Engage Agentic AI Design: Next 4 Steps Implementation Plan

Executive Summary

This document outlines the specific design and implementation tasks for the next 4 critical steps to enhance BrightMove's existing agentic AI foundation. Based on the current state analysis, we have a solid foundation with AWS Bedrock integration, basic AI evaluation endpoints, and frontend integration. The next steps focus on adding LangChain orchestration, implementing the "Wiz" agent persona, expanding agentic capabilities, and integrating Twilio for messaging.

Current State Assessment

✅ Already Implemented

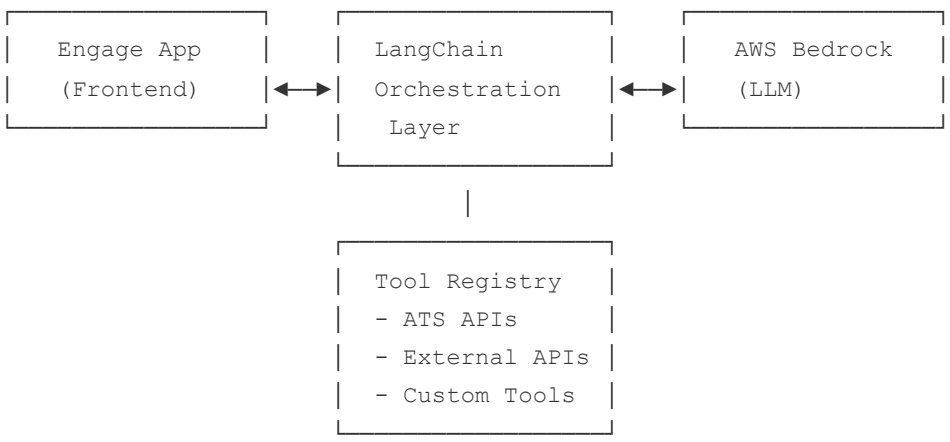
- AWS Bedrock integration in `AiAgenticService`
- Basic AI evaluation endpoints (`/agent/recruiter/evaluate`)
- Generative AI endpoints (`/ai/job-description`, `/ai/email`)
- Frontend integration in Engage app
- MongoDB audit logging
- Server-Sent Events (SSE) for streaming responses

🔄 Next 4 Steps to Implement

1. **LangChain Integration & Orchestration Layer**
2. **Wiz Agent Persona Implementation**
3. **Twilio Integration for Messaging**
4. **Advanced Agentic Capabilities Expansion**

Step 1: LangChain Integration & Orchestration Layer

Architecture Overview



Implementation Tasks

1.1 Create LangChain Service Layer

`apps/brightmove-ats/brightmove-common/src/main/java/com/bm/ats/ai/LangChainOrchestrationService.java`

```
@Service @Slf4j public class LangChainOrchestrationService { private final BedrockRuntimeClient bedrockClient; private final ToolRegistry toolRegistry; private final AgentMemoryService memoryService; public AgentResponse executeAgentWorkflow( UserModel user,
```

```
AgentWorkflowRequest request, List tools ) { // 1. Initialize
LangChain agent with tools // 2. Set up conversation memory // 3.
Execute workflow with Bedrock // 4. Return structured response }
public StreamingAgentResponse executeStreamingWorkflow( UserModel
user, AgentWorkflowRequest request, SseEmitter emitter ) { //
Streaming version for real-time responses } }
```

1.2 Create Tool Registry

apps/brightmove-ats/brightmove-common/src/main/java/com/bm/ats/ai/tools/AgentToolRegistry.java

```
@Component public class AgentToolRegistry { private final Map tools =
new ConcurrentHashMap<>(); @PostConstruct public void
registerDefaultTools() { registerTool(new ATSDataTool());
registerTool(new EmailCompositionTool()); registerTool(new
CandidateSearchTool()); registerTool(new JobDescriptionTool());
registerTool(new CalendarSchedulingTool()); } public List
getToolsForWorkflow(String workflowType) { // Return appropriate
tools based on workflow } }
```

LangChain-Specific Instructions

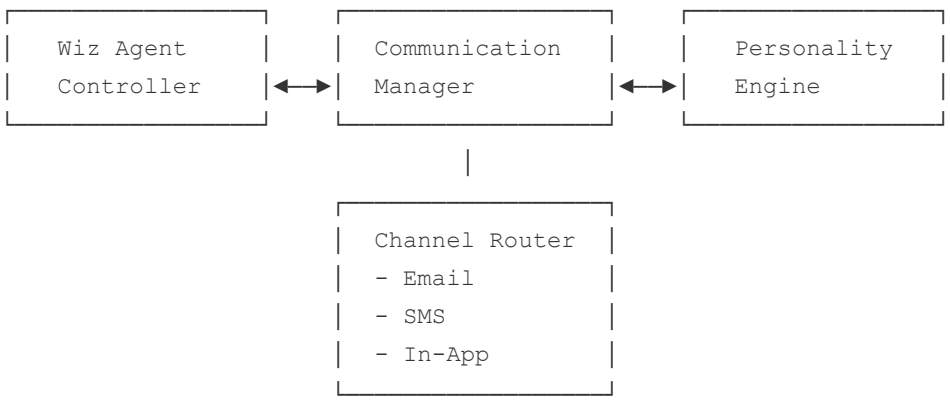
> ### 1. Install LangChain4j Dependencies
>
> ```
> <!-- Add to build.gradle --> implementation
> 'dev.langchain4j:langchain4j:0.27.1' implementation
> 'dev.langchain4j:langchain4j-bedrock:0.27.1' implementation
> 'dev.langchain4j:langchain4j-memory:0.27.1'
> ```

> ### 2. Configure LangChain4j with Bedrock

```
@Configuration public class LangChainConfig { @Bean public
BedrockChatModel bedrockChatModel(BedrockRuntimeClient client) {
return BedrockChatModel.builder() .client(client)
.model("anthropic.claude-3-sonnet-20240229-v1:0") .build(); }
@Bean public AgentMemoryService memoryService() { return new
InMemoryAgentMemoryService(); } }
```

Step 2: Wiz Agent Persona Implementation

Architecture Overview



Implementation Tasks

2.1 Create Wiz Agent Service

`apps/brightmove-ats/brightmove-common/src/main/java/com/bm/ats/ai/wiz/WizAgentService.java`

```
@Service @Slf4j public class WizAgentService { private final
LangChainOrchestrationService orchestrationService; private final
CommunicationManager communicationManager; private final
PersonalityEngine personalityEngine; private final ChannelRouter
channelRouter; public WizResponse handleCommunication( UserModel
user, CommunicationRequest request ) { // 1. Analyze communication
```

```
context // 2. Apply Wiz personality // 3. Route to appropriate channel // 4. Execute communication workflow } public WizResponse manageConversation( UserModel user, ConversationRequest request ) { // Handle ongoing conversation management } }
```

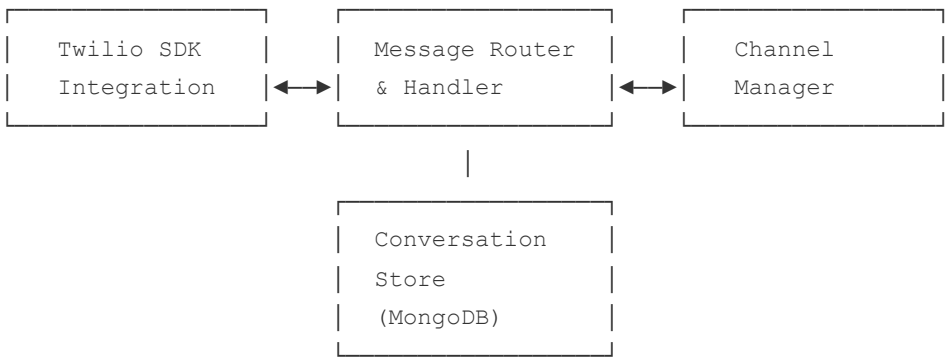
LangChain-Specific Instructions for Wiz Agent

1. Create Wiz Agent Chain

```
@Component public class WizAgentChain { private final ChatLanguageModel model; private final List wizTools; public WizAgentChain(BedrockChatModel model, AgentToolRegistry toolRegistry) { this.model = model; this.wizTools = toolRegistry.getToolsForWorkflow("wiz_communication"); } public String executeWizWorkflow(String input, UserModel user) { // Create LangChain4j agent with Wiz-specific tools Agent agent = Agent.builder() .chatLanguageModel(model) .tools(wizTools) .memory(new InMemoryChatMemory()) .build(); return agent.execute(input); } }
```

Step 3: Twilio Integration for Messaging

Architecture Overview



Implementation Tasks

3.1 Create Twilio Service

apps/brightmove-ats/brightmove-common/src/main/java/com/bm/ats/messaging/TwilioService.java

```
@Service @Slf4j public class TwilioService { private final TwilioClient twilioClient; private final ConversationStore conversationStore; private final MessageRouter messageRouter; public MessageResponse sendMessage( UserModel user, MessageRequest request ) { // 1. Validate message request // 2. Route to appropriate Twilio service // 3. Store conversation state // 4. Return response } public void handleIncomingMessage( TwilioWebhookRequest webhook ) { // Handle incoming SMS/WhatsApp messages } }
```

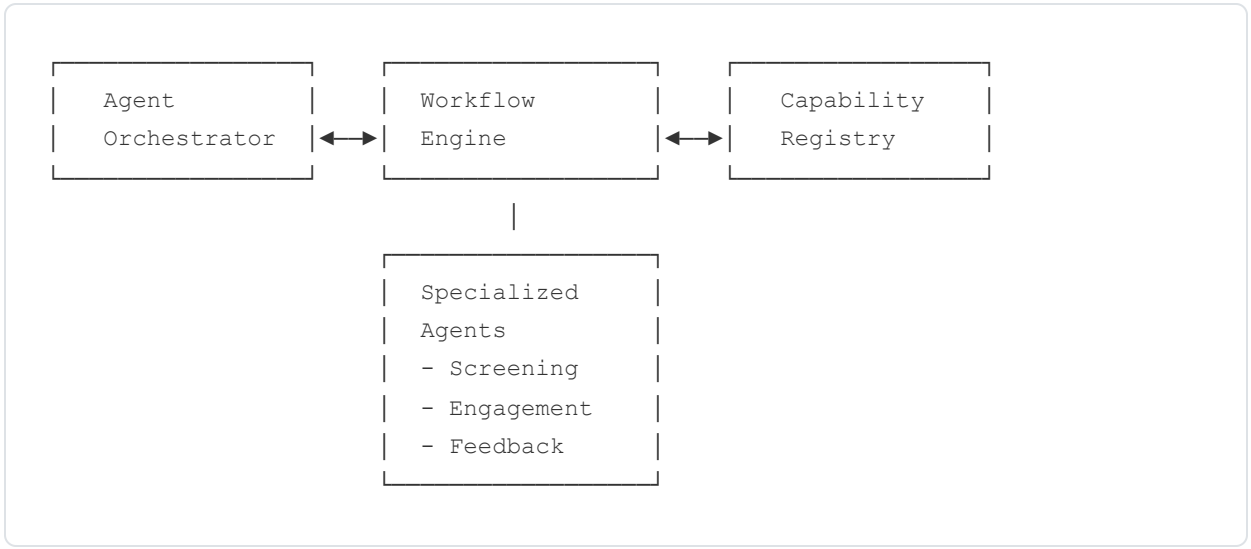
LangChain-Specific Instructions for Twilio Integration

1. Create Twilio Tool for LangChain

```
@Component public class TwilioMessagingTool implements AgentTool { private final TwilioService twilioService; @Override public String getName() { return "twilio_messaging"; } @Override public ToolResult execute(Map parameters, UserModel user) { String phoneNumber = (String) parameters.get("phone_number"); String message = (String) parameters.get("message"); MessageRequest request = MessageRequest.builder() .to(phoneNumber) .body(message) .build(); MessageResponse response = twilioService.sendMessage(user, request); return ToolResult.success(response); } }
```

Step 4: Advanced Agentic Capabilities Expansion

Architecture Overview



Implementation Tasks

4.1 Create Agent Orchestrator

apps/brightmove-ats/brightmove-common/src/main/java/com/bm/ats/ai/orchestration/AgentOrchestrator.java

```
@Service @Slf4j public class AgentOrchestrator { private final Map agents; private final WorkflowEngine workflowEngine; private final CapabilityRegistry capabilityRegistry; public AgentResponse orchestrateWorkflow( UserModel user, WorkflowRequest request ) { // 1. Analyze workflow requirements // 2. Select appropriate agents // 3. Execute workflow with LangChain // 4. Return coordinated response } }
```

LangChain-Specific Instructions for Advanced Capabilities

> ### 1. Create Multi-Agent LangChain Setup
>
> ```
> @Component public class MultiAgentOrchestrator { private final Map agents; private final BedrockChatModel model; public MultiAgentOrchestrator(BedrockChatModel model, List
> ```

```
specializedAgents) { this.model = model; this.agents =
createAgents(specializedAgents); } private Map createAgents(List
specializedAgents) { Map agentMap = new HashMap<>(); for
(SpecializedAgent specializedAgent : specializedAgents) { Agent
agent = Agent.builder() .chatLanguageModel(model)
.tools(specializedAgent.getSpecializedTools()) .memory(new
InMemoryChatMemory()) .build();
agentMap.put(specializedAgent.getAgentType(), agent); } return
agentMap; } public String executeMultiAgentWorkflow(String input,
List agentTypes, UserModel user) { // Execute workflow across
multiple agents String result = input; for (String agentType :
agentTypes) { Agent agent = agents.get(agentType); result =
agent.execute(result); } return result; } }
```

Deployment Instructions

1. Database Schema Updates

```
-- Create new tables for agentic features CREATE TABLE
agent_conversations ( id VARCHAR(36) PRIMARY KEY, user_id VARCHAR(36)
NOT NULL, conversation_sid VARCHAR(255), channel_type VARCHAR(50),
status VARCHAR(50), created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ); CREATE TABLE
agent_workflows ( id VARCHAR(36) PRIMARY KEY, workflow_type
VARCHAR(100) NOT NULL, user_id VARCHAR(36) NOT NULL, status
VARCHAR(50), result JSON, created_at TIMESTAMP DEFAULT
CURRENT_TIMESTAMP );
```

2. Configuration Updates

apps/brightmove-ats/brightmove-web/src/main/resources/application.yml

```
langchain: bedrock: model: anthropic.claude-3-sonnet-20240229-v1:0
max-tokens: 4096 temperature: 0.7 twilio: account-sid:
${TWILIO_ACCOUNT_SID} auth-token: ${TWILIO_AUTH_TOKEN} phone-number:
```

```
${TWILIO_PHONE_NUMBER} agent: wiz: personality-config: classpath:config/wiz-personality.json memory-retention-days: 30 workflow: max-concurrent: 10 timeout-seconds: 300
```

3. New REST Endpoints

apps/brightmove-ats/brightmove-web/src/main/java/com/bm/ats/controller/agent/AdvancedAgentController.java

```
@RestController @RequestMapping("/agent/v2") @Slf4j public class AdvancedAgentController { private final AgentOrchestrator agentOrchestrator; private final WizAgentService wizAgentService; private final TwilioService twilioService; @PostMapping("/workflow") public AgentResponse executeWorkflow( @RequestHeader(value = ApiHeaders.USER_API_KEY) String userApiKey, @RequestBody WorkflowRequest request ) { UserModel user = userService.getUserByApiKey(userApiKey); return agentOrchestrator.orchestrateWorkflow(user, request); } @PostMapping("/wiz/communicate") public WizResponse communicate( @RequestHeader(value = ApiHeaders.USER_API_KEY) String userApiKey, @RequestBody CommunicationRequest request ) { UserModel user = userService.getUserByApiKey(userApiKey); return wizAgentService.handleCommunication(user, request); } @PostMapping("/twilio/webhook") public void handleTwilioWebhook(@RequestBody TwilioWebhookRequest webhook) { twilioService.handleIncomingMessage(webhook); } }
```

4. Frontend Integration Updates

apps/engage-app/src/components/WizAgent.tsx

```
interface WizAgentProps { conversationId?: string; channelType: 'email' | 'sms' | 'in-app'; } export const WizAgent: React.FC = ({ conversationId, channelType }) => { const [messages, setMessages] = useState([]); const [isTyping, setIsTyping] = useState(false); const sendMessage = async (content: string) => { setIsTyping(true); try { const response = await api.post('/agent/v2/wiz/communicate', { content, channelType, conversationId }); setMessages(prev =>
```

```
[...prev, response.data]); } catch (error) { console.error('Error
sending message:', error); } finally { setIsTyping(false); } };
return (
); };
```

Testing Strategy

Unit Tests

- Test each specialized agent independently
- Mock LangChain responses
- Test tool execution

Integration Tests

- Test agent orchestration
- Test Twilio integration
- Test conversation flow

End-to-End Tests

- Test complete workflow from frontend to backend
- Test multi-agent scenarios
- Test error handling and recovery

Monitoring & Observability

1. Metrics to Track

- Agent response times
- Workflow success rates
- Twilio message delivery rates
- User engagement metrics

2. Logging Strategy

- Structured logging for all agent interactions
- Audit trail for AI decisions
- Performance monitoring

3. Alerting

- Agent failure alerts
- High latency alerts
- Twilio delivery failure alerts

Risk Mitigation

Technical Risks

- **LangChain Version Compatibility**: Pin specific versions and test thoroughly
- **Bedrock Rate Limits**: Implement retry logic and circuit breakers
- **Twilio Costs**: Monitor usage and implement rate limiting

Business Risks

- **AI Response Quality**: Implement human-in-the-loop for critical decisions
- **Data Privacy**: Ensure all AI interactions are logged and auditable
- **User Adoption**: Provide clear value proposition and training

Success Metrics

Technical Metrics

Agent response time < 2 seconds

Business Metrics

User engagement increase > 20%

Workflow success rate > 95%

System uptime > 99.9%

Communication efficiency improvement > 30%

Customer satisfaction score > 4.5/5

Timeline

Week 1-2	**LangChain Integration** - Set up LangChain4j dependencies - Create orchestration service - Implement tool registry
Week 3-4	**Wiz Agent Implementation** - Create Wiz agent service - Implement personality engine - Add communication manager
Week 5-6	**Twilio Integration** - Set up Twilio SDK - Create message router - Implement conversation store
Week 7-8	**Advanced Capabilities** - Create agent orchestrator - Implement specialized agents - Add workflow templates
Week 9-10	**Testing & Deployment** - Comprehensive testing - Performance optimization - Production deployment

Appendix: Analysis Process

Documents Analyzed

- Current ATS source code structure
- Existing AI service implementations
- Engage app frontend architecture
- BrightMove technical stack documentation

Key Decisions Made

1. **LangChain4j over Python LangChain**: Better integration with existing Java Spring stack
2. **Modular Agent Architecture**: Allows for independent development and testing
3. **Twilio as Primary Messaging Platform**: Leverages existing Twilio SDK integration
4. **MongoDB for Conversation Storage**: Consistent with existing audit logging approach

Assumptions

- AWS Bedrock will remain the primary LLM provider
- Existing ATS data models will be sufficient for agentic features
- Twilio pricing model will remain cost-effective for the use case
- LangChain4j will provide the necessary orchestration capabilities

Gaps Identified

- Need for comprehensive testing framework for AI agents
- Requirement for human-in-the-loop validation system
- Need for advanced monitoring and observability tools
- Requirement for AI bias detection and mitigation tools