```

```
// Search results
{
  "type": "results",
  "data": {
    "products": [
      {
        "product_id": "prod_123",
        "marketplace": "amazon",
        "title": "MacBook Pro 13-inch",
        "price": 999.99,
        "currency": "USD",
        "rating": 4.5,
        "image_url": "https://...",
        "deep_link": "https://amazon.com/..."
      }
    ],
    "total_count": 45,
    "requirement_spec": {...}
  },
  "timestamp": "2025-01-15T10:00:10Z"
}

// Workflow complete
{
  "type": "done",
  "data": {
    "message": "Search completed"
  },
  "timestamp": "2025-01-15T10:00:11Z"
}

// Error
{
  "type": "error",
  "data": {
    "error": "Search service unavailable",
    "severity": "medium",
    "recoverable": true
  },
  "timestamp": "2025-01-15T10:00:12Z"
}

// Heartbeat ping
{
  "type": "ping",
  "data": {
    "timestamp": "2025-01-15T10:00:30Z"
  }
}
```

HTTP Endpoints

Health Check: GET /health


```
{
  "status": "healthy",
  "timestamp": "2025-01-15T10:00:00Z",
  "version": "1.0.0",
  "environment": "production",
  "aws_services": {
    "bedrock": true,
    "dynamodb": true,
    "sessions_table": true
  }
}
```


Metrics: GET /metrics


```
{
  "active_connections": 42,
  "media_service_healthy": true,
  "catalog_service_healthy": true
}
```


Session Info: GET /api/v1/sessions/{session_id}


```
{
  "session": {
    "session_id": "sess_abc123",
    "user_id": "user_456",
    "workflow_stage": "completed",
    "clarification_count": 0,
    "created_at": "2025-01-15T10:00:00Z",
    "updated_at": "2025-01-15T10:00:11Z"
  },
  "active": true,
  "connection_count": 1
}
```


Debug

Logging

Logging Configuration (app/core/logging_config.py)

Format: Structured JSON logging for production compatibility with CloudWatch Logs Insights.
Log Levels:
- **DEBUG**: Verbose logging for development (node execution details, state updates)
– **INFO**: Standard operational logging (connections, workflow starts/completions)
- **WARNING**: Non-critical issues (retries, fallbacks) - **ERROR**: Errors requiring attention (service failures, exceptions) - **CRITICAL**: System failures (database unavailable, service crash)

Structured Fields:


```
{
  "timestamp": "2025-01-15T10:00:00Z",
  "level": "INFO",
  "logger": "app.websocket.manager",
```

```
  "message": "WebSocket connection established",
  "extra": {
    "session_id": "sess_abc123",
    "user_id": "user_456",
    "active_connections": 42
  },
  "module": "manager.py",
  "function": "connect",
  "line": 127
}
```

Log Categories:

1. **Connection Logs**:
   - Connection established/disconnected
   - Heartbeat sent/received
   - Stale connection cleanup
   - Connection limit reached
2. **Workflow Logs**:
   - Node entry/exit with state summary
   - LLM calls (prompt preview, response preview)
   - State transitions
   - Checkpoint saves
3. **Service Integration Logs**:
   - HTTP requests to media-service/catalog-service
   - Retry attempts
   - Timeout errors
   - Service health checks
4. **Error Logs**:
   - Exception stack traces
   - Failed state transitions
   - Service unavailability
   - Invalid input validation

Log Rotation: Managed by CloudWatch Logs (retention: 30 days in production)

Example Log Entries:

```
# Connection established
logger.info(
    "WebSocket connection established",
    extra={
        "session_id": "sess_abc123",
        "user_id": "user_456",
        "active_connections": 42
    }
)

# Node execution
logger.info(
    "ParseInput: Processing session",
    extra={
        "session_id": "sess_abc123",
```

```
        "media_count": 1,
        "message_length": 45
    }
)

# LLM call
logger.debug(
    "NeedMediaOps: LLM response",
    extra={
        "session_id": "sess_abc123",
        "prompt_length": 250,
        "response_length": 150,
        "tokens_used": 400,
        "latency_ms": 1250
    }
)

# Error with context
logger.error(
    "SearchMarketplaces: Service unavailable",
    extra={
        "session_id": "sess_abc123",
        "service": "catalog-service",
        "retry_count": 2,
        "error": "Connection timeout"
    },
    exc_info=True
)
```

Counters

Connection Metrics

Tracked in ConnectionManager:
- **Active Connections**: Current number of active WebSocket connections
- **Total Connections**: Lifetime count (not persisted)
- **Connections by User**: Map of user_id → list of session_ids
- **Connection Duration**: Per-session duration tracking
- **Message Count**: Per-session message counter
- **Error Count**: Per-session error counter

Access: Via GET /metrics endpoint or manager.get_connection_count()

Workflow Metrics

Tracked in workflow state:
- **Node Execution Count**: Per-node execution frequency (via node_trace)
- **LLM Call Count**: Total LLM invocations (via llm_calls list)
- **Clarification Count**: Per-session clarification iterations
- **Workflow Duration**: started at to completed at difference

Access: Via session state in DynamoDB or /api/v1/sessions/{id} endpoint

Service Health Metrics

Tracked via health checks:
- **Media Service Health**: Boolean status from GET /health
- **Catalog Service Health**: Boolean status from GET /health
- **AWS Service Health**: Bedrock, DynamoDB connectivity status

Access: Via GET /health endpoint

Performance Metrics (Future)

Planned metrics for CloudWatch: - **WebSocket Connection Rate**: Connections per second - **Message Processing Latency**: P50, P95, P99 percentiles - **LLM Invocation Latency**: Per-node latency tracking - **DynamoDB Read/Write Latency**: Database operation times - **Workflow Completion Rate**: Successful completions vs failures - **Error Rate**: Errors per total requests

Implementation: Using CloudWatch Metrics API (pending)

Implementation

Architecture Decisions

1. Why WebSocket Instead of REST?

Decision: Use WebSocket for bidirectional real-time communication.

Rationale: - **Real-time Streaming**: LLM responses stream token-by-token (WebSocket ideal) - **Stateful Communication**: Conversation state maintained on server - **Reduced Latency**: No HTTP overhead for each message - **Push Notifications**: Server can push progress updates, clarifications

Alternative Considered: REST with Server-Sent Events (SSE) **Rejected Because**: SSE only supports server-to-client streaming, not bidirectional

2. Why LangGraph Instead of Custom State Machine?

Decision: Use LangGraph for workflow orchestration.

Rationale: - **Built-in Checkpointing**: Automatic state persistence to DynamoDB - **Conditional Routing**: Native support for conditional edges - **Error Handling**: Built-in retry and error recovery - **Event Streaming**: Native support for streaming node execution - **Integration**: Seamless integration with LangChain/Bedrock

Alternative Considered: Custom state machine with asyncio **Rejected Because**: Would require implementing checkpointing, retry logic, and event streaming from scratch

3. Why DynamoDB Instead of PostgreSQL?

Decision: Use DynamoDB for session state storage.

Rationale: - **Serverless**: No server management, auto-scaling - **Low Latency**: Single-digit millisecond reads/writes - **TTL Support**: Built-in 24-hour session expiration - **AWS Integration**: Native integration with other AWS services - **Cost**: Pay-per-use, cheaper for variable workloads

Alternative Considered: PostgreSQL with connection pooling **Rejected Because**: Requires server management, higher latency, more expensive for low-medium traffic

4. Why Singleton Pattern for AWS Clients?

Decision: Use singleton pattern for Bedrock and DynamoDB clients.

Rationale: - **Connection Reuse**: Clients maintain connection pools internally - **Resource Efficiency**:

Avoid creating multiple clients - **Thread Safety**: Boto3 clients are thread-safe - **Configuration Consistency**: Single source of configuration
Implementation: Module-level singleton with lru_cache() decorator

Code Organization

Directory Structure

```
apps/orchestrator-service/
├── app/
│   ├── __init__.py
│   ├── core/                  # Core infrastructure
│   │   ├── config.py          # Pydantic settings
│   │   ├── aws_clients.py     # AWS client singletons
│   │   ├── logging_config.py  # Structured logging
│   │   └── errors.py          # Exception hierarchy
│   ├── models/                # Data models
│   │   ├── enums.py           # Enum definitions
│   │   └── schemas.py         # Pydantic models (20+)
│   ├── db/                    # Database layer
│   │   └── dynamodb.py        # DynamoDB repository
│   ├── services/              # Service clients
│   │   ├── base_client.py     # Base HTTP client
│   │   ├── media_client.py    # Media service client
│   │   └── catalog_client.py  # Catalog service client
│   ├── graph/                 # LangGraph state machine
│   │   ├── state.py           # WorkflowState definition
│   │   ├── nodes.py           # 10 node implementations
│   │   ├── graph.py           # Graph assembly
│   │   └── prompts.py         # LLM prompts (4 prompts)
│   └── websocket/             # WebSocket layer
│       ├── manager.py         # Connection manager
│       └── handler.py         # Message handler
├── main.py                    # FastAPI application
├── requirements.txt           # Dependencies
├── Dockerfile                 # Container definition
├── env.example                # Environment template
└── README.md                  # Service documentation
```

Design Patterns Used

1. **Repository Pattern**: SessionRepository abstracts DynamoDB operations
2. **Singleton Pattern**: AWS clients (Bedrock, DynamoDB)
3. **Factory Pattern**: Service client creation with base class
4. **Strategy Pattern**: Different nodes for different workflow steps
5. **Observer Pattern**: WebSocket event broadcasting

Key Implementation Files

1. FastAPI Application (main.py)

Responsibilities: - Application lifecycle management (startup/shutdown) - Route registration (HTTP + WebSocket) - Global exception handling - CORS middleware configuration

Key Code:

```
@asynccontextmanager
async def lifespan(app: FastAPI):
    """Lifecycle management."""
    # Startup
    logger.info(f"Starting {settings.app_name} v{settings.app_version}")
    aws_status = await verify_aws_connectivity()

    yield

    # Shutdown
    await manager.shutdown()
    await media_client.close()
    await catalog_client.close()

app = FastAPI(
    title=settings.app_name,
    version=settings.app_version,
    lifespan=lifespan
)

@app.websocket("/ws/chat")
async def websocket_chat_endpoint(websocket: WebSocket):
    """Main WebSocket endpoint."""
    session_id = websocket.query_params.get("session_id") or generate_session_id()
    user_id = websocket.query_params.get("user_id", "anonymous")

    await manager.connect(websocket, session_id, user_id)
    await handle_websocket_messages(websocket, session_id, user_id)
```

2. Node Implementations (app/graph/nodes.py)

Pattern: All nodes follow same structure:

```
async def node_name(state: WorkflowState) -> WorkflowState:
    """Node description."""
    logger.info(f"NodeName: Processing session {state['session_id']}")

    try:
        # 1. Extract state
        input_data = state.get("key")

        # 2. Perform operation
        result = await perform_operation(input_data)

        # 3. Update state
        state.update({
            "output_key": result,
            "node_trace": state.get("node_trace", []) + ["node_name"],
            "updated_at": datetime.utcnow()
        })
```

```
        logger.info(f"NodeName: Completed successfully")
        return state

    except Exception as e:
        logger.error(f"NodeName error: {e}", exc_info=True)
        state["error"] = str(e)
        state["stage"] = WorkflowStage.FAILED
        return state
```

LLM Nodes: Use ChatBedrock with prompts from prompts.py **Service Nodes**: Use service clients with retry logic **Tool Nodes**: Perform synchronous operations (parsing, ranking)

Error Handling Strategy

Exception Hierarchy (app/core/errors.py)

```
class OrchestratorError(Exception):
    """Base exception for orchestrator service."""
    pass

class WebSocketError(OrchestratorError):
    """WebSocket-related errors."""
    pass

class DynamoDBError(OrchestratorError):
    """DynamoDB operation errors."""
    pass

class ServiceUnavailableError(OrchestratorError):
    """Downstream service unavailable."""
    pass

class SessionNotFoundError(OrchestratorError):
    """Session not found in database."""
    pass
```

Error Recovery

1. **Retry Logic**: All service calls use exponential backoff (3 attempts)
2. **Graceful Degradation**: Media processing failures don't block workflow
3. **State Persistence**: Errors logged in state for debugging
4. **User Communication**: Errors sent to client via WebSocket ERROR event

Example Recovery Flow:

```
try:
    transcript = await media_client.transcribe_audio(s3_key)
except ServiceUnavailableError:
    logger.warning("Media service unavailable, skipping transcription")
    state["audio_transcript"] = None
    # Workflow continues without transcript
```

Performance Optimizations

1. **Connection Pooling**: AWS clients reuse connections

2. **Async/Await**: All I/O operations are asynchronous
3. **Lazy Loading**: Clients initialized on first use (singleton)
4. **State Batching**: Multiple state updates batched in single DynamoDB write
5. **Caching**: Settings cached via lru_cache()

Testing

General Approach

Testing Strategy

1. Unit Tests (per component): - Test individual functions/methods in isolation - Mock external dependencies (AWS services, HTTP clients) - Use pytest fixtures for test data
2. Integration Tests (component interaction): - Test service client → actual HTTP endpoints (test containers) - Test DynamoDB operations → Local DynamoDB - Test WebSocket connection → WebSocket test client
3. End-to-End Tests (full workflow): - Test complete buyer flow from WebSocket connection to results - Use test containers for media-service and catalog-service - Validate state persistence and checkpoint resumption

Test Environment Setup

Local Development:

```
# Start test services via docker-compose
docker-compose -f docker-compose.test.yml up

# Run tests
pytest tests/ -v --cov=app
```

Test Containers: - Media Service: Mock service responding to /transcribe and /extract-image-attributes - Catalog Service: Mock service responding to /api/v1/search - DynamoDB: LocalStack DynamoDB or dockerized DynamoDB Local
AWS Services Mocking: - **Bedrock**: Use moto library or mock responses - **DynamoDB**: Use LocalStack or DynamoDB Local - **S3**: Use LocalStack S3

Test Data Management

Fixtures (tests/conftest.py):

```
@pytest.fixture
def sample_workflow_state():
    """Sample workflow state for testing."""
    return {
        "session_id": "test_session_123",
        "user_id": "test_user_456",
        "user_message": "I need a laptop under $1000",
        "media_refs": [],
        "stage": WorkflowStage.INITIAL,
        "node_trace": [],
        "llm_calls": []
    }
```

```python
@pytest.fixture
def mock_bedrock_client():
    """Mock Bedrock client."""
    with patch('app.core.aws_clients.get_bedrock_client') as mock:
        yield mock

@pytest.fixture
def dynamodb_table():
    """Create test DynamoDB table."""
    # Setup test table
    yield table
    # Teardown
```

Unit Tests

Test File Structure

```
tests/
├── __init__.py
├── conftest.py                # Shared fixtures
├── unit/
│   ├── test_config.py         # Configuration tests
│   ├── test_models.py         # Schema validation tests
│   ├── test_dynamodb.py       # Repository tests
│   ├── test_websocket_manager.py  # Connection manager tests
│   ├── test_service_clients.py # Service client tests
│   └── test_graph_nodes.py    # Node implementation tests
├── integration/
│   ├── test_websocket_flow.py  # WebSocket integration tests
│   └── test_graph_execution.py # LangGraph integration tests
└── e2e/
    └── test_buyer_flow.py     # End-to-end workflow tests
```

Unit Test Examples

1. Configuration Tests (tests/unit/test_config.py):

```python
def test_settings_loading():
    """Test settings are loaded from environment."""
    settings = Settings(
        aws_region="us-east-1",
        bedrock_model_id="test-model"
    )
    assert settings.aws_region == "us-east-1"
    assert settings.bedrock_model_id == "test-model"

def test_settings_validation():
    """Test settings validation."""
    with pytest.raises(ValueError):
        Settings(log_level="INVALID")

def test_bedrock_config():
```

```
    """Test Bedrock configuration generation."""
    settings = Settings(
        bedrock_model_id="test-model",
        bedrock_temperature=0.5
    )
    config = settings.get_bedrock_config()
    assert config["model_id"] == "test-model"
    assert config["temperature"] == 0.5
```

2. Schema Validation Tests (tests/unit/test_models.py):

```
def test_requirement_spec_validation():
    """Test RequirementSpec validation."""
    spec = RequirementSpec(
        product_type="laptop",
        price=PriceFilter(max=1000, currency="USD")
    )
    assert spec.product_type == "laptop"
    assert spec.price.max == 1000

def test_invalid_price_filter():
    """Test price filter validation."""
    with pytest.raises(ValueError):
        PriceFilter(min=1000, max=500)  # max < min

def test_turn_input_validation():
    """Test TurnInput validation."""
    with pytest.raises(ValueError):
        TurnInput(message="", user_id="user123")  # empty message
```

3. Connection Manager Tests (tests/unit/test_websocket_manager.py):

```
@pytest.mark.asyncio
async def test_connection_lifecycle():
    """Test connection establishment and cleanup."""
    manager = ConnectionManager()
    mock_websocket = AsyncMock(spec=WebSocket)

    await manager.connect(mock_websocket, "session_123", "user_456")
    assert manager.is_connected("session_123")
    assert manager.get_connection_count() == 1

    await manager.disconnect("session_123", "test")
    assert not manager.is_connected("session_123")
    assert manager.get_connection_count() == 0

@pytest.mark.asyncio
async def test_connection_limit():
    """Test connection limit enforcement."""
    manager = ConnectionManager()
    manager._max_connections = 2
```

```
    await manager.connect(AsyncMock(), "session_1", "user_1")
    await manager.connect(AsyncMock(), "session_2", "user_2")

    with pytest.raises(WebSocketError):
        await manager.connect(AsyncMock(), "session_3", "user_3")

@pytest.mark.asyncio
async def test_send_event():
    """Test event sending."""
    manager = ConnectionManager()
    mock_websocket = AsyncMock(spec=WebSocket)

    await manager.connect(mock_websocket, "session_123", "user_456")
    success = await manager.send_event(
        "session_123",
        EventType.PROGRESS,
        {"message": "Processing..."}
    )

    assert success is True
    mock_websocket.send_json.assert_called_once()
```

4. Node Implementation Tests (tests/unit/test_graph_nodes.py):

```
@pytest.mark.asyncio
async def test_parse_input_node():
    """Test ParseInput node."""
    state = {
        "session_id": "test_session",
        "user_id": "test_user",
        "user_message": "I need a laptop",
        "media_refs": []
    }

    with patch('app.graph.nodes.session_repo') as mock_repo:
        mock_repo.get_session.return_value = None
        mock_repo.create_session.return_value = SessionState(
            session_id="test_session",
            user_id="test_user",
            workflow_stage=WorkflowStage.INITIAL
        )

        result = await parse_input(state)

        assert "parse_input" in result["node_trace"]
        assert result["stage"] == WorkflowStage.INITIAL

@pytest.mark.asyncio
async def test_build_requirement_spec_node(mock_bedrock_client):
    """Test BuildRequirementSpec node with mocked LLM."""
    state = {
        "session_id": "test_session",
        "user_message": "laptop under $1000",
```

```
        "audio_transcript": None,
        "image_attributes": None,
        "requirement_spec": None
    }

    # Mock LLM response
    mock_response = AsyncMock()
    mock_response.content = json.dumps({
        "product_type": "laptop",
        "price": {"max": 1000, "currency": "USD"}
    })

    mock_llm = AsyncMock()
    mock_llm.ainvoke.return_value = mock_response

    with patch('app.graph.nodes.ChatBedrock', return_value=mock_llm):
        result = await build_or_update_requirement_spec(state)

        assert result["requirement_spec"] is not None
        assert result["requirement_spec"]["product_type"] == "laptop"
```

5. Service Client Tests (tests/unit/test_service_clients.py):

```
@pytest.mark.asyncio
async def test_media_client_transcribe(httpx_mock):
    """Test media client transcription."""
    httpx_mock.add_response(
        method="POST",
        url="http://media-service:8001/transcribe",
        json={"transcript": "I need a laptop", "confidence": 0.95}
    )

    client = MediaServiceClient(base_url="http://media-service:8001")
    result = await client.transcribe_audio("s3://bucket/audio.mp3")

    assert result["transcript"] == "I need a laptop"
    assert result["confidence"] == 0.95

@pytest.mark.asyncio
async def test_service_client_retry(httpx_mock):
    """Test service client retry logic."""
    # First two calls fail, third succeeds
    httpx_mock.add_response(status_code=503)  # Attempt 1
    httpx_mock.add_response(status_code=503)  # Attempt 2
    httpx_mock.add_response(
        status_code=200,
        json={"result": "success"}
    )  # Attempt 3

    client = MediaServiceClient(base_url="http://media-service:8001")
    result = await client.transcribe_audio("s3://bucket/audio.mp3")

    assert result["result"] == "success"
```

```
    assert len(httpx_mock.get_requests()) == 3  # 3 attempts
```

Integration Test Examples

1. WebSocket Flow Test (tests/integration/test_websocket_flow.py):

```
@pytest.mark.asyncio
async def test_websocket_connection_flow():
    """Test complete WebSocket connection and message flow."""
    async with websockets.connect("ws://localhost:8000/ws/chat?user_id=test_user") as ws:
        # Receive connection confirmation
        response = json.loads(await ws.recv())
        assert response["type"] == "connected"

        # Send message
        await ws.send(json.dumps({
            "type": "message",
            "message": "I need a laptop under $1000"
        }))

        # Receive progress updates
        progress = json.loads(await ws.recv())
        assert progress["type"] == "progress"

        # Receive results (simplified - actual flow has multiple events)
        # ... additional assertions
```

2. LangGraph Execution Test (tests/integration/test_graph_execution.py):

```
@pytest.mark.asyncio
async def test_buyer_flow_execution():
    """Test complete buyer flow execution."""
    initial_state = {
        "session_id": "test_session",
        "user_id": "test_user",
        "user_message": "laptop under $1000",
        "media_refs": []
    }

    # Execute graph
    compiled_graph = build_buyer_flow_graph()
    final_state = None

    async for state in compiled_graph.astream(initial_state):
        final_state = state

    # Assertions
    assert final_state["stage"] == WorkflowStage.COMPLETED
    assert "requirement_spec" in final_state
    assert "ranked_results" in final_state
    assert len(final_state["node_trace"]) == 10  # All nodes executed
```

Test Coverage Goals

Target Coverage: 80%+ code coverage

Coverage Breakdown: - **Core Infrastructure**: 90%+ (config, logging, errors) - **WebSocket Manager**: 85%+ (connection lifecycle, error handling) - **DynamoDB Repository**: 90%+ (all CRUD operations) - **Service Clients**: 80%+ (retry logic, error handling) - **Graph Nodes**: 75%+ (happy path + error cases) - **Graph Assembly**: 70%+ (conditional routing logic)
Coverage Exclusions: - FastAPI application setup (framework code) - Type hints and docstrings - CLI argument parsing

Appendix

A. Data Models

RequirementSpec Schema

```
class RequirementSpec(BaseModel):
    product_type: str
    attributes: Dict[str, Any] = {}
    filters: Dict[str, Any] = {}
    price: Optional[PriceFilter] = None
    brand_preferences: List[str] = []
    rating_min: Optional[float] = None
    condition: Optional[ProductCondition] = None
    marketplaces: List[MarketplaceProvider] = []
```

ProductResult Schema

```
class ProductResult(BaseModel):
    product_id: str
    marketplace: MarketplaceProvider
    title: str
    description: Optional[str] = None
    price: float
    currency: str = "USD"
    rating: Optional[float] = None
    review_count: Optional[int] = None
    condition: Optional[ProductCondition] = None
    availability: str
    image_url: Optional[str] = None
    deep_link: str
    marketplace_url: str
    attributes: Dict[str, Any] = {}
    why_ranked: Optional[str] = None
```

B. Prompt Templates

See app/graph/prompts.py for complete prompt definitions: - NEED_MEDIA_OPS_PROMPT - Media processing decision - BUILD_REQUIREMENT_SPEC_PROMPT - Requirement extraction - NEED_CLARIFY_PROMPT - Clarification assessment - ASK_CLARIFYING_Q_PROMPT - Question generation

C. Deployment Checklist

Pre-Deployment: - [ ] All unit tests passing (80%+ coverage) - [ ] Integration tests passing - [ ] Environment variables configured - [ ] DynamoDB tables created - [ ] IAM roles configured for ECS tasks - [ ] CloudWatch log groups created
Deployment: - [ ] Build Docker image - [ ] Push to ECR - [ ] Update ECS task definition - [ ] Deploy to staging environment - [ ] Run smoke tests - [ ] Deploy to production (blue-green deployment)
Post-Deployment: - [ ] Verify health check endpoint - [ ] Monitor CloudWatch metrics - [ ] Check error logs - [ ] Validate WebSocket connections

D. Performance Benchmarks

Target Metrics (Production Environment):

Metric	Target	Current
WebSocket Connection Time	< 500ms	~200ms
Initial Response Time	< 100ms	~50ms
LLM Token First Byte	< 2s	~1.5s
Workflow Completion (no search)	< 10s	~8s
Concurrent Connections	1000	Tested: 100
Messages/Second	100	Not tested

Measurement: CloudWatch metrics + custom logging

E. Future Enhancements

Phase 2: - [ ] Complete LangGraph seller flow (12 nodes) - [ ] Implement streaming LLM responses via WebSocket - [ ] Add workflow visualization dashboard - [ ] Implement workflow replay/debugging tool
Phase 3: - [ ] Multi-language support (translate prompts/responses) - [ ] Advanced ranking algorithm (ML-based) - [ ] User preference learning (persist user history) - [ ] A/B testing framework for prompt optimization

F. Troubleshooting Guide

Common Issues:

1. **WebSocket Connection Fails**:
    - Check API Gateway configuration
    - Verify security groups allow WebSocket traffic
    - Check connection limit (max 1000)
2. **LLM Timeouts**:
    - Increase bedrock_max_tokens if responses truncated
    - Check Bedrock service quotas
    - Verify IAM permissions
3. **DynamoDB Errors**:
    - Check table exists and IAM permissions
    - Verify region configuration
    - Check read/write capacity
4. **Service Unavailable**:
    - Verify service URLs in environment variables
    - Check service health endpoints
    - Review retry logic (3 attempts with backoff)

Debug Commands:

```
# Check service health
curl http://localhost:8000/health | jq
```

```
# View active connections (debug mode)
curl http://localhost:8000/debug/connections | jq

# Check logs
docker logs orchestrator-service --tail 100

# Test WebSocket connection
python tests/integration/test_websocket_flow.py
```