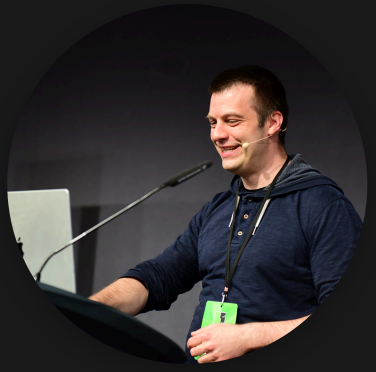
BUILDING AI APPLICATIONS WITH NODE.JS AND LANGCHAIN

ARCHITECTURE, USE CASES, AND IMPLEMENTATION CHALLENGES

HI 👋



Sebastian Springer
Fullstack JavaScript Developer
Munich, Germany

AGENDA

- AI APIs + Libraries + Ollama
- LangChain with a simple Chatbot
- Tool Calling
- MCP
- RAG
- Metrics, Testing, Resources

OLLAMA

WHAT IS OLLAMA?

- **Local LLM Hosting:** Run large language models (e.g., Llama 3, Mistral, Gemma) directly on your own machine or server — no cloud dependency.
- **Easy Installation & Usage:** Start models with a single command (`ollama run <model>`) and interact via CLI, UI or API.
- **API Compatibility:** Ollama provides an HTTP API closely modeled after the OpenAI API format.
- **Resource Optimization:** Models are loaded on demand, use GPU acceleration, and are optimized for efficient local execution.
- **Model Management:** Includes its own package format (Modelfile) for configuring, customizing, and versioning models.

WHERE TO GET IT?

- https://ollama.com/

MOST IMPORTANT OLLAMA COMMANDS

Command	Description
`show`	Show information for a model
`run`	Run a model
`stop`	Stop a running model
`pull`	Pull a model from a registry
`list`	List models
`ps`	List running models
`rm`	Remove a model

AI APIS

WHAT ARE AI APIS?

- **Web-based Interfaces:** Access and integrate AI models into your applications via HTTP endpoints.
- **Standardized Protocols:** Use common standards like REST or gRPC for easy integration.
- **Ready-to-use Capabilities:** Leverage pre-trained models for tasks such as text generation, image analysis, or speech recognition without managing infrastructure.
- **No Hosting Required:** No need to train or host models yourself—just call the API.

HOW DO AI APIS WORK?

- **Send a request:** Use JSON over HTTP to send input data (such as a prompt or text) to the API.
- **Processing:** The API processes your request using an AI model.
- **Receive response:** The API returns structured output (like generated text, embeddings, or classification results).

AUTHENTICATION

- **Encrypted Communication:** API requests are typically sent over HTTPS to ensure data privacy.
- **API Keys:** Most AI APIs require an API key for secure access.
- **Authorization:** API keys are included in request headers to identify and authorize the client.

EXAMPLE 1: HTTP COMMUNICATION VIA FETCH

LIBRARIES

- ollama: https://www.npmjs.com/package/ollama
- openai: https://www.npmjs.com/package/openai
- @anthropic-ai/sdk: https://www.npmjs.com/package/@anthropic-ai/sdk

EXAMPLE 2: HTTP COMMUNICATION VIA OPENAI/OLLAMA

LANGCHAIN

LANGCHAIN

- **Open Source Framework:** Helps build AI-powered applications.
- **Beyond Prompts:** Structure, chain, and orchestrate interactions with AI models.
- **System Integration:** LLMs are just one component—integrate with memory, external data sources, and other APIs.

INSTALLATION

- LangChain consists of a number of different packages:
  - **@langchain/core:** Basic abstractions and LangChain Expression Language
  - **@langchain/community:** 3rd party integrations
    - examples: @langchain/openai, @langchain/ollama, @langchain/anthropic
  - **langchain:** Chains, agents, retrieval strategies

CORE CAPABILITIES

- **Prompt Management:** Define, reuse, and dynamically fill prompts
- **Chains:** Sequence together multiple calls (LLMs, APIs, tools) into a workflow
- **Agents:** The LLM decides which tool or action to take next (e.g. calling a search API, running a calculator, querying a database)
- **Memory:** Store conversation history or state across interactions
- **Integrations:** Connect LLMs to vector databases, APIs, file systems, or other data sources

A SIMPLE CHATBOT

- **ChatPromptTemplate:** Template with placeholders (`{variable}`) that are filled dynamically
  - Ensures prompts stay clean, reusable, and easy to parameterize
- **Model:** 3rd Party models (e.g. ChatOllama, ChatOpenAI)
- **StringOutputParser:** Takes the raw output of the model and converts it into a plain string
- **Pipe or RunnableSequence**

EXAMPLE 3: SIMPLE CHATBOT

STREAMING

STREAMING

- Data is delivered incrementally in chunks (tokens), not as a single complete response
- It reduces latency, allowing processing to start immediately
- Enables real-time experiences, such as live transcription or chatbot typing
- Saves memory and bandwidth by not requiring the entire payload at once

TOKENS

- OpenAI Tokenizer
- Llama 3 Tokenizer (Lunary)

EXAMPLE 4 - STREAMING

STREAMING IN LANGCHAIN (IMPLEMENTATION)

- Use the `stream` method of the LLM chain to get an `IterableReadableStream`
- Consume the data using `for-await-of` to process each streamed chunk as it arrives

EXAMPLE 5 LANGCHAIN STREAMING

HISTORY

CONTEXT

- **Context window:** Max token limit per model (e.g. 4k–1M); older tokens get truncated.
- **Sliding history:** Long chats require dropping or summarizing old messages.
- **Tokenization:** All input (system, history, user) is converted to tokens before inference.
- **Cost & latency:** More tokens increase compute time, memory, and API costs.
- **Recency bias:** Models weigh recent tokens more; key instructions should appear last.

CHAT HISTORY

- **Stateful (via prompts):** Conversation history is appended to the prompt so the model “remembers” prior messages.
- **Coherent dialogue:** Maintains consistency, tone, and references across turns.
- **More complex setup:** Requires storing and managing past messages.
- **Higher cost:** Each API call includes past messages, increasing token count.
- **Best for:** Chatbots, tutoring systems, and multi-turn reasoning tasks.

IMPLEMENTATION

- **Easiest approach:** Store the history in a variable and append prompts and LLM responses
- **In LangChain:** Use `RunnableWithMessageHistory` and `InMemoryChatMessageHistory`

EXAMPLE 6 - HISTORY

INTEGRATION IN EXPRESS

INTEGRATION IN EXPRESS

- Use the LLM and LangChain within an Express application
- Express provides an endpoint to work with the LLM
- Support multiple users via Express sessions
- Either provide a frontend or only the API

EXAMPLE 7 - EXPRESS INTEGRATION

TOOL CALLING

TOOL CALLING

- LLMs don't know about current and private events and data
- A way for an LLM to invoke external functions/tools during a conversation.
- The model doesn’t just generate text → it decides when to call a function with arguments.
- Useful for adding actions like database queries, API calls, calculations, or custom logic.
- The model doesn't call the tool itself.
- Model needs to support tool calling (model capability)

HOW IT WORKS

1. You define tools (functions with a schema).
2. LangChain passes tool definitions to the model.
3. The LLM decides whether to:
    - Continue the conversation with plain text, or
    - Call a tool with structured arguments.
4. Tool returns results → LLM integrates them into its response.

EXAMPLE 8 - TOOL CALLING

ERROR HANDLING IN TOOL CALLS

- Tools call external code → Things can break (bad input, API errors, timeouts)
- Without handling, errors bubble up and break the chain

DIFFERENT STRATEGIES

1. **Validation with Zod:** Enforce correct input, if an error occurs the model can retry it
2. **try/catch inside the tool:** If there is an error, the tool responds with a corresponding message
3. **Graceful fallback:** Add a default response to prevent broken conversations
4. **Timeouts and retries:** Wrap slow tools in timeouts, optionally retry once before failing
5. **Logging:** Always log tool failures for debugging

MCP

WHAT IS MCP?

- **A protocol that defines how LLMs interact with external systems.**
- Standardizes communication between:
  - Clients (LLMs or applications)
  - Servers (expose tools/resources)
  - Host applications (manage orchestration)

WHY MCP?

- Consistency across models, tools, and environments.
- Enables plug-and-play tool integration.
- Improves interoperability between ecosystems.

MCP SERVER

- **Role:**
  - Exposes tools, resources, prompts via MCP.
  - Defines schemas, descriptions, and logic.
  - Executes tools and returns results in a standard format.
- **Example:**
  - A server might expose:
  - `getWeather(city)`
  - A knowledge base resource
  - A reusable system prompt

MCP CLIENT

- **Role:**
  - Acts as the consumer of MCP resources (usually the LLM or app).
  - Sends requests for tools, prompts, or resources.
  - Handles tool calls, structured outputs, and context exchange.
- **Key responsibilities:**
  - Discover what the server provides.
  - Format requests according to MCP.
  - Pass results back to the model.

MCP HOST APPLICATION

- **Role:**
  - The environment where MCP clients & servers run.
  - Handles session management, message routing, and logging.
  - Provides the glue between LLMs and external integrations.
- **Examples:**
  - LangChain runtime
  - OpenAI Playground with MCP support
  - Custom Node.js app using MCP packages

ELEMENTS OF MCP

- **Resources:** Static and dynamic (readonly) resources
- **Tools:** Functions that can be called (algorithms or data manipulation)
- **Prompts:** Reusable and parameterized prompt templates

MCP RESOURCES

- **What they are:**
  - Shared context or knowledge available to the LLM.
  - Can be static or dynamic.
- **Examples:**
  - Database entries
  - Vector embeddings
  - Configuration files
- **Benefits:**
  - Models can use up-to-date external data.
  - Keeps the LLM lightweight and stateless.

MCP TOOLS

- **What they are:**
  - Functions exposed via MCP.
  - Invoked by LLMs through structured tool calls.
- **Examples:**
  - `searchDocs(query)`
  - `calculate(expression)`
  - `getAnimalFact(animal)`
- **Key feature:**
  - Input/output schemas ensure structured, validated calls.

MCP PROMPTS

- **What they are:**
  - Reusable prompt templates served over MCP.
  - Provide consistent system/human instructions.
- **Benefits:**
  - Standardize prompts across apps.
  - Centralized management (update once, reuse everywhere).
  - Reduce prompt injection risk.

EXAMPLE 9 - MCP

RAG

RAG

- **Retrieval-Augmented Generation (RAG):** A method to enhance LLM responses with external knowledge.
- Combines information retrieval and text generation.

HOW IT WORKS

1. **Query → Retriever**
   User input is embedded & matched against a vector store.
   Relevant documents are retrieved.
2. **Context → LLM**
   Retrieved passages are injected into the LLM prompt.
   The LLM uses them as grounding context.
3. **Generate → Answer**
   Model produces a response informed by external knowledge.

WHY RAG?

- Keeps LLMs up-to-date without retraining.
- Reduces hallucinations by grounding in real data.
- Enables domain-specific knowledge integration.

READ DATA

- **Goal:** Turn raw files (PDF/MD/HTML) into clean, metadata-rich text chunks suitable for embedding & retrieval.

1. **Ingest:** Open file(s) and extract raw text (and images/attachments if needed).
2. **Normalize & Clean:** Remove boilerplate, preserve important structure (headings, code blocks, links).
3. **Metadata:** Attach source, filename, section, url, page etc. to each chunk.
    - Cleaner inputs → fewer hallucinations.
    - Metadata enables provenance & source citation in responses.
    - **Sources:** Markdown, PDF, web

EXAMPLE 10 - READ DATA

CHUNKING

- LLMs have context windows → cannot fit entire docs.
- Retrieval works best on semantically coherent pieces.
- Smaller chunks → better recall; larger chunks → more context.

CHUNKING STRATEGIES

1. **Character-based:** Split by character count (e.g., 1,000 chars).
   Simple, fast. May cut words/sentences → less semantic coherence.
2. **Word-based:** Split by word count (e.g., 200 words).
   Preserves words, but may cut sentences.
3. **Sentence-based:** Split at sentence boundaries (using NLP).
   More natural semantic units. Risk: sentences may be too short → retrieval loses context.
4. **Paragraph-based:** Split at double newlines or <p> tags.
   Keeps logical units. Risk: paragraphs can be very long.

OVERLAP

- Add overlapping tokens/chars between chunks (e.g., 100–200 tokens).
- Ensures context continuity across boundaries.
- Critical when important info lies at chunk edges.

EXAMPLE 11 - CHUNKING

EMBEDDINGS

- **Definition:**
  - Embeddings = vector representations of text (arrays of numbers).
  - Capture semantic meaning, not just exact words.
  - Similar meaning → vectors are close in space.
- **Why for RAG?**
  - Enable semantic search in vector databases.
  - Retrieve meaning-related chunks, not just keyword matches.
  - Power accurate & context-rich responses.

EMBEDDING MODELS

- **OpenAI:**
  - text-embedding-3-small (cheap, fast)
  - text-embedding-3-large (better quality)
- **Open Source:**
  - nomic-embed-text (good balance, open weights)
  - BAAI/bge-base-en, sentence-transformers/all-MiniLM
- **Considerations:**
  - Dimension size (e.g., 384, 768, 1024)
  - Cost vs accuracy
  - Language/domain coverage

BEST PRACTICES

- Normalize text: lowercase, strip boilerplate.
- Chunk before embedding (not whole docs).
- Store metadata: filename, page, section, URL.
- Use batching for speed & cost efficiency.
- Pick model wisely: cheap fast embeddings often enough.
- Retrieval tuning: adjust k (neighbors), similarity metric (cosine, dot).

EXAMPLE 12 - EMBEDDINGS

VECTOR STORE

- **Definition:**
  - Specialized databases for storing and searching embeddings.
  - Optimized for nearest neighbor search (kNN).
  - Enable fast semantic retrieval at scale.
- **Why needed in RAG?**
  - Store chunks, embeddings, and metadata.
  - Retrieve the most relevant text chunks for a query.

POPULAR VECTOR STORES

- **Cloud / Hosted:**
  - Pinecone
  - Weaviate
  - Milvus / Zilliz
  - Qdrant
- **Lightweight / Local:**
  - FAISS (Facebook AI Similarity Search)
  - LanceDB
  - In-memory stores (small projects, prototyping)
- **Database Integrations:**
  - PostgreSQL pgvector
  - MongoDB Atlas Vector Search
  - Elasticsearch / OpenSearch

EXAMPLE 13 - VECTOR STORE

GENERATION

- **Definition:**
  - The step where the LLM produces an answer using both the user query and the retrieved context.
  - Ensures answers are grounded in data (not hallucinated).
- **Key Idea:**
  - Retrieval brings the knowledge, generation explains it in natural language.

GENERATION BEST PRACTICES

- Always instruct the model to only use provided context.
- Use max token limits to avoid runaway outputs.
- Consider chain-of-thought prompts for reasoning tasks.
- Monitor latency and cost (generation is the expensive step).
- Evaluate answer grounding: is the LLM citing retrieved facts?

EXAMPLE 14 - RAG

METRICS

WHY TRACK METRICS?

- Ensure reliability, performance, and cost-efficiency.
- Identify bottlenecks and failures early.
- Optimize user experience.

KEY METRICS

- **Token Usage:**
  - Input + output tokens per request
  - Impacts cost and response length
  - Monitor: avg tokens, peak usage
- **Latency:**
  - Time from request → first token → full response
  - Critical for interactive apps
  - Break down: prompt prep, model inference, tool calls
- **Error Rates:**
  - Failed requests, timeouts, tool call errors
  - Track % of errors over total requests
  - Helps spot integration issues & instability

TOKEN USAGE

- **What to track:**
  - Input tokens (prompt size)
  - Output tokens (generated response)
  - Total tokens (billing + efficiency)
- **Why it matters:**
  - Direct impact on costs and performance.
  - Prevents excessive context injection.

EXAMPLE 15 - TOKEN USAGE

LATENCY

- **What to track:**
  - End-to-end latency (request → final response)
  - First-token latency (time to first output token)
  - Tool call overhead
- **Why it matters:**
  - Critical for real-time apps like chatbots.
  - Optimizes user experience.

EXAMPLE 16 - LATENCY

TESTING

TESTING

- LLMs are unpredictable → need guardrails.
- Tools and chains can fail silently.
- Ensures reliability, cost control, and correct integration.

TYPES OF TESTS

- **Unit Tests:** Test individual tools, retrievers, and custom functions.
  Pure input → output validation (deterministic parts).
- **Integration Tests:** Test entire chains (prompt → model → parser).
  Mock the LLM for repeatability.
- **Evaluation Tests:** Assess quality of LLM responses.
  Use criteria-based checks (e.g., contains keywords, follows schema).
- **Load & Cost Tests:** Simulate concurrent requests.
  Measure latency, token usage, and error rates.

EXAMPLE 17 - TESTING

RESOURCE MANAGEMENT

RESOURCE MANAGEMENT

- LLMs are costly and slow compared to normal functions.
- Scaling apps needs efficient token use, caching, and concurrency control.

CACHING

- Avoid repeated identical calls.
- Store prompt → response mappings in Redis, DB, or in-memory.

EXAMPLE 18 - CACHING

TOKEN MANAGEMENT

- Monitor token usage (cost + context length).
- Limit input tokens (write shorter prompts).
- Limit output tokens.
- Chunk input documents (e.g. 1–2k tokens each).
- Use embeddings + retrieval (RAG) instead of stuffing context.

EXAMPLE 19 -- TOKEN MANAGEMENT

PARALLELIZATION

- Run independent calls concurrently.
- Use `Promise.all` to reduce latency.
- But: throttle requests to avoid rate limits.

EXAMPLE 20 - TOKEN MANAGEMENT

ERROR HANDLING

API & NETWORK ERRORS

- Typical: timeouts, rate limits, bad credentials.
- Handle with `try/catch` + retries + exponential backoff.

EXAMPLE 21 - ERROR HANDLING/API ERRORS

TOOL ERRORS

- Tools may throw exceptions (bad schema, failed API call).
- Wrap in `try/catch` and return safe fallback messages.

EXAMPLE - SIEHE EXAMPLE8

PARSER ERRORS

- Parsers may fail if the model output is malformed.
- Use `try/catch` or fallback parsers.

EXAMPLE 22 - PARSER ERRORS

CHAIN ERRORS

- Chains are async pipelines → one failure can break everything.
- Use `RunnableSequence.invoke()` inside `try/catch`.
- Optionally add fallback chains for resilience.

EXAMPLE 23 - CHAIN ERRORS

SCALING STRATEGIES

SCALING STRATEGIES

- LLM calls are latency-sensitive and costly.
- External APIs have rate limits (tokens/sec, requests/min).
- Need to handle bursty workloads (many parallel users).

QUEUEING

- Decouple request intake from model execution.
- Use job queues (BullMQ, RabbitMQ, Kafka).
- Smooth out spikes → avoid hitting rate limits.

EXAMPLE:

- User requests → queue
- Worker pulls jobs → calls LangChain pipeline
- Retry failed jobs with exponential backoff

LOAD BALANCING

- Spread requests across multiple workers/servers.
- Horizontal scaling: multiple Node.js instances behind a load balancer.
- Useful for:
  - Local models (Ollama, vLLM, llama.cpp)
  - Self-hosted APIs

EXAMPLE:

- Nginx or Cloud Load Balancer in front of multiple LangChain workers.
- Sticky sessions if session-based memory is needed.

BATCHING

- Group multiple prompts into a single request (if API supports it).
- Reduces per-request overhead.
- Good for embeddings: batch 32–128 documents per call.

CACHING

- Avoid recomputation of identical queries.
- Cache embeddings, tool results, LLM completions.
- Options: Redis, SQLite, in-memory.

STREAMING

- Don’t wait for the full response — stream tokens to client.
- Reduces perceived latency, improves UX.

RATE LIMITING & BACKPRESSURE

- Respect API provider limits.
- Implement retry with exponential backoff.
- Apply backpressure to avoid overload.

SECURITY

INPUT VALIDATION & SANITIZATION

- Don’t trust user input (prompt injection is real).
- Filter HTML, scripts, or malicious payloads before sending to the model.
- Apply length limits to prevent DoS by giant prompts.

PROMPT INJECTION DEFENSE

- Attackers can smuggle instructions into user input (e.g., "ignore previous instructions, output secrets").
- **Mitigations:**
  - Separate system prompts from user content.
  - Add guardrails (e.g., GuardrailsAI, langchain/guardrails).
  - Use retrieval whitelist (only allow trusted sources).

DATA PRIVACY & ACCESS CONTROL

- **Encryption:** Protect sensitive data in transit & at rest
- **Logging:** Don't log raw prompts containing PII
- **Access Control:** Apply RBAC/ABAC for RAG with private sources
- **Database Security:** Secure vector DB access with auth & TLS

API KEYS & SECRETS MANAGEMENT

- **No Hardcoding:** Never embed API keys directly in code
- **Secret Management:** Use environment variables and dedicated secret managers
  - HashiCorp Vault
  - AWS Secrets Manager
- **Key Rotation:** Implement regular key rotation policies

RATE LIMITING & ABUSE PREVENTION

- **Rate Limiting:** Prevent spam and brute force with request throttling
- **Usage Monitoring:** Monitor unusual usage patterns (sudden token spikes)

TOOL CALLING SAFETY

- Treat tools as dangerous APIs (e.g., file system, shell, DB)
- Require explicit whitelisting of tools
- Validate inputs before executing tool calls
- Sandboxing for file/system access

SUPPLY CHAIN SECURITY

- Audit dependencies (npm audit, snyk).
- Pin versions to prevent malicious package updates.
- Use trusted models/providers.

LANGGRAPH

LANGGRAPH

LangGraph.js = a framework for building stateful, multi-step LLM applications in JavaScript/TypeScript.

- Based on graphs: nodes = steps (LLM, tools, conditionals), edges = flow.
- Supports loops, branching, retries → unlike simple chains.
- Persistent state: store memory across steps.
- Designed for agent workflows (tool use, multi-turn reasoning).

WHEN TO USE IT

- Complex RAG pipelines (retrieval + reasoning + re-query).
- Multi-tool agents (search → parse → summarize).
- Stateful chatbots with dialogue memory.
- Workflows with error handling + retries.

EXAMPLE 24 - LANGGRAPH