AIMA ChatBot - Comprehensive Documentation

This document provides an in-depth explanation of every process, component, and feature in the AIMA ChatBot system.

Table of Contents

Architecture Overview

Three-Layer Hybrid System

AIMA ChatBot uses a **three-layer architecture** designed for efficiency, intelligence, and personalization:

Layer 1: Local Pattern Matching (Reflex Layer)

Purpose: Provide instant, zero-latency responses to common queries.

How it works:

1. User input is normalized (lowercased, trimmed)
2. Input is compared against patterns in `local/patterns.json`
3. Three matching strategies are used in order:
   - **Exact Match**: Direct string comparison
   - **Regex Match**: Pattern-based matching (e.g., `\b(hello|hi|hey)\b`)
   - **Fuzzy Match**: Uses Levenshtein distance via `fuzzywuzzy` library (threshold: 80%)
4. If match confidence ≥ 70% (configurable), response is returned immediately
5. **Multi-Intent Enhancement**: If input contains multiple sentences (split on `.`, `?`, `!`, `;`), each segment is matched separately and responses are combined

Benefits:

- Zero API cost
- Works offline
- Instant response (<1ms)
- Configurable priority

Layer 2: Cloud AI (Reasoning Layer)

Purpose: Handle complex queries requiring intelligence, reasoning, or knowledge.

How it works:

1. If no local match found, request is forwarded to Google Gemini API
2. **Context Assembly**:
    - System instruction (defines bot persona)
    - Recent conversation history (last 10 exchanges by default)
    - User profile facts (injected as "System Note")
3. API call to `gemini-2.5-flash` model
4. Response extraction and validation
5. **Smart Caching**: Only successful (non-error) responses are cached
6. **Auto-Learning**: If enabled, successful responses are saved as new patterns

Benefits:

- Access to LLM capabilities
- Contextual awareness
- Continuously improving (auto-learning)

Layer 3: Long-Term Memory (Persistence Layer)

Purpose: Remember user-specific information across sessions.

How it works:

1. During initialization, user identity is determined:
    - Default: OS username (via `os.getlogin()`)
    - Override: `--user` CLI argument
2. User profile loaded from `data/users/{username}.json`
3. During conversation:
    - **Fact Extraction**: Regex patterns detect user facts (e.g., "My name is X")
    - Facts are saved to profile immediately
4. AI context injection: Profile facts are prepended to every API request
5. Profile is persistent across sessions (survives restarts)

Benefits:

- Personalized responses
- Multi-user support
- Privacy (each user has separate profile)

Core Components

1. `main.py` - The CLI Orchestrator

Role: Entry point and user interaction handler.

Key Functions:

`ChatbotCLI.__init__(user_override)`

- Instantiates `ChatbotConfig`
- Creates `HybridChatbot` instance with optional user override
- Initializes session state

`setup()`

Process:

1. Checks for `GEMINI_API_KEY` environment variable
2. If not found, prompts user for manual input
3. Calls `chatbot.initialize(api_key)`
4. Displays setup status and help tips

`run()` **- Main Event Loop**

Process:

```
while running:
    1. Display prompt: "You: "
    2. Capture user input
    3. Check for commands (quit, help, stats, etc.)
    4. If normal input:
       - Show typing indicator (optional)
       - Call chatbot.process_input()
       - Display formatted response
    5. Handle KeyboardInterrupt (Ctrl+C)
```

Command Handlers

- `help`: Displays command list and usage guide
- `stats`: Shows session metrics (queries, cache hits, local vs AI breakdown)
- `config`: Displays current configuration values
- `clear`: Clears conversation history
- `train`: Enters interactive training mode to teach new patterns
- `autolearn [on/off]`: Toggles automatic pattern learning

---

2. `core/chatbot.py` - The Brain

Role: Central orchestrator for all bot logic.

`HybridChatbot.__init__(config, user_override)`

Initialization sequence:

1. Stores config reference
2. Creates `InputParser` instance
3. Creates `IntentSplitter` instance
4. Creates `UserManager` with identity resolution
5. Creates `PatternMatcher` with patterns file path
6. Creates `GeminiClient` (uninitialized)
7. Creates `ChatbotLogger` with configured log level
8. Creates `ResponseCache` with TTL settings
9. Initializes conversation history list
10. Sets session start timestamp
11. Initializes statistics counters

`initialize(api_key)` - Startup Sequence

Process:

1. **Config Validation**:
   - Checks threshold values (0-1 range)
   - Validates temperature (0-2 range)
   - Ensures positive integers for history/cache limits
2. **Gemini Initialization** (if API key provided):
   - Attempts to create `genai.Client(api_key)`
   - On failure: logs warning, continues if `graceful_degradation=True`
3. **History Loading**:
   - If `save_conversations=True` and `clear_history_on_restart=False`
   - Loads conversation history from `data/conversation_history.json`
4. Logs success message with username

`process_input(user_input)` - Main Processing Pipeline

Complete Flow:

```
1. INPUT VALIDATION
   ├─ Check minimum length (default: 1 char)
   ├─ Check maximum length (default: 1000 chars)
   ├─ Sanitize for XSS patterns (if enabled)
   └─ Return error message if invalid

2. INPUT PARSING
   ├─ Normalize text (lowercase, trim)
   ├─ Tokenize (split into words)
   └─ Create ParsedInput object

3. CACHE CHECK
   ├─ If caching enabled
   ├─ Lookup normalized text in cache
   ├─ Check TTL expiration
   └─ Return cached response if valid
```

```
4. MULTI-INTENT HANDLING
   ├─ Split input on punctuation: . ? ! ;
   ├─ If 2+ segments found:
   │  ├─ For each segment:
   │  │  ├─ Parse segment
   │  │  ├─ Attempt pattern match
   │  │  └─ Collect response
   │  ├─ If ALL segments matched locally:
   │  │  ├─ Combine responses with spaces
   │  │  ├─ Tag as [LOCAL:multi]
   │  │  └─ RETURN early
   │  └─ Otherwise: continue to full match

5. LOCAL PATTERN MATCHING (Full String)
   ├─ Query PatternMatcher with full parsed input
   ├─ Check confidence threshold (default: 0.7)
   ├─ If matched:
   │  ├─ Cache response
   │  ├─ Add to history
   │  └─ RETURN as [LOCAL:regex/exact/fuzzy]

6. AI FALLBACK
   ├─ If fallback_to_ai=True AND client initialized:
   │  ├─ Get recent context (last N exchanges)
   │  ├─ Get user profile context
   │  ├─ Combine: [history..., "System Note: {user facts}"]
   │  ├─ Call gemini_client.generate_response()
   │  ├─ Validate response for errors
   │  ├─ Cache if not error
   │  ├─ Auto-learn pattern (if enabled + valid)
   │  ├─ Extract user facts via regex
   │  ├─ Add to history
   │  └─ RETURN as [GEMINI]

7. NO MATCH FALLBACK
   └─ Return default_error_response
```

Error Handling:

- All exceptions caught at top level
- Logged with full traceback
- Error counter incremented
- Returns either error details (if `verbose_errors=True`) or generic message

---

3. `core/intent_splitter.py` - The Segmenter

Role: Splits compound inputs into logical segments.

`IntentSplitter.split(text)`

Algorithm:

```
Pattern: r'[.?!;]+'  # One or more sentence terminators

Process:
1. re.split(pattern, text)
   → ["Hello", "", "How are you", ""]
2. Filter empty strings
   → ["Hello", "How are you"]
3. Strip whitespace from each
4. Return list
```

Examples:

- `"Hello! Thanks."` → `["Hello", "Thanks"]`
- `"Hi. How are you? What's up?"` → `["Hi", "How are you", "What's up"]`
- `"Simple text"` → `["Simple text"]`

---

4. `utils/math_solver.py` - The Calculator

Role: Safely evaluates arithmetic expressions without API calls.

`MathSolver.is_math_expression(text)` - Detection

Algorithm:

```
1. Clean input:
   - Replace ^ with **
   - Replace x or × with *
   - Replace ÷ with /

2. Regex check: r'^[\d\s\+\-\*/\(\)\^\%\.]+$'
   Must contain ONLY: numbers, operators, spaces, parentheses

3. Validation:
   - Must have at least one number
   - Must have at least one operator

4. Return True if valid math expression
```

Examples:

- `"2 + 2"` → True
- `"10 * 5 + 3"` → True
- `"hello world"` → False
- `"x = 5"` → False (variables not allowed)

`MathSolver.solve(expression)` - Evaluation

Process:

```
1. Normalize expression:
   cleaned = "2 + 2"

2. Parse with AST (Abstract Syntax Tree):
   tree = ast.parse(cleaned, mode='eval')
   # Creates safe parse tree: BinOp(Add, Num(2), Num(2))

3. Recursively evaluate tree:
   - If Num node: return number
   - If BinOp node:
       left = eval(left_node)
       right = eval(right_node)
       return OPERATOR[op_type](left, right)
   - If UnaryOp: handle +/- signs

4. Format result:
   - Strip trailing zeros from decimals
   - Return (numeric_value, formatted_string)

Example:
   solve("2 + 2") → (4, "4")
   solve("10 / 3") → (3.333333, "3.333333")
```

Supported Operators:

```
ast.Add: operator.add        # +
ast.Sub: operator.sub        # -
ast.Mult: operator.mul       # *
ast.Div: operator.truediv    # /
ast.Pow: operator.pow        # ** or ^
ast.Mod: operator.mod        # %
ast.FloorDiv: operator.floordiv  # //
```

Safety Features:

- **No code execution**: Uses AST parsing only
- **No function calls**: Only arithmetic operators allowed
- **No variables**: Only numbers and operators
- **Exception handling**: Returns None on invalid input
- **Protected against**: Division by zero, syntax errors, type errors

Example Evaluations:

```
"2 + 2"           → 4
"10 * 5 + 3"      → 53
"100 / 4"         → 25
```

```
"2^8"             → 256
"(5 + 3) * 2"     → 16
"10 % 3"          → 1
"2 ** 10"         → 1024
"sqrt(16)"        → None (functions not supported)
```

---

5. `core/user_manager.py` - The Memory Manager

Role: Manages user identity and persistent facts.

`UserManager.__init__(base_dir, user_override)`

Identity Resolution Process:

```
IF user_override provided:
    username = user_override
ELSE:
    TRY:
        username = os.getlogin()  # Windows: gets logged-in user
    EXCEPT:
        username = "default_user"  # Fallback for edge cases
```

Profile File Structure:

```
{
  "username": "muaza",
  "created_at": "2026-01-15T12:00:00",
  "facts": {
    "name": "Muaz",
    "favorite_lang": "Python",
    "age": "25"
  },
  "preferences": {}
}
```

`set_fact(key, value)` - Fact Storage

Process:

1. Update `self.profile["facts"][key] = value`
2. Call `save_profile()` immediately (auto-save)
3. Log fact storage

`get_context_string()` - Context Generation

Output Format:

```
User Profile (muaza):
- name: Muaz
- favorite_lang: Python
```

Purpose: This string is injected into AI prompts to provide personalization.

---

5. `ai/gemini_client.py` - The API Connector

Role: Interface with Google Gemini API.

`initialize(api_key)` - Client Setup

Process:

```
try:
    self.client = genai.Client(api_key=api_key)
    self.initialized = True
except ImportError:
    # google-genai not installed
    return False
except Exception as e:
    # Invalid API key or network error
    log error
    return False
```

`generate_response(prompt, context, temperature)` - API Call

Complete Process:

1. Pre-flight Checks:

```
if not self.initialized:
    return "AI service not initialized"

if not _check_rate_limit():
    return "Rate limit exceeded. Please wait."
```

2. Prompt Construction:

```
System: {self.config.system_instruction}

Context:
{context[0]}
{context[1]}
```

```
...
{context[-1]}
System Note: {user_profile}

User: {prompt}
```

3. API Request:

```
response = self.client.models.generate_content(
    model=self.config.gemini_model,  # "gemini-2.5-flash"
    contents=full_prompt
)
```

4. Response Extraction:

```
text = self._extract_text(response)
```

`_extract_text(resp)` **- Robust Extraction**: Handles multiple response formats:

- Direct string: `return resp`
- Has `.text` attribute: `return resp.text`
- Has `.outputs` list: Extract from `outputs[0].content[0].text`
- Dictionary: Navigate nested structure
- Fallback: `str(resp)`

5. Error Handling:

```
Checks error message for keywords:
- "429" or "resource exhausted" → "API Limit" message
- "503" or "unavailable" → "Server Busy" message
- "404" → "Model not found" message
Default: Returns config.default_error_response
```

`_check_rate_limit()` **- Rate Limiting**

Algorithm:

```
current_time = time.time()

if current_time - last_request_time >= 60:
    # Window expired, reset
    request_count = 0
    last_request_time = current_time
```

```
if request_count >= max_requests_per_minute:
    return False  # Blocked

request_count += 1
return True
```

---

6. `local/pattern_matcher.py` - The Pattern Engine

Role: Match user input against local patterns and knowledge base.

`PatternMatcher.match(parsed_input)` - Matching Process

Complete Algorithm:

Step 1: Cache Check

```
if text in match_cache:
    return cached_result  # Instant return
```

Step 2: Standard Pattern Matching

```
for category, data in patterns.items():
    for pattern in data["patterns"]:

        # Regex Match
        if _is_regex_pattern(pattern):
            if re.search(pattern, text, re.IGNORECASE):
                response = random.choice(data["responses"])
                confidence = 1.0
                match_type = "regex"
                CACHE and RETURN

        # Exact Match
        else:
            if pattern.lower() in text:
                response = random.choice(data["responses"])
                confidence = 1.0
                match_type = "exact"
                CACHE and RETURN
```

Step 3: Knowledge Base Search

```
for entry in knowledge_base:
    # Tag Matching (Priority)
    for tag in entry["tags"]:
        score = fuzz.ratio(tag.lower(), text)
```

```
            if score >= 85:
                RETURN entry["content"] immediately

        # Content Matching
        score = fuzz.token_set_ratio(entry["content"].lower(), text)
        if score > best_score:
            best_score = score
            best_content = entry["content"]

    if best_score >= 85:
        RETURN best_content
```

Step 4: Fuzzy Matching

```
if use_fuzzy_matching:
    for category, data in patterns.items():
        for pattern in data["patterns"]:
            if not is_regex(pattern):
                score = fuzz.partial_ratio(text, pattern.lower())
                if score >= fuzzy_threshold (default: 80):
                    track best match

    if best_match found:
        confidence = score / 100.0
        match_type = "fuzzy"
        RETURN
```

Step 5: No Match

```
return MatchResult(matched=False, ...)
```

---

7. `utils/cache.py` - The Response Cache

Role: LRU cache with TTL expiration.

`ResponseCache` - Structure

```
cache = OrderedDict({
    "hello": {
        "value": "Hi there!",
        "timestamp": 1705319400.0
    },
    "what time": {
        "value": "Current time is...",
        "timestamp": 1705319450.0
```

```
    }
})
```

`get(key)` **- Cache Retrieval**

Process:

```
1. Check if key exists
2. If exists:
   a. Check TTL: current_time - timestamp > cache_ttl_seconds
   b. If expired: DELETE entry, return None
   c. If valid:
      - Move to end (LRU update)
      - Return value
3. Return None if not found
```

`set(key, value)` **- Cache Storage**

Process with Thread Safety:

```
with self.lock:  # Acquire lock
    1. If cache at capacity (>= cache_max_size):
       - Remove oldest: popitem(last=False)

    2. Store new entry:
       cache[key] = {
           "value": value,
           "timestamp": current_time
       }
# Lock released
```

Data Flow & Processing

Complete Request Lifecycle

Example: User types "Hello! My name is Muaz."

```
┌─────────────────────────────────────────────────────────┐
│ USER INPUT: "Hello! My name is Muaz."                   │
└─────────────────────────────────────────────────────────┘
                          ↓
┌─────────────────────────────────────────────────────────┐
│ main.py: ChatbotCLI.run()                               │
│ - Captures input from CLI                               │
│ - Checks for commands (none)                            │
```

```
└──────────────────────────────────────────────────────────┘
                          ↓
┌──────────────────────────────────────────────────────────┐
│ core/chatbot.py: process_input()                         │
│                                                          │
│ Step 1: INPUT VALIDATION                                 │
│ ✓ Length OK (1-1000 chars)                               │
│ ✓ No XSS patterns                                        │
│                                                          │
│ Step 2: PARSING                                          │
│ → InputParser.parse()                                    │
│   - Normalized: "hello! my name is muaz."                │
│   - Tokens: ["hello", "my", "name", "is", "muaz"]        │
│                                                          │
│ Step 3: CACHE CHECK                                      │
│ → ResponseCache.get("hello! my name is muaz.")           │
│   Result: None (not cached)                              │
└──────────────────────────────────────────────────────────┘
                          ↓
┌──────────────────────────────────────────────────────────┐
│ Step 4: MULTI-INTENT SPLIT                               │
│ → IntentSplitter.split()                                 │
│   Input: "Hello! My name is Muaz."                       │
│   Segments: ["Hello", "My name is Muaz"]                 │
│                                                          │
│ → Try matching EACH segment:                             │
│   Segment 1: "Hello"                                     │
│   ├─ PatternMatcher.match("hello")                       │
│   └─ ✓ Match: "greeting" → "Hi there!"                   │
│                                                          │
│   Segment 2: "My name is Muaz"                           │
│   ├─ PatternMatcher.match("my name is muaz")             │
│   └─ ✗ No local match                                    │
│                                                          │
│ Result: Not ALL segments matched → Continue              │
└──────────────────────────────────────────────────────────┘
                          ↓
┌──────────────────────────────────────────────────────────┐
│ Step 5: FULL STRING PATTERN MATCH                        │
│ → PatternMatcher.match("hello! my name is muaz.")        │
│                                                          │
│ Checking patterns:                                       │
│ - Greetings: r"\b(hello|hi|hey)\b" → Partial match       │
│   (only "hello" part matches, not full string)           │
│                                                          │
│ Result: No high-confidence match (< 70% threshold)       │
└──────────────────────────────────────────────────────────┘
                          ↓
┌──────────────────────────────────────────────────────────┐
│ Step 6: AI FALLBACK                                      │
│ → GeminiClient.generate_response()                       │
│                                                          │
│ 6a. RATE LIMIT CHECK                                     │
│ ✓ Request count OK                                       │
```

```
│                                                          │
│ 6b. CONTEXT ASSEMBLY                                     │
│ - Recent history (last 10 exchanges)                     │
│ - User profile context:                                  │
│   "System Note: User Profile (muaza):"                   │
│   (empty - first interaction)                            │
│                                                          │
│ 6c. PROMPT CONSTRUCTION                                  │
│ System: You are a helpful CLI assistant...               │
│                                                          │
│ Context:                                                 │
│ [Previous exchanges if any]                              │
│                                                          │
│ User: Hello! My name is Muaz.                            │
│                                                          │
│ 6d. API CALL                                             │
│ POST https://generativelanguage.googleapis.com/...       │
│ Model: gemini-2.5-flash                                  │
│                                                          │
│ 6e. RESPONSE                                             │
│ API Returns: "Hello Muaz! Nice to meet you. How can..."  │
│                                                          │
│ 6f. ERROR CHECK                                          │
│ ✓ No error keywords detected                             │
│                                                          │
│ 6g. CACHE STORAGE                                        │
│ → ResponseCache.set(key, "Hello Muaz! Nice to...")       │
└──────────────────────────────────────────────────────────┘
                          ↓
┌──────────────────────────────────────────────────────────┐
│ Step 7: AUTO-LEARNING (if enabled)                       │
│ Error keywords check: PASS                               │
│ → chatbot.learn_pattern()                                │
│   Pattern: "hello! my name is muaz."                     │
│   Response: "Hello Muaz! Nice to meet you..."            │
│   Saved to: patterns.json                                │
└──────────────────────────────────────────────────────────┘
                          ↓
┌──────────────────────────────────────────────────────────┐
│ Step 8: USER FACT EXTRACTION                             │
│ → Regex: r"my name is\s+([a-zA-Z]+)"                     │
│   Match: "Muaz"                                          │
│                                                          │
│ → UserManager.set_fact("name", "Muaz")                   │
│   Saved to: data/users/muaza.json                        │
│   {                                                      │
│     "username": "muaza",                                 │
│     "facts": {"name": "Muaz"}                            │
│   }                                                      │
└──────────────────────────────────────────────────────────┘
                          ↓
┌──────────────────────────────────────────────────────────┐
│ Step 9: HISTORY LOGGING                                  │
│ → chatbot._add_to_history()                              │
```

```
│   Entry: {                                                  │
│     "timestamp": "2026-01-15T16:00:00",                    │
│     "user": "Hello! My name is Muaz.",                     │
│     "bot": "Hello Muaz! Nice to meet you...",              │
│     "source": "GEMINI"                                      │
│   }                                                         │
│ → _save_history() if persistence enabled                   │
└─────────────────────────────────────────────────────────────┘
                          ↓
┌─────────────────────────────────────────────────────────────┐
│ Step 10: RESPONSE FORMATTING                               │
│ → _format_response()                                       │
│   if show_response_source:                                 │
│      "[GEMINI] Hello Muaz! Nice to meet you..."            │
│   else:                                                     │
│      "Hello Muaz! Nice to meet you..."                     │
└─────────────────────────────────────────────────────────────┘
                          ↓
┌─────────────────────────────────────────────────────────────┐
│ main.py: Display Response                                  │
│ OUTPUT: "Bot: [GEMINI] Hello Muaz! Nice to meet you..."    │
└─────────────────────────────────────────────────────────────┘
```

Feature Deep Dive

Multi-Intent Handling

Detailed Example:

Input: `"Hi! Thanks for your help. Bye!"`

Processing:

```
1. Split on [.!?;]:
   → ["Hi", "Thanks for your help", "Bye"]

2. Match each segment:

   Segment 1: "Hi"
   ├─ Check patterns.json
   ├─ Match: greetings pattern r"\b(hi|hello|hey)\b"
   ├─ Response: "Hello! How can I help you?"
   └─ Confidence: 1.0

   Segment 2: "Thanks for your help"
   ├─ Check patterns.json
   ├─ Match: gratitude pattern r"\b(thank|thanks)\b"
   ├─ Response: "You're welcome!"
   └─ Confidence: 1.0
```

```
   Segment 3: "Bye"
   ├─ Check patterns.json
   ├─ Match: farewell pattern r"\b(bye|goodbye)\b"
   ├─ Response: "Take care! See you later."
   └─ Confidence: 1.0

3. All segments matched!
   → Combine responses:
   "Hello! How can I help you? You're welcome! Take care! See you later."

4. Return as [LOCAL:multi]
```

When Multi-Intent Fails: If ANY segment doesn't match locally, the entire input falls back to single-string matching or AI.

Long-Term Memory System

Architecture:

```
data/
└── users/
    ├── muaza.json          ← Your profile
    ├── alice.json          ← Alice's profile
    └── default_user.json   ← Fallback profile
```

Fact Extraction Logic:

Current Regex Patterns:

```
# Name extraction
pattern = r"my name is\s+([a-zA-Z]+)"
example = "My name is Muaz" → extracts "Muaz"

# Future patterns (can be added):
# Age: r"i am\s+(\d+)\s+years old"
# Location: r"i live in\s+([\w\s]+)"
# Occupation: r"i am a\s+([\w\s]+)"
```

Context Injection Example:

Profile State:

```
{
  "facts": {
    "name": "Muaz",
    "favorite_lang": "Python"
```

```
  }
}
```

AI Prompt (What Gemini Sees):

```
System: You are a helpful CLI assistant...

Context:
User: Hi
Bot: Hello!
System Note: User Profile (muaza):
- name: Muaz
- favorite_lang: Python

User: What's my favorite programming language?
```

AI Response: "Based on your profile, your favorite programming language is Python!"

Auto-Learning System

Complete Process:

Trigger: User asks "What time does the library close?"

Flow:

```
1. No local match → AI responds:
   "The library typically closes at 8 PM."

2. Error validation:
   Check for keywords: ["error", "sorry", "unable", ...]
   Result: PASS (no errors)

3. Length check:
   Response length > 5 characters
   Result: PASS

4. learn_pattern() called:

   a. Sanitize pattern:
      Input: "What time does the library close?"
      Escaped: "What\\ time\\ does\\ the\\ library\\ close\\?"

   b. Smart boundary generation:
      Start char: 'W' (word char) → add \b
      End char: '?' (non-word char) → no \b
      Final regex: r"\bWhat\ time\ does\ the\ library\ close\?"

   c. Generate unique category:
```

```
       ID: "learned_a3f8b219"

   d. Create pattern entry:
      {
        "learned_a3f8b219": {
          "patterns": [r"\bWhat\ time..."],
          "responses": ["The library typically closes at 8 PM."],
          "priority": 9
        }
      }

   e. Save to patterns.json

   f. Reload PatternMatcher

5. Next time user asks same question:
   → Instant local match (saves API call)
```

Configuration System

`config.py` - All Settings Explained

API Configuration:

```
gemini_api_key: Optional[str]
  # Source: Environment variable GEMINI_API_KEY
  # Fallback: None (prompts user at runtime)

gemini_model: str = "gemini-2.5-flash"
  # Available models: gemini-2.5-flash, gemini-2.0-flash-exp, etc.
  # Trade-off: flash (fast/cheap) vs pro (smart/expensive)

api_timeout: int = 30
  # Max seconds to wait for API response
  # Increase for slow networks

max_retries: int = 3
  # Number of retry attempts on network errors

retry_delay: float = 1.0
  # Seconds to wait between retries
```

Pattern Matching Configuration:

```
pattern_match_threshold: float = 0.7
  # Minimum confidence (0.0-1.0) to accept local match
  # Lower = more permissive, Higher = stricter
```

```
use_fuzzy_matching: bool = True
  # Enable Levenshtein distance matching
  # Handles typos: "helo" → "hello"

fuzzy_match_threshold: int = 80
  # Minimum fuzzy score (0-100)
  # 100 = exact match, 80 = allows ~20% difference

case_sensitive: bool = False
  # If True, "Hello" != "hello"
```

Response Configuration:

```
max_response_length: int = 2000
  # Truncate AI responses longer than this
  # Prevents terminal overflow

response_temperature: float = 0.7
  # AI creativity (0.0-2.0)
  # 0.0 = deterministic, 2.0 = highly creative

enable_local_priority: bool = True
  # If True: Check local patterns BEFORE AI
  # If False: Always use AI (expensive!)

fallback_to_ai: bool = True
  # If True: Use AI when no local match
  # If False: Return error message

enable_auto_learning: bool = True
  # Automatically save AI responses as patterns

knowledge_file: str = "local/knowledge_base.json"
  # Path to structured knowledge base

min_knowledge_score: int = 85
  # Minimum score for knowledge base matches

system_instruction: str = "..."
  # Defines AI persona and behavior
```

Conversation Management:

```
max_history_length: int = 50
  # Maximum conversation exchanges to remember
  # Oldest entries are dropped

context_window_size: int = 10
  # Number of recent exchanges sent to AI
```

```
  # Smaller = less context, faster/cheaper

enable_context: bool = True
  # If False, AI has no memory of previous turns

clear_history_on_restart: bool = False
  # If True, conversation history is not persisted
```

Caching:

```
enable_response_cache: bool = True
  # Cache successful responses

cache_ttl_seconds: int = 3600
  # Cache expiration (1 hour default)
  # Older entries are auto-deleted

cache_max_size: int = 1000
  # Maximum cache entries
  # LRU eviction when full
```

Logging:

```
log_level: str = "INFO"
  # Options: DEBUG, INFO, WARNING, ERROR, CRITICAL

log_to_file: bool = True
  # Save logs to file

log_file_path: str = "logs/chatbot.log"
  # Log file location

log_conversations: bool = True
  # Log all exchanges to file
```

Error Handling & Reliability

Error Categories and Responses

1. Import Errors:

```
try:
    from google import genai
except ImportError:
    genai = None
    # Bot continues in local-only mode
```

2. API Errors:

Rate Limit (429):

```
Detection: "429" in error message OR "resource exhausted"
Response: "⚠ [API Limit] You're sending requests too fast..."
Retry: Manual (after rate limit window expires)
```

Server Error (503):

```
Detection: "503" OR "unavailable" OR "overloaded"
Response: "⚠ [Server Busy] Google's AI service is overloaded..."
Retry: Automatic (via max_retries)
```

Model Not Found (404):

```
Detection: "404" OR "not found"
Response: "⚠ [Config Error] The model '...' was not found..."
Fix: Update gemini_model in config.py
```

3. File Errors:

```python
# All file operations use try/except:
try:
    with open(file, 'r', encoding='utf-8') as f:
        data = json.load(f)
except FileNotFoundError:
    # Use default patterns or empty data
except JSONDecodeError:
    # Log error, return empty
    logger.error("Corrupted JSON file")
```

4. Encoding Errors:

```python
# Fixed: All file operations use UTF-8
with open(file, 'r', encoding='utf-8') as f:
    # Prevents Windows cp1252 errors
```

Storage & Persistence

File Structure

```
AIMA-chatbot/
├── data/
│   ├── users/
│   │   ├── muaza.json                ← Your profile
│   │   └── {username}.json           ← Per-user profiles
│   ├── conversation_history.json     ← Chat history
│   └── user_preferences.json         ← (Future use)
│
├── local/
│   ├── patterns.json                 ← Pattern database
│   └── knowledge_base.json           ← Factual knowledge
│
├── logs/
│   └── chatbot.log                   ← Application logs
│
└── core/, ai/, utils/                ← Code modules
```

JSON Formats

patterns.json:

```
{
  "greetings": {
    "patterns": [
      "\\b(hello|hi|hey)\\b",
      "good morning",
      "what's up"
    ],
    "responses": [
      "Hello! How can I help you?",
      "Hi there!",
      "Hey! What's up?"
    ],
    "priority": 10
  },
  "learned_a3f8b219": {
    "patterns": ["\\bWhat\\ time\\ does\\ the\\ library\\ close\\?"],
    "responses": ["The library closes at 8 PM."],
    "priority": 9
  }
}
```

knowledge_base.json:

```
[
  {
```

```
    "tags": ["university", "location", "address"],
    "content": "Jamia Millia Islamia is located in New Delhi, India.",
    "sources": ["official_website"]
  }
]
```

user profile (data/users/muaza.json):

```
{
  "username": "muaza",
  "created_at": "2026-01-15T12:00:00",
  "facts": {
    "name": "Muaz",
    "favorite_lang": "Python"
  },
  "preferences": {}
}
```

conversation_history.json:

```
[
  {
    "timestamp": "2026-01-15T16:00:00",
    "user": "Hello",
    "bot": "Hi there! How can I help you?",
    "source": "LOCAL"
  },
  {
    "timestamp": "2026-01-15T16:01:00",
    "user": "What's Python?",
    "bot": "Python is a programming language...",
    "source": "GEMINI"
  }
]
```

Performance Characteristics

Response Times

Scenario	Typical Time	Notes
Local exact match	<1ms	Instant
Local fuzzy match	1-5ms	Depends on pattern count
Cache hit	<1ms	Instant

Scenario	Typical Time	Notes
Multi-intent (all local)	1-10ms	N × local match time
AI call (no context)	500-1500ms	Network dependent
AI call (with context)	600-2000ms	More tokens to process

Memory Usage

Component	Typical Size
Base application	~50 MB
Loaded patterns	~1-5 MB
Conversation history	~10-100 KB
Response cache	~1-10 MB
Per-user profile	~1-10 KB

API Cost Optimization

Strategies Used:

1. **Local-first**: Checks patterns before API
2. **Caching**: Avoids duplicate API calls
3. **Auto-learning**: Converts expensive AI calls to cheap local lookups
4. **Context pruning**: Only sends last N exchanges (not entire history)
5. **Flash model**: Uses cheaper model tier

Cost Example (Gemini 2.5 Flash pricing: ~$0.075 per 1M input tokens):

- Local response: $0
- Cached response: $0
- AI response (~500 tokens): ~$0.000037
- 1000 AI responses: ~$0.037

This documentation provides a complete technical reference for the AIMA ChatBot system. For quick-start guides, see `README.md`. For architectural overview, see `TECHNICAL_OVERVIEW.md`.