



ENTERPRISE SEARCH PLATFORM

TURING

Intelligent Search & AI

D O C U M E N T A T I O N

v 2026.2

May 11, 2026



viglet

viglet.org

Turing ES

Enterprise Search Platform

PRODUCT Turing ES — Enterprise Search

VERSION 2026.2

GENERATED May 11, 2026

WEBSITE viglet.org

SOURCE github.com/openturing

LICENSE Apache License 2.0

Turing ES is an enterprise search platform that combines powerful indexing, natural language processing, and AI-powered retrieval to deliver intelligent search experiences. This document covers architecture, installation, configuration, connectors, and the complete API reference.

Table of Contents

What is Turing ES?	1
Core Concepts	5
Turing ES — Architecture Overview	14
Viglet Turing ES: Installation Guide	31
Configuration Reference	46
Search Engine	65
Semantic Navigation	70
DSL Query API	111
DSL Compatibility Matrix	138
SPA Pages	152
Integration	157
AEM Connector	164
Import & Export	167
Generative AI & LLM Configuration	173
What is RAG?	177
LLM Instances	193
Embedding Stores	199
Embedding Models	210
Assets	218
Tool Calling	229
Custom Tools	236
MCP Servers	252
AI Agents	256
Personas	271
Chat	285

Chat Flow	298
Intent	314
Chat Analytics	322
Token Usage	341
Administration Guide	346
Logging	361
Observability — Prometheus & Grafana	370
Developer Guide	384
React SDK	398
REST API Reference	411
GraphQL API	437
Authentication	456
Social Login (OAuth2)	459
Security & Keycloak	466

What is Turing ES?

Viglet Turing ES is an open-source enterprise search platform. It helps organizations make their content findable, understandable, and interactive — through keyword search, faceted navigation, and generative AI conversations.

Whether you have thousands of documents on a file server, pages on a CMS, records in a database, or a combination of all of them, Turing ES indexes everything into a single search experience that your users can explore naturally.

What can you do with Turing ES?

Search your content

Index content from multiple sources and expose a unified, faceted search experience. Users can filter results by category, date, author, or any attribute you define — and always find what they are looking for.

Ask questions, get answers

Enable generative AI on your search site and let users ask questions in natural language. Turing ES retrieves the most relevant documents and uses an LLM to generate a grounded, accurate response — not a hallucination.

Build AI Agents

Compose AI Agents that combine a language model with a curated set of tools: search your content, browse the web, run code, query financial data, call external systems via MCP, and more. Each agent appears as a chat tab ready to assist users.

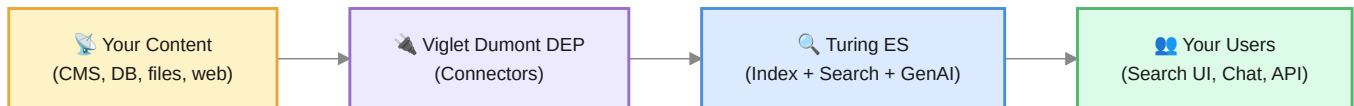
Connect any content source

Turing ES receives content from **Viglet Dumont DEP**, a companion application that provides connectors for web crawlers, databases, file systems, AEM, and WordPress. Connectors run independently and push content to Turing ES through its REST API.

Integrate with any stack

Consume Turing ES from your application via **REST API**, **GraphQL**, the **Java SDK** (available on Maven Central), or the **JavaScript / TypeScript SDK** [@viglet/turing-sdk](#) (available on npm).

How it works at a glance



Content flows from its sources through Dumont DEP connectors into Turing ES, where it is indexed and made available to users through search interfaces, chat, and APIs.

Key concepts

These are the main building blocks you will work with in Turing ES. You do not need to understand all of them before getting started — come back to each one as you need it.

CONCEPT	WHAT IT IS	LEARN MORE
Semantic Navigation Site	The central configuration object. Defines what content is indexed, how it is searched, and how results are presented.	Core Concepts
Connector	A component in Dumont DEP that extracts content from a source and sends it to Turing ES.	Core Concepts
Facets	Filterable attributes shown alongside search results (e.g., category, date, author).	Core Concepts
Spotlight	Curated results pinned to specific search terms.	Semantic Navigation
Targeting Rules	Rules that show different results to different users based on their profile.	Semantic Navigation
Merge Providers	Rules that combine documents from two different connectors into one enriched result.	Semantic Navigation
RAG	Retrieval-Augmented Generation — finding relevant documents and using them to ground an LLM's response.	GenAI & LLM
AI Agent	A named assistant that combines an LLM with a set of tools and knowledge sources.	GenAI & LLM
Assets	A file manager backed by MinIO where you upload documents to feed the Knowledge Base — files are automatically indexed as vector embeddings.	Assets
Chat	The conversational AI interface — three tabs for direct LLM chat, Semantic Navigation search, and each configured AI Agent.	Chat
Token Usage	A dashboard showing LLM token consumption by model, day, and month — useful for monitoring AI costs.	Token Usage

Where to go next

Not sure where to start? Here is a suggested path depending on what you want to do:

I want to understand how Turing ES works → Read [Core Concepts](#) first, then [Architecture Overview](#).

I want to set up Turing ES → Go to the [Installation Guide](#).

I want to configure search for my content → Start with [Core Concepts](#) then go to [Semantic Navigation](#).

I want to add generative AI to my search → Read [GenAI & LLM Configuration](#).

I want to use the AI chat interface → Go to [Chat](#) for direct LLM, Semantic Navigation, and AI Agent tabs.

I want to upload documents to the Knowledge Base → Go to [Assets](#) to manage files and train the RAG index.

I want to monitor LLM usage and costs → Go to [Token Usage](#).

I want to secure Turing ES with SSO → Go directly to [Security & Keycloak](#).

Core Concepts

This page explains the fundamental concepts of Turing ES in plain terms. No configuration files, no code — just the mental model you need before diving into the technical documentation.

Semantic Navigation Site

The **Semantic Navigation Site** (or SN Site) is the central object in Turing ES. Everything revolves around it.

Think of an SN Site as a configured search experience. It defines:

What content is indexed — which fields each document has (title, text, URL, date, image, and custom fields)

How content is searched — which fields are searchable, how results are ranked, how many results per page

How results are navigated — which fields become filterable facets (e.g., category, author, date range)

How results are presented — field mappings for display, highlighting of matched terms

Advanced behaviors — Spotlights for curated results, Targeting Rules for personalization, Merge Providers for combining content from multiple sources

AI capabilities — whether GenAI and RAG are enabled, and which language model to use

A single Turing ES instance can host multiple SN Sites, each independently configured. For example, you could have one site for your public website, another for your internal knowledge base, and a third for your product documentation — all on the same server.

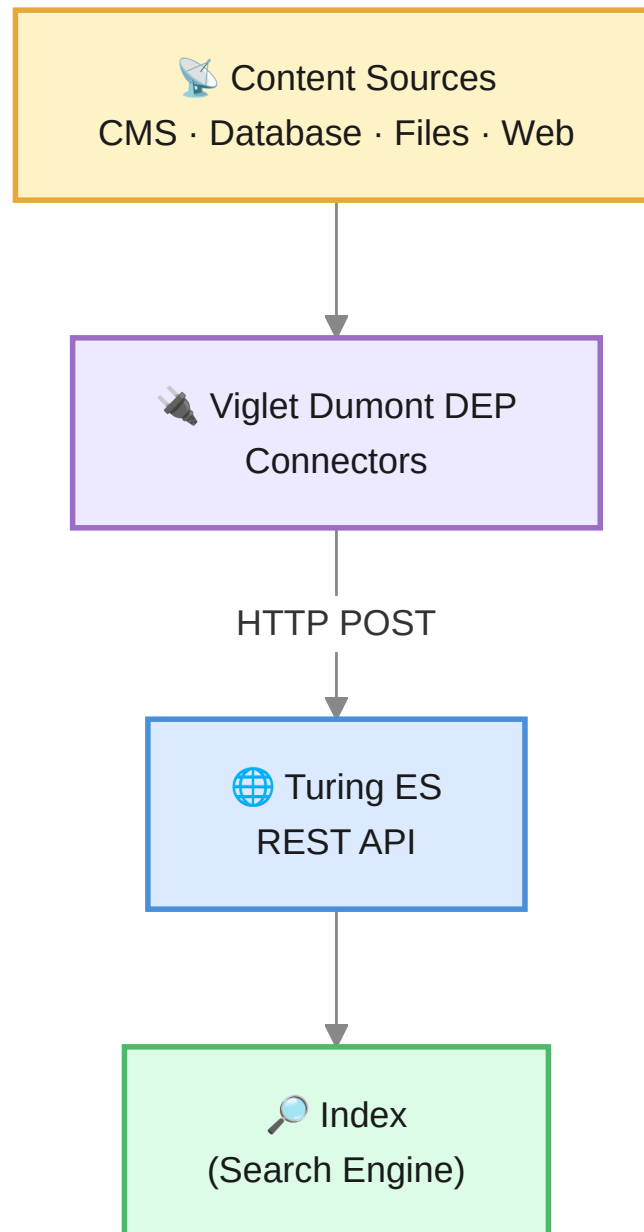
Content Ingestion

Content does not flow into Turing ES on its own. It needs connectors.

Viglet Dumont DEP

Viglet Dumont DEP is a separate application that manages connectors — the components responsible for extracting content from its original source and sending it to Turing ES. Dumont DEP and Turing ES are separate projects that work together: Dumont handles *getting* content, Turing

handles *indexing and searching* it.



How a connector works

Each connector in Dumont DEP:

- Connects to a content source (a website, a database table, a file folder, a CMS)

- Extracts documents according to its configuration (which pages to crawl, which SQL query to run, which folder to scan)

- Sends each document to the Turing ES REST API, targeting a specific SN Site

Turing ES receives the document, validates it against the SN Site configuration, and creates an indexing job. The job processes the document and writes it to the configured search engine (Apache Solr, Elasticsearch, or Apache Lucene) — making it immediately searchable. Apache Solr is the primary and default backend, but Turing ES also supports Elasticsearch and Lucene as alternative search engine backends.

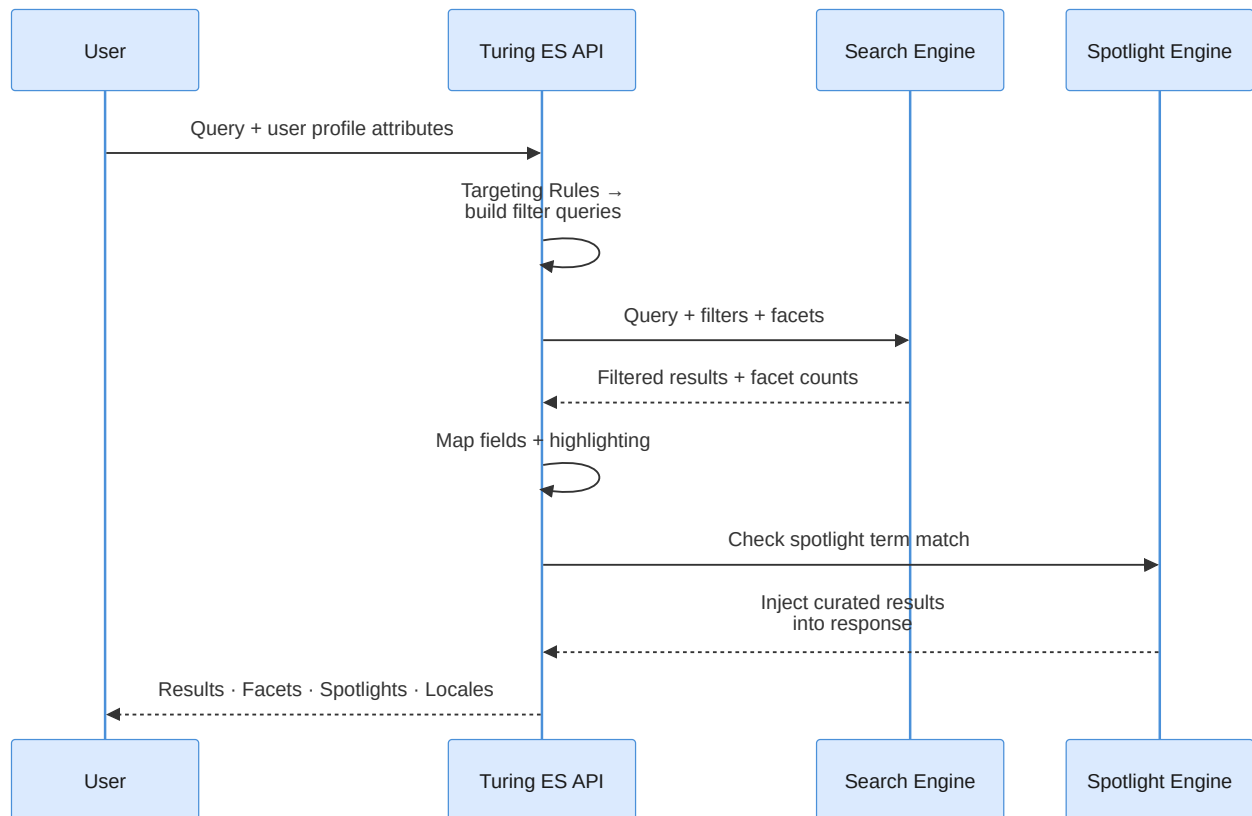
What happens when two connectors index the same content

Sometimes the same real-world document exists in two systems with complementary information. For example, your CMS has structured metadata (author, tags, content type) while your web crawler has the full rendered text of the same page. Neither connector alone gives you a complete document.

Merge Providers solve this by instructing Turing ES to detect when two connectors have indexed the same document — using a shared field as a join key — and merge them into one enriched result. See [Semantic Navigation](#) to learn how to configure this.

Search

When a user searches on an SN Site, this is what happens — in simple terms:



The query arrives at the Turing ES API along with the user's profile attributes (if Targeting Rules are configured)

Targeting Rules run first — they translate the user's profile attributes into additional filter queries (e.g., Solr `fq` parameters when using the default Solr backend), so only content relevant to that user is retrieved

Turing ES executes the search engine query with those filters, facet counts, and result ranking applied together

Results come back from the search engine already filtered; Turing ES maps fields and applies highlighting

Spotlights are checked last — if the query matches a spotlight term, curated documents are injected at their configured positions in the response

The final response — results, facets, spotlights, locales, spell check, and similar documents — goes back to the client

Facets

Facets are filterable dimensions shown alongside search results. A user searching for "annual report" might see facets for *Year*, *Department*, and *Document Type*, and click to narrow results to "Finance" documents from "2024".

Facets are configured per SN Site, at the field level. Any indexed field can become a facet — the configuration defines how it is labeled, sorted, and counted.

Spotlights

A Spotlight is a curated result pinned to a specific search term. When someone searches for "benefits", you can ensure your HR benefits page always appears at the top — regardless of how it ranks organically.

Spotlights are configured per SN Site and managed through the admin console. See [Spotlights](#) for details.

Targeting Rules

Targeting Rules let you show different results to different users from the same index. A document tagged as `audience: internal` only appears for users whose profile includes that attribute. Users without a matching profile always see untagged content.

See [Targeting Rules](#) for a full explanation.

Generative AI

Turing ES integrates generative AI in two places: directly on **SN Sites** (search + summarization) and through standalone **AI Agents** (open-ended chat with tools).

RAG on an SN Site

When GenAI is enabled for an SN Site, indexed documents are also stored as **vector embeddings** — a mathematical representation of their meaning, not just their words. When a user asks a question, Turing ES:

- Finds the most semantically relevant documents (not just keyword matches) using the vector embeddings
- Passes those documents as context to a language model
- Returns a natural-language answer grounded in your actual content

This is called **Retrieval-Augmented Generation (RAG)**. The LLM does not make things up — it answers based on what is in your index.

Knowledge Base

Beyond SN Sites, Turing ES provides a **Knowledge Base** — a file and folder interface in the admin console (backed by MinIO) where you can upload documents directly. These files are also indexed as vector embeddings and become available to AI Agents as a searchable knowledge source, independent of any connector or SN Site.

The Knowledge Base is managed through the **Assets** page: drag-and-drop upload, folder navigation, inline preview, and batch AI training with real-time progress. Uploaded files are indexed automatically on upload and unindexed on deletion.

AI Agents

An **AI Agent** is a named assistant that you compose from three ingredients:

- A **language model** (Anthropic Claude, OpenAI, Gemini, Ollama, or others)
- A set of **tools** it can call (search your SN Sites, query the Knowledge Base, browse the web, run Python code, get financial data, and more)
- Optionally, one or more **MCP Servers** — external services that provide additional tools via the Model Context Protocol

Each AI Agent appears as its own tab in the **Chat** interface. You can have a "Research Assistant" that combines SN Site search with web browsing, a "Data Analyst" that can run Python code and query your knowledge base, and a "Support Agent" that only sees your product documentation — all on the same platform.

See [AI Agents](#) for configuration, [Tool Calling](#) for the full tool reference, and [MCP Servers](#) for connecting external tools.

The Admin Console

Everything described above — SN Sites, connectors, Spotlights, Targeting Rules, Merge Providers, AI Agents, MCP Servers, and the Knowledge Base — is configured and managed through the **Turing ES Admin Console**: a browser-based React application served by Turing ES itself.

The admin console also exposes application logs (when MongoDB is configured), search metrics, and top search terms, giving you operational visibility without needing access to the server.

Consuming Turing ES from Your Application

Turing ES exposes its search, chat, and indexing capabilities through four integration options. Choose the one that best fits your technology stack.

REST API

The primary integration method. All search, indexing, and administration operations are available as HTTP endpoints. The search response is a self-describing JSON object with pre-built navigation links — no URL construction needed on the client side.

```
GET https://<your-host>/api/sn/{siteName}/search?q=your+query
```

Suitable for any language or platform that can make HTTP requests.

GraphQL

Turing ES also exposes a GraphQL endpoint for clients that prefer a graph-based query model. Useful when you want to request only specific fields or combine multiple queries in a single request.

```
POST https://<your-host>/graphql
```

Java SDK

An official Java client library is available on **Maven Central**. It provides a typed API for performing searches, submitting documents for indexing, and interacting with SN Sites — without dealing with raw HTTP or JSON.

```
<dependency>
  <groupId>com.viglet.turing</groupId>
  <artifactId>turing-java-sdk</artifactId>
  <version>${turing.version}</version>
</dependency>
```

JavaScript / TypeScript SDK

An official JavaScript SDK is available on **npm**. It is TypeScript-ready and covers the same search and indexing operations as the Java SDK, suited for web applications and Node.js backends.

```
npm install @viglet/turing-sdk
```

Ready to go deeper?

I WANT TO...	GO TO
Understand the full system architecture	Architecture Overview
Configure Spotlights, Targeting Rules, or Merge Providers	Semantic Navigation
Set up GenAI and RAG	GenAI & LLM Configuration
Configure AI Agents and tools	AI Agents, Tool Calling
Use the AI chat interface	Chat
Upload documents to the Knowledge Base	Assets
Monitor LLM token consumption	Token Usage
Authenticate via API Key	Authentication
Secure Turing ES with Keycloak	Security & Keycloak
Install Turing ES	Installation Guide

Turing ES — Architecture Overview

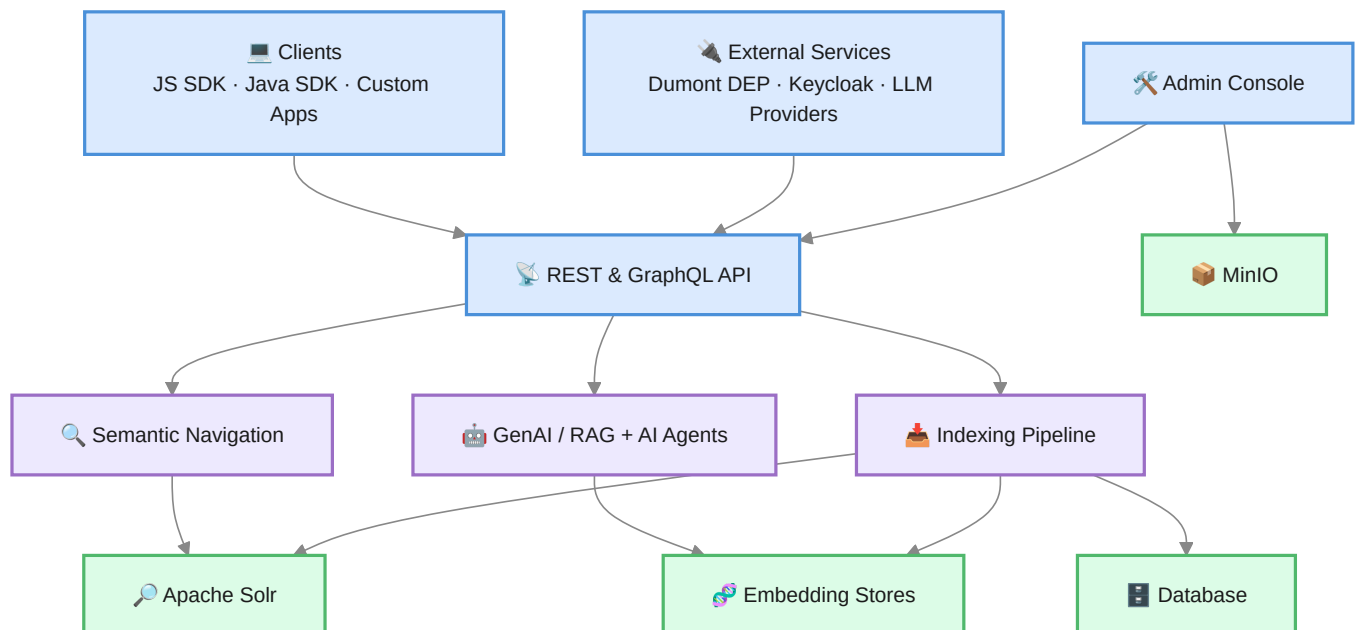
Introduction

Viglet Turing ES is an open-source enterprise search platform that combines semantic navigation, generative AI, tool calling, and AI Agents. It allows organizations to index content from multiple sources, apply Retrieval-Augmented Generation (RAG) over that content, and expose rich search experiences through REST and GraphQL APIs.

Content ingestion is handled by **Viglet Dumont DEP**, a separate project that runs connectors independently and delivers indexed documents to Turing ES via an asynchronous message queue.

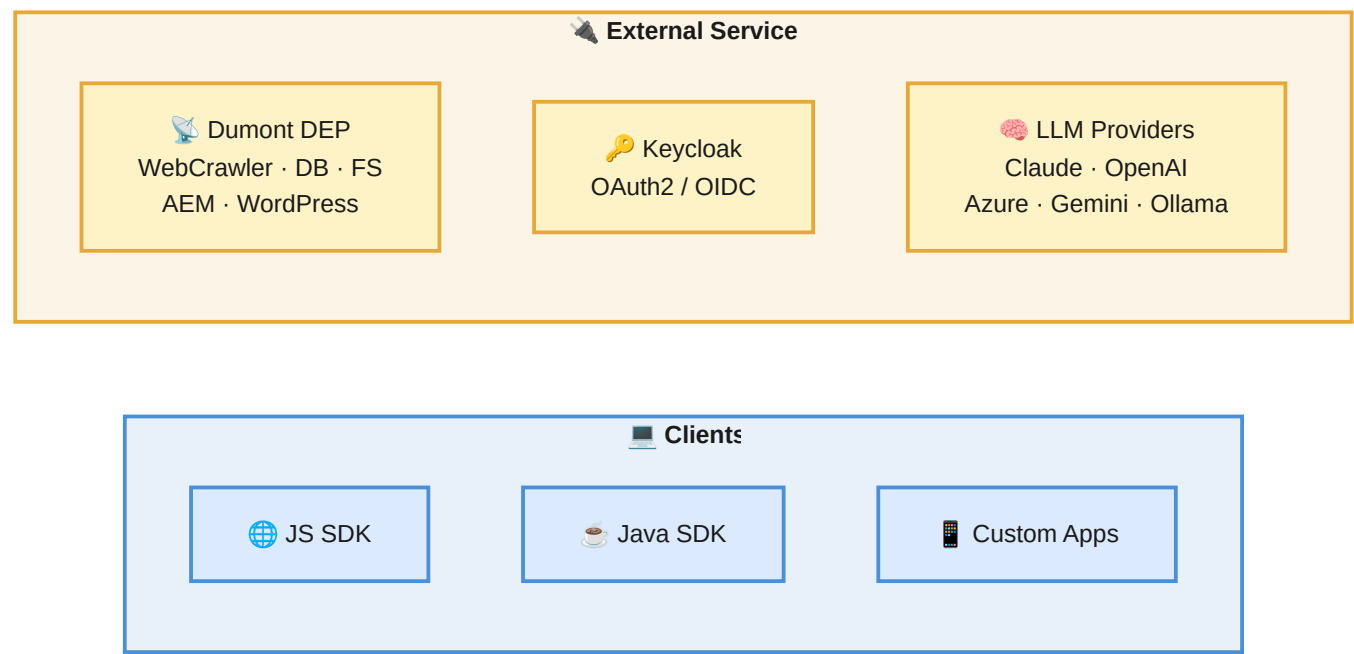
This document describes the system's components, internal modules, and the two core data flows: **indexing** and **search**.

High-Level Component Diagram



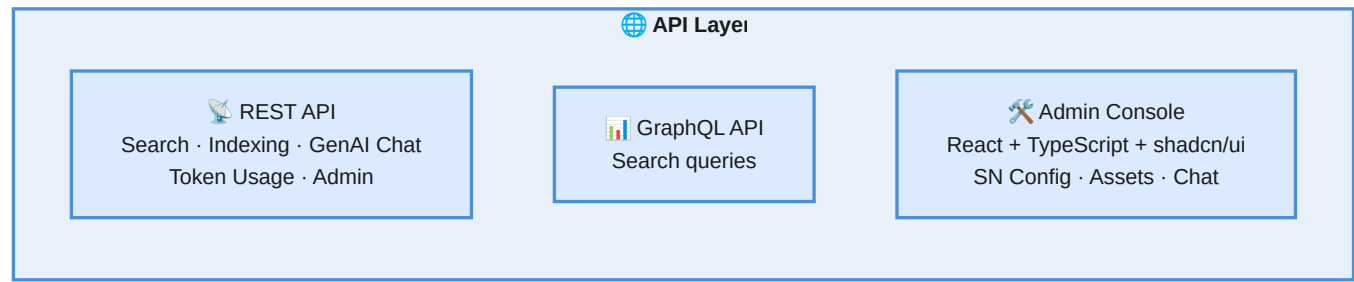
The architecture is organized into **four layers**: Clients & External Services at the top, an API gateway in the middle, Core Modules for business logic, and Backends & Storage at the bottom. Each layer is detailed below.

Clients & External Services



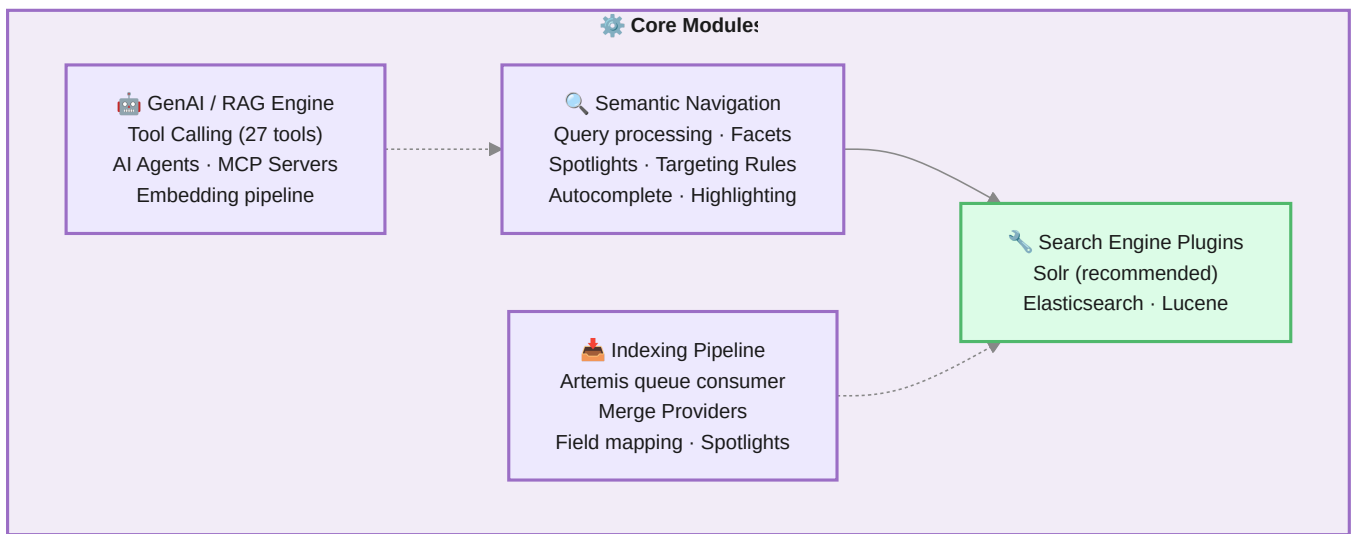
COMPONENT	DESCRIPTION
JS SDK / Java SDK	Typed clients for search and indexing — available on npm and Maven Central
Dumont DEP	Separate application that runs connectors and sends documents to Turing ES via REST API
Keycloak	Optional OAuth2 / OIDC provider for production SSO
LLM Providers	Six supported vendors: Anthropic Claude, OpenAI, Azure OpenAI, Google Gemini, Gemini (OpenAI-compatible), Ollama

API Layer



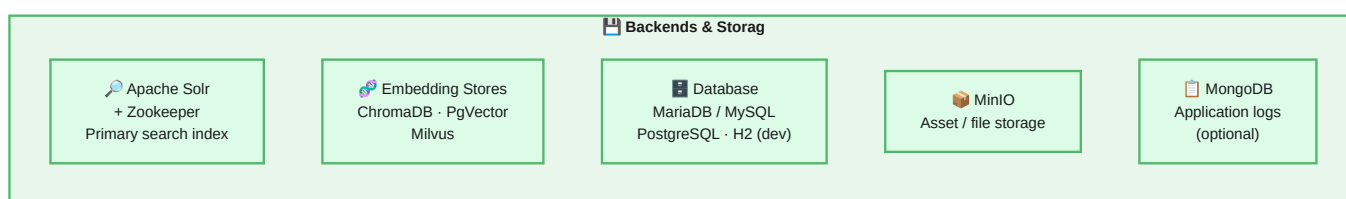
COMPONENT	PACKAGE	DESCRIPTION
REST API	api	Controllers for search, indexing, GenAI chat, agents, token usage, and administration
GraphQL API	api	Search query resolvers as an alternative to REST
Admin Console	turing-react	SN Site configuration, Assets file manager (MinIO), Chat interface, Token Usage dashboard
Security	spring/security	Session-based auth (console) + API Key header (REST) + optional Keycloak OAuth2/OIDC

Core Modules



MODULE	PACKAGE	RESPONSIBILITY
Semantic Navigation	<code>sn</code>	Core search orchestration: query processing, facets, spotlights, targeting rules, autocomplete
Search Engine Plugins	<code>se / plugins/se</code>	Abstraction layer over Solr (recommended), Elasticsearch, and Lucene
GenAI / RAG	<code>genai</code>	RAG over SN content and MinIO assets; LLM context building and invocation
Tool Calling	<code>genai/tool</code>	27 native tools: SN search (15), RAG/KB (4), web crawler (2), finance (2), weather (1), image search (1), datetime (1), code interpreter (1); plus MCP servers
LLM Providers	<code>genai/provider/llm</code>	Pluggable provider factory: Anthropic, OpenAI, Azure OpenAI, Gemini, Gemini-OpenAI, Ollama
AI Agents	<code>agent</code>	Composition of LLM Instance + Tools + MCP Servers into deployable assistants
Indexing Pipeline	<code>indexer</code>	Receives documents via Artemis, applies Merge Providers, writes to Solr and embedding stores
Message Queue	<code>artemis</code>	Apache Artemis — async communication between Dumont DEP and the indexing pipeline
OCR	<code>ocr</code>	Text extraction from PDFs, Word documents, and images via Apache Tika
Persistence	<code>persistence</code>	JPA entities, repositories, DTOs, and MapStruct mappers for all domain objects

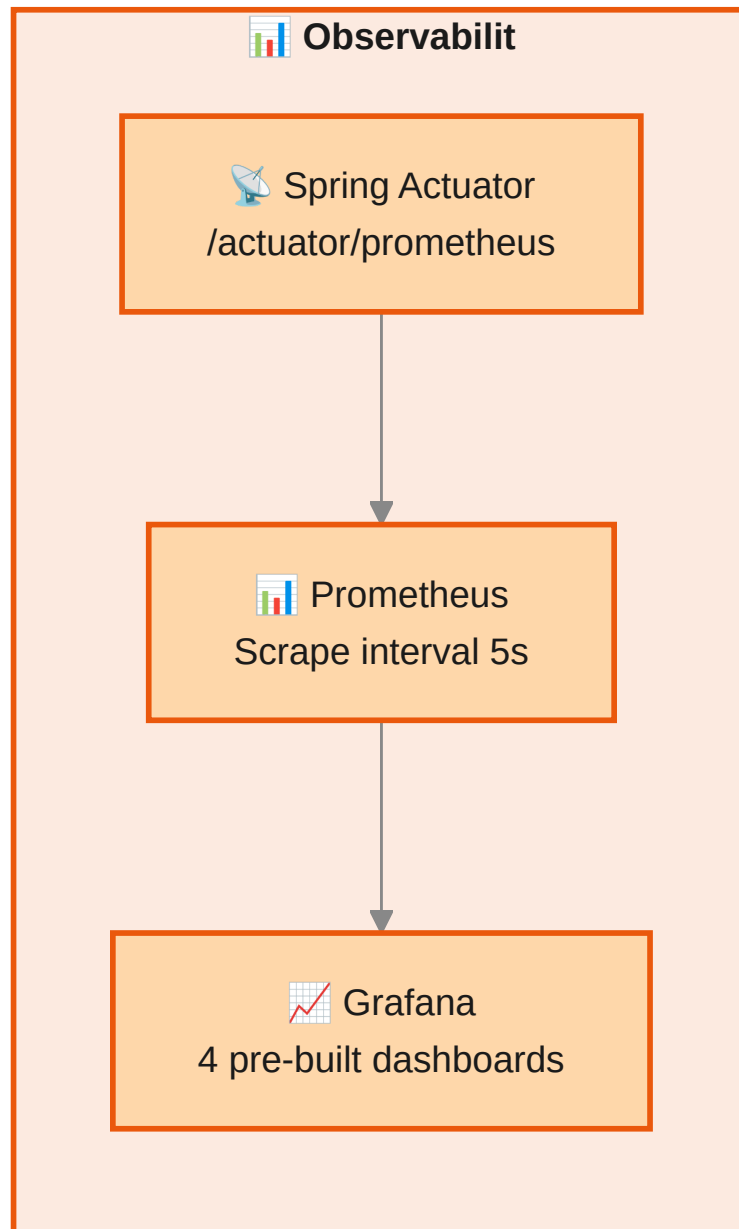
Backends & Storage



BACKEND	PURPOSE	NOTES
Apache Solr	Primary search index	SolrCloud mode with Zookeeper in production
Embedding Stores	Vector storage for RAG	One active per deployment — details
Database	Configuration, metadata, spotlights	H2 for dev; MariaDB/MySQL for production
MinIO	Asset/file object storage	Managed via admin console Assets file manager
MongoDB	Application log persistence	Optional — custom Logback appender, browsable in admin console
MongoDB / Redis (Chat Analytics)	Chat session metadata + AI enrichment	Optional, engine-agnostic via turing.logging.engine ; powers Chat Analytics

Observability Layer

Independent of all the above, an optional observability layer instruments every LLM call, search, snapshot lookup, and chat-analytics enrichment cycle. **Spring Boot Actuator** exposes [/actuator/prometheus](#); **Prometheus** scrapes it; **Grafana** renders four pre-built dashboards (search health, search insights, search-term detail, chat insights).



COMPONENT	ROLE	NOTES
Actuator	Metrics endpoint	Built into Spring Boot; no extra dependency
Micrometer + Observation API	Metric/trace emission	Stable metric names defined in <code>TurMeterNames</code> ; same observation produces both timer + span (OTel bridge)
Prometheus	Time-series scrape & store	<code>containers/prometheus/prometheus.yml</code> is the local-dev config
Grafana	Dashboards over metrics + Chat Analytics	Provisioning under <code>containers/grafana/</code>

The Chat Analytics dashboard reads from Turing ES's REST API (engine-agnostic JSON over HTTP) via the **Infinity datasource** — independent of Prometheus. See [Observability](#) for full coverage.

Indexing Flow

Content ingestion is handled externally by **Viglet Dumont DEP**. Each connector runs as an independent process and sends documents to Turing ES via its **REST API**. The API receives the request, validates it against the target Semantic Navigation Site configuration, and creates an indexing job that is queued internally via Apache Artemis for asynchronous processing.

The **Semantic Navigation Site** is the central configuration artifact that drives the entire indexing behavior: it defines which Solr instance to use, which fields the documents carry, how those fields are mapped and used (title, text, URL, date, image, facets, etc.), how search will behave, and which spotlights are configured. The indexing pipeline reads this configuration to know exactly what to do with each incoming document.

Key indexing concepts

REST API as entry point: Dumont DEP connectors send documents to Turing ES via REST API, not by writing directly to Solr or the queue. The API is the single integration point — it validates the request, loads the target SN Site configuration, and enqueues an indexing job via Apache Artemis for asynchronous processing.

SN Site as the indexing blueprint: Every indexing job is bound to a Semantic Navigation Site. The site configuration defines the complete indexing contract: which Solr collection to write to, which document fields exist and how they are typed, which fields become facets, how highlighting and autocomplete will work, and which spotlights are active. The pipeline reads this configuration at job execution time to determine field mappings, facet assignments, and spotlight handling.

Spotlight persistence: When an incoming document matches a configured Spotlight — for example, its URL or ID matches a spotlight term — the pipeline indexes the document in Solr normally and also persists the spotlight content (title, description, URL, position) in the relational database. This ensures the spotlight data remains available for injection even if the document is later removed from the Solr index.

Viglet Dumont DEP: The connector system that feeds Turing ES. It runs as a separate application and manages its own connector lifecycle (schedules, credentials, field mappings). Connectors currently available in Dumont DEP include WebCrawler (Nutch-based), Database, FileSystem, AEM, and WordPress. Refer to the Dumont DEP documentation for connector configuration.

Merge Providers: When two Dumont DEP connectors independently index different representations of the same real-world document — for example, AEM indexing structured metadata from `model.json` and WebCrawler indexing the rendered HTML of the same page — the

Merge Provider identifies them as the same document using a configured join key and merges their fields before writing to Solr. See [Semantic Navigation](#) for a detailed explanation.

Embedding stores: If Generative AI is enabled for an SN Site, a vector embedding is generated for each indexed document and written to the configured embedding store. Turing ES supports three embedding backends via Spring AI: **ChromaDB**, **PgVector** (PostgreSQL extension), and **Milvus**. Only one is active per deployment. The default embedding store and embedding model are defined globally in **Administration → Settings**.

MinIO asset indexing: Turing ES includes an **Assets** file manager in the admin console, backed by MinIO as the object storage layer. Files are uploaded via drag-and-drop, organized into folders, and automatically indexed as vector embeddings on upload (and unindexed on deletion). A batch "Train AI with Assets" operation processes all files using Apache Tika for text extraction, chunking at 1,024 characters, and storing embeddings in the active vector store.

Application logs in MongoDB: When MongoDB is configured, Turing ES ships with a custom Logback appender that extends `ch.qos.logback`. Every log entry generated by the application — including indexing events, search requests, errors, and system events — is persisted to MongoDB in addition to standard output. These logs are exposed in the admin console, giving administrators full visibility into application behavior without requiring access to the server file system or a separate log management tool.

Search Flow

The search flow is synchronous and request-driven. Every request goes through a structured pipeline before a response is returned to the client.

Key search concepts

Site-scoped execution: Every search request is scoped to a named Semantic Navigation Site. The site configuration defines which search engine backend to use, how many results per page, facet definitions, field mappings, whether Spotlights and Targeting Rules are active, and whether GenAI is enabled.

Plugin abstraction: The orchestrator does not query the search backend directly. An intermediate plugin layer translates the abstract search context into backend-specific queries, supporting **Apache Solr**, **Elasticsearch**, and **Lucene** as backends. Solr is the recommended production backend as it provides the most complete feature set, including full support for facets, spotlights, targeting rules, highlighting, and autocomplete. Elasticsearch and Lucene are available as alternatives with a reduced feature set.

Spotlight injection: After retrieving organic results, the system checks whether any configured Spotlights match the current query. Matching Spotlights insert curated documents at specific positions in the result list. See [Semantic Navigation](#) for details.

Targeting Rules: Results are filtered based on the requesting user's profile attributes, passed in the request context. Targeting Rules translate those attributes into additional Solr filter queries. See [Semantic Navigation](#) for details.

Field mapping and highlighting: Before returning results, each document's fields are remapped to a canonical set of display fields configured per site (title, description, text, date, image, URL). Highlighting wraps matched terms with configurable HTML tags (default: `<mark>`).

Metrics: After assembling the response, query metrics (search term, result count, site, timestamp) are logged asynchronously to avoid adding latency to the response.

Technology Stack

LAYER	TECHNOLOGY	NOTES
Runtime	Java 21	Minimum supported version
Framework	Spring Boot + Spring AI	Application container; Spring AI powers LLM and embedding integrations
Search Engine	Apache Solr (recommended), Elasticsearch, Lucene	Solr is the primary production backend with the most complete feature set; Elasticsearch and Lucene are supported as alternatives
Solr Coordination	Apache Zookeeper	Required for Solr in production (SolrCloud mode)
Message Broker	Apache Artemis	Asynchronous indexing queue (embedded in Turing ES)
Database	H2 / MariaDB / MySQL / PostgreSQL	H2 for development; MariaDB or MySQL recommended for production
Embedding Stores	ChromaDB / PgVector / Milvus	Via Spring AI; one backend active per deployment
Asset Store	MinIO	External service; configured in <code>application.yaml</code> (host, user, password); object storage for files managed via the admin console folder UI
Log Store	MongoDB	Optional; custom Logback appender (<code>ch.qos.logback</code>) persists all application logs to MongoDB, accessible via the admin console
LLM Providers	Anthropic Claude, OpenAI, Azure OpenAI, Google Gemini, Gemini (OpenAI-compatible), Ollama	Configured per site or globally; one active at a time

LAYER	TECHNOLOGY	NOTES
Tool Calling	27 native tools across 7 categories + MCP (external servers)	Semantic Nav (15), RAG/KB (4), Web Crawler (2), Finance (2), Weather (1), Image Search (1), DateTime (1), Code Interpreter (1); MCP via HTTP or stdio
Identity Management	Keycloak	OAuth2 / OpenID Connect; optional for deployments without SSO
Load Balancer	Apache HTTP Server	Optional; required for high-availability cluster deployments
Connector System	Viglet Dumont DEP	Separate application; sends documents to Turing ES via REST API, which queues them internally to Artemis
Build System	Apache Maven	Multi-module project
Frontend	React + TypeScript + shadcn/ui + Vite	Admin console (<code>turing-react</code>) — includes SN configuration, <code>Assets</code> manager, <code>Chat</code> interface, <code>Token Usage</code> dashboard
Containerization	Docker / Docker Compose	Available in <code>containers/</code> directory
Orchestration	Kubernetes	Manifests available in <code>k8s/</code> directory
API Protocols	REST + GraphQL	Swagger UI available in development mode
Java SDK	<code>turing-java-sdk</code>	Available on Maven Central; typed client for search and indexing
JavaScript SDK	<code>@viglet/turing-sdk</code>	Available on npm; TypeScript-ready client for web and Node.js

Deployment Topologies

Turing ES supports several deployment configurations. Each builds on the previous one — start with what you need and add components as requirements grow.

Development

Minimal setup for local development and evaluation.

```
Turing ES (H2 embedded) + Apache Solr
```

Turing ES starts with an embedded H2 database. No external database is needed. Not suitable for production.

Simple Production

Recommended baseline for production environments.

```
Turing ES + Apache Solr + Zookeeper + MariaDB / MySQL
```

Solr runs in SolrCloud mode coordinated by Zookeeper, enabling index replication and fault tolerance at the Solr layer. MariaDB or MySQL provides durable persistence for Turing ES configuration and metadata.

Production with Security (SSO)

For environments that require integration with an identity provider or corporate SSO.

```
Turing ES + Apache Solr + Zookeeper + MariaDB / MySQL + Keycloak
```

Keycloak handles authentication via OAuth2 / OpenID Connect. Users log in through Keycloak and receive tokens that are validated by Turing ES. See [Security & Keycloak](#) for configuration details.

Production with Log UI

For environments where administrators need visibility into application behavior directly from the Turing ES admin console — without access to server logs or external log tools.

```
Turing ES + Apache Solr + Zookeeper + MariaDB / MySQL + MongoDB
```

When MongoDB is configured, a custom Logback appender persists every log entry generated by the application to MongoDB. The admin console exposes these logs in a searchable interface, showing errors, warnings, indexing events, and system activity in real time.

High Availability

For environments requiring horizontal scaling and zero-downtime deployments.

```
Apache HTTP Server (load balancer)
├─ Turing ES node 1
├─ Turing ES node 2
└─ Turing ES node N
Apache Solr + Zookeeper (cluster)
MariaDB / MySQL (primary + replica)
Keycloak (optional)
MongoDB (optional)
```

Multiple Turing ES instances run behind Apache HTTP Server configured as a reverse proxy and load balancer. Solr runs as a multi-node SolrCloud cluster. The database should be configured with at least one replica for redundancy.

Related Pages

PAGE	DESCRIPTION
Installation Guide	Setup with Docker, JAR, or build from source
Configuration Reference	All application.yaml properties
Integration	Manage content connectors in the admin console
Dumont DEP — Architecture	Connector-side architecture — pipeline engine, message queue, and indexing plugins
Dumont DEP — Connectors	Available connectors and deployment types

Viglet Turing ES: Installation Guide

Viglet Turing ES (<https://viglet.org/turing>) is an open source enterprise search platform (<https://github.com/openviglet>) with Semantic Navigation and Generative AI as its main features. All content is indexed in Apache Solr as the primary search engine.

Installing Java

Turing ES requires Java.

NOTE

Turing ES only supports Java 21.

During installation, Turing ES reads the JAVA_HOME and PATH variables to install the necessary services. Therefore, you must modify the variables as specified here:

On Windows:

Set the JAVA_HOME system variable to point to the JAVA location.

Attach the PATH system variable with `%JAVA_HOME%/bin`.

On Unix:

Create an environment variable called JAVA_HOME.

Provide the JAVA bin path in the PATH variable.

Turing ES Requirements

The information below describes the resource modes available to run Turing ES.

NOTE

Use a screen resolution of at least 1024 x 768 for optimum clarity.

This section describes the minimum and suggested features for systems on which you can configure Turing ES.

Hardware Requirements

MINIMUM	RECOMMENDED
Single 1.5 GHz processor, 1 GB available memory	Dual 2 GHz processors with Hyper-Threading enabled, 2 GB available memory

In addition, the following is required:

- 1GB of disk space for software.

- A compatible version of a database.

Installing and Configuring Databases

Turing ES supports multiple database platforms, each with specific unique requirements to maximize effectiveness with Turing ES. The following sections describe these requirements.

Turing ES Database Table Requirements

Turing ES uses multiple tables in multiple databases (also called database schemas by some vendors such as Oracle). You must create new databases / schemas that will contain the sets of tables described below.

Oracle Database Requirements

This section describes configuration requirements for an existing database.

To set up an Oracle database:

- Install and configure a single database instance for use by Turing ES.

- Create a database schema to contain the Turing ES system tables.

- Configure the Turing ES admin user for the database. For Oracle, the database user with the grant of the CONNECT and RESOURCE roles. Make sure the user has UNLIMITED TABLESPACE and CREATE VIEW privileges.

Microsoft SQL Server Database Requirements

This section discusses the steps required to configure Microsoft SQL Server for your Turing ES databases.

When creating the Turing ES database, use the following options:

You must install the Database Engine feature and Management Tools are recommended. All other features are optional.

Set the Collation configuration variable to `SQL_Latin1_General_CP1_CS_AS` or `Latin1_General / Case-sensitive / Accent-Sensitive`.

Set authentication mode to mixed mode.

Turing ES system user cannot be sa.

Configuring the SQL Server Database and Schemas

NOTE

The database and usernames provided are examples only.

To create the SQL Server database:

Select Databases > New Database.

Enter the database name (eg turing) and the size of the data and log files according to the environment.

Go to the Options page and set the following values:

Collation = `Latin1_General_CS_AS` or `SQL_Latin1_General_CP1_CS_AS`

Miscellaneous > Allow Snapshot Isolation = True

Miscellaneous > Is Read Committed Snapshot On = True

```
USE master
GO
ALTER DATABASE turing
SET SINGLE_USER
WITH ROLLBACK IMMEDIATE
GO
ALTER DATABASE turing
SET ALLOW_SNAPSHOT_ISOLATION ON
GO
ALTER DATABASE turing
SET READ_COMMITTED_SNAPSHOT ON
GO
ALTER DATABASE turing
SET MULTI_USER
GO
```

Under Security > Logins, select New Login and set the following attributes:

Enter the Login name (for example, turing)
Deselect the Enforce Password Expiration option.
Set the Default database.

On the User Mapping page, set the following attributes for the turing database:

Default user = turing
Default schema = turing
With the turing database is selected, select the option for db_owner

Click OK.

PostgreSQL Database Requirements

This section discusses the steps required to configure PostgreSQL for your Turing ES databases:

- Installing and configuring the PostgreSQL Database
- Configuring the Turing ES user, database, and schema

Installing and configuring the PostgreSQL Database

This section describes the steps required to install and configure a PostgreSQL database for your Turing ES environment. You will also need to configure the Turing user, database and schemas.

To install the PostgreSQL database:

Navigate to the following URL and follow the instructions provided for your supported version of PostgreSQL: <https://www.postgresql.org/docs/manuals/>

To configure the PostgreSQL instance:

Open the `postgresql.conf` file (located in `<PostgreSQL_installDir>/<version>/data/`) and set the following parameters:

```
max_connections = 10
max_prepared_transactions = 10
shared_buffers = 512MB
autovacuum = off
```

! IMPORTANT

The autovacuum daemon is used to reclaim memory/storage space the database is no longer using. You must re-enable this utility after you have finished installing and configuring Turing ES.

i NOTE

Configuration values will vary for higher transaction environments and numbers of users.

`max_prepared_transactions` should be at least as large as `max_connections` so that every session can have a prepared transaction pending.

Configuring the Turing user, database, and schemas

i NOTE

You must be a PostgreSQL super-user or have createuser privileges to create Turing users. To create and configure the Turing ES database, you must be a super-user or have create privileges.

! IMPORTANT

When you create a database table in PostgreSQL, the field name must be lowercase.

To configure the Turing ES user:

Create a user as the owner of the Turing ES database:

```
CREATE USER turing PASSWORD <password> with createuser;
```

Create a Turing ES database:

```
CREATE DATABASE turing_db;  
ALTER DATABASE turing_db OWNER TO turing;
```

To create and access the Turing ES user and schema:

Log in to the turing (database) using the turing account:

```
$psql -U turing -d turing_db
```

Create the Turing schema and grant ownership to the Turing user:

```
CREATE SCHEMA turing;  
ALTER SCHEMA turing OWNER TO turing;
```

Set the PostgreSQL search_path variable:

```
ALTER USER turing SET SEARCH_PATH = turing;
```

Turing Utils

First you need install Turing Utils, that contains sample configurations and script to facilitate the use of Turing ES, so go to <https://viglet.org/turing/download/> and click on "Integration > Utils" link to download it.

Extract the turing-utils.zip to `/appl/viglet/turing/utils`:

```
mkdir -p /appl/viglet/turing/utils  
unzip turing-utils.zip -d /appl/viglet/turing/utils
```

Keycloak

For production deployments that require SSO, Keycloak integrates with Turing ES via OAuth2 / OpenID Connect. Full setup instructions — including database creation, keystore generation, Keycloak configuration, realm and client setup, JVM properties, and Apache reverse proxy configuration — are covered in the [Security & Keycloak](#) guide.

Solr Configuration

You need to configure Solr to be able to store Semantic Navigation data.

With Solr running, go to Turing Utils and create the Solr collection that will be used by Turing ES:

```
cd <SOLR_DIR>/bin
./solr create_collection -c turing -n turing -d
/appl/viglet/turing/utils/solr/<VERSION>/turing/<LANGUAGE>
```

Start the Solr service after the collection is created.

Installing Turing ES

Turing ES Download

Option 1 — Docker (fastest)

```
docker pull openviglet/turing:2026.2
docker run -p 2700:2700 openviglet/turing:2026.2
```

Option 2 — JAR download

Go to <https://viglet.org/turing/download/> and click on "Download Turing ES" button to download `viglet-turing.jar`.

Copy the `viglet-turing.jar` to `/appl/viglet/turing/server`:

```
mkdir -p /appl/viglet/turing/server
cp viglet-turing.jar /appl/viglet/turing/server
```

Option 3 — Build from source

```
git clone https://github.com/openviglet/turing-ce.git
cd turing
mvn clean package -pl turing-app
cp turing-app/target/viglet-turing.jar /appl/viglet/turing/server/
```

Database

By default, Turing uses H2 as its embedded database, but if you prefer another database you can create an external properties file and reference it at startup using `--spring.config.additional-location`.

Create `/appl/viglet/turing/server/viglet-turing.properties`:

MySQL

```
spring.datasource.url=jdbc:mariadb://localhost:3306/turing
spring.datasource.username=turing
spring.datasource.password=turing
spring.datasource.driver-class-name=org.mariadb.jdbc.Driver
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL5InnoDBDialect
```

Oracle Database

```
spring.datasource.url=jdbc:oracle:thin:@localhost:1521/turing
spring.datasource.username=turing
spring.datasource.password=turing
spring.datasource.driver-class-name=oracle.jdbc.OracleDriver
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.Oracle10gDialect
```

PostgreSQL

```
spring.datasource.url=jdbc:postgresql://localhost:5432/turing
spring.datasource.username=turing
spring.datasource.password=turing
spring.datasource.driver-class-name=org.postgresql.Driver
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.PostgreSQL94Dialect
```

Then start Turing ES referencing the properties file:

```
java -Xmx1g -Xms1g -jar viglet-turing.jar \
  --spring.config.additional-location=file:/appl/viglet/turing/server/viglet-turing.properties
```

Creating Turing ES Service on Linux

As root, create a `/etc/systemd/system/turing.service` file with the following lines:

```
[Unit]
Description=Viglet Turing ES
After=syslog.target

[Service]
User=turing
Group=turing
WorkingDirectory=/appl/viglet/turing/server
ExecStart=/appl/java/jdk21/bin/java -Xmx1g -Xms1g \
  -jar /appl/viglet/turing/server/viglet-turing.jar \
  --spring.config.additional-location=file:/appl/viglet/turing/server/viglet-
turing.properties
SuccessExitStatus=143

[Install]
WantedBy=multi-user.target
```

SPRING BOOT 4

Starting with Spring Boot 4, JAR files are no longer directly executable (`./viglet-turing.jar`). You must launch Turing ES using `java -jar` explicitly. The `.conf` file pattern used in older versions is no longer supported — use `--spring.config.additional-location` to load external properties instead.

Enable and start the service:

```
systemctl daemon-reload
systemctl enable turing.service
systemctl start turing.service
```

Other useful commands:


```
systemctl stop turing.service
systemctl restart turing.service
systemctl status turing.service
```

Starting Turing ES

When Turing ES is started for the first time, it will perform initial setup.

```
$ java -jar viglet-turing.jar  
  
|_| |_| | _ _ _ _ _ ( ) _ _ _ _ _ | |_| / |_|  
| | | | | | | | | | \ / _ \ | | | \ _ \  
|_| \ , _ ||_| | |_|_|_|_| \_, | |_|_|_|_/ 2026.2.4  
      |_/_/  
  
:: Copyright (C) 2016-2026 Viglet Turing Enterprise Search  
  
:: Built with Spring Boot :: 3.5.0  
  
First Time Configuration ...  
Configuration finished.
```

Accessing the Turing Console

Turing provides remote access to administration, configuration, and management through its Web application interfaces. Once setup is complete, the Console become browser-accessible through the following URL: `http://<host>:<port>/console` where `<host>:<port>` is the listening host and port for the Turing ES. The default port is **2700**.

The default username is **admin**. The password is set via the `TURING_ADMIN_PASSWORD` environment variable before first startup — see the [Administration Guide](#) for details.

Appendix A: Installation Modes

Turing ES Server

Simple

Minimal setup using an embedded H2 database. Suitable for local development and evaluation only — not for production.

Prerequisites:

- Linux server
- Java 21
- 50 GB HDD
- 2 GB RAM

Target Audience: Development and testing environments.

Docker Compose

Turing ES and its dependencies installed via Docker Compose, including:

- MariaDB — Turing ES system tables
- Solr — Semantic Navigation search backend
- Nginx — reverse proxy on port 80
- Turing ES

Prerequisites:

- Linux server with Docker and Docker Compose installed
- 50 GB HDD
- 4 GB RAM

Target Audience: Teams that want a complete environment without manual service installation. Suitable for QA and production.

Kubernetes

Turing ES and its dependencies deployed via Kubernetes manifests (available in the [k8s/](#) directory), including:

MariaDB — Turing ES system tables

Solr — Semantic Navigation search backend

Nginx — reverse proxy

Turing ES

Prerequisites:

Linux server with Kubernetes, or a cloud Kubernetes service (GKE, EKS, AKS)

100 GB storage

4 GB RAM

Target Audience: Cloud deployments requiring horizontal scaling and infrastructure-as-code.

Manual Installation

Each service installed individually following this guide:

MariaDB — Turing ES system tables

Solr + Zookeeper — Semantic Navigation search backend

Apache HTTP Server — reverse proxy (required for Keycloak integration)

Turing ES

Prerequisites:

One to four Linux servers

50-100 GB storage per server

Minimum 2 GB RAM per server

Connectors

Content ingestion is handled by **Viglet Dumont DEP**, a separate application. Available connectors include WebCrawler (Apache Nutch), Database, FileSystem, AEM, and WordPress. Refer to the [Dumont DEP documentation](#) for connector setup.

Prerequisites:

A server with access to the content sources to be indexed

50 GB storage

Appendix B: Docker Compose

You can start the Turing ES using MariaDB, Solr and Nginx.

```
./gradlew turing-app:build -x test -i --stacktrace  
docker-compose up
```

NOTE

If you have problems with permissions on directories, run `chmod -R 777 volumes`

Docker Commands

Turing

```
docker exec -it turing /bin/bash
```

Solr

```
docker exec -it turing-solr /bin/bash
```

Check logs:

```
docker-compose exec turing-solr cat /opt/solr/server/logs/solr.log  
# or  
docker-compose exec turing-solr tail -f /opt/solr/server/logs/solr.log
```

MariaDB

```
docker exec -it turing-mariadb /bin/bash
```

Nginx

```
docker exec -it turing-nginx /bin/bash
```

Configuration Reference

This page documents every significant property in the Turing ES `application.yaml` file. The file uses standard Spring Boot configuration — any property can be overridden via environment variables, JVM system properties (`-Dkey=value`), or a separate `application-production.yaml` file in the working directory.

OVERRIDE PATTERN

To override a property at runtime without editing the file, use the environment variable convention: replace `.` and `-` with `_` and uppercase everything. For example, `turing.storage.type` becomes `TURING_STORAGE_TYPE=minio`.

Full Default Configuration

```
spring:
  datasource.hikari:
    maximum-pool-size: 20
    minimum-idle: 5
    idle-timeout: 600000
    pool-name: TurConnectionPool
  jmx.enabled: true
  profiles:
    active: production
  h2:
    console:
      enabled: false
      path: /h2
      settings:
        web-allow-others: true
    datasource:
      url:
jdbc:h2:file:./store/db/turingDB;DATABASE_TO_UPPER=false;CASE_INSENSITIVE_IDENTIFIERS=true
    username: sa
    password: ""
    driver-class-name: org.h2.Driver
  liquibase:
    change-log: classpath:db/changelog/db.changelog-master.yaml
    enabled: true
    default-schema: PUBLIC
  jpa:
    properties:
      hibernate:
        "[globally_quoted_identifiers]": true
        "[format_sql]": false
        "[enable_lazy_load_no_trans]": true
        "[generate_statistics]": false
        "[jakarta.persistence.sharedCache.mode]": ALL
    hibernate:
      ddl-auto: none
      show-sql: false
  jackson:
    mapper:
      "[DEFAULT_VIEW_INCLUSION]": true
  thymeleaf:
    mode: HTML
    check-template: true
    check-template-location: true
    prefix: classpath:public/
    suffix: .html
```

```
artemis:
  mode: embedded
  broker-url: localhost:61616
  embedded:
    enabled: true
    persistent: true
    data-directory: store/queue
    queues: indexing.queue
  pool:
    max-connections: 10
jms:
  template:
    default-destination: indexing.queue
servlet:
  multipart:
    max-file-size: 1024MB
    max-request-size: 1024MB
mvc:
  async:
    request-timeout: 3600000
output:
  ansi:
    enabled: always
graphql:
  graphiql:
    enabled: true
    path: /graphiql
  http.path: /graphql
  cors:
    allowed-origins: ${turing.allowedOrigins}
    allowed-methods: GET,POST
    allowed-headers: "*"

server:
  port: ${PORT:2700}
  compression:
    enabled: true
  mime-types: application/json,text/css,application/javascript
  min-response-size: 2048
tomcat:
  accesslog:
    enabled: false
    suffix: .log
    prefix: access_log
    file-date-format: .yyyy-MM-dd
    directory: logs
  basedir: store
  threads:
    max: 600
  max-swallow-size: 200MB
```



```
max-http-form-post-size: 200MB
maxHttpResponseHeaderSize: 800KB

turing:
  ai.crypto.key: sample-key-for-crypto
  allowedOrigins: http://localhost:4200,http://localhost:5173
  multi-tenant: false
  keycloak: false
  url: http://localhost:2700
  open-browser: true
  jms.concurrency: 1-1
  code-interpreter:
    python-executable: ""
  solr:
    timeout: 30000
    cloud: false
    commit:
      within: 10000
      enabled: false
  elasticsearch:
    timeout: 30000
  search:
    metrics.enabled: true
    cache:
      ttl:
        seconds: 86400000
        enabled: false
  storage:
    type: none
    filesystem:
      path: ./store/assets
    minio:
      endpoint: http://localhost:9000
      access-key: admin
      secret-key: minha_senha_forte
      bucket: turing-assets
  mongodb:
    enabled: false
    uri: mongodb://localhost:27017
    logging:
      database: turingLog
      collection:
        server: server
        aem: aem
        indexing: indexing
      purge:
        days: 30

springdoc:
  pathsToMatch: /api/**
```

```
swagger-ui:
  path: /swagger-ui.html

management:
  endpoints:
    web:
      exposure:
        include: "*"

logging:
  config: classpath:logback-spring.xml
  level:
    org:
      springframework: INFO
      apache: INFO
      "[apache.activemq]": ERROR
    com:
      viglet: INFO
    dev:
      langchain4j: INFO
      ai4j:
        openai4j: INFO
  file:
    name: store/logs/turing.log
  logback:
    rollingpolicy:
      max-file-size: 25MB
      max-history: 10
```

Property Reference

Server

PROPERTY	DEFAULT	DESCRIPTION
<code>server.port</code>	<code>\${PORT:2700}</code>	HTTP port. Override with the <code>PORT</code> environment variable.
<code>server.compression.enabled</code>	<code>true</code>	Enables gzip compression for JSON, CSS, and JavaScript responses
<code>server.compression.min-response-size</code>	<code>2048</code>	Minimum response size in bytes before compression is applied
<code>server.tomcat.threads.max</code>	<code>600</code>	Maximum number of Tomcat worker threads
<code>server.tomcat.basedir</code>	<code>store</code>	Base directory for Tomcat internal files
<code>server.tomcat.max-swallow-size</code>	<code>200MB</code>	Maximum size of request body Tomcat will absorb
<code>server.tomcat.max-http-form-post-size</code>	<code>200MB</code>	Maximum size of HTML form POST bodies
<code>server.tomcat.maxHttpResponseBodySize</code>	<code>800KB</code>	Maximum size of HTTP response headers
<code>server.tomcat.accesslog.enabled</code>	<code>false</code>	Enable Tomcat access log file

Database (Spring Datasource)

The default database is **H2** (embedded, file-based) — suitable for development and simple deployments. For production, replace with MariaDB, MySQL, or PostgreSQL.

PROPERTY	DEFAULT	DESCRIPTION
<code>spring.datasource.url</code>	<code>jdbc:h2:file:./storage/db/turingDB</code>	JDBC URL. Change to your production DB URL.
<code>spring.datasource.username</code>	<code>sa</code>	Database username
<code>spring.datasource.password</code>	<code>""</code>	Database password
<code>spring.datasource.driver-class-name</code>	<code>org.h2.Driver</code>	JDBC driver class
<code>spring.datasource.hikari.maximum-pool-size</code>	<code>20</code>	Maximum connections in the HikariCP pool
<code>spring.datasource.hikari.minimum-idle</code>	<code>5</code>	Minimum idle connections kept alive
<code>spring.datasource.hikari.idle-timeout</code>	<code>600000</code>	Time (ms) a connection can remain idle before being removed
<code>spring.h2.console.enabled</code>	<code>false</code>	Enable H2 web console — do not enable in production

Switching to MariaDB / MySQL:

```
spring:
  datasource:
    url: jdbc:mariadb://localhost:3306/turingdb
    username: turing
    password: your_password
    driver-class-name: org.mariadb.jdbc.Driver
```

Message Queue (Apache Artemis)

Apache Artemis runs **embedded inside Turing ES** by default — no separate installation is required. For high-availability or multi-node deployments, you can switch to an external Artemis broker.

PROPERTY	DEFAULT	DESCRIPTION
<code>spring.artemis.mode</code>	<code>embedded</code>	<code>embedded</code> (default) or <code>native</code> (external broker)
<code>spring.artemis.broker-url</code>	<code>localhost:61616</code>	Used when <code>mode: native</code> to point to an external broker
<code>spring.artemis.embedded.enabled</code>	<code>true</code>	Enables the embedded broker
<code>spring.artemis.embedded.persistent</code>	<code>true</code>	Persist queue messages to disk (survives restarts)
<code>spring.artemis.embedded.data-directory</code>	<code>store/queue</code>	Directory for persisted queue data
<code>spring.artemis.embedded.queues</code>	<code>indexing.queue</code>	Queue names to create on startup
<code>spring.artemis.pool.max-connections</code>	<code>10</code>	Maximum JMS connection pool size
<code>spring.jms.template.default-destination</code>	<code>indexing.queue</code>	Default JMS destination for indexing jobs
<code>turing.jms.concurrency</code>	<code>1-1</code>	JMS listener concurrency range (min-max consumers). Increase for higher indexing throughput.



EXTERNAL ARTEMIS FOR SCALABILITY

To use an external Artemis broker (e.g., for multi-node deployments), set `spring.artemis.mode: native` and point `broker-url` to your broker. The embedded broker should be disabled in that case (`spring.artemis.embedded.enabled: false`).

File Uploads

PROPERTY	DEFAULT	DESCRIPTION
<code>spring.servlet.multipart.max-file-size</code>	<code>1024MB</code>	Maximum size of a single uploaded file
<code>spring.servlet.multipart.max-request-size</code>	<code>1024MB</code>	Maximum total size of a multipart request

GraphQL

PROPERTY	DEFAULT	DESCRIPTION
<code>spring.graphql.http.path</code>	<code>/graphql</code>	GraphQL endpoint path
<code>spring.graphql.graphiql.enabled</code>	<code>true</code>	Enable GraphiQL browser IDE
<code>spring.graphql.graphiql.path</code>	<code>/graphiql</code>	GraphiQL IDE path
<code>spring.graphql.cors.allowed-origins</code>	<code>\${turing.allowedOrigins}</code>	CORS allowed origins for GraphQL

Async / Timeout

PROPERTY	DEFAULT	DESCRIPTION
<code>spring.mvc.async.request-timeout</code>	<code>3600000</code>	Maximum time (ms) for async requests — 1 hour. Covers streaming LLM responses.

Turing ES Core

PROPERTY	DEFAULT	DESCRIPTION
<code>turing.url</code>	<code>http://localhost:2700</code>	Public base URL of this Turing ES instance
<code>turing.allowedOrigins</code>	<code>http://localhost:4200, http://localhost:5173</code>	CORS allowed origins for the REST API and GraphQL. Add your frontend URL here.
<code>turing.keycloak</code>	<code>false</code>	Set <code>true</code> to enable Keycloak OAuth2/OIDC. See Security & Keycloak .
<code>turing.multi-tenant</code>	<code>false</code>	Enable multi-tenant mode
<code>turing.open-browser</code>	<code>true</code>	Automatically open the admin console in the browser on startup
<code>turing.ai.crypto.key</code>	<code>sample-key-for-crypto</code>	Encryption key for stored AI provider credentials. Change this in production.
<code>turing.code-interpreter.python-executable</code>	<i>(auto-detected)</i>	Absolute path to the Python 3 binary used by the Code Interpreter GenAI tool. When blank, Turing searches standard OS locations automatically.

CHANGE THE CRYPTO KEY

The `turing.ai.crypto.key` is used to encrypt LLM provider API keys stored in the database. Always set a strong, unique value in production.

Solr

PROPERTY	DEFAULT	DESCRIPTION
<code>turing.solr.timeout</code>	<code>30000</code>	Connection and read timeout (ms) for Solr requests
<code>turing.solr.cloud</code>	<code>false</code>	Set <code>true</code> for SolrCloud mode with Zookeeper
<code>turing.solr.commit.within</code>	<code>10000</code>	Soft commit time (ms) — Solr commits changes within this window
<code>turing.solr.commit.enabled</code>	<code>false</code>	Enable explicit commits after indexing. Leave <code>false</code> to use Solr's auto-commit.

Elasticsearch

PROPERTY	DEFAULT	DESCRIPTION
<code>turing.elasticsearch.timeout</code>	<code>30000</code>	Connection and read timeout (ms) for Elasticsearch requests

Search Cache

PROPERTY	DEFAULT	DESCRIPTION
<code>turing.search.metrics.enabled</code>	<code>true</code>	Record search metrics (query terms, result counts, timestamps)
<code>turing.search.cache.enabled</code>	<code>false</code>	Enable server-side search result caching
<code>turing.search.cache.ttl.seconds</code>	<code>86400000</code> <code>0</code>	Cache TTL in seconds (default ~1000 days — effectively no expiry when enabled)

Storage (Assets, Knowledge Base & Pages)

Storage powers the **Assets** file manager, the RAG Knowledge Base, and **SPA Pages**. Disabled by default (`none`). Turing supports three pluggable storage backends:

TYPE	DESCRIPTION
<code>none</code>	Disabled — Assets and Pages are hidden from the sidebar
<code>minio</code>	MinIO object storage (recommended for production)
<code>filesystem</code>	Local filesystem directory (simple deployments, development)

Common

PROPERTY	DEFAULT	DESCRIPTION
<code>turing.storage.type</code>	<code>none</code>	Storage backend: <code>none</code> , <code>minio</code> , or <code>filesystem</code>

MinIO Backend

PROPERTY	DEFAULT	DESCRIPTION
<code>turing.storage.minio.endpoint</code>	<code>http://localhost:9000</code>	MinIO server URL
<code>turing.storage.minio.access-key</code>	<code>admin</code>	MinIO access key
<code>turing.storage.minio.secret-key</code>	<code>minha_senha_forte</code>	MinIO secret key
<code>turing.storage.minio.bucket</code>	<code>turing-assets</code>	Bucket name — created automatically on startup if it does not exist

Filesystem Backend

PROPERTY	DEFAULT	DESCRIPTION
<code>turing.storage.filesystem.path</code>	<code>./store/assets</code>	Base directory for stored files. Created automatically if it does not exist.

The filesystem backend includes **path traversal protection** — all paths are resolved against the base directory and any attempt to escape it is blocked.

NOTE

The **Assets** and **Pages** sections only appear in the sidebar when `turing.storage.type` is set to `minio` or `filesystem`.

MongoDB (Application Logs)

MongoDB is used to persist application logs, making them accessible from the admin console. Disabled by default.

PROPERTY	DEFAULT	DESCRIPTION
<code>turing.mongodb.enabled</code>	<code>false</code>	Set <code>true</code> to enable MongoDB log persistence
<code>turing.mongodb.uri</code>	<code>mongodb://localhost:27017</code>	MongoDB connection URI
<code>turing.mongodb.logging.database</code>	<code>turingLog</code>	Database name for logs
<code>turing.mongodb.logging.collection.server</code>	<code>server</code>	Collection for general server logs
<code>turing.mongodb.logging.collection.aem</code>	<code>aem</code>	Collection for AEM-specific logs
<code>turing.mongodb.logging.collection.indexing</code>	<code>indexing</code>	Collection for indexing pipeline logs
<code>turing.mongodb.logging.purge.days</code>	<code>30</code>	Automatically delete log entries older than this many days

API Documentation (SpringDoc)

PROPERTY	DEFAULT	DESCRIPTION
<code>springdoc.pathsToMatch</code>	<code>/api/**</code>	API paths included in the OpenAPI spec
<code>springdoc.swagger-ui.path</code>	<code>/swagger-ui.html</code>	Swagger UI path

Logging

PROPERTY	DEFAULT	DESCRIPTION
<code>logging.file.name</code>	<code>store/logs/turing.log</code>	Log file path
<code>logging.logback.rollingpolicy.max-file-size</code>	<code>25MB</code>	Maximum size of each log file before rotation
<code>logging.logback.rollingpolicy.max-history</code>	<code>10</code>	Number of rotated log files to keep
<code>logging.level.com.viglet</code>	<code>INFO</code>	Log level for Turing ES application code
<code>logging.level.org.springframework</code>	<code>INFO</code>	Log level for Spring framework
<code>logging.level.dev.langchain4j</code>	<code>INFO</code>	Log level for LangChain4j (LLM library)

Keycloak (OAuth2 / OIDC)

When `turing.keycloak: true`, uncomment and configure the following block:

```
spring:
  security:
    oauth2:
      client:
        registration:
          keycloak:
            client-id: demo-app
            client-secret: your-client-secret
            scope: openid
            authorization-grant-type: authorization_code
            redirect-uri: http://localhost:2700/login/oauth2/code/demo-app
        provider:
          keycloak:
            issuer-uri: http://localhost:8080/realms/demo
            jwk-set-uri: http://localhost:8080/realms/demo/protocol/openid-
connect/certs
            user-info-uri: http://localhost:8080/realms/demo/protocol/openid-
connect/userinfo
          resourceserver:
            jwt:
              issuer-uri: http://localhost:8080/realms/demo
```

For full Keycloak setup instructions, see [Security & Keycloak](#).

GenAI / LLM (Database Settings)

LLM providers, models, and GenAI behavior are managed through the **Administration Console** and stored in the database — not in `application.yaml`. The following settings are available via **Settings > Global Settings** in the admin UI:

SETTING	DEFAULT	DESCRIPTION
Default LLM	<i>(none)</i>	Global default LLM instance used when a Semantic Navigation site does not specify one
LLM Cache Enabled	false	Cache LLM responses to reduce API calls for repeated queries
LLM Cache TTL (ms)	3600000 (1 hour)	How long cached LLM responses remain valid
LLM Cache Regenerate	false	When true, regenerate cached responses in the background
RAG Enabled	false	Enable Retrieval-Augmented Generation globally
Default Embedding Model	<i>(none)</i>	Global default embedding model for vector operations
Default Embedding Store	<i>(none)</i>	Global default embedding store instance
Python Executable	<i>(auto-detected)</i>	Path to Python 3 for the Code Interpreter tool (also overridable via turing.code-interpreter.python-executable in YAML)

Supported LLM providers: OpenAI, Ollama, Anthropic (Claude), Google Gemini, Google Gemini (OpenAI-compatible), and Azure OpenAI.

LLM instances are created under **GenAI > LLM** in the admin console, where you configure the provider, API key, base URL, and model name.

Embedding Store (Database Settings)

Embedding store instances are configured through the **Administration Console** under **GenAI > Embedding Store**. Each instance specifies a vendor, connection URL, credentials, and optional provider-specific options via a JSON field.

Supported vendors:

VENDOR	PLUGIN ID	KEY PROVIDER OPTIONS
ChromaDB	chroma	baseUrl (default http://localhost:8000), collectionName , tenantName , databaseName , initializeSchema , keyToken , basicUsername , basicPassword
Milvus	milvus	baseUrl , collectionName , databaseName , token , embeddingDimension , metricType , indexType , indexParameters , initializeSchema
PGVector	pgvector or	JDBC URL (set as instance URL), tableName (default vector_store), schemaName (default public), dimensions , distanceType , indexType , initializeSchema

Provider options are passed as a JSON object in the **Provider Options** field when creating or editing an embedding store instance.

Common Production Overrides

A minimal production override file (`application-production.yaml`) covering the most important changes:

```
spring:
  profiles:
    active: production
  datasource:
    url: jdbc:mariadb://db-host:3306/turingdb
    username: turing
    password: strong_db_password
    driver-class-name: org.mariadb.jdbc.Driver
  h2:
    console:
      enabled: false

turing:
  url: https://search.yourcompany.com
  allowedOrigins: https://search.yourcompany.com
  ai.crypto.key: your-strong-random-key-here
  storage:
    type: minio
    minio:
      endpoint: http://minio:9000
      access-key: your-minio-user
      secret-key: your-minio-password
  mongodb:
    enabled: true
    uri: mongodb://mongo-host:27017

server:
  tomcat:
    accesslog:
      enabled: true
```


Search Engine

The **Search Engine** page (`/admin/se/instance`) manages the search backends that power Semantic Navigation Sites. It is accessible from the **Enterprise Search** section of the sidebar.

Each **Search Engine instance** is a configured connection to a search backend. Semantic Navigation Sites bind to a specific instance when their cores (collections) are created. Turing ES supports three backends via a plugin architecture: **Apache Solr** (recommended), **Apache Lucene** (embedded), and **Elasticsearch**.

Instance Listing

The page displays all configured instances as a grid of cards (title and description). Use the **"New"** button to add an instance. When no instances exist, a blank state guides you to create the first one.

Create / Edit Form

General Information

FIELD	REQUIRED	DESCRIPTION
Title	✓	Display name for this instance — shown in SN Site dropdowns
Description		Free-text notes about this instance

Connection Settings

FIELD	REQUIRED	DESCRIPTION
Vendor	✓	Backend type: <code>SOLR</code> , <code>LUCENE</code> , or <code>ES</code>
Endpoint URL	✓	Connection URL for the backend service

Default endpoint URLs per vendor:

VENDOR	DEFAULT ENDPOINT URL
SOLR (Apache Solr)	<code>http://localhost:8983/solr</code>
LUCENE (embedded Apache Lucene)	<code>/data/turing/lucene</code>
ES (Elasticsearch)	<code>http://localhost:9200</code>

Instance Detail Pages

After creating an instance, its detail page exposes three navigation sections.

Settings

Editing form for all fields listed above, plus a **Delete instance** button. An instance cannot be deleted if any SN Site cores are still using it.

Cores (Collections)

Manages the search indices — called **cores** in Solr terminology — within this search engine instance. Each Semantic Navigation Site locale maps to one core.

Core Listing

The list shows all cores found in the connected backend:

COLUMN	DESCRIPTION
Name	Core identifier (e.g., <code>my-site_en_US</code>)
Documents	Number of indexed documents (<code>numDocs</code>) reported by the engine
Sites	Badges showing each SN Site and locale using this core, with a country flag for quick recognition

Cores are grouped by locale pattern: `{base}_{lang}_{COUNTRY}` (e.g., `product-docs_pt_BR`). A **search/filter** field at the top narrows the list by core name.

Core Actions

ACTION	DESCRIPTION
Delete core	Permanently removes the core and all its indexed data. Blocked if the core is in use by any SN Site — the UI shows which sites and locales are preventing deletion.
Clear documents	Erases all indexed documents from the core without removing the core itself. Useful for a clean re-index without reconfiguring site associations.

Create a New Core

FIELD	REQUIRED	DESCRIPTION
Name / Name Prefix	✓	Core name or prefix
Locale	✓	Language and country (e.g., <code>en_US</code> , <code>pt_BR</code>)
Append locale to name		Toggle — when enabled, the locale is automatically appended to the prefix, generating the canonical name (e.g., prefix <code>my-site</code> + locale <code>en_US</code> → <code>my-site_en_US</code>)

A **name preview** shows the final core name as you type, before saving.

NAMING CONVENTION

Use the `{site-name}_{lang}_{COUNTRY}` pattern consistently. This makes it easy to identify which site and locale each core belongs to when browsing multiple instances.

System Information

Live monitoring panel for the connected backend:

ITEM	DESCRIPTION
Status badge	UP (green) or DOWN (red) — real-time connectivity check
Engine version	Solr or Elasticsearch version string
Lucene version	Underlying Lucene library version
Operating system	OS name and version of the engine host
Java version	JVM version running the search engine
JVM memory	Heap usage reported by the engine
Other properties	Additional dynamic properties returned by the engine's status API

Plugin Architecture

Turing ES uses a plugin architecture to support multiple search backends behind a unified interface. The active plugin is resolved at runtime based on the vendor configured per instance. If a vendor is unrecognised, the factory falls back to Solr.

For the full interface reference — all methods across search, index management, schema management, and document operations — and instructions on implementing a new backend, see [Developer Guide → Search Engine Plugin Architecture](#).

Protections

SCENARIO	BEHAVIOUR
Delete core in use	Blocked — the UI displays a list of SN Sites and locales currently using the core
Delete instance in use	Should be avoided — removing an instance with active sites will break indexing and search for those sites

Repository-level **caching** is enabled for search engine instances to avoid repeated database reads during high-frequency searches.

Related Pages

PAGE	DESCRIPTION
Administration Guide	Full console reference
Semantic Navigation	How SN Sites use cores and search engines
Architecture Overview	Solr, Elasticsearch, and Lucene in the system architecture
Configuration Reference	Solr and Elasticsearch timeout settings in <code>application.yaml</code>

Semantic Navigation

The **Semantic Navigation** page (`/admin/sn`) manages the search experiences that power every user-facing search in Turing ES. It is accessible from the **Enterprise Search** section of the sidebar.

Each **Semantic Navigation Site** (SN Site) is a complete, independently configured search experience: it defines which content is indexed, which search engine backend is used, how results are presented, which AI features are enabled, and how the front-end should render the response. A single Turing ES instance can host multiple SN Sites.

For a conceptual overview of what SN Sites are and how they relate to the rest of the system, see [Core Concepts](#).

Site Listing

The page displays all configured SN Sites as a grid of cards (title and description). Use the **"New"** button to create a new site. When no sites exist, a blank state guides you to create the first one.

Site Configuration

Clicking a site opens its configuration, organized into tabs. Each tab covers a distinct aspect of the site's behaviour.

Settings

General identity and search engine binding for the site.

ATTRIBUTE	DESCRIPTION
Name	Identifier for the site — used in API URLs (e.g., <code>/api/sn/{Name}/search</code>)
Description	Human-readable description shown in the console
Search Engine	The Search Engine instance that stores and searches content for this site. See Search Engine
Thesaurus	Enables thesaurus-based query expansion — matching synonyms and related terms
Search Template	SPA Page used as the search interface for this site. See Search Template below

Search Template

The **Search Template** setting allows you to assign a deployed [SPA Page](#) as the custom search interface for an SN site. When configured, visiting the site's search URL serves the selected SPA instead of the default built-in search page.

SEARCH URL	BEHAVIOR
<code>GET /sn/{siteName}</code>	Serves the SPA Page selected in Search Template
<code>GET /sn/{siteName}/any/path</code>	Serves the SPA with client-side routing fallback to <code>index.html</code>

How it works:

Deploy a SPA via **Enterprise Search > Pages** (upload a ZIP with your built React/Vue/Angular app)

Open the SN site's **Settings** tab

Select the deployed page from the **Search Template** dropdown

The site's search URL (`/sn/{siteName}`) now serves your custom SPA

When no template is selected (or storage is disabled), the site uses the default built-in search interface.



BUILD WITH THE REACT SDK

Use the **Turing React SDK** (`@viglet/turing-react-sdk`) to build custom search SPAs with hooks for search, facets, pagination, and autocomplete. Deploy the built app via Pages and assign it as the Search Template — the SPA communicates with the Turing REST API on the same origin, so no CORS configuration is needed.



STORAGE REQUIRED

The Search Template dropdown only appears when a **storage backend** is configured. If no SPA Pages have been deployed yet, the dropdown shows a message indicating no pages are available.

Multi Languages

Defines which locales are active for this site and maps each locale to a search engine core (collection).

ATTRIBUTE	DESCRIPTION
Language	Locale code for this language variant (e.g., <code>en_US</code> , <code>pt_BR</code>)
Core	The Solr core (or Elasticsearch index) where content for this locale is stored and searched

Each locale is indexed and searched independently. The `Accept-Language` request header or the `_setlocale` query parameter selects the active locale at search time.

Open Search button: Next to each locale row, the **Open Search** button opens the HTML search page for that locale at:

```
GET http://localhost:2700/sn/<SITE_NAME>
```

Behavior

Controls how the search engine processes queries and assembles results for this site. All settings are managed via `GET / PUT /api/sn/{id}`.

General

SETTING	DESCRIPTION
Rows per Page	Number of results returned per search request
Exact Match	When enabled, terms wrapped in quotes trigger an exact phrase match instead of a tokenized search

Wildcard

SETTING	DESCRIPTION
Wildcard No Results	Automatically appends a wildcard () to the query when the original search returns no results, broadening the match
Wildcard Always	Always appends a wildcard to every search term, regardless of result count

Use **Wildcard No Results** to recover gracefully from zero-result queries.

WILDCARD ALWAYS REDUCES PRECISION

Wildcard Always appends  to every query term regardless of result count. This increases recall — more documents match — but significantly reduces relevance precision. Avoid using it in sites where ranking quality matters. Prefer **Wildcard No Results** as a safer fallback.

Facets

SETTING	DESCRIPTION
Facet	Enables or disables faceted navigation for this site
Items per Facet	Maximum number of values displayed per facet in the result panel
Facet Sort	How facet values are ordered: by document count (COUNT) or alphabetically (ALPHABETICAL)
Facet Type	How different facets combine with each other: AND or OR . Default: AND
Facet Item Type	How values within the same facet combine with each other: AND or OR . Default: AND

Understanding Facet Type vs Facet Item Type

These two settings control **different levels** of the filter query. Think of a search where the user has selected filters from two facets:

Category facet: selected **News** and **Blog**

Year facet: selected **2024**

Facet Item Type controls how **News** and **Blog** combine (they're items **within the same facet**):

FACET ITEM TYPE	FILTER LOGIC	PRACTICAL EFFECT
OR (recommended)	category: "News" OR category: "Blog"	Show documents that are News or Blog — broadens results within this facet
AND	category: "News" AND category: "Blog"	Show documents that are News and Blog simultaneously — narrows results (often returns 0 results for single-value fields)

Facet Type controls how the Category group and the Year group combine (they're **different facets**):

FACET TYPE	FILTER LOGIC	PRACTICAL EFFECT
AND (recommended)	(category filter) AND (year filter)	Results must match both facets — narrows results progressively as more facets are selected
OR	(category filter) OR (year filter)	Results can match either facet — broadens results when more facets are selected

The four combinations

User selects: Category = [News, Blog] + Year = [2024]

Facet Type = AND, Facet Item Type = OR (recommended default)

Solr fq: (category:"News" OR category:"Blog") AND (year:2024)

"Show me News or Blog articles, but only from 2024"

→ Progressive filtering: each new facet narrows results

→ Multiple values in same facet broaden within that dimension

Facet Type = AND, Facet Item Type = AND

Solr fq: (category:"News" AND category:"Blog") AND (year:2024)

"Show me documents tagged both News AND Blog, from 2024"

→ Only useful for multi-valued fields (tags, categories)

→ Single-value fields return 0 results with multiple selections

Facet Type = OR, Facet Item Type = OR

Solr fq: (category:"News" OR category:"Blog") OR (year:2024)

"Show me anything that is News, Blog, OR from 2024"

→ Very broad: adding more facets increases results

→ Useful for discovery/exploration interfaces

Facet Type = OR, Facet Item Type = AND

Solr fq: (category:"News" AND category:"Blog") OR (year:2024)

"Show me documents tagged both News AND Blog, OR anything from 2024"

→ Uncommon combination



RECOMMENDED CONFIGURATION

For most search interfaces, use **Facet Type = AND** with **Facet Item Type = OR**. This gives users the intuitive behavior of narrowing results as they add facets, while allowing multiple selections within a single facet to broaden within that dimension. This is the standard pattern used by e-commerce sites and enterprise search portals.

Override priority

The facet operators are resolved with a three-level priority:

Request-level (highest) — the `fq0operator` and `fqItem0operator` fields in the POST body override everything

Field-level — each field can set its own `Facet Type` and `Facet Item Type` in the Fields configuration (set to `DEFAULT` to inherit from site)

Site-level (fallback) — the values configured here in the Behavior tab

This allows fine-grained control: for example, the site default can be `Facet Type = AND`, while a `tags` field overrides its `Facet Item Type` to `OR` because tags are multi-valued and should always use OR logic.

Secondary Facets

Any field configured as a facet can additionally be promoted to a **Secondary Facet** in the field's settings. Secondary facets are returned in a separate `widget.secondaryFacet` section of the search response, completely independent from the regular `widget.facet` section. This gives the front-end full control over how to render them — for example, a `content_type` field configured as a secondary facet can be rendered as navigation tabs ("All · Articles · Videos · Downloads") while the remaining facets appear as sidebar filters. The field must be a facet first; secondary facet is a promotion, not a replacement.

Highlighting

SETTING	DESCRIPTION
HL	Enables term highlighting in search result snippets
HL Pre	Opening HTML tag wrapping matched terms (e.g., <code><mark></code> or <code></code>)
HL Post	Closing HTML tag (e.g., <code></mark></code> or <code></code>)

The HL Pre and HL Post values are injected directly around matched terms in the `text` and `description` fields of each result. Your front-end application receives the pre-highlighted HTML and renders it as-is.

Spell Check

SETTING	DESCRIPTION
Spell Check	Enables spelling suggestion when the query term is not found or returns few results
Spell Check Fixes	When enabled, automatically re-runs the search using the corrected term and returns those results instead of showing a "Did you mean?" prompt

More Like This (MLT)

SETTING	DESCRIPTION
MLT	Enables the More Like This feature, which finds documents semantically similar to a given result

When MLT is enabled, each search result can be expanded to show related documents — useful for discovery experiences and content recommendation.

Spotlight

SETTING	DESCRIPTION
Spotlight with Results	When enabled, curated Spotlight documents are displayed alongside organic results. When disabled, Spotlights are shown only when the query matches a spotlight term and no organic results are returned

Default Field Mappings

These settings map the SN Site's canonical display fields to the actual field names in the Solr index. They allow the front-end application to always reference a consistent set of fields regardless of how connectors named the fields at indexing time.

SETTING	DESCRIPTION
Title	Field used as the result title
Text	Field used as the main body / snippet text
Description	Field used as the result description
Date	Field used as the result date
Image	Field used as the result thumbnail image URL
URL	Field used as the result link
Default Field	The field Solr uses for full-text search when no specific field is targeted in the query
Exact Match Field	The field used when the user searches with quoted terms (exact match mode)

Fields

Lists all fields configured for this site. Each field defines how a document attribute is indexed, searched, and presented.

Field Listing

COLUMN	DESCRIPTION
Field	Field name as it appears in the Solr schema
Enabled	Whether this field is active
MLT	Whether this field is included in More Like This queries
Facets	Whether this field is exposed as a facet filter
Secondary Facet	Whether this facet is promoted to the <code>secondaryFacet</code> section of the search response
Highlighting	Whether highlighted snippets are returned for this field

Field Detail

Clicking a field opens its detail form:

ATTRIBUTE	DESCRIPTION
Name	Field identifier — must match the Solr schema field name
Description	Purpose or contents of this field
Type	Data type: <code>INT</code> , <code>LONG</code> , <code>FLOAT</code> , <code>DOUBLE</code> , <code>CURRENCY</code> , <code>STRING</code> , <code>DATE</code> , or <code>BOOL</code>
Multi Valued	Whether the field holds an array of values
Facet Name	Display label shown in the search UI facet panel
Facet	Whether to expose this field as a filter facet
Secondary Facet	Promotes this facet to the <code>secondaryFacet</code> section of the search response (e.g., for rendering as navigation tabs)
Highlighting	Whether to return highlighted snippets for this field
MLT	Whether this field participates in More Like This queries
Enabled	Whether this field is active
Required	Whether the field must be present when indexing a document
Default Value	Value used when a document is indexed without this field



SECONDARY FACETS

A secondary facet is a regular facet that is additionally returned in a separate `secondaryFacet` section of the search response. This lets the front-end render it differently — for example, as horizontal tabs ("All · Articles · Videos") — while other facets appear as sidebar filters. The field must have **Facet** enabled first; **Secondary Facet** is a promotion, not a replacement.

Field-Level Facet Overrides

Beyond the standard settings, each field can override the site-level facet behavior:

FIELD SETTING	DESCRIPTION
Facet Type	Overrides the site-level operator between this facet and others (AND / OR)
Facet Item Type	Overrides the site-level operator between values within this facet (AND / OR)
Facet Range	Enables date range faceting for this field. Instead of listing discrete values, the facet groups documents by predefined date periods: day , month , or year

A date field configured with **Facet Range** groups results into date period buckets rather than listing individual values. The available granularities are **day**, **month**, and **year** — selected in the field configuration. The response returns one facet item per period that has at least one matching document, each with its pre-built filter link.

Custom Facets

Custom Facets let you define the exact items that appear in a facet, each with its own filter rule, label, and position — rather than relying on the values that happen to exist in the index. They are created from an existing field and appear automatically in the search response alongside regular facets, following the site's behavior settings.

This is useful when the raw field values are not user-friendly, when you want to group multiple values into a named bucket, or when you need range-based navigation (e.g., price ranges, date periods, score thresholds).

Custom Facet item operators

Each item in a Custom Facet is defined by an operator that generates its Solr filter query:

OPERATOR	BEHAVIOR	EXAMPLE
EQUAL	Matches documents where the field equals the specified value exactly	<code>content_type = "video"</code>
BETWEEN	Matches documents where the field value falls within a range (inclusive)	<code>price between 100 and 500</code>
GREATER_THAN	Matches documents where the field value is above the threshold	<code>score > 80</code>
LESS_THAN	Matches documents where the field value is below the threshold	<code>date < 2020-01-01</code>

Custom Facet item configuration

SETTING	TYPE	DESCRIPTION
label	<code>string</code>	The display name shown in the facet panel (e.g., "Last 30 days", "Under \$100")
operator	<code>enum</code>	The filter operator: <code>EQUAL</code> , <code>BETWEEN</code> , <code>GREATER_THAN</code> , or <code>LESS_THAN</code>
rangeStart	<code>BigDecimal</code>	Start of a numeric range (used with <code>BETWEEN</code> and <code>GREATER_THAN</code>)
rangeEnd	<code>BigDecimal</code>	End of a numeric range (used with <code>BETWEEN</code> and <code>LESS_THAN</code>)
rangeStartDate	<code>Instant</code>	Start of a date range (used with <code>BETWEEN</code> and <code>GREATER_THAN</code> for date fields)
rangeEndDate	<code>Instant</code>	End of a date range (used with <code>BETWEEN</code> and <code>LESS_THAN</code> for date fields)
position	<code>integer</code>	The order in which this item appears within the facet

How it works

Once a Custom Facet is saved, Turing ES generates the facet items automatically on every search request. Each item appears in the `widget.facet` (or `widget.secondaryFacet`) section of the response with its pre-built filter link, exactly like a regular facet value. The front-end renders it without any special handling — Custom Facets are indistinguishable from regular facets in the response structure.

Example: Price range facet

A field `price` (numeric type) configured as a Custom Facet with four items:

LABEL	OPERATOR	RANGESTART	RANGEEND
Under \$50	LESS_THAN	—	50
\$50 – \$200	BETWEEN	50	200
\$200 – \$500	BETWEEN	200	500
Over \$500	GREATER_THAN	500	—

The search response returns these four items as a facet, each with a pre-built filter link — exactly like regular facet values.

Example: Score threshold facet

A field `relevance_score` configured as a Custom Facet:

LABEL	OPERATOR	RANGESTART	RANGEEND
High relevance	GREATER_THAN	80	—
Medium relevance	BETWEEN	50	80
Low relevance	LESS_THAN	—	50

Example: Date range facet

For date fields, use `rangeStartDate` and `rangeEndDate` (Instant values) instead of the numeric fields:

LABEL	OPERATOR	RANGESTARTDATE	RANGEENDDATE
This year	GREATER_THAN	2026-01-01T00:00:00Z	—
Last year	BETWEEN	2025-01-01T00:00:00Z	2025-12-31T23:59:59Z
Older	LESS_THAN	—	2025-01-01T00:00:00Z

i CUSTOM FACETS USE EXPLICIT VALUES, NOT SOLR EXPRESSIONS

Custom Facet items use concrete `BigDecimal` values (for numeric fields) or `Instant` timestamps (for date fields). Solr query expressions like `NOW-7DAYS` are not supported — use absolute values instead.

Merge Providers

Merge Providers enable Turing ES to detect when two connectors have indexed the same real-world document and merge their fields into a single enriched Solr document.

What they are

Merge Providers solve the problem of having two independent connectors that index different representations of the same real-world document. Without merging, the same content ends up as two separate entries in the search index — one from each connector — with incomplete information in each.

A classic example: an **AEM connector** indexes structured data from the CMS (`model.json`), capturing content metadata such as content type, author, tags, and structured fields. A **WebCrawler** indexes the same page as rendered HTML, capturing the full readable text. Both connectors run independently and neither has access to the data the other collects.

Merge Providers instruct Turing ES to detect when both connectors have indexed the same document and merge their content into a single, enriched Solr document.

How they work

A Merge Provider is configured with:

A join key — a pair of fields, one from each connector, whose values identify the same document. For example, the `url` field in the AEM document equals the `id` field in the WebCrawler document.

A set of field overwrite rules — which fields from the source connector (the one arriving second) should overwrite fields in the destination document (the one already in the index).

When the indexing pipeline receives a document from connector B and finds that a document from connector A already exists in the index with a matching join key value, it fetches the existing document, applies the field overwrite rules, and writes the merged result back to Solr.

AEM Connector indexes:

```
id:      "aem-12345"
url:     "https://example.com/products/widget"
title:   "Widget Pro"
author:  "Marketing"
tags:    ["product", "hardware"]
text:    "" ← empty, AEM doesn't have full text
```

WebCrawler indexes:

```
id:      "https://example.com/products/widget"
text:    "The Widget Pro is a high-performance..." ← full page text
title:   "Widget Pro - Example"
```

Merge Provider configuration:

```
Join key:      AEM.url = WebCrawler.id
Overwrite field: text ← WebCrawler.text → AEM document
```

Result in Solr:

```
id:      "aem-12345"
url:     "https://example.com/products/widget"
title:   "Widget Pro"
author:  "Marketing"
tags:    ["product", "hardware"]
text:    "The Widget Pro is a high-performance..." ← from WebCrawler
```

The final document in the index has the structured metadata from AEM and the full text from the WebCrawler, enabling both faceted navigation (using AEM's tags and metadata) and full-text search (using the crawled content).

Configuration

Each Merge Provider entry defines:

FIELD	DESCRIPTION
providerFrom	The source connector whose incoming document provides field values
providerTo	The destination connector whose existing document in the index will be updated
relationFrom	The field in the source document used as the join key
relationTo	The field in the destination document used as the join key
locale	The locale this merge provider applies to
description	Optional description of the merge provider's purpose

Each entry has a set of **overwrittenFields** — the field names from the source that replace values in the destination document.

Important considerations

CONNECTOR INDEXING ORDER MATTERS

The merge is triggered when the **source** connector's document arrives and a matching **destination** document already exists in the index. If the destination connector runs after the source, no merge will occur until the source re-indexes. Always schedule the **destination connector first** so the document is already in Solr when the source arrives.

Join key uniqueness: The field used as the join key must uniquely identify a document within its connector's output. If multiple documents share the same join key value, the merge behavior is undefined.

Field type compatibility: The fields being merged must have compatible types in the Solr schema. Merging a multi-value field into a single-value field, or a date into a string, will cause indexing errors.

Merge scope: Merge Providers are site-scoped. The same connector can participate in different merge configurations across different SN Sites.

Targeting Rules

Targeting Rules are a search-time personalization mechanism that filters results based on user profile attributes — group, role, segment, country, department, or any indexed field. They allow different users to see different content from the same Solr index without maintaining separate cores or running multiple queries.

There is no admin UI for Targeting Rules. Rules are passed by the client in the body of the `POST /api/sn/{siteName}/search` request and applied dynamically at query time. The Solr schema only needs to have the targeting attributes indexed on the documents.

Common use cases:

Corporate portal: show HR documents only to employees in the HR department

E-commerce: show promotions for the user's country and loyalty segment

Intranet: restrict confidential documents to users with the correct access group

SaaS platform: filter documentation by the user's subscription tier

How the pipeline works

Four rule types

Targeting Rules are sent in the POST request body (see [POST Body Parameters](#)). The four types differ in how values are combined across attributes:

1. `targetingRules` — simple list

A flat array of `attribute:value` strings. Values for the **same attribute** are combined with **OR**; different attributes are combined with **AND**.

```
{
  "query": "benefits",
  "targetingRules": ["department:HR", "department:Finance",
    "clearance:confidential"]
}
```

Result: documents where `(department=HR OR department=Finance)` AND `(clearance=confidential OR no clearance)`.

2. `targetingRulesWithCondition` — map with default logic

A map of `attribute → list of values`. Behaves identically to `targetingRules` but uses a structured map format instead of a flat list. Values for the **same attribute** are combined with **OR**; different attributes are combined with **AND**.

```
{
  "query": "benefits",
  "targetingRulesWithCondition": {
    "department": ["HR", "Finance"],
    "clearance": ["confidential"]
  }
}
```

Result: equivalent to the `targetingRules` example above — `(department=HR OR department=Finance) AND (clearance=confidential OR no clearance)`.

3. `targetingRulesWithConditionAND` — explicit AND

A map of `attribute → list of values`. Every attribute group must be satisfied simultaneously — AND between groups, OR within each group.

```
{
  "query": "promotions",
  "targetingRulesWithConditionAND": {
    "country": ["BR"],
    "language": ["pt"]
  }
}
```

Result: documents matching `country=BR AND language=pt` (or documents with no restrictions on either attribute).

4. `targetingRulesWithConditionOR` — explicit OR

A map of `attribute → list of values`. Any condition is sufficient — OR across all groups.

```
{
  "query": "discount",
  "targetingRulesWithConditionOR": {
    "segment": ["premium", "gold"],
    "loyalty": ["active"]
  }
}
```

Result: documents matching any of the conditions — `segment=premium`, `segment=gold`, or `loyalty=active`. More permissive than AND.

Solr filter query generation

Turing ES converts the rule type into Solr `fq` clauses automatically.

AND logic (each attribute group becomes a clause, all clauses joined with AND):

```
(group:admin OR group:user OR (*:* NOT group:*))
AND
(role:editor OR (*:* NOT role:*))
```

Each clause includes `(*:* NOT attribute:*)` — documents that are not tagged with the attribute are always included (the fallback clause). This ensures untagged, unrestricted content is always visible regardless of the active rules.

OR logic (all attribute-value pairs flattened, joined with OR):

```
(attr1:val1 OR attr2:val2)
OR
(*:* NOT attr1:* AND NOT attr2:*)
```

Documents that have none of the targeting attributes are also included.

The fallback clause

Both AND and OR methods include `(*:* NOT attribute:*)` to ensure documents that were never tagged with a targeting attribute are always returned. This means:

Adding targeting rules to a search request does **not** hide untagged content

Only documents explicitly tagged with a **conflicting** attribute value are filtered out

Documents with no targeting attributes are always visible

Practical examples

Example 1 — Corporate portal with department-scoped content

```
POST /api/sn/portal/search
{
  "query": "benefits",
  "targetingRules": ["department:HR", "department:Finance"]
}
```

Returns documents tagged for HR or Finance departments, plus all untagged public documents. A Marketing employee does not see HR-restricted content.

Example 2 — E-commerce: country and language (AND)

```
{
  "query": "promotions",
  "targetingRulesWithConditionAND": {
    "country": ["BR"],
    "language": ["pt"]
  }
}
```

Solr `fq: (country:BR OR (*:* NOT country:*)) AND (language:pt OR (*:* NOT language:*))`

Returns only documents for Brazil **and** in Portuguese, plus documents with no country or language restriction. A document tagged `country:US` is filtered out.

Example 3 — Internal system with role and group (AND)

```
{
  "query": "reports",
  "targetingRulesWithConditionAND": {
    "role": ["admin", "manager"],
    "group": ["sales"]
  }
}
```

Solr fq: (role:admin OR role:manager OR (:* NOT role:*)) AND (group:sales OR (:* NOT group:*))

Documents visible to admins or managers (or no role restriction), and within the sales group (or no group restriction).

Example 4 — Promotional content by segment (OR)

```
{
  "query": "discount",
  "targetingRulesWithConditionOR": {
    "segment": ["premium", "gold"],
    "loyalty": ["active"]
  }
}
```

Returns documents matching any condition — premium, gold, or active loyalty. More permissive than AND; used when any segment match is sufficient.

Example 5 — Intranet with access group control

```
{
  "query": "security policy",
  "targetingRules": ["access_group:it", "access_group:security",
    "clearance:confidential"]
}
```

Returns documents accessible to IT or Security groups, AND with clearance=confidential (or no clearance). Documents tagged `access_group:directors` remain hidden.

Indexing requirements

The targeting attributes must be **indexed fields** in the Solr schema and **populated at indexing time** by the Dumont DEP connector. Add the desired targeting fields to the SN Site's Fields configuration so they are included in the schema.

If a document is indexed without a targeting attribute (the field is absent or empty), it is treated as unrestricted and always visible — the fallback clause ensures this.

Metrics

Every search request with targeting rules is recorded in `sn_site_metric_access_trs`, allowing analytics on which rule combinations were applied, how often, and how they affect result volume.

Spotlights

Spotlights are curated search results that are pinned to specific search terms. When a user's query matches a spotlight term, configured documents are injected into the result list at defined positions, before the organic (ranked) results at those positions.

What they are

Spotlights are used for:

- Promoting official pages for important queries (e.g., "benefits" always surfaces the HR benefits page first)
- Pinning announcements or featured content to relevant search terms
- Ensuring a specific URL appears for a known high-traffic query

How matching works

Each Spotlight is associated with one or more **terms**. When a search request arrives, the system checks whether the query string contains any of the spotlight terms as a substring. The match is case-insensitive.

For example, a spotlight with the term `annual report` will match the queries `annual report 2024`, `download annual report`, and `annual report Q3` — any query that contains the phrase.

Spotlight terms are cached in memory to avoid repeated database lookups on every search request, making the matching step fast regardless of the number of spotlights configured.

How injection works

Each document within a Spotlight has a configured **position** — the ordinal slot in the result list where it should appear. When a spotlight match is detected, the system iterates through the result list and inserts each spotlight document at its configured position, shifting organic results down.

If a spotlight references a document by its Solr ID and that document exists in the index, the full indexed document is retrieved and injected with its live metadata. If the document is not found in the index (e.g., it was deleted or not yet indexed), the system falls back to the raw content configured directly in the spotlight — a manually defined title, description, URL, and type.

Search request: "annual report"



Spotlight term match found



Position 1 — Spotlight Doc A (pinned)
Position 2 — Organic result #1 (shifted)
Position 3 — Organic result #2 (shifted)
Position 4 — Spotlight Doc B (pinned)
Position 5 — Organic result #3 (shifted)

Configuring Spotlights

Each spotlight definition contains:

FIELD	DESCRIPTION
name	Descriptive label for the spotlight (admin only)
description	Optional notes about the spotlight's purpose
language	Locale for this spotlight (spotlights are scoped by language)
managed	☒ for managed mode (references indexed documents), ☐ for unmanaged (raw content)
provider	Source provider (default: <code>TURING</code>)
terms	One or more search terms that trigger this spotlight. Matching is case-insensitive substring matching

A document entry within a spotlight defines:

FIELD	DESCRIPTION
position	The slot in the result list where this document is injected (1-based)
referenceId	The <code>id</code> field value of the target document in the Solr index (optional — used in managed mode)
title	Fallback title if the document is not found in the index
content	Fallback description (max 2000 characters)
link	Fallback URL
type	Content type label shown in the UI

Spotlight modes

Spotlights operate in two modes:

Managed mode: The spotlight references documents that exist in the Solr index. The system retrieves live metadata from the index at query time, so the result always reflects the current state of the document (updated title, description, etc.).

Unmanaged mode: The spotlight stores its own raw content (title, description, URL, type) independently of the index. This is useful for pinning external URLs or content that is not indexed in Turing ES.

Custom Sort

Custom Sort allows administrators to define named, multi-level sort options beyond the built-in "relevance", "newest", and "oldest". Each custom sort consists of one or more sort levels (field + direction), applied in priority order to the Solr query.

Creating a Custom Sort

Navigate to the SN site's **Custom Sort** tab

Click **New**

Fill in the **Name** (used as the sort identifier in API requests) and optional **Description**

Add one or more **Sort Levels**:

SETTING	DESCRIPTION
Field	The Solr field to sort by (selected from the site's configured fields)
Direction	ASC (ascending) or DESC (descending)
Position	Priority order — position 0 is the primary sort, position 1 is the first tiebreaker, and so on

Use the move up/down buttons to reorder levels.

How It Works

When a search request includes `sort={customSortName}`, the query builder looks up the custom sort by name and applies each level to the Solr query in position order:

```
# Custom sort "price_then_date" with two levels:
#   Position 0: price ASC
#   Position 1: date DESC
#
# Solr query: sort=price asc, date desc
```

Built-in Sort Options

These are always available alongside any custom sorts:

VALUE	BEHAVIOR
relevance	Default Solr relevance scoring (no explicit sort)
newest	Sorts by the site's default date field, descending
oldest	Sorts by the site's default date field, ascending

Sort Options API

The endpoint `GET /api/sn/{siteName}/search/sort-options` returns all available sort options — built-in and custom — as a list of `{value, label}` pairs. This is consumed by the [React SDK's `useTuringSortOptions`](#) hook to populate sort dropdowns in the search UI.

```
[
  { "value": "relevance", "label": "Relevance" },
  { "value": "newest", "label": "Newest" },
  { "value": "oldest", "label": "Oldest" },
  { "value": "price_asc", "label": "Price: Low to High" },
  { "value": "title_sort", "label": "Title A-Z" }
]
```

REST API

METHOD	ENDPOINT	DESCRIPTION
GET	<code>/api/sn/{siteId}/custom-sort</code>	List all custom sorts
GET	<code>/api/sn/{siteId}/custom-sort/{id}</code>	Get a custom sort with its levels
GET	<code>/api/sn/{siteId}/custom-sort/fields</code>	List available fields for sorting
POST	<code>/api/sn/{siteId}/custom-sort</code>	Create a new custom sort
PUT	<code>/api/sn/{siteId}/custom-sort/{id}</code>	Update a custom sort
DELETE	<code>/api/sn/{siteId}/custom-sort/{id}</code>	Delete a custom sort

Search Rules

Search Rules are a powerful automation mechanism that dynamically modifies search behavior based on conditions evaluated at query time. When a rule's conditions match the incoming search request, its actions are applied to the Solr query — overriding sort, adding filters, changing facets, or boosting results — all without any client-side logic.

Rules use **first-match semantics**: they are evaluated in position order, and only the first matching rule's actions are applied.

Rule Structure

Each rule has three parts:

PART	DESCRIPTION
Details	Name, description, position (priority), and enabled/disabled toggle
Conditions	One or more conditions grouped by parameter — all groups must match (AND between groups)
Actions	One or more actions applied when the rule matches

Conditions

Conditions define **when** a rule triggers. Each condition evaluates a search parameter against a value using an operator.

Parameters

PARAMETER	DESCRIPTION	EXAMPLE
QUERY	The user's search query (<code>q</code> parameter)	User searched for "education"
FILTER_QUERY	Active facet filters (<code>fq</code> parameter)	User filtered by <code>category:books</code>
SORT	Current sort specification	Sort is "newest"
LOCALE	Current locale	Locale is <code>pt_BR</code>

Operators

OPERATOR	DESCRIPTION
<code>EQUALS</code>	Exact match
<code>CONTAINS</code>	Value contains the substring
<code>STARTS_WITH</code>	Value starts with the substring
<code>MATCHES_ANY</code>	Value contains the substring (alias for CONTAINS)
<code>IS_EMPTY</code>	Parameter has no value

Condition Grouping

Conditions are grouped by parameter. Within a group, conditions are combined using the **logic operator** (`AND` or `OR`). Between different parameter groups, `AND` is always used.

Example: "When query contains 'laptop' OR 'notebook', AND locale is pt_BR"

Condition Group 1 (QUERY, OR):

- QUERY CONTAINS "laptop"
- QUERY CONTAINS "notebook"

Condition Group 2 (LOCALE):

- LOCALE EQUALS "pt_BR"

Evaluation: (Group 1 OR) AND (Group 2) → both groups must match

NOTE

`SORT` and `LOCALE` parameters always use OR logic internally, regardless of the configured logic operator.

Actions

Actions define **what happens** when the rule matches. Multiple actions can be applied by a single rule.

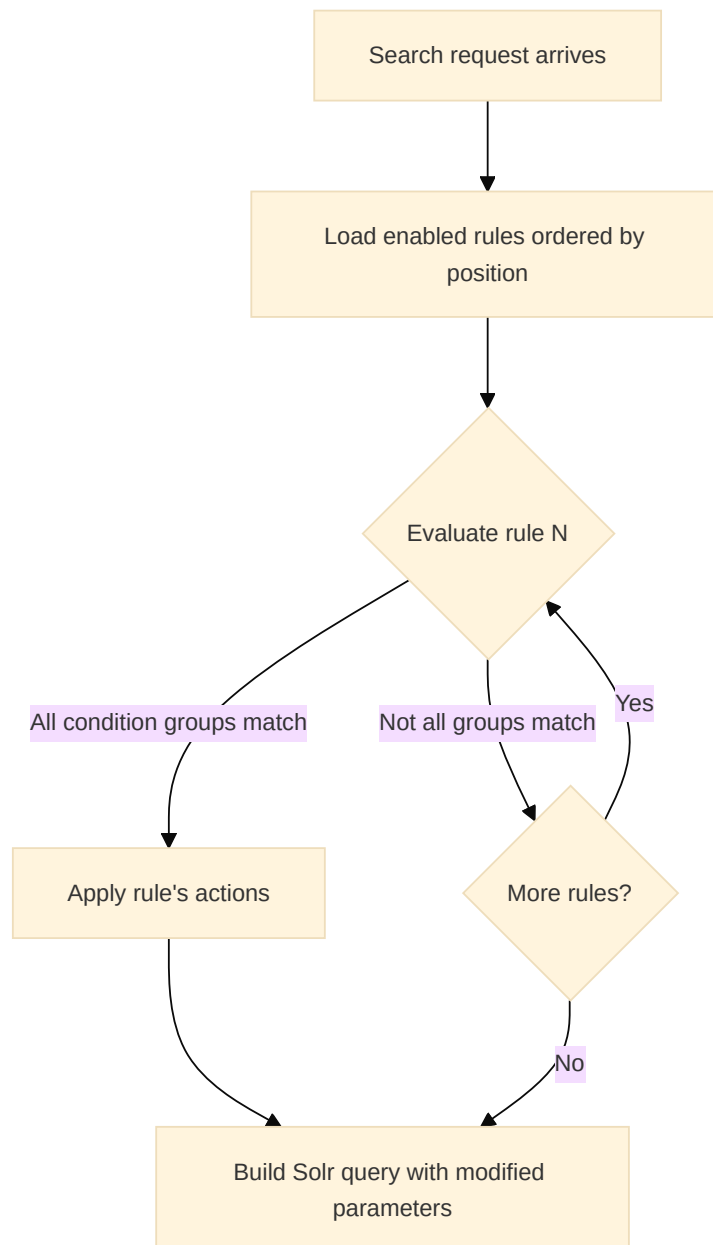
ACTION	DESCRIPTION	VALUE FORMAT
<code>SET_SORT</code>	Override the sort order	Sort name (e.g., <code>newest</code>), custom sort name, or <code>field:direction</code>
<code>SET_ROWS</code>	Change the number of results per page	Integer (e.g., <code>20</code>)
<code>ADD_FACETS</code>	Add facets to the response	Comma-separated field names (e.g., <code>brand,color</code>)
<code>REMOVE_FACETS</code>	Hide facets from the response	Comma-separated facet names
<code>ADD_FILTER_QUERY</code>	Automatically apply a filter	Filter expression (e.g., <code>type:article</code>)
<code>SET_BOOST_QUERY</code>	Boost specific documents	Boost expression (e.g., <code>featured:true^10</code>)



SET_SORT AND SET_ROWS ARE SINGLE-USE

Each rule can have at most one `SET_SORT` and one `SET_ROWS` action. The UI prevents adding duplicates.

Evaluation Pipeline



Rules are evaluated **before** the Solr query is built, allowing them to modify sort, facets, filters, and boost queries before they are sent to the search engine.

Examples

Example 1 — Force sort for product queries

When the user searches for "cheap" or "affordable", sort by price ascending:

PART	CONFIGURATION
Condition	QUERY CONTAINS "cheap" OR QUERY CONTAINS "affordable"
Action	SET_SORT → <code>price_asc</code> (a custom sort)

Example 2 — Add filters for a specific locale

When the locale is `pt_BR`, automatically filter to Portuguese content:

PART	CONFIGURATION
Condition	LOCALE EQUALS "pt_BR"
Action	ADD_FILTER_QUERY → <code>language:pt</code>

Example 3 — Boost featured content and limit results

When the query is empty (wildcard search), boost featured content and show fewer results:

PART	CONFIGURATION
Condition	QUERY IS_EMPTY
Actions	SET_BOOST_QUERY → <code>featured:true^10</code> , SET_ROWS → <code>5</code>

Example 4 — Show specific facets for filtered searches

When the user filters by `category:electronics`, add brand and price facets:

PART	CONFIGURATION
Condition	FILTER_QUERY EQUALS <code>category:electronics</code>
Action	ADD_FACETS → <code>brand,price_range</code>

REST API

METHOD	ENDPOINT	DESCRIPTION
GET	/api/sn/{siteId}/search-rule	List all rules (ordered by position)
GET	/api/sn/{siteId}/search-rule/{id}	Get a rule with conditions and actions
GET	/api/sn/{siteId}/search-rule/fields	List available fields for conditions
POST	/api/sn/{siteId}/search-rule	Create a new rule
PUT	/api/sn/{siteId}/search-rule/{id}	Update a rule
DELETE	/api/sn/{siteId}/search-rule/{id}	Delete a rule

Top Search Terms

Displays reports of the most frequently searched terms for this site. Turing ES records every search query and aggregates statistics across four time windows:

REPORT	DESCRIPTION
Today	Top 50 terms searched today
This Week	Top 50 terms for the current week
This Month	Top 50 terms for the current month
All Time	Top 50 most searched terms of all time

Each report shows the term and its search count for the selected period. Use this data to identify popular queries for Spotlight configuration, content gaps, and relevance tuning.

Result Ranking

Configures boost expressions that influence search relevance. Each ranking expression boosts documents matching a set of conditions by a configured weight. Expressions are converted into Solr boost queries in the format `(condition)^weight` and applied at query time.

Ranking Expression Form

General Information

ATTRIBUTE	DESCRIPTION
Name	Name for this ranking rule
Description	What this rule does and why

Ranking Conditions

A dynamic list of one or more field-value conditions. At least one condition is required.

ATTRIBUTE	DESCRIPTION
Attribute	An indexed field in the document, selected from the site's configured fields
Condition	<code>Is</code> (equals) or <code>Is not</code> (not equals)
Value	The value to compare against

Multiple conditions are combined with AND logic — all conditions must match for the boost to apply.

Ranking Weight

A slider from **0** to **10** that sets the boost intensity. Higher weight causes matching documents to rank higher in results.

Example: A condition `type Is news` with weight `8` causes documents of type "news" to rank significantly higher than other results.

AI Insights

The AI Insights tab displays an AI-generated natural language summary of this SN Site — aggregating information about its configuration, fields, behavior settings, and indexed content. The summary is generated on demand and gives administrators a quick overview of what the site is configured to do, without having to navigate through all the tabs.

Click **Generate** to trigger the summary. The response streams in progressively as it is generated.

Generative AI

Configures the RAG (Retrieval-Augmented Generation) system for this site, enabling conversational AI responses grounded in the indexed documents. The tab is divided into four sections.

Site Prompt

A field for generating an automatic description of the site using AI, with streaming support. The generated description can overwrite an existing one after a confirmation dialog.

RAG Activation

ATTRIBUTE	DESCRIPTION
Enabled	Enables the generative AI chat endpoint for this site (<code>GET /api/sn/{siteName}/chat</code>)

AI Model & Embedding Configuration

If not configured here, the site inherits the global defaults from **Administration → Settings**.

ATTRIBUTE	DESCRIPTION
LLM Instance	The language model used to generate responses (e.g., Anthropic Claude, OpenAI GPT-4o). See LLM Instances
Embedding Store	The vector store where document embeddings are persisted: ChromaDB, PgVector, or Milvus
Embedding Model	The model used to vectorize documents at indexing time and queries at search time

CHANGING THE EMBEDDING MODEL

Changing the Embedding Model invalidates all existing embeddings for this site. All indexed content must be re-indexed after this change. The Knowledge Base training status in [Assets](#) will reflect the stale state.

System Prompt

An editor for the prompt template sent to the LLM on each chat request. The template must include two required variables:

VARIABLE	DESCRIPTION
<code>{{question}}</code>	The user's question
<code>{{information}}</code>	The context retrieved from indexed documents via similarity search

Saving is blocked if either variable is missing from the template.

How it works: When documents are indexed, Turing ES extracts title, summary, and body text and stores their embeddings in the configured vector store. When a user sends a chat request (`GET /api/sn/{siteName}/chat?q=...`), the system performs a similarity search, builds the prompt with the retrieved context, sends it to the LLM, and returns the generated response via Server-Sent Events (SSE). See [Chat](#) for the front-end experience.

Search Response Structure

The full search response schema — including `pagination`, `queryContext`, `results`, and the `widget` object (containing `facet`, `secondaryFacet`, `similar`, `spellCheck`, `spotlights`, `locales`, and `cleanUpFacets`) — is documented in the [REST API Reference → Search Response Structure](#).

Related Pages

PAGE	DESCRIPTION
REST API Reference	Search, autocomplete, spell check, and all other API endpoints
Search Engine	Manage the Solr/Elasticsearch/Lucene backends that SN Sites connect to
Assets	Knowledge Base files for RAG — requires an embedding-capable LLM Instance
LLM Instances	Configure the language models used by the Generative AI tab
Chat	Front-end chat interface for the Semantic Navigation tab
Architecture Overview	How SN Sites fit into the overall system architecture

DSL Query API

Turing ES provides an **Elasticsearch-compatible Query DSL** endpoint that translates queries to the configured search engine backend (Solr, Lucene, or Elasticsearch). This allows you to use the same query syntax regardless of the underlying engine.

Endpoint

```
POST /api/sn/{siteName}/_search?locale=en
GET  /api/sn/{siteName}/_search?locale=en
```

PARAMETER	TYPE	DEFAULT	DESCRIPTION
<code>siteName</code>	path	required	The Semantic Navigation site name
<code>locale</code>	query	<code>en</code>	Search locale (e.g., <code>en</code> , <code>pt</code> , <code>en_US</code>)

The `GET` method performs a `match_all` search. The `POST` method accepts a JSON request body following the Elasticsearch `_search` format.

Quick Start

Simple Match Query

```
curl -X POST "http://localhost:2700/api/sn/mySite/_search?locale=en" \
-H "Content-Type: application/json" \
-d '{
  "query": {
    "match": { "title": "enterprise search" }
  }
}'
```

Bool Query with Filters and Pagination

```
curl -X POST "http://localhost:2700/api/sn/mySite/_search?locale=en" \
-H "Content-Type: application/json" \
-d '{
  "query": {
    "bool": {
      "must": [{ "match": { "title": "machine learning" } }],
      "filter": [
        { "term": { "status": "published" } },
        { "range": { "date": { "gte": "2025-01-01" } } }
      ],
      "must_not": [{ "term": { "category": "draft" } }]
    }
  },
  "from": 0,
  "size": 10,
  "sort": [{ "date": "desc" }, "_score"],
  "_source": ["title", "url", "date"],
  "highlight": {
    "fields": { "title": {}, "body": {} },
    "pre_tags": ["<em>"],
    "post_tags": ["</em>"]
  },
  "aggs": {
    "by_category": { "terms": { "field": "category", "size": 10 } }
  }
}'
```


Response Format

```
{
  "took": 15,
  "timed_out": false,
  "hits": {
    "total": { "value": 42, "relation": "eq" },
    "max_score": 1.5,
    "hits": [
      {
        "_id": "doc1",
        "_score": 1.5,
        "_source": { "title": "ML Guide", "url": "/ml-guide", "date": "2025-06-01" },
        "highlight": { "title": ["<em>Machine Learning</em> Guide"] }
      }
    ]
  },
  "aggregations": {
    "by_category": {
      "buckets": [
        { "key": "tutorial", "doc_count": 18 },
        { "key": "research", "doc_count": 12 }
      ]
    }
  }
}
```

Supported Queries

Full-text Queries

match

Full-text search on a single field with optional operator.

```
{ "query": { "match": { "title": "enterprise search" } } }
```

With operator:

```
{ "query": { "match": { "title": { "query": "enterprise search", "operator":  
"and" } } } }
```

multi_match

Search across multiple fields.

```
{  
  "query": {  
    "multi_match": {  
      "query": "artificial intelligence",  
      "fields": ["title", "body", "abstract"]  
    }  
  }  
}
```

match_phrase

Exact phrase search with optional slop (word distance).

```
{ "query": { "match_phrase": { "title": { "query": "enterprise search",  
"slop": 2 } } } }
```

match_phrase_prefix

Autocomplete-style phrase matching.

```
{ "query": { "match_phrase_prefix": { "title": { "query": "enter",  
"max_expansions": 10 } } } }
```

match_bool_prefix

Combines bool query for leading terms with prefix on the last term.

```
{ "query": { "match_bool_prefix": { "title": "quick brown f" } } }
```

combined_fields

Search across multiple fields as if they were one combined field.

```
{
  "query": {
    "combined_fields": {
      "query": "enterprise search",
      "fields": ["title", "body"],
      "operator": "and"
    }
  }
}
```

query_string

Lucene query string syntax.

```
{ "query": { "query_string": { "query": "title:search AND status:published",
  "default_field": "title" } } }
```

simple_query_string

Simple query string syntax (no exceptions on parse errors).

```
{
  "query": {
    "simple_query_string": {
      "query": "\"enterprise search\" +platform",
      "fields": ["title", "body"],
      "default_operator": "AND"
    }
  }
}
```

Term-level Queries

term

Exact match on a field (not analyzed).

```
{ "query": { "term": { "status": "published" } } }
```

terms

Match any of multiple values.

```
{ "query": { "terms": { "status": ["published", "review"] } } }
```

terms_set

Terms with dynamic `minimum_should_match`.

```
{
  "query": {
    "terms_set": {
      "tags": {
        "terms": ["search", "ai", "nlp"],
        "minimum_should_match_field": "required_matches"
      }
    }
  }
}
```

range

Range queries with `gte`, `gt`, `lte`, `lt`.

```
{ "query": { "range": { "date": { "gte": "2025-01-01", "lte": "2026-12-31" } } } }
```

exists

Check if a field has a value.

```
{ "query": { "exists": { "field": "thumbnail" } } }
```

prefix

Prefix matching.

```
{ "query": { "prefix": { "title": "ent" } } }
```

wildcard

Wildcard pattern matching (***** and **?**).

```
{ "query": { "wildcard": { "title": "enter*" } } }
```

regexp

Regular expression matching.

```
{ "query": { "regexp": { "title": { "value": "enter.*ch", "flags": "ALL" } } } }
```

fuzzy

Fuzzy (approximate) matching.

```
{ "query": { "fuzzy": { "title": { "value": "serch", "fuzziness": 2 } } } }
```

ids

Match specific document IDs.

```
{ "query": { "ids": { "values": ["doc1", "doc2", "doc3"] } } }
```

Compound Queries

bool

Combine multiple clauses with **must**, **should**, **must_not**, and **filter**.

```
{
  "query": {
    "bool": {
      "must": [{ "match": { "title": "search" } }],
      "should": [{ "match": { "category": "premium" } }],
      "must_not": [{ "term": { "status": "archived" } }],
      "filter": [{ "range": { "date": { "gte": "2025-01-01" } } }],
      "minimum_should_match": 1
    }
  }
}
```

constant_score

Wraps a filter query and assigns a constant score.

```
{ "query": { "constant_score": { "filter": { "term": { "status": "published" } } }, "boost": 1.5 } }
```

dis_max

Returns documents matching any query, using the best-matching score.

```
{
  "query": {
    "dis_max": {
      "queries": [
        { "match": { "title": "search" } },
        { "match": { "body": "search" } }
      ],
      "tie_breaker": 0.3
    }
  }
}
```

boosting

Boost positive matches while demoting negative matches.

```
{
  "query": {
    "boosting": {
      "positive": { "match": { "title": "search" } },
      "negative": { "term": { "status": "deprecated" } },
      "negative_boost": 0.2
    }
  }
}
```

function_score

Custom scoring with multiple functions.

```
{
  "query": {
    "function_score": {
      "query": { "match_all": {} },
      "functions": [
        { "filter": { "term": { "category": "premium" } }, "weight": 2.0 },
        {
          "field_value_factor": {
            "field": "popularity",
            "factor": 1.2,
            "modifier": "log1p",
            "missing": 1
          }
        }
      ],
      "score_mode": "sum",
      "boost_mode": "multiply",
      "max_boost": 10
    }
  }
}
```

script_score

Score using a script.

```
{
  "query": {
    "script_score": {
      "query": { "match": { "title": "search" } },
      "script": {
        "source": "_score * doc['popularity'].value",
        "lang": "painless",
        "params": { "factor": 2 }
      }
    }
  }
}
```

pinned

Pin specific documents to the top of results.


```
{
  "query": {
    "pinned": {
      "ids": ["featured-doc-1", "featured-doc-2"],
      "organic": { "match": { "title": "search" } }
    }
  }
}
```

Nested & Join Queries

nested

Search within nested objects.

```
{
  "query": {
    "nested": {
      "path": "comments",
      "query": { "match": { "comments.text": "great article" } },
      "score_mode": "avg"
    }
  }
}
```

has_child

Find parent documents by child document criteria.

```
{
  "query": {
    "has_child": {
      "type": "answer",
      "query": { "match": { "body": "elasticsearch" } },
      "min_children": 1,
      "max_children": 10
    }
  }
}
```

has_parent

Find child documents by parent document criteria.

```
{
  "query": {
    "has_parent": {
      "parent_type": "question",
      "query": { "match": { "title": "search" } },
      "score": true
    }
  }
}
```

Geo Queries

geo_distance

Filter by distance from a point.

```
{
  "query": {
    "geo_distance": {
      "distance": "10km",
      "location": { "lat": 40.73, "lon": -73.93 }
    }
  }
}
```

geo_bounding_box

Filter by bounding box.

```
{
  "query": {
    "geo_bounding_box": {
      "location": {
        "top_left": { "lat": 40.73, "lon": -74.1 },
        "bottom_right": { "lat": 40.01, "lon": -71.12 }
      }
    }
  }
}
```

geo_shape

Filter by geographic shape.

```
{
  "query": {
    "geo_shape": {
      "location": {
        "shape": { "type": "envelope", "coordinates": [[-74.1, 40.73],
[-71.12, 40.01]] },
        "relation": "within"
      }
    }
  }
}
```

Vector Search

knn

K-nearest neighbors vector search (semantic search).

```
{
  "query": {
    "knn": {
      "field": "embedding",
      "query_vector": [0.1, 0.2, 0.3, 0.4],
      "k": 10,
      "num_candidates": 100,
      "similarity": 0.8,
      "filter": { "term": { "status": "published" } }
    }
  }
}
```

Span Queries

`span_term`, `span_near`, `span_or`, `span_not`, `span_first`

Low-level positional queries for advanced use cases.

```
{
  "query": {
    "span_near": {
      "clauses": [
        { "span_term": { "title": "enterprise" } },
        { "span_term": { "title": "search" } }
      ],
      "slop": 2,
      "in_order": true
    }
  }
}
```

Specialized Queries

`more_like_this`

Find similar documents.

```
{
  "query": {
    "more_like_this": {
      "fields": ["title", "body"],
      "like": "enterprise search platform with AI capabilities",
      "min_term_freq": 1,
      "min_doc_freq": 1
    }
  }
}
```

intervals

Fine-grained control over term proximity.

```
{
  "query": {
    "intervals": {
      "body": {
        "match": {
          "query": "enterprise search",
          "max_gaps": 2,
          "ordered": true
        }
      }
    }
  }
}
```

rank_feature, **distance_feature**, **percolate**, **wrapper**

Additional specialized queries for ranking features, distance-based scoring, reverse search, and base64-encoded queries.

Aggregations

Bucket Aggregations

terms

Group documents by field values.

```
{
  "aggs": {
    "categories": { "terms": { "field": "category", "size": 10 } }
  }
}
```

range

Group documents by numeric or date ranges.

```
{
  "aggs": {
    "price_ranges": {
      "range": {
        "field": "price",
        "ranges": [
          { "key": "cheap", "to": 50 },
          { "key": "mid", "from": 50, "to": 200 },
          { "key": "expensive", "from": 200 }
        ]
      }
    }
  }
}
```

date_histogram

Group documents by date intervals.

```
{
  "aggs": {
    "monthly": {
      "date_histogram": {
        "field": "date",
        "calendar_interval": "month",
        "format": "yyyy-MM"
      }
    }
  }
}
```

histogram

Group documents by numeric intervals.

```
{
  "aggs": {
    "price_histogram": { "histogram": { "field": "price", "interval": 50 } }
  }
}
```

filter / filters

Single or multiple named filters.

```
{
  "aggs": {
    "status_counts": {
      "filters": {
        "filters": {
          "active": { "term": { "status": "active" } },
          "archived": { "term": { "status": "archived" } }
        }
      }
    }
  }
}
```

`significant_terms`, `rare_terms`, `multi_terms`, `auto_date_histogram`

Additional bucket aggregations for statistical analysis.

`nested` / `reverse_nested`

Aggregations scoped to nested objects.

`composite`

Paginated aggregations for large cardinality fields.

```
{
  "aggs": {
    "paginated": {
      "composite": {
        "size": 100,
        "sources": [
          { "category": { "terms": { "field": "category" } } }
        ]
      }
    }
  }
}
```

`sampler` / `diversified_sampler`, `adjacency_matrix`, `variable_width_histogram`

Advanced bucket aggregations for sampling, matrix analysis, and adaptive histograms.

Metric Aggregations

`avg`, `sum`, `min`, `max`

Basic numeric operations.


```
{
  "aggs": {
    "avg_price": { "avg": { "field": "price" } },
    "total_sales": { "sum": { "field": "amount" } },
    "cheapest": { "min": { "field": "price" } },
    "most_expensive": { "max": { "field": "price" } }
  }
}
```

stats / **extended_stats**

Multiple statistics in one aggregation.

```
{
  "aggs": {
    "price_stats": { "stats": { "field": "price" } }
  }
}
```

Response:

```
{
  "price_stats": {
    "values": { "count": 100, "min": 5.0, "max": 999.0, "avg": 125.5, "sum":
12550.0 }
  }
}
```

cardinality

Count distinct values.

```
{
  "aggs": {
    "unique_users": { "cardinality": { "field": "user_id",
    "precision_threshold": 1000 } }
  }
}
```

percentiles

Percentile distribution.

```
{
  "aggs": {
    "latency_percentiles": {
      "percentiles": { "field": "latency", "percents": [50, 95, 99] }
    }
  }
}
```

percentile_ranks

Rank of specific values.

```
{
  "aggs": {
    "latency_ranks": {
      "percentile_ranks": { "field": "latency", "values": [100, 200, 500] }
    }
  }
}
```

value_count, top_hits, top_metrics

Count, top documents, and top metrics per bucket.

median_absolute_deviation, boxplot, string_stats, matrix_stats, t_test, rate, scripted_metric

Advanced statistical aggregations.

Sub-aggregations

Nest aggregations within bucket aggregations for multi-dimensional analysis.

```
{
  "aggs": {
    "by_category": {
      "terms": { "field": "category", "size": 10 },
      "aggs": {
        "avg_price": { "avg": { "field": "price" } },
        "top_docs": { "top_hits": { "size": 3 } }
      }
    }
  }
}
```

Request Features

Pagination

```
{ "from": 0, "size": 10 }
```

Cursor-based pagination (search_after)

```
{ "search_after": ["2026-01-01", "doc123"], "size": 10, "sort": [{ "date":
"desc" }, "_id"] }
```

Scroll API

```
{ "scroll": "5m", "size": 1000 }
```

Sorting

```
{ "sort": [{ "date": "desc" }, { "title": "asc" }, "_score"] }
```

Source Filtering

```
{ "_source": ["title", "url", "date"] }  
{ "_source": false }  
{ "_source": { "includes": ["title", "url"] } }
```

Highlighting

```
{  
  "highlight": {  
    "fields": { "title": {}, "body": {} },  
    "pre_tags": ["<mark>"],  
    "post_tags": ["</mark>"],  
    "fragment_size": 150,  
    "number_of_fragments": 3  
  }  
}
```

Post Filter

Filter hits without affecting aggregations.

```
{  
  "query": { "match": { "title": "shoes" } },  
  "post_filter": { "term": { "color": "red" } },  
  "aggs": {  
    "all_colors": { "terms": { "field": "color" } }  
  }  
}
```

Rescore

Re-score top N results with a secondary query.

```
{
  "query": { "match": { "title": "search" } },
  "rescore": [{
    "window_size": 50,
    "query": {
      "rescore_query": { "match_phrase": { "title": "enterprise search" } },
      "query_weight": 0.7,
      "rescore_query_weight": 1.2
    }
  ]
}
```

Collapse (Field Collapsing)

Deduplicate results by a field.

```
{
  "query": { "match": { "title": "search" } },
  "collapse": {
    "field": "user_id",
    "inner_hits": { "name": "recent", "size": 3 }
  }
}
```

Suggest

Term, phrase, and completion suggestions.

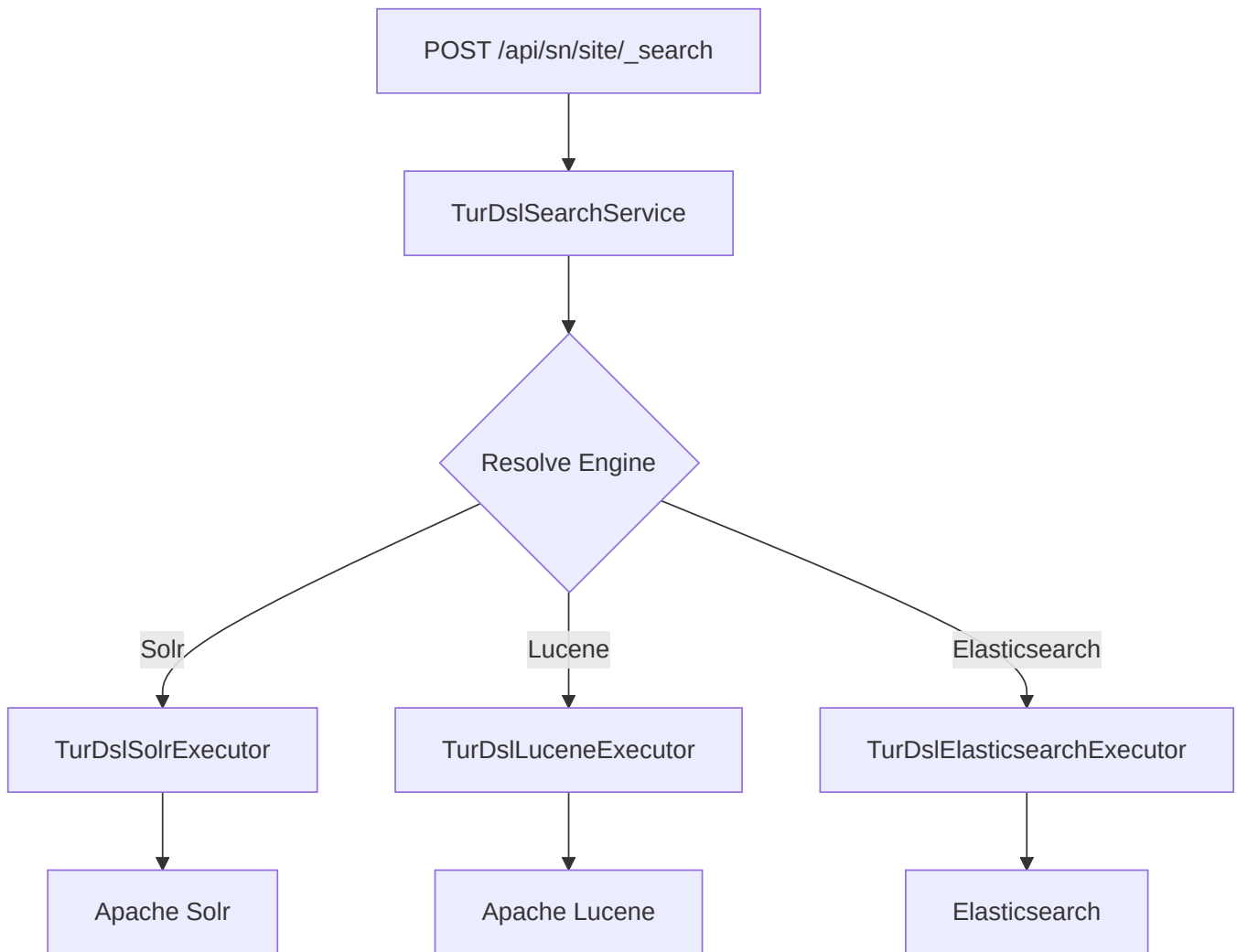
```
{
  "suggest": {
    "title-suggest": {
      "text": "enterprse",
      "term": { "field": "title" }
    },
    "autocomplete": {
      "text": "ent",
      "completion": { "field": "suggest_field", "size": 5, "skip_duplicates":
true }
    }
  }
}
```

Additional Features

FEATURE	EXAMPLE
<code>min_score</code>	<code>"min_score": 0.5</code>
<code>timeout</code>	<code>"timeout": "5s"</code>
<code>explain</code>	<code>"explain": true</code>
<code>profile</code>	<code>"profile": true</code>
<code>version</code>	<code>"version": true</code>
<code>script_fields</code>	<code>"script_fields": {"x": {"script": {"source": "doc['price'].value * 2"}}}</code>
<code>stored_fields</code>	<code>"stored_fields": ["title", "date"]</code>
<code>docvalue_fields</code>	<code>"docvalue_fields": ["date", "price"]</code>
<code>indices_boost</code>	<code>"indices_boost": [{"index1": 1.5}]</code>
<code>track_total_hits</code>	<code>"track_total_hits": true</code> or <code>"track_total_hits": 10000</code>
<code>preference</code>	<code>"preference": "_local"</code>
<code>routing</code>	<code>"routing": "user123"</code>
<code>terminate_after</code>	<code>"terminate_after": 1000</code>
<code>seq_no_primary_term</code>	<code>"seq_no_primary_term": true</code>
<code>pit</code>	<code>"pit": {"id": "abc", "keep_alive": "5m"}</code>

Architecture

The DSL Query API uses a **Strategy Pattern** with three executor implementations:



The engine is resolved automatically from the SN site's Search Engine instance configuration. The same JSON request body works across all three engines.

How it Works

Request arrives at the `_search` endpoint

TurDslSearchService resolves the site and its engine type via

`TurSearchEnginePluginFactory`

The appropriate **executor** translates the DSL query to the native engine format:

Elasticsearch: Direct 1:1 mapping using the Java API client

Solr: Translates to Solr query syntax, facets, and parameters

Lucene: Translates to Lucene `Query` objects (BooleanQuery, TermQuery, etc.)

Response is built in Elasticsearch-compatible format regardless of engine

See Also

[Search Engine Configuration](#)

[Semantic Navigation](#)

[REST API Reference](#)

[DSL Compatibility Matrix](#)

DSL Compatibility Matrix

Compatibility of every Turing ES DSL Query feature across the three supported search engine backends.

Legend

BADGE	LEVEL	DESCRIPTION
	Native	Direct 1:1 mapping to the engine API
	Translated	Converted to equivalent engine syntax
	Partial	Basic support with limitations
	Fallback	Best-effort approximation
	N/A	Not supported by the engine

Queries (40 types)

Full-text Queries

QUERY	ES	SOLR	LUCENE
<code>match</code>	N	T	T
<code>multi_match</code>	N	T	T
<code>match_phrase</code>	N	T	T
<code>match_phrase_prefix</code>	N	T	T
<code>match_bool_prefix</code>	N	T	T
<code>combined_fields</code>	N	T	T
<code>query_string</code>	N	T	T
<code>simple_query_string</code>	N	T	T
<code>match_all</code>	N	T	T

Term-level Queries

QUERY	ES	SOLR	LUCENE
term	N	T	T
terms	N	T	T
terms_set	N	F	F
range	N	T	T
exists	N	T	T
prefix	N	T	T
wildcard	N	T	T
regexp	N	T	T
fuzzy	N	T	T
ids	N	T	T

Compound Queries

QUERY	ES	SOLR	LUCENE
<code>bool</code>	N	T	T
<code>constant_score</code>	N	T	T
<code>dis_max</code>	N	T	T
<code>boosting</code>	N	T	T
<code>function_score</code>	N	P	P
<code>script_score</code>	N	F	F
<code>pinned</code>	N	T	T

Nested & Join Queries

QUERY	ES	SOLR	LUCENE
<code>nested</code>	N	F	F
<code>has_child</code>	N	F	F
<code>has_parent</code>	N	F	F

Geo Queries

QUERY	ES	SOLR	LUCENE
<code>geo_distance</code>	N	T	F
<code>geo_bounding_box</code>	N	T	F
<code>geo_shape</code>	N	F	F

Vector Search

QUERY	ES	SOLR	LUCENE
knn	N	T	N

Span Queries

QUERY	ES	SOLR	LUCENE
span_term	N	T	T
span_near	N	T	T
span_or	N	T	T
span_not	N	T	T
span_first	N	F	F

Specialized Queries

QUERY	ES	SOLR	LUCENE
more_like_this	N	T	T
intervals	N	T	T
rank_feature	N	F	T
distance_feature	N	F	F
wrapper	N	F	F
percolate	N	F	F

Aggregations (35 types)

Bucket Aggregations

AGGREGATION	ES	SOLR	LUCENE
terms	N	T	T
range	N	T	—
date_histogram	N	—	—
histogram	N	—	—
filter	N	T	T
filters	N	T	T
significant_terms	N	T	T
rare_terms	N	P	T
nested	N	—	—
reverse_nested	N	—	—
auto_date_histogram	N	—	—
multi_terms	N	—	—
composite	N	—	—
sampler	N	—	—
diversified_sampler	N	—	—
adjacency_matrix	N	—	—
geo_distance	N	—	—

AGGREGATION	ES	SOLR	LUCENE
variable_width_histogram	N	—	—

Metric Aggregations

AGGREGATION	ES	SOLR	LUCENE
avg	N	T	T
sum	N	T	T
min	N	T	T
max	N	T	T
cardinality	N	T	T
value_count	N	T	T
stats	N	T	T
extended_stats	N	T	T
percentiles	N	T	T
percentile_ranks	N	T	T
top_hits	N	—	—
top_metrics	N	—	—
median_absolute_deviation	N	—	—
boxplot	N	—	—
string_stats	N	—	—
matrix_stats	N	—	—
t_test	N	—	—
rate	N	—	—

AGGREGATION	ES	SOLR	LUCENE
<code>scripted_metric</code>	N	—	—
<code>geo_bounds</code>	N	—	—
<code>geo_centroid</code>	N	—	—

Request Features (29 fields)

FEATURE	ES	SOLR	LUCENE
<code>from</code> / <code>size</code>	N	T	T
<code>sort</code>	N	T	T
<code>_source</code>	N	T	T
<code>highlight</code>	N	T	T
<code>aggs</code>	N	T	T
<code>post_filter</code>	N	T	T
<code>min_score</code>	N	T	P
<code>search_after</code>	N	—	—
<code>collapse</code>	N	T	—
<code>suggest</code>	N	P	—
<code>rescore</code>	N	P	—
<code>timeout</code>	N	T	—
<code>explain</code>	N	T	—
<code>script_fields</code>	N	—	—
<code>indices_boost</code>	N	—	—
<code>track_total_hits</code>	N	T	T
<code>stored_fields</code>	N	T	—
<code>scroll</code>	N	—	—

FEATURE	ES	SOLR	LUCENE
profile	N	—	—
pit	N	—	—
docvalue_fields	N	—	—
version	N	—	—
seq_no_primary_term	N	—	—
preference	N	—	—
routing	N	—	—
terminate_after	N	—	—

Response Fields

FIELD	ES	SOLR	LUCENE
<code>took</code>	N	T	T
<code>timed_out</code>	N	T	T
<code>hits.total</code>	N	T	T
<code>hits.max_score</code>	N	T	T
<code>hits._id</code>	N	T	T
<code>hits._score</code>	N	T	T
<code>hits._source</code>	N	T	T
<code>hits.highlight</code>	N	T	T
<code>hits._explanation</code>	N	—	—
<code>hits.fields</code>	N	—	—
<code>hits._version</code>	N	—	—
<code>hits._seq_no</code>	N	—	—
<code>hits.sort</code>	N	—	—
<code>aggregations</code>	N	T	T
<code>suggest</code>	N	—	—
<code>_scroll_id</code>	N	—	—
<code>profile</code>	N	—	—
<code>_shards</code>	N	—	—

Coverage Summary

CATEGORY	TOTAL	ES	SOLR	LUCENE
Queries	40	N 100%	T 100%	T 100%
Aggregations	35	N 100%	T 49%	T 31%
Request Features	29	N 100%	T 41%	T 21%
Response Fields	18	N 100%	T 44%	T 44%



RECOMMENDATION

For **full DSL compatibility**, use Elasticsearch as the search engine backend. Solr and Lucene provide excellent coverage for the most common queries and aggregations, with best-effort translations for advanced features.

Engine Selection Guide

USE CASE	RECOMMENDED
Full Elasticsearch DSL compatibility	N Elasticsearch
Production enterprise search with Solr infrastructure	T Solr
Embedded search without external dependencies	T Lucene
Vector / semantic search (knn)	N ES or N Lucene
Advanced aggregations (composite, nested, geo)	N Elasticsearch
Simple faceted search with metric aggs	T Solr or T Lucene

See Also

[DSL Query API](#)

[Search Engine Configuration](#)

[REST API Reference](#)

SPA Pages

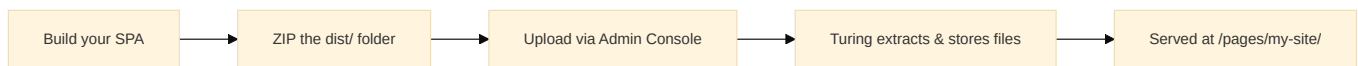
SPA Pages allows you to deploy and host compiled Single Page Applications (React, Vue, Angular, etc.) directly inside Turing ES. Upload a ZIP file containing your built app and it is served at a dedicated URL with proper client-side routing support.

The **Pages** section (`/console/page`) is available in the **Enterprise Search** section of the sidebar and is only visible when a **storage backend** is configured (`turing.storage.type` is `minio` or `filesystem`).

! STORAGE REQUIRED

SPA Pages requires a storage backend to be configured. See [Storage Configuration](#).

How It Works



Build your frontend app (e.g., `npm run build`)

ZIP the output folder (`dist/` , `build/` , etc.)

Upload the ZIP in the Admin Console under **Pages**

Access the deployed app at `http://your-turing-url/pages/{siteName}/`

Turing handles SPA routing automatically — if a requested file doesn't exist, it falls back to `index.html` , allowing client-side routers (React Router, Vue Router, etc.) to work correctly.

Deploying a Site

Upload via Admin Console

Navigate to **Enterprise Search > Pages** in the sidebar

Click **Upload SPA Site**

Select the ZIP file containing your built app

Confirm or edit the **site name** (auto-derived from the filename)

Click **Deploy**

The site name is normalized to lowercase alphanumeric characters, dots, and hyphens. For example, `My App v2.zip` becomes `my-app-v2`.

RE-DEPLOYMENT

Uploading a ZIP with the same site name replaces the existing deployment entirely. The previous version is deleted before the new one is extracted.

Upload via REST API

```
curl -X POST "http://localhost:2700/api/page?siteName=my-site" \  
  -H "Cookie: JSESSIONID=your-session" \  
  -F "file=@dist.zip"
```

Turing Manifest

You can include an optional `turing-manifest.json` file at the root of your ZIP to provide metadata about the deployment:

```
{  
  "name": "My Search App",  
  "version": "1.2.0",  
  "author": "Your Team",  
  "description": "Custom search experience for our enterprise portal",  
  "framework": "React",  
  "buildTool": "Vite",  
  "snSite": "enterprise-site",  
  "locale": "en_US",  
  "repository": "https://github.com/your-org/your-app",  
  "buildDate": "2026-04-08T10:00:00Z"  
}
```

All fields are optional. When present, the manifest metadata is displayed on the site card in the Admin Console.

FIELD	DESCRIPTION
<code>name</code>	Display name of the application
<code>version</code>	App version
<code>author</code>	Author or team name
<code>description</code>	Short description
<code>framework</code>	Frontend framework (React, Vue, Angular, etc.)
<code>buildTool</code>	Build tool used (Vite, Webpack, etc.)
<code>snSite</code>	Associated Turing Semantic Navigation site
<code>locale</code>	Default locale
<code>repository</code>	Source code repository URL
<code>buildDate</code>	ISO 8601 build timestamp

Site Management

Listing Sites

The Pages admin page displays a card grid of all deployed sites. Each card shows:

- Site name** and manifest metadata (if available)
- Preview link** — opens the deployed site in a new tab
- Delete button** — removes the site and all its files

Deleting a Site

Click the delete button on a site card and confirm the dialog. This removes all files under the site's storage prefix.

REST API

METHOD	ENDPOINT	DESCRIPTION
GET	/api/page	List all deployed sites with manifests
POST	/api/page?siteName={name}	Upload and deploy a ZIP file
DELETE	/api/page/{siteName}	Delete a deployed site

Caching Behavior

Turing applies different caching strategies depending on the file:

FILE	CACHE-CONTROL	REASON
index.html	no-cache, no-store, must-revalidate	Always serves the latest version after re-deployment
All other files	public, max-age=31536000, immutable	Static assets (JS, CSS, images) are cached for 1 year

This follows the standard SPA caching pattern: hashed asset filenames ensure cache-busting on new deployments, while `index.html` is never cached so users always get the latest entry point.

Use with Turing React SDK

SPA Pages is designed to work with the **Turing React SDK**. A typical workflow:

Create a React app using the Turing React SDK (`@viglet/turing-react-sdk`)

Build the app (`npm run build`)

Add a `turing-manifest.json` to the build output with the `snSite` field pointing to your Semantic Navigation site

ZIP the build output and deploy via Pages

The deployed app communicates with the Turing REST API on the same origin, so no CORS configuration is needed.

Related Pages

PAGE	DESCRIPTION
Storage Configuration	Configure the storage backend required by Pages
React SDK	Build custom search experiences to deploy via Pages
Semantic Navigation	Configure the search sites that your SPA consumes
REST API	Full API reference for programmatic deployments

Integration

The **Integration** page (`/admin/integration/instance`) manages connectors that import and index content from external sources — such as AEM, web crawlers, or other content repositories — into the Turing ES search engine. It is accessible from the **Enterprise Search** section of the sidebar.

Each **Integration instance** represents a configured connection to a content connector application. The connector runs as a separate process (for example, Viglet Dumont DEP) and communicates with Turing ES via REST. Turing ES acts as a proxy: the API path `/api/v2/integration/{integrationId}/**` forwards requests to the configured connector endpoint, with built-in SSRF protection.

Instance Listing

The listing page shows all configured Integration instances as a grid of cards, each displaying the integration's title and description. A button at the top of the page opens the creation form.

Creating or Editing an Integration

The creation and edit form is divided into two sections.

General Information

FIELD	DESCRIPTION
Integration Name	Human-readable identifier for this integration (required)
Short Description	Optional description of the integration's purpose

Connection Details

FIELD	DESCRIPTION
Integration Type	Connector type — AEM or Web Crawler
Endpoint	URL of the connector application
Enabled	Toggle to activate or deactivate this integration

Integration Detail — Sections

After an integration is created, its detail page provides navigation to several sections. The sections below are common to all connector types. For AEM-specific configuration, see the [Turing ES AEM Connector](#) overview and the full [Dumont DEP AEM Connector](#) documentation.

Settings

The Settings section contains the same form fields as the creation form (General Information and Connection Details), allowing you to edit the integration name, description, endpoint, type, and enabled state.

Monitoring

The Monitoring section is a real-time dashboard for tracking the indexing pipeline. It displays indexing events as they are received and processed.

Filters:

FILTER	DESCRIPTION
Date From / Date To	Restrict results to a time window
Object ID	Filter by a specific document ID
Status	Filter by processing status (see status table below)
Environment	Author or Publish
Language	Locale code (for example, <code>en_US</code>)
Sites	Filter by target SN Site

Auto-refresh: configurable intervals — Off, 1s, 5s, 10s, 30s, 1m, 5m.

Result columns: Date, Object ID, Status, Environment, Language, Sites.

Results are paginated and sortable.

Indexing Status Values

STATUS	MEANING
PREPARE_INDEX	Preparing to index the document
PREPARE_UNCHANGED	No changes detected since last indexing
PREPARE_REINDEX	Preparing a reindexation
PREPARE_FORCED_REINDEX	Forced reindexation triggered
RECEIVED_AND_SENT_TO_TURING	Document received by the connector and forwarded to Turing ES
SENT_TO_QUEUE	Document placed in the Artemis processing queue
RECEIVED_FROM_QUEUE	Document consumed from the queue by the indexing pipeline
INDEXED	Document successfully indexed in Solr
FINISHED	Operation finished
DEINDEXED	Document removed from the index
NOT_PROCESSED / IGNORED	Document skipped due to an Indexing Rule or connector decision

Indexing Stats

The Indexing Stats section provides a table of completed bulk indexing operations for this integration.

COLUMN	DESCRIPTION
Start Time	Timestamp when the operation started
Source	Source that was operated on
Operation	<code>INDEX_ALL</code> or <code>REINDEX_ALL</code>
Documents	Number of documents processed
Duration	Total elapsed time
Docs/min	Throughput (documents per minute)

Double Check

The Double Check section validates the consistency between the connector's content and the Turing ES search index. It detects drift between what the connector knows about and what is actually indexed.

Select a **Source** to inspect

Results are shown in two views:

Missing — content that exists in the connector but is not present in the index

Extra — content that is present in the index but no longer exists in the connector

Results are grouped by Solr core in an accordion, listing the affected document paths

Use Double Check after a partial failure, a forced reindex, or when users report missing or stale search results.

System Information

Displays live diagnostic information from the remote connector application.

Status badge: UP (green) or DOWN (red) — indicates whether the connector endpoint is reachable.

Application:

ITEM	DESCRIPTION
Application	Connector application name
Version	Connector build version
Java Version	JVM version running the connector
Vendor	JVM vendor
JVM	JVM identifier
OS	Operating system name and version

Memory and disk gauges (progress bars with total, used, and free values):

GAUGE	DESCRIPTION
Physical Memory (RAM)	Host system memory
JVM Heap Memory	Java heap utilisation and limits
Disk Space	Available storage on the connector's host volume

Architecture

Turing ES acts as a transparent proxy to the connector. The API path `/api/v2/integration/{integrationId}/**` forwards all requests to the configured connector endpoint, including authentication headers. This design keeps the connector application decoupled — it does not need to be publicly accessible, only reachable from the Turing ES server.

Built-in SSRF protection validates the endpoint before forwarding. Requests to private IP ranges, loopback addresses, or disallowed schemes are rejected.

Related Pages

PAGE	DESCRIPTION
AEM Connector	AEM-specific configuration — sources, content types, environments, indexing rules, and the indexing manager
Semantic Navigation	Configure the SN Sites that receive indexed content
Architecture Overview	End-to-end indexing flow from connector to Solr
REST API Reference	API endpoints for programmatic indexing
Dumont DEP — Connectors	Available connectors (Web Crawler, AEM, Database, FileSystem, WordPress) and how to deploy them
Dumont DEP — REST API	Connector-side API for triggering indexing, monitoring, and source management

AEM Connector

The **Adobe Experience Manager (AEM)** connector allows Turing ES to index content from AEM author and publish instances. The connector runs as a separate process (**Dumont DEP**) and communicates with Turing ES via REST. Turing ES acts as a proxy — the API path `/api/v2/integration/{integrationId}/**` forwards requests to the configured connector endpoint.

For general integration management — creating instances, monitoring, statistics, and system information — see [Integration](#).

Overview

The AEM connector is a two-part system:

COMPONENT	WHERE IT RUNS	DOCUMENTATION
AEM Event Listener	Inside AEM (OSGi bundle) — sends indexing events to the connector	Dumont DEP — AEM Event Listener
AEM Connector Plugin	Dumont DEP process — fetches content from AEM and indexes it into Turing ES	Dumont DEP — AEM Connector
Integration Instance	Turing ES admin console — proxy configuration and monitoring	Integration

Configuration

All AEM-specific source configuration — connection details, root paths, content types, author/publish environments, locale mappings, delta tracking, indexing rules, and the Indexing Manager — is documented in the [Dumont DEP — AEM Connector](#) page.

The Turing ES admin console provides the UI for managing these settings under **Enterprise Search → Integration → [your AEM instance]**.

Monitoring & Diagnostics

The following features are available in the Turing ES admin console for each AEM integration instance (see [Integration](#) for details):

FEATURE	DESCRIPTION
Monitoring	Real-time dashboard for tracking indexing events with filters and auto-refresh
Indexing Stats	Table of completed bulk indexing operations with throughput metrics
Double Check	Validates consistency between the connector's content and the Turing ES search index
System Information	Live status, version, memory, and disk information from the connector
AEM Logging	AEM-specific logs in Logging → AEM (see Logging)

Related Pages

PAGE	DESCRIPTION
Integration	General integration management — instances, monitoring, statistics, and system information
Semantic Navigation	Configure the SN Sites that receive indexed content
Architecture Overview	End-to-end indexing flow from connector to Solr
REST API Reference	API endpoints for programmatic indexing
Dumont DEP — AEM Connector	How the AEM connector works — source configuration, indexing flow, event listeners, API triggering
Dumont DEP — AEM Event Listener	Install the OSGi event listener bundle inside AEM for real-time indexing
Dumont DEP — Extending AEM	Custom attribute extractors, content processors, and configuration JSON reference

Import & Export

Turing ES supports exporting and importing **Semantic Navigation site configurations** as ZIP archives. This allows you to migrate sites between environments, create backups, or share configurations across teams.

Export

Export creates a ZIP archive containing a complete snapshot of one or all SN Sites, including field definitions, facet settings, spotlights, ranking expressions, merge providers, locale configuration, and GenAI settings.

Export a Single Site

```
GET /api/sn/{id}/export
```

Export All Sites

```
GET /api/sn/export
```

Both endpoints require authentication via the **Key** header. The response is a downloadable ZIP file containing an **export.json** with the full site configuration.

Example:

```
curl "http://localhost:2700/api/sn/export" \
  -H "Key: <YOUR_API_TOKEN>" \
  -o turing-sites-export.zip
```

Export Contents

The `export.json` inside the ZIP archive contains:

SECTION	DESCRIPTION
<code>snSite</code> <code>s</code>	Complete SN Site configurations — fields, facets, spotlights, ranking expressions, merge providers, locales, and GenAI settings
<code>llm</code>	LLM instance references used by sites with GenAI enabled
<code>store</code>	Embedding store instance references
<code>se</code>	Search engine instance references

Import

The import page (`/admin/exchange/import`) accepts a ZIP file previously exported from Turing ES and recreates the site configurations in the target environment.

Using the Admin Console

Navigate to **Import** in the sidebar

Drag and drop a ZIP file onto the upload area, or click to browse

The upload progress is displayed in real time (percentage bar)

On success, you are automatically redirected to the SN Sites listing after 2 seconds

Supported formats: `.zip` files exported from Turing ES.

UI states:

STATE	DESCRIPTION
Idle	Dashed border with upload icon — ready for file drop or click
Uploading	Blue background with spinning loader and progress percentage
Success	Green border with checkmark — "Import Completed" message and auto-redirect
Error	Red border with error message and "Try Again" button

Using the REST API

```
POST /api/import
```

Accepts a `multipart/form-data` request with a `file` parameter containing the ZIP archive. Authentication required.

Example:

```
curl -X POST "http://localhost:2700/api/import" \  
  -H "Key: <YOUR_API_TOKEN>" \  
  -F "file=@turing-sites-export.zip"
```

Content Import (SN Job Items)

In addition to site configuration import, Turing ES accepts **document-level indexing jobs** via the SN Import API. This is used by connectors and custom integrations to push content into the search index.

JSON Import

```
POST /api/sn/import
```

Accepts a JSON body with a list of job items — documents to index, deindex, or commit.

Example:

```
curl -X POST "http://localhost:2700/api/sn/import" \  
  -H "Key: <YOUR_API_TOKEN>" \  
  -H "Content-Type: application/json" \  
  -d '{  
    "snJobItems": [  
      {  
        "locale": "en_US",  
        "turSNJobAction": "CREATE",  
        "siteNames": ["Sample"],  
        "attributes": {  
          "id": "doc-001",  
          "title": "Getting Started Guide",  
          "text": "This guide explains how to get started...",  
          "url": "https://example.com/getting-started"  
        }  
      }  
    ]  
  }'
```

ZIP Import (with file attachments)

```
POST /api/sn/import/zip
```

Accepts a `multipart/form-data` ZIP file containing an `export.json` with job items. Document attributes can reference embedded files using the `file://` protocol — Turing ES extracts text from these files using Apache Tika before indexing.

Job Actions

ACTION	DESCRIPTION
<code>CREATE</code>	Index or update a document in the search engine
<code>DELETE</code>	Remove a document from the index
<code>COMMIT</code>	Commit pending changes to the search engine

Job Item Structure

FIELD	TYPE	DESCRIPTION
<code>locale</code>	<code>string</code>	Document locale (e.g., <code>en_US</code>)
<code>turSNJobAction</code>	<code>string</code>	<code>CREATE</code> , <code>DELETE</code> , or <code>COMMIT</code>
<code>siteNames</code>	<code>string[]</code>	Target SN Site names
<code>attributes</code>	<code>map</code>	Document fields — must include <code>id</code>
<code>checksum</code>	<code>string</code>	Optional content checksum for change detection
<code>environment</code>	<code>string</code>	Optional — <code>AUTHOR</code> or <code>PUBLISHING</code>

Related Pages

PAGE	DESCRIPTION
Semantic Navigation	SN Site configuration
Integration	Content connectors that automate document import
REST API Reference	Full API endpoint reference
Developer Guide	Java SDK and JavaScript SDK for programmatic import

Generative AI & LLM Configuration

Turing ES integrates Generative AI throughout the platform via **Spring AI**, providing a unified abstraction layer over multiple LLM providers and embedding backends. The GenAI capabilities are organized around four connected concepts: **LLM Instances**, **Tool Calling**, **MCP Servers**, and **AI Agents** — which together define how the system reasons, retrieves, and responds.

RAG (Retrieval-Augmented Generation) is the primary pattern used to ground LLM responses in real content: both the **Knowledge Base** (files stored in MinIO) and **Semantic Navigation Sites** can serve as RAG sources.

Global Settings

Before configuring individual LLM Instances, AI Agents, or RAG sources, establish platform-wide defaults in **Administration** → **Settings**. These defaults apply wherever no specific override is configured.

SETTING	DESCRIPTION
Default LLM Instance	The LLM provider used when no specific instance is selected in an AI Agent or SN Site
Default Embedding Store	Which vector database backend to use (ChromaDB, PgVector, or Milvus)
Default Embedding Model	The embedding model used to generate vector representations during indexing and at query time
RAG Enabled by Default	Whether RAG is pre-enabled for new Semantic Navigation Sites
Python Path	Absolute path to the Python executable used by the Code Interpreter tool
Email Settings	SMTP configuration for system notifications

⚠ CHANGING THE EMBEDDING MODEL REQUIRES FULL RE-INDEXING

The embedding store and embedding model must be consistent across indexing and query time. Changing the default embedding model after documents have been indexed will cause embedding dimension mismatches and incorrect similarity results. A full re-indexing of all content is required.

LLM Instances

An **LLM Instance** is a named, configured connection to an LLM provider. Multiple instances can coexist — different AI Agents, SN Sites, and the Chat interface can each use a different instance independently.

Turing ES supports **six vendor types**: OLLAMA, OPENAI, ANTHROPIC, GEMINI, GEMINI_OPENAI, and AZURE_OPENAI.

VENDOR	DEFAULT MODEL	EMBEDDING	TOOL CALLING
OLLAMA	mistral	✓	✓
OPENAI	gpt-4o-mini	✓	✓
ANTHROPIC	claude-sonnet-4-20250514	✗	✓
GEMINI	gemini-2.0-flash	✗	✓
GEMINI_OPENAI	gemini-2.0-flash	✗	✓
AZURE_OPENAI	gpt-4o	✓	✓

For the full UI reference — form sections, provider-specific options, generation parameters, capability matrix, and API key security — see [LLM Instances](#).

Embedding Stores & Models

Turing ES delegates vector storage to one of three backends (ChromaDB, PgVector, Milvus) and uses an embedding model to generate and query vectors. The choice of backend and model affects both storage cost and similarity quality.

See [Embedding Stores](#) for backend comparison and [Embedding Models](#) for provider support, model selection guidance, and configuration details.

RAG Architecture

Retrieval-Augmented Generation (RAG) grounds LLM responses in indexed content. Rather than relying on training data alone, Turing ES retrieves the most semantically relevant documents and injects them into the prompt before generation.

Turing ES supports two RAG sources:

By default, Turing ES retrieves the **top 10** most similar chunks with a similarity **threshold of 0.7**. Documents below the threshold are excluded from the context.

Knowledge Base (Assets)

The Knowledge Base is built from files managed in the **Assets** section — a file manager backed by MinIO. Files are extracted with Apache Tika, truncated to 100,000 characters, split into 1,024-character chunks, embedded, and stored in the active embedding store.

Full documentation is available on the [Assets](#) page.

RAG for Semantic Navigation Sites

When GenAI is enabled for an SN Site, indexed documents are also embedded and stored in the vector store alongside their Solr representation. This allows AI Agents and the SN Site chat to retrieve relevant content via semantic similarity. Configure this in the site's **Generative AI** section — see [Semantic Navigation → Generative AI](#).

ⓘ RE-INDEXING EXISTING CONTENT

Documents indexed before GenAI was enabled do not have embeddings. A full re-indexing of the site is required to make existing content available for RAG queries.

Tool Calling

Turing ES includes **27 native tools** across 7 categories that AI Agents can invoke autonomously: Semantic Navigation (15), RAG / Knowledge Base (4), Web Crawler (2), Finance (2), Weather (1), Image Search (1), DateTime (1), and Code Interpreter (1).

See [Tool Calling](#) for the full tool reference.

MCP Servers

AI Agents can connect to any external server implementing the **Model Context Protocol (MCP)** to access tools beyond the 27 native ones — company-internal systems, proprietary APIs, or public MCP services. Configured in **Administration → MCP Servers**.

See [MCP Servers](#) for transport types, configuration form, and examples.

AI Agents

An **AI Agent** combines a specific LLM Instance, a set of tools, and MCP Servers into a single deployable assistant. Each agent appears as a separate tab in the Chat interface and can be purpose-built for specific roles — enterprise search, data research, IT operations, and more.

See [AI Agents](#) for configuration, composition examples, and the agent execution flow.

What is RAG?

RAG (Retrieval-Augmented Generation) is the technique that makes AI responses accurate and grounded in your actual content instead of relying solely on the model's training data.

The core idea is simple: **before the LLM generates a response, the system retrieves the most relevant documents from your content and includes them in the prompt.** The LLM then answers based on real, up-to-date information rather than what it memorized during training.

The Problem RAG Solves

Large Language Models are trained on vast amounts of text, but they have critical limitations:

Knowledge cutoff — They don't know about content created after their training date

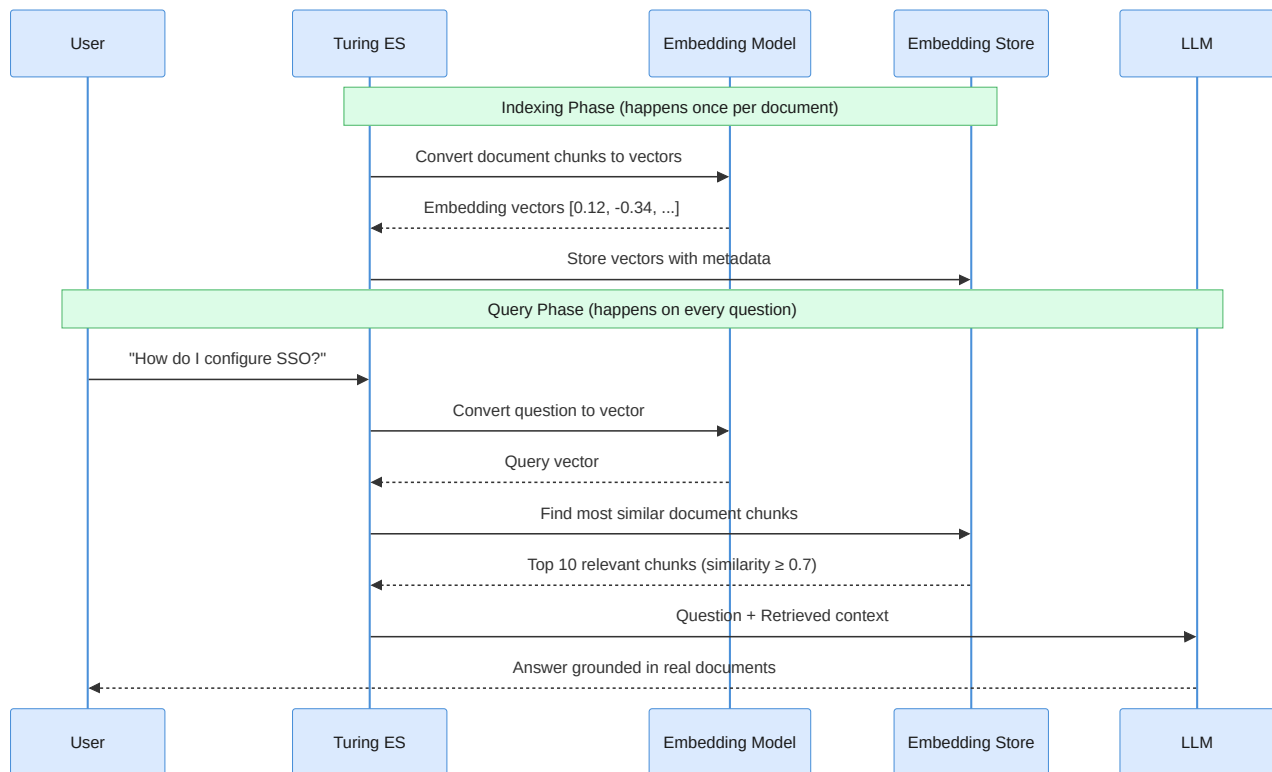
No access to private data — They've never seen your internal documents, product specs, or company policies

Hallucination — When they don't know an answer, they may generate plausible-sounding but incorrect information

Generic responses — Without specific context, answers are broad and imprecise

RAG eliminates these problems by giving the LLM access to your actual content at query time.

How RAG Works — Step by Step



Phase 1 — Indexing (one-time per document)

Document ingestion — Content enters Turing ES via connectors (AEM, web crawler) or file uploads (Assets/Knowledge Base)

Text extraction — Apache Tika extracts text from PDFs, DOCX, XLSX, HTML, and other formats

Chunking — The extracted text is split into chunks (default: 1,024 characters) to fit within embedding model limits

Embedding — Each chunk is passed through the **Embedding Model**, which converts text into a high-dimensional numerical vector (e.g., a 1,536-dimension array of floats)

Storage — The vectors are stored in the **Embedding Store** alongside metadata (source file, chunk position, original text)

Phase 2 — Retrieval (on every user question)

Query embedding — The user's question is converted into a vector using the **same Embedding Model**

Similarity search — The **Embedding Store** finds the vectors most similar to the query vector (cosine similarity)

Threshold filtering — Only chunks with similarity ≥ 0.7 are included (configurable)

Top-K selection — The top 10 most relevant chunks are selected

Phase 3 — Generation

Prompt construction — Turing ES builds a prompt containing the user's question plus the retrieved document chunks as context

LLM generation — The LLM reads the context and generates an answer based on the real content

Streaming response — The answer is streamed back to the user via SSE

The Three Components

LLM (Large Language Model)

The LLM is the "brain" that reads the retrieved context and generates a natural language response. It does the reasoning, summarization, and articulation — but it does **not** search or retrieve content on its own.

In Turing ES, LLMs are configured as **LLM Instances** supporting six providers:

PROVIDER	EXAMPLE MODELS
OpenAI	GPT-4o, GPT-4o-mini
Anthropic	Claude Sonnet 4
Ollama	Mistral, Llama, Qwen
Google Gemini	Gemini 2.0 Flash
Azure OpenAI	GPT-4o (Azure-hosted)

The LLM does not store knowledge. It only processes what's given to it in the prompt. RAG ensures the prompt contains the right content.

Embedding Model

The Embedding Model is a specialized neural network that converts text into numerical vectors (arrays of floating-point numbers). These vectors capture the **semantic meaning** of the text — similar concepts end up with similar vectors, even if the words are different.

Example:

TEXT	VECTOR (SIMPLIFIED)
"How to configure single sign-on"	[0.82, -0.15, 0.44, ...]
"SSO setup and authentication"	[0.80, -0.13, 0.46, ...] ← very similar!
"Weather forecast for tomorrow"	[-0.31, 0.67, -0.22, ...] ← very different

The embedding model is used **twice**: once during indexing (to embed document chunks) and once at query time (to embed the user's question). Both must use the **same model** — otherwise the vectors are incompatible and similarity search fails.

In Turing ES, providers that support embedding include **OpenAI**, **Ollama**, and **Azure OpenAI**. See [Embedding Models](#) for provider details and model selection.

⚠ SAME MODEL FOR INDEXING AND QUERYING

Changing the embedding model after documents have been indexed causes dimension mismatches and incorrect results. A full re-indexing is required after any model change.

Embedding Store (Vector Database)

The Embedding Store is a specialized database optimized for storing and querying high-dimensional vectors. Unlike a traditional database that matches exact values, an embedding store finds the **most similar** vectors using distance metrics (cosine similarity, dot product, or Euclidean distance).

Turing ES supports three backends:

BACKEND	BEST FOR	HOW IT WORKS
ChromaDB	Development, small/medium deployments	Lightweight, open-source, connects via HTTP API
PgVector	Teams already using PostgreSQL	PostgreSQL extension — embeddings in the same DB as app data
Milvus	Large-scale production, high throughput	Purpose-built vector DB with advanced indexing (IVF, HNSW)

What happens inside the embedding store:

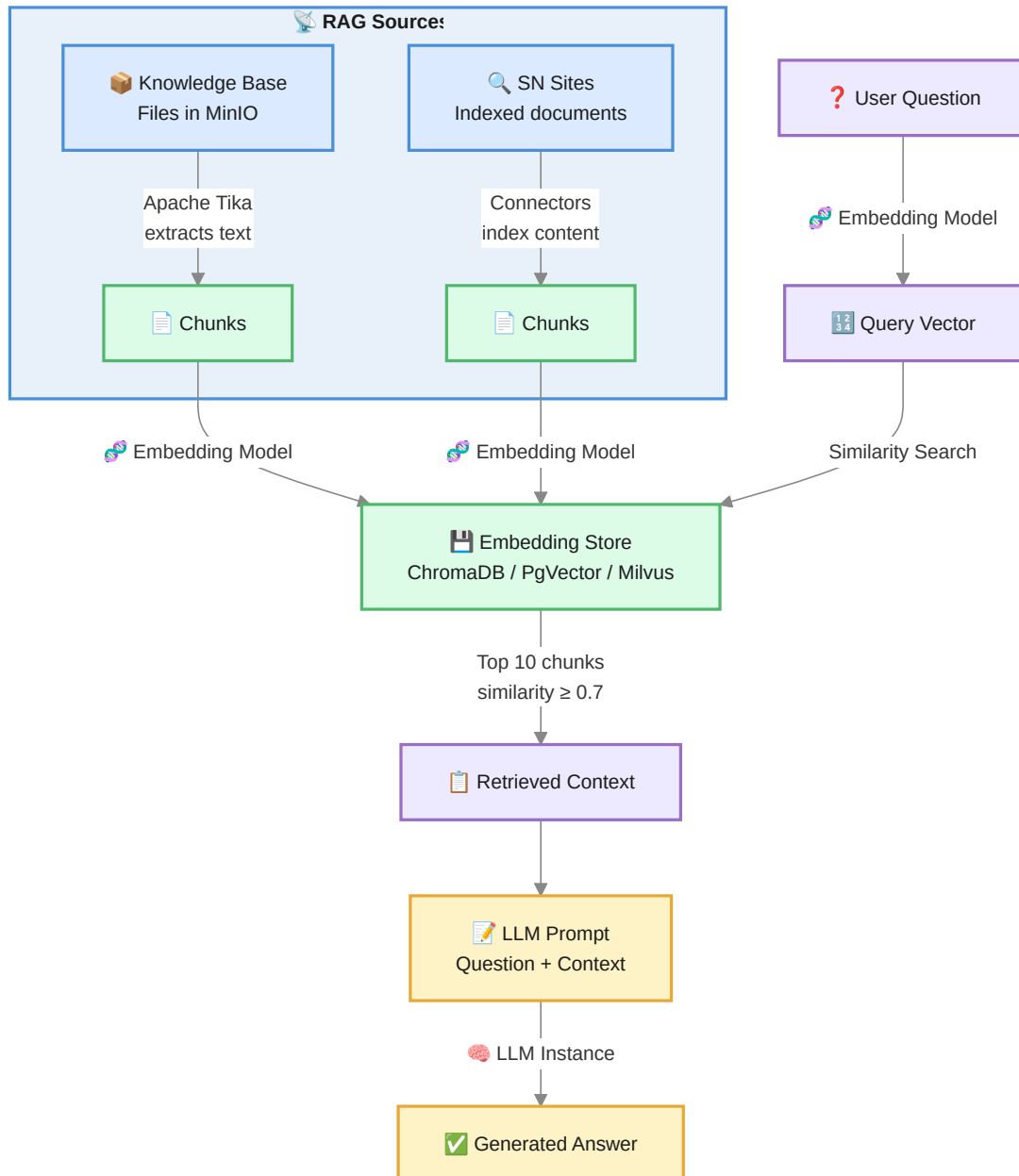
Query vector: [0.82, -0.15, 0.44, ...]

Stored vectors:

doc_chunk_42: [0.80, -0.13, 0.46, ...] → similarity: 0.97 ✓ (returned)
doc_chunk_17: [0.75, -0.20, 0.39, ...] → similarity: 0.89 ✓ (returned)
doc_chunk_91: [-0.31, 0.67, -0.22, ...] → similarity: 0.12 ✗ (below threshold)

RAG in Turing ES

Turing ES supports two RAG sources that can be used independently or together:



Knowledge Base (Assets)

Files uploaded to **Assets** are stored in MinIO, extracted with Apache Tika (supporting PDF, DOCX, XLSX, PPTX, HTML, TXT, and more), truncated to 100,000 characters, split into 1,024-character chunks, embedded, and stored in the active embedding store.

The Knowledge Base is queried by the `search_knowledge_base` tool — available in the Chat interface and configurable per AI Agent.

Semantic Navigation Sites

When **GenAI** is enabled for an SN Site, indexed documents are also embedded alongside their Solr representation. This allows the SN Site chat and AI Agents to retrieve relevant content via semantic similarity.

Configure this in **Semantic Navigation → [Site] → Generative AI**. See [Semantic Navigation](#).

⚠ RE-INDEXING REQUIRED

Documents indexed before GenAI was enabled do not have embeddings. A full re-indexing of the site is required to make existing content available for RAG.

AI Agents + Semantic Navigation: The Full Loop

Here is the question that determines whether your AI feels alive: *when a user asks a question, does the agent answer from its training data, or does it search **your** content first?*

Turing ES gives the agent the tools to do both — and the wiring is what makes them work together as one experience.

The two retrieval surfaces

An AI Agent that's wired for SN can reach the index through **two different routes**, each suited to a different shape of question.

ROUTE	NATIVE TOOL	WHAT IT DOES	BEST WHEN
Keyword + facets	<code>search_site</code> (and the helpers <code>list_sites</code> , <code>get_site_fields</code> , <code>get_valid_filter_values</code>)	Sends a Solr query to the SN Site backend, with optional facet filters and locale. Returns documents with snippets and metadata.	The user asks for something <i>describable in your taxonomy</i> — a product category, a brand, a date range
Semantic similarity (RAG)	<code>search_knowledge_base</code> and the SN site's vector lookup	Embeds the user's question, retrieves the top-K most similar document chunks from the embedding store, and includes them in the prompt	The user asks something fuzzy — " <i>what does our policy say about X</i> ", " <i>is there a way to do Y</i> "

Both call the *same indexed content*. The difference is **how** they search it. A site can have both enabled at once — the agent picks the right tool for the question.

What happens at chat time

Here's the sequence when a user asks an SN-grounded agent: "*Which of our products are best for cold-storage logistics?*"

Tool catalog presented to the LLM. The agent has `search_site`, `find_similar_documents`, and `search_knowledge_base` attached. Their JSON schemas are part of the prompt; the model can pick any of them.

LLM picks `search_site`. It's a domain question with a clear filter (cold storage). The model emits a tool call: `search_site(site="products", query="cold storage logistics", locale="en_US")`.

Solr handles the query. Turing ES translates the tool arguments into a Solr query against the products SN Site, applies the configured facets, returns the top results with snippets.

The model reads the snippets. They're fed back as the tool's result. The model now has *real product names, real descriptions, real URLs*.

Possibly a second tool call. If the snippets are thin, the model may chain `find_similar_documents(documentId="<one-of-the-results>")` to broaden the recommendation. Or `search_knowledge_base("cold storage compliance")` to pull policy context.

Final answer. The model composes the response in the agent's voice (and its **Persona**, if attached), with concrete product names and links — nothing hallucinated, everything grounded.

If the question had been different — say, *"Why does this product spec say it tolerates -25°C?"* — the LLM would have picked `search_knowledge_base` first, not `search_site`. The RAG embeddings carry semantics that keyword search doesn't.

The Semantic Navigation chat tab

The **Chat** interface has a dedicated **Semantic Navigation** tab where these tools are pre-wired without you having to build an agent. Use it when you want raw, grounded search-with-explanation. The system prompt for that tab includes:

The list of available SN Sites and their locales,
The configured facets per site (so the model knows what filters it can apply),
A directive to **never answer outside the indexed content**.

It's the Turing ES equivalent of *"talk to my site"*.

Designing an SN-backed agent

A few patterns that work well:

Limit which sites an agent can search. A sales agent shouldn't be searching the internal HR site. Even though `list_sites` returns all of them, you can shape behavior with the agent's system prompt: *"Only use the products and pricing SN Sites. Ignore others."*

Match agent persona to site content. A persona that says *"speak like a brand voice"* paired with an HR knowledge base will sound off. Pair brand-voice personas with marketing/product sites; pair instructional personas with internal docs.

Combine SN search with Custom Tools for action. Search returns information; a **Custom Tool** (or MCP) takes action. *"Find me the right product"* → `search_site` → *"and book me a demo on it"* → custom tool that hits your CRM.

The ANN Search Page: See Your Vectors

There's one question RAG operators always end up asking: *"Did the embedding store actually receive what I think it did?"*

The **ANN Search** page (`/ann/{siteName}`) is the answer. It's a developer-facing search page that hits the **vector store directly** — no LLM, no chat, no agent. You type a query, it embeds it, and shows you the raw top-K matches from the embedding store, with similarity scores and full metadata.

Think of it as the "*view source*" of your RAG pipeline.

What it shows

COLUMN	MEANING
Result content	The actual chunk text the embedding store returned
Score	Similarity score (cosine, typically 0.0-1.0)
Metadata	The document's source — URL, file path, locale, custom fields you indexed
Filters	Faceted metadata so you can narrow the query (e.g., <code>locale = pt_BR</code> , <code>source = aem</code>)
Pagination	Browse beyond the immediate top-K when you want to see what's <i>just</i> below the threshold
View raw JSON	The full vector record as it lives in ChromaDB / PgVector / Milvus

When to open it

SCENARIO	WHAT ANN SEARCH TELLS YOU
"My agent isn't finding the policy doc I just uploaded."	Search the page for the doc's title. If it's not there → indexing failed. If it's there with score 0.3 → embedding model is mismatched or chunks too coarse.
"My RAG answers feel generic."	Search a sample question. Look at the top 3 chunks. If they're all the same boilerplate, your chunking is too coarse. If they're irrelevant, the embedding model isn't right for your content.
"I need to verify locale filtering works."	Search with the locale facet. If the results aren't locale-scoped, the metadata isn't being indexed or the filter isn't being applied.
"After a re-index, did everything come back?"	Search a known document. If hit count drops vs. before, indexing dropped chunks.
"I want to debug a single bad answer."	Take the user's exact question, paste it into ANN Search, look at the top 10 chunks. Now you know exactly what context the LLM saw.

How it relates to the chat

ANN Search shares the same backing infrastructure as the agent's `search_knowledge_base` tool — same embedding model, same vector store, same query pipeline. **What you see in ANN Search is exactly what the LLM would have seen** when answering a related question.

This makes ANN Search the canonical debugging surface. Reproduce the chat answer there, and the cause is one of three things:

- The chunks the LLM got are wrong → fix indexing or chunking,

- The chunks were right but the LLM ignored them → tighten the agent system prompt or persona,

- The chunks were right and the LLM used them but the user is asking something not covered → product gap, not a tech gap.

The first two you fix in Turing. The third is a roadmap signal.

! ANN SEARCH IS RAG-ONLY

The ANN page only works on SN Sites where **GenAI is enabled** (i.e., embeddings are being indexed). If a site shows "*ANN Search is not available for this site (RAG is disabled)*", the site doesn't have RAG turned on. Enable it in **Semantic Navigation** → **[Site]** → **Generative AI** and re-index.

REST API behind the page

The ANN page is just a UI over a small REST contract. Your own tools can call it directly:

```
POST /ann/{siteName}/search
Content-Type: application/json

{
  "query": "cold storage compliance",
  "locale": "en_US",
  "topK": 10,
  "filters": { "source": ["aem"] }
}
```

Response:

```
{
  "query": "cold storage compliance",
  "locale": "en_US",
  "topK": 10,
  "page": 0,
  "pageSize": 10,
  "totalHits": 42,
  "hasMore": true,
  "results": [
    {
      "id": "doc-789-chunk-3",
      "content": "Storage at temperatures below -20°C requires...",
      "score": 0.92,
      "metadata": { "url": "...", "locale": "en_US", "source": "aem" }
    }
  ],
  "facets": {
    "source": [{ "value": "aem", "count": 38 }, { "value": "manual", "count":
4 } ]
  }
}
```

This is also a useful endpoint for *automated quality checks* — a CI job that verifies critical content is still embedded after a re-index.

Practical Example

Imagine a company with an internal knowledge base containing HR policies, product documentation, and engineering guides.

Without RAG:

User: "What is our vacation policy for remote employees?"

LLM: "Typically, companies offer 15-20 days of PTO..." (*generic, possibly wrong for this company*)

With RAG:

User: "What is our vacation policy for remote employees?"

Turing ES retrieves: Chunk from `hr-policies-2025.pdf` (page 12): "Remote employees are entitled to 25 days of paid leave per year, plus 5 floating holidays. Requests must be submitted via the HR portal at least 14 days in advance..."

LLM (with context): "According to our HR policies, remote employees receive 25 days of paid leave per year plus 5 floating holidays. Leave requests must be submitted via the HR portal at least 14 days in advance." (accurate, sourced from actual company data)

Key Configuration

SETTING	WHERE	DEFAULT	IMPACT
Embedding Store	Administration → Settings	—	Which vector database to use
Embedding Model	Administration → Settings	Provider default	Quality and dimensions of vectors
Default LLM Instance	Administration → Settings	—	Which model generates responses
RAG Enabled	Administration → Settings	Off	Whether new SN Sites have RAG by default
Top-K results	Internal	10	How many chunks are retrieved
Similarity threshold	Internal	0.7	Minimum similarity to include a chunk

Related Pages

PAGE	DESCRIPTION
Generative AI & LLM Configuration	Global GenAI settings and architecture overview
LLM Instances	Configure LLM providers
Embedding Stores	Vector database backends (ChromaDB, PgVector, Milvus)
Embedding Models	Provider support and model selection guidance
Assets	Knowledge Base file management
AI Agents	Compose agents with RAG tools
Tool Calling	RAG / Knowledge Base tools reference

LLM Instances

The **Language Model** page (`/admin/llm/instance`) is the central place to configure the AI models that power the Turing ES Generative AI features. It is accessible from the **Generative AI** section of the sidebar.

Each **LLM Instance** is a named, configured connection to an LLM provider. Multiple instances can coexist — different AI Agents, SN Sites, and the Chat interface can each use a different instance. This allows you to, for example, use a fast local Ollama model for low-stakes tasks and Anthropic Claude Sonnet for complex reasoning agents.

Instance Listing

The page displays all configured instances as a grid of cards (title and description). Use the **"New language model instance"** button to create a new one.

Create / Edit Form

The form is organized into **5 colour-coded sections** for quick visual orientation.

1. General Information (blue)

FIELD	REQUIRED	DESCRIPTION
Title	✓	Display name for this instance — appears in dropdowns and agent configuration
Vendor	✓	Select the LLM provider. Selecting a vendor applies sensible defaults to Endpoint URL and Model Name automatically.
Description		Free-text notes about this instance's purpose

2. Model Settings (purple)

FIELD	REQUIRED	DESCRIPTION
Endpoint URL	✓	Base URL for the provider API (e.g., <code>https://api.openai.com</code> , <code>http://localhost:11434</code>)
Model Name	✓	Specific model identifier (e.g., <code>gpt-4o-mini</code> , <code>mistral</code> , <code>claude-sonnet-4-20250514</code>)
API Key		Provider API key — stored encrypted in the database. Leave blank when editing to keep the existing key.

⚠ API KEY SECURITY

The API Key field is **write-only**. It is stored encrypted and never returned in API responses. When editing an existing instance, leaving the field blank preserves the previously saved key. The encryption key is configured in `turing.ai.crypto.key` in `application.yaml`. See [Configuration Reference](#).

3. Generation Parameters (emerald)

Fine-tune how the model generates responses. Defaults are appropriate for most use cases.

FIELD	DESCRIPTION	NOTES
Temperature	Randomness of the output (0.0 = deterministic, 1.0 = very creative)	Applies to all vendors
Top P	Nucleus sampling — restricts token selection to the top P probability mass	Applies to all vendors
Seed	Fixed seed for reproducible outputs	Only available for OLLAMA , OPENAI , and AZURE_OPENAI

4. Advanced Options (amber)

FIELD	DESCRIPTION
Response Format	Output format: <code>TEXT</code> (default) or <code>JSON</code>
Supported Capabilities	Comma-separated list of feature flags (e.g., <code>RESPONSE_FORMAT_JSON_SCHEMA</code>)
Timeout	Maximum time to wait for a response, in ISO 8601 duration format (e.g., <code>PT60S</code> = 60 seconds)
Max Retries	Number of retry attempts on transient failures. Default: <code>3</code>
Provider Options (Visual)	Vendor-specific fields rendered dynamically based on the selected vendor (see Provider Options below)
Provider Options (JSON)	Raw JSON override for any vendor-specific setting — useful for advanced configurations not exposed in the visual fields

5. Status (slate)

FIELD	DESCRIPTION
Enabled	Toggle to activate or deactivate this instance. Disabled instances are not available for selection in agents or sites.
Tools Enabled	Toggle to allow this instance to use function calling (tools such as web search, code interpreter, etc.)

Supported Vendors

Six vendor types are supported. When a vendor is selected in the form, the Endpoint URL and Model Name fields are pre-filled with the defaults shown below.

VENDOR	DEFAULT ENDPOINT	DEFAULT MODEL
OLLAMA	http://localhost:11434	mistral
OPENAI	https://api.openai.com	gpt-4o-mini
ANTHROPIC	https://api.anthropic.com	claude-sonnet-4-20250514
GEMINI	(Google native API)	gemini-2.0-flash
GEMINI_OPENAI	https://generativelanguage.googleapis.com/v1beta/openai	gemini-2.0-flash
AZURE_OPENAI	(configured via provider options)	gpt-4o

Provider Options

Each vendor exposes additional fields in the **Provider Options (Visual)** section. These fields appear dynamically when that vendor is selected.

VENDOR	AVAILABLE FIELDS
OLLAMA	embeddingModel , topK , repeatPenalty , numPredict , stop
OPENAI	embeddingModel , maxTokens
ANTHROPIC	topK , maxTokens
GEMINI	topK , maxTokens
GEMINI_OPENAI	maxTokens
AZURE_OPENAI	deploymentName , embeddingDeploymentName , maxTokens

💡 AZURE OPENAI

For Azure OpenAI, the `deploymentName` provider option is required — it specifies the name of your deployed model in the Azure portal. The endpoint must also be set to your Azure OpenAI resource URL (e.g., `https://my-resource.openai.azure.com`).

Capabilities by Vendor

Not all vendors support all features. The table below shows which capabilities are available per vendor:

VENDOR	CHAT	EMBEDDING	TOOL CALLING	SEED
OLLAMA	✓	✓ (configurable)	✓	✓
OPENAI	✓	✓ (<code>text-embedding-3-small</code>)	✓	✓
ANTHROPIC	✓	✗	✓	✗
GEMINI	✓	✗	✓	✗
GEMINI_OPENAI	✓	✗	✓	✗
AZURE_OPENAI	✓	✓ (<code>text-embedding-ada-002</code>)	✓	✓

ⓘ EMBEDDING VENDORS

If you need embedding support (for RAG and the Knowledge Base), use **OLLAMA**, **OPENAI**, or **AZURE_OPENAI**. The other vendors can be used for chat and tool calling but not for vector generation.

Security

API Keys are handled with care at every layer:

Stored encrypted — the key is encrypted before being persisted to the database.

Never returned — the API Key field is write-only: it flows in on save but never comes back in API responses or GET endpoints.

Edit safely — leaving the API Key field blank when editing an instance preserves the existing encrypted value without modification.

Encryption key — configured via `turing.ai.crypto.key` in `application.yaml`. **Always set a strong, unique value in production** — the default is a placeholder and must be changed before handling real API keys.

Caching

LLM Instance data is cached at the repository layer to avoid repeated database reads during high-throughput inference:

`turLLMInstancefindAll` — caches the full list of instances

`turLLMInstancefindById` — caches individual instance lookups

Vendor metadata is also cached

Cache entries are invalidated automatically on create, update, or delete.

Related Pages

PAGE	DESCRIPTION
Generative AI & LLM Configuration	Conceptual overview of RAG, embeddings, tool calling, and agents
Chat	Using the chat interface with configured LLM instances
Token Usage	Monitor token consumption per instance
Assets	Knowledge Base files — requires an embedding-capable instance
Configuration Reference	<code>turing.ai.crypto.key</code> and other application settings

Embedding Stores

An **Embedding Store** is the specialized vector database that persists and queries document embeddings — the numerical vectors generated by the **Embedding Model**. It enables similarity search, finding the documents most semantically related to a user's query.

Turing ES supports three backends via Spring AI. The active backend is set globally in **Administration** → **Settings** and can be overridden per Semantic Navigation Site in its **Generative AI** tab.

Supported Backends

ChromaDB

A lightweight, open-source vector database ideal for development and small to medium deployments.

- Self-hosted, connects via its HTTP API

- Zero infrastructure overhead for teams already running Python tooling

- No special schema setup required — Turing ES manages the collections automatically

- Multi-tenant and multi-database support

Docker Compose quickstart:

```
services:
  chroma:
    image: chromadb/chroma:latest
    ports:
      - "8000:8000"
```

CONFIGURATION	DEFAULT	DESCRIPTION
Base URL	<code>http://localhost:8000</code>	Chroma HTTP API endpoint
Collection Name	<code>turing</code>	Target collection name
Tenant Name	<code>default_tenant</code>	Chroma tenant identifier
Database Name	<code>default_database</code>	Chroma database name
Key Token	—	Bearer token for authentication
Basic Username / Password	—	Basic auth credentials

Authentication: ChromaDB supports two methods — **Bearer token** and **Basic auth**. Configure either via the credential field or the provider options. When both are present, the credential field takes precedence.

PgVector

PostgreSQL with the `pgvector` extension — the best choice for deployments that already use PostgreSQL as their primary database.

Avoids an additional infrastructure dependency

Embeddings live in the same database as your application data

Supports standard PostgreSQL backup, replication, and access control

Connection pooling via HikariCP (max 5 connections per store)

Enable the required extensions in your PostgreSQL instance:

```
CREATE EXTENSION IF NOT EXISTS vector;
CREATE EXTENSION IF NOT EXISTS hstore;
CREATE EXTENSION IF NOT EXISTS "uuid-oss";
```


CONFIGURATION	DEFAULT	DESCRIPTION
JDBC URL	—	PostgreSQL connection string (e.g., <code>jdbc:postgresql://localhost:5432/turing</code>)
Username	—	Database user
Password	—	Database password
Table Name	<code>vector_store</code>	Table where embeddings are stored
Schema Name	<code>public</code>	PostgreSQL schema
Dimensions	—	Vector dimensionality — must match the <code>embedding model</code>
Distance Type	—	<code>COSINE_DISTANCE</code> , <code>EUCLIDEAN_DISTANCE</code> , or <code>INNER_PRODUCT</code>
Index Type	—	<code>HNSW</code> or <code>IVFFLAT</code>

Table schema created by Turing ES:

```
CREATE TABLE IF NOT EXISTS "vector_store" (  
  id      UUID DEFAULT uuid_generate_v4() PRIMARY KEY,  
  content TEXT,  
  metadata JSON,  
  embedding VECTOR  
);
```

Milvus

A purpose-built, cloud-native vector database designed for high-scale similarity search.

Recommended for large corpora or high-throughput deployments

Supports distributed operation, horizontal scaling, and advanced index management

Managed cloud offering available (Zilliz Cloud)

CONFIGURATION	DEFAULT	DESCRIPTION
Base URL	<code>http://localhost:19530</code>	Milvus service URI
Collection Name	<code>turing</code>	Target collection
Database Name	—	Optional database name
Token	—	Authentication token (<code>username:password</code> format)
Embedding Dimension	—	Vector dimensionality — must match the embedding model
Metric Type	—	<code>COSINE</code> , <code>L2</code> , or <code>IP</code> (inner product)
Index Type	—	<code>HNSW</code> , <code>IVF_FLAT</code> , <code>IVF_SQ8</code> , or <code>DISKANN</code>
Index Parameters	—	JSON string with index-specific params (e.g., <code>{"M":16,"efConstruction":200}</code>)

Store Comparison

FEATURE	CHROMADB	PGVECTOR	MILVUS
Best for	Dev / small-medium	PostgreSQL shops	Large-scale production
Infrastructure	Standalone container	PostgreSQL extension	Dedicated cluster
Scaling	Single node	PostgreSQL replication	Horizontal / distributed
Index types	Automatic	HNSW, IVFFLAT	HNSW, IVF_FLAT, IVF_SQ8, DISKANN
Distance metrics	Cosine, L2	Cosine, L2, Inner Product	Cosine, L2, Inner Product
Multi-tenant	Yes (tenant + database)	Via schema	Yes (database + collection)
Authentication	Token or Basic Auth	JDBC credentials	Token
Managed cloud	—	Any managed PostgreSQL	Zilliz Cloud

Create / Edit Form

Navigate to **Generative AI → Embedding Stores** to manage store instances.

General Information

FIELD	REQUIRED	DESCRIPTION
Title	Yes	Display name for this store instance
Description	Yes	Brief description of its purpose
Vendor	Yes	Select the backend: ChromaDB, PgVector, or Milvus. Selecting a vendor applies default values for Endpoint URL and Collection Name.

Connection

FIELD	REQUIRED	DESCRIPTION
Endpoint URL	Yes	Base URL for the store backend
Collection Name		Name of the collection or table (vendor-specific default applied)
Credential		Authentication token or password — stored encrypted. Leave blank when editing to keep the existing value.

Provider Options

Each vendor exposes additional configuration fields in the **Provider Options** section. These fields appear dynamically when a vendor is selected. A raw JSON editor is also available for advanced configurations.

Status

FIELD	DESCRIPTION
Enabled	Toggle to activate or deactivate this store. Disabled stores are not available for selection.

Vendor defaults applied on selection:

VENDOR	DEFAULT URL	DEFAULT COLLECTION
ChromaDB	<code>http://localhost:8000</code>	<code>turing</code>
PgVector	<code>jdbc:postgresql://localhost:5432/turing</code>	<code>vector_store</code>
Milvus	<code>http://localhost:19530</code>	<code>turing</code>

Collection Management

Each store instance provides a **Collections** page where you can view and manage vector collections.

The collections table shows:

COLUMN	DESCRIPTION
Collection Name	Name of the collection or table
ID	Internal identifier
Document Count	Number of distinct source documents embedded
Chunk Count	Total number of embedding chunks stored

Available actions per collection:

ACTION	DESCRIPTION
Create	Create a new empty collection in the store
Clear	Remove all embeddings from the collection while keeping the collection structure
Delete	Remove the entire collection and all its data

CLEARING OR DELETING COLLECTIONS

Clearing or deleting a collection permanently removes all stored embeddings. A full re-indexing of all content is required to rebuild the vectors. This action cannot be undone.

System Information

The **System Info** endpoint returns backend-specific metadata for monitoring and diagnostics:

BACKEND	INFORMATION RETURNED
ChromaDB	ChromaDB version
PgVector	PostgreSQL version, pgvector extension version
Milvus	Milvus server version

REST API

Store instances are managed via the REST API at `/api/store`.

Store Instance Endpoints

METHOD	ENDPOINT	DESCRIPTION
GET	<code>/api/store</code>	List all store instances (ordered by title)
GET	<code>/api/store/structure</code>	Get the structure template for a new instance
GET	<code>/api/store/{id}</code>	Get a specific store instance
POST	<code>/api/store</code>	Create a new store instance
PUT	<code>/api/store/{id}</code>	Update an existing store instance
DELETE	<code>/api/store/{id}</code>	Delete a store instance

Collection Endpoints

METHOD	ENDPOINT	DESCRIPTION
GET	/api/store/{id}/collections	List all collections in a store
POST	/api/store/{id}/collections/{name}	Create a new collection
DELETE	/api/store/{id}/collections/{name}	Delete a collection
DELETE	/api/store/{id}/collections/{name}/clear	Clear all embeddings from a collection

System Info Endpoint

METHOD	ENDPOINT	DESCRIPTION
GET	/api/store/{id}/system-info	Get store backend version and status

Store Vendor Endpoints

METHOD	ENDPOINT	DESCRIPTION
GET	/api/store/vendor	List all available vendors
GET	/api/store/vendor/{id}	Get a specific vendor

Global Configuration

Set the default embedding store in **Administration** → **Settings**:

SETTING	DESCRIPTION
Default Embedding Store	Which vector database backend to use (ChromaDB, PgVector, or Milvus)

Individual Semantic Navigation Sites can override this setting in their **Generative AI** tab. The Knowledge Base always uses the global default.

Security

Credentials are handled with care at every layer:

Stored encrypted — credentials are encrypted before being persisted to the database

Never returned — the `credential` field is transient and write-only. It flows in on save but never comes back in API responses

Edit safely — leaving the credential field blank when editing preserves the existing encrypted value

Per-vendor auth — ChromaDB supports Bearer token or Basic Auth; Milvus uses token-based auth; PgVector uses JDBC credentials with connection pooling

Caching

Store instance and vendor data is cached at the repository layer to avoid repeated database reads:

`turStoreInstancefindAll` — caches the full list of instances

`turStoreInstancefindById` — caches individual instance lookups

`turStoreVendorfindAll` / `turStoreVendorfindById` — caches vendor metadata

Cache entries are invalidated automatically on create, update, or delete.

Related Pages

PAGE	DESCRIPTION
Embedding Models	Configure the models that generate vectors stored here
What is RAG?	How embedding stores fit into the RAG pipeline
GenAI & LLM Configuration	RAG architecture, RAG sources, and system overview
LLM Instances	Configure the LLM providers that supply embedding APIs
Assets	Knowledge Base files that are indexed into the Embedding Store
Semantic Navigation	Generative AI tab: per-site embedding overrides

Embedding Models

An **Embedding Model** converts text into a high-dimensional numerical vector that captures its semantic meaning. These vectors enable similarity search — finding documents conceptually related to a query even when they share no common keywords.

Turing ES uses embedding models in two phases:

Indexing — document chunks are vectorized and stored in the **Embedding Store**

Querying — the user's question is vectorized with the same model and compared against stored vectors

SAME MODEL FOR INDEXING AND QUERYING

The embedding model must remain consistent across both phases. Changing the model after documents have been indexed causes dimension mismatches and incorrect similarity results. A **full re-indexing** of all content is required whenever the embedding model changes.

Supported Providers

Embedding model support depends on the LLM vendor configured in the **LLM Instance**. Not all vendors provide an embedding API.

PROVIDER	EMBEDDING SUPPORT	EXAMPLE MODELS	DEFAULT MODEL
OpenAI	Yes	<code>text-embedding-3-small</code> , <code>text-embedding-3-large</code> , <code>text-embedding-ada-002</code>	<code>text-embedding-3-small</code>
Azure OpenAI	Yes	Deployment name of an embedding model in your Azure resource	<code>text-embedding-ada-002</code>
Ollama	Yes	<code>omic-embed-text</code> , <code>mxbai-embed-large</code> , <code>all-minilm</code> , <code>stella-v5</code>	(configurable)
Anthropic	No	—	—
Gemini	No	—	—
Gemini (OpenAI-compatible)	No	—	—

OpenAI

Connects to the OpenAI API (default: `https://api.openai.com`) using your API key. OpenAI offers three embedding model families:

MODEL	DIMENSIONS	NOTES
<code>text-embedding-3-small</code>	1,536	Best cost-performance balance for most deployments
<code>text-embedding-3-large</code>	3,072	Higher quality, larger storage footprint
<code>text-embedding-ada-002</code>	1,536	Legacy model — use <code>3-small</code> for new deployments

Azure OpenAI

Uses the same OpenAI embedding models, hosted on your Azure tenant. Configuration requires:

Endpoint — your Azure OpenAI resource URL (e.g., `https://my-resource.openai.azure.com`)

Embedding Deployment Name — the deployment name created in the Azure portal

API Key — Azure API key (stored encrypted)

Ollama (Local)

Runs embedding models locally via [Ollama](#). No API key required for local deployments — ideal for air-gapped environments or development.

MODEL	DIMENSIONS	NOTES
<code>nomic-embed-text</code>	768	Good general-purpose model, lightweight
<code>mxbai-embed-large</code>	1,024	Higher quality, more resource-intensive
<code>all-minilm</code>	384	Very lightweight, fast inference

Pull a model before using it:

```
ollama pull nomic-embed-text
```

Local Transformers (ONNX)

Turing ES also supports running embedding models locally via ONNX Runtime, without an external LLM provider. This is useful for deploying custom or fine-tuned models.

SETTING	DESCRIPTION
Model Path	Absolute path to the <code>.onnx</code> model file
Tokenizer Path	Absolute path to <code>tokenizer.json</code>
Enable GPU	Toggle GPU acceleration via ONNX Runtime
Batch Size	Number of texts to embed per batch

Navigate to **Generative AI → Embedding Models** to manage embedding model configurations.

FIELD	REQUIRED	DESCRIPTION
Model Name	Yes	Display name for this embedding model
Description		Free-text notes about the model's purpose

FIELD	REQUIRED	DESCRIPTION
LLM Instance	Yes*	Select the LLM Instance that provides the embedding API. Not required for Local Transformers.
Model Reference	Yes	Technical model identifier (e.g., <code>text-embedding-3-large</code> , <code>gpt-4o</code> , <code>gpt-4o-mini</code> , <code>gpt-4o-2024-08-07</code> , <code>gpt-4o-2024-05-13</code> , <code>gpt-4o-2024-03-16</code> , <code>gpt-4o-2024-01-25</code> , <code>gpt-4o-2023-11-06</code> , <code>gpt-4o-2023-08-01</code> , <code>gpt-4o-2023-07-12</code> , <code>gpt-4o-2023-06-13</code> , <code>gpt-4o-2023-05-07</code> , <code>gpt-4o-2023-04-09</code> , <code>gpt-4o-2023-03-16</code> , <code>gpt-4o-2023-02-15</code> , <code>gpt-4o-2023-01-27</code> , <code>gpt-4o-2023-01-19</code> , <code>gpt-4o-2023-01-10</code> , <code>gpt-4o-2023-01-03</code> , <code>gpt-4o-2022-12-01</code> , <code>gpt-4o-2022-11-06</code> , <code>gpt-4o-2022-10-27</code> , <code>gpt-4o-2022-10-19</code> , <code>gpt-4o-2022-10-10</code> , <code>gpt-4o-2022-09-01</code> , <code>gpt-4o-2022-08-01</code> , <code>gpt-4o-2022-07-01</code> , <code>gpt-4o-2022-06-01</code> , <code>gpt-4o-2022-05-01</code> , <code>gpt-4o-2022-04-01</code> , <code>gpt-4o-2022-03-01</code> , <code>gpt-4o-2022-02-01</code> , <code>gpt-4o-2022-01-01</code> , <code>gpt-4o-2021-12-01</code> , <code>gpt-4o-2021-11-01</code> , <code>gpt-4o-2021-10-01</code> , <code>gpt-4o-2021-09-01</code> , <code>gpt-4o-2021-08-01</code> , <code>gpt-4o-2021-07-01</code> , <code>gpt-4o-2021-06-01</code> , <code>gpt-4o-2021-05-01</code> , <code>gpt-4o-2021-04-01</code> , <code>gpt-4o-2021-03-01</code> , <code>gpt-4o-2021-02-01</code> , <code>gpt-4o-2021-01-01</code> , <code>gpt-4o-2020-12-01</code> , <code>gpt-4o-2020-11-01</code> , <code>gpt-4o-2020-10-01</code> , <code>gpt-4o-2020-09-01</code> , <code>gpt-4o-2020-08-01</code> , <code>gpt-4o-2020-07-01</code> , <code>gpt-4o-2020-06-01</code> , <code>gpt-4o-2020-05-01</code> , <code>gpt-4o-2020-04-01</code> , <code>gpt-4o-2020-03-01</code> , <code>gpt-4o-2020-02-01</code> , <code>gpt-4o-2020-01-01</code> , <code>gpt-4o-2019-12-01</code> , <code>gpt-4o-2019-11-01</code> , <code>gpt-4o-2019-10-01</code> , <code>gpt-4o-2019-09-01</code> , <code>gpt-4o-2019-08-01</code> , <code>gpt-4o-2019-07-01</code> , <code>gpt-4o-2019-06-01</code> , <code>gpt-4o-2019-05-01</code> , <code>gpt-4o-2019-04-01</code> , <code>gpt-4o-2019-03-01</code> , <code>gpt-4o-2019-02-01</code> , <code>gpt-4o-2019-01-01</code> , <code>gpt-4o-2018-12-01</code> , <code>gpt-4o-2018-11-01</code> , <code>gpt-4o-2018-10-01</code> , <code>gpt-4o-2018-09-01</code> , <code>gpt-4o-2018-08-01</code> , <code>gpt-4o-2018-07-01</code> , <code>gpt-4o-2018-06-01</code> , <code>gpt-4o-2018-05-01</code> , <code>gpt-4o-2018-04-01</code> , <code>gpt-4o-2018-03-01</code> , <code>gpt-4o-2018-02-01</code> , <code>gpt-4o-2018-01-01</code> , <code>gpt-4o-2017-12-01</code> , <code>gpt-4o-2017-11-01</code> , <code>gpt-4o-2017-10-01</code> , <code>gpt-4o-2017-09-01</code> , <code>gpt-4o-2017-08-01</code> , <code>gpt-4o-2017-07-01</code> , <code>gpt-4o-2017-06-01</code> , <code>gpt-4o-2017-05-01</code> , <code>gpt-4o-2017-04-01</code> , <code>gpt-4o-2017-03-01</code> , <code>gpt-4o-2017-02-01</code> , <code>gpt-4o-2017-01-01</code> , <code>gpt-4o-2016-12-01</code> , <code>gpt-4o-2016-11-01</code> , <code>gpt-4o-2016-10-01</code> , <code>gpt-4o-2016-09-01</code> , <code>gpt-4o-2016-08-01</code> , <code>gpt-4o-2016-07-01</code> , <code>gpt-4o-2016-06-01</code> , <code>gpt-4o-2016-05-01</code> , <code>gpt-4o-2016-04-01</code> , <code>gpt-4o-2016-03-01</code> , <code>gpt-4o-2016-02-01</code> , <code>gpt-4o-2016-01-01</code> , <code>gpt-4o-2015-12-01</code> , <code>gpt-4o-2015-11-01</code> , <code>gpt-4o-2015-10-01</code> , <code>gpt-4o-2015-09-01</code> , <code>gpt-4o-2015-08-01</code> , <code>gpt-4o-2015-07-01</code> , <code>gpt-4o-2015-06-01</code> , <code>gpt-4o-2015-05-01</code> , <code>gpt-4o-2015-04-01</code> , <code>gpt-4o-2015-03-01</code> , <code>gpt-4o-2015-02-01</code> , <code>gpt-4o-2015-01-01</code> , <code>gpt-4o-2014-12-01</code> , <code>gpt-4o-2014-11-01</code> , <code>gpt-4o-2014-10-01</code> , <code>gpt-4o-2014-09-01</code> , <code>gpt-4o-2014-08-01</code> , <code>gpt-4o-2014-07-01</code> , <code>gpt-4o-2014-06-01</code> , <code>gpt-4o-2014-05-01</code> , <code>gpt-4o-2014-04-01</code> , <code>gpt-4o-2014-03-01</code> , <code>gpt-4o-2014-02-01</code> , <code>gpt-4o-2014-01-01</code> , <code>gpt-4o-2013-12-01</code> , <code>gpt-4o-2013-11-01</code> , <code>gpt-4o-2013-10-01</code> , <code>gpt-4o-2013-09-01</code> , <code>gpt-4o-2013-08-01</code> , <code>gpt-4o-2013-07-01</code> , <code>gpt-4o-2013-06-01</code> , <code>gpt-4o-2013-05-01</code> , <code>gpt-4o-2013-04-01</code> , <code>gpt-4o-2013-03-01</code> , <code>gpt-4o-2013-02-01</code> , <code>gpt-4o-2013-01-01</code> , <code>gpt-4o-2012-12-01</code> , <code>gpt-4o-2012-11-01</code> , <code>gpt-4o-2012-10-01</code> , <code>gpt-4o-2012-09-01</code> , <code>gpt-4o-2012-08-01</code> , <code>gpt-4o-2012-07-01</code> , <code>gpt-4o-2012-06-01</code> , <code>gpt-4o-2012-05-01</code> , <code>gpt-4o-2012-04-01</code> , <code>gpt-4o-2012-03-01</code> , <code>gpt-4o-2012-02-01</code> , <code>gpt-4o-2012-01-01</code> , <code>gpt-4o-2011-12-01</code> , <code>gpt-4o-2011-11-01</code> , <code>gpt-4o-2011-10-01</code> , <code>gpt-4o-2011-09-01</code> , <code>gpt-4o-2011-08-01</code> , <code>gpt-4o-2011-07-01</code> , <code>gpt-4o-2011-06-01</code> , <code>gpt-4o-2011-05-01</code> , <code>gpt-4o-2011-04-01</code> , <code>gpt-4o-2011-03-01</code> , <code>gpt-4o-2011-02-01</code> , <code>gpt-4o-2011-01-01</code> , <code>gpt-4o-2010-12-01</code> , <code>gpt-4o-2010-11-01</code> , <code>gpt-4o-2010-10-01</code> , <code>gpt-4o-2010-09-01</code> , <code>gpt-4o-2010-08-01</code> , <code>gpt-4o-2010-07-01</code> , <code>gpt-4o-2010-06-01</code> , <code>gpt-4o-2010-05-01</code> , <code>gpt-4o-2010-04-01</code> , <code>gpt-4o-2010-03-01</code> , <code>gpt-4o-2010-02-01</code> , <code>gpt-4o-2010-01-01</code> , <code>gpt-4o-2009-12-01</code> , <code>gpt-4o-2009-11-01</code> , <code>gpt-4o-2009-10-01</code> , <code>gpt-4o-2009-09-01</code> , <code>gpt-4o-2009-08-01</code> , <code>gpt-4o-2009-07-01</code> , <code>gpt-4o-</code>

These fields appear only when **Transformers (Local)** is selected as the provider type:

FIELD	DESCRIPTION
Model Path	Path to the ONNX model file (<code>.onnx</code> extension)
Tokenizer Path	Path to the tokenizer file (<code>tokenizer.json</code>)
Batch Size	Number of texts processed per inference batch
Enable GPU	Toggle hardware acceleration

Status

FIELD	DESCRIPTION
Enabled	Toggle to activate or deactivate this model. Disabled models are not available for selection.

Choosing a Model

The embedding model determines two things:

- Dimensionality** — the number of dimensions in the vector (e.g., 384, 768, 1,536, 3,072). Higher dimensions capture more nuance but require more storage.
- Semantic quality** — how well the model captures meaning. Larger models generally produce better similarity results at the cost of slower indexing.

Recommendations

SCENARIO	RECOMMENDED MODEL	PROVIDER
General production	text-embedding-3-small	OpenAI
Maximum quality	text-embedding-3-large	OpenAI
Local / air-gapped	nomic-embed-text	Ollama
Resource-constrained	all-minilm	Ollama
Azure enterprise	text-embedding-ada-002 deployment	Azure OpenAI
Custom fine-tuned	Your .onnx model	Local Transformers

For most deployments, a mid-sized model such as text-embedding-3-small (OpenAI) or nomic-embed-text (Ollama) provides a good balance between quality and performance.

Global Configuration

Set the default embedding model in **Administration** → **Settings**:

SETTING	DESCRIPTION
Default Embedding Model	The embedding model used to generate vectors at indexing and query time

Individual Semantic Navigation Sites can override this setting in their **Generative AI** tab.

The Knowledge Base always uses the global default.

REST API

Embedding models are managed via the REST API at `/api/embedding-model`.

METHOD	ENDPOINT	DESCRIPTION
GET	<code>/api/embedding-model</code>	List all embedding models
GET	<code>/api/embedding-model/structure</code>	Get the structure template for a new model
GET	<code>/api/embedding-model/{id}</code>	Get a specific embedding model
POST	<code>/api/embedding-model</code>	Create a new embedding model
PUT	<code>/api/embedding-model/{id}</code>	Update an existing embedding model
DELETE	<code>/api/embedding-model/{id}</code>	Delete an embedding model

Per-Site Override

Each Semantic Navigation Site can override the global embedding model in its **Generative AI** tab. This allows different sites to use different models — for example, a multilingual site might use a model optimized for cross-language embeddings while a technical site uses a domain-specific

model.

The site-level configuration includes:

SETTING	DESCRIPTION
Embedding Model	Overrides the global default for this site
Embedding Store	Overrides the global store backend for this site
LLM Instance	The chat/reasoning model for this site's GenAI features

Caching

Embedding model data is cached at the repository layer to avoid repeated database reads during high-throughput indexing:

`turEmbeddingModelfindAll` — caches the full list of models

`turEmbeddingModelfindById` — caches individual model lookups

Cache entries are invalidated automatically on create, update, or delete.

Related Pages

PAGE	DESCRIPTION
Embedding Stores	Vector database backends (ChromaDB, PgVector, Milvus)
What is RAG?	How embedding models fit into the RAG pipeline
LLM Instances	Configure the LLM providers that supply embedding APIs
Assets	Knowledge Base files indexed using the embedding model
Semantic Navigation	Per-site GenAI and embedding overrides
GenAI & LLM Configuration	Global settings and architecture overview

Assets

Assets is the file manager that feeds the RAG Knowledge Base in Turing ES. Content managers and administrators use it to upload documents (PDFs, Word files, spreadsheets, and more) that are automatically indexed as vector embeddings and made available to AI Agents for semantic search. It is the bridge between your files and the Knowledge Base — everything uploaded here becomes part of what the AI can reference when answering questions.

The **Assets** section (`/console/asset`) is a file manager with built-in RAG training capabilities. It is available in the **Management** section of the sidebar and is only visible when a **storage backend is configured** (`turing.storage.type` is `minio` or `filesystem`).

Assets serves as the Knowledge Base for AI Agents — every file uploaded here can be indexed as vector embeddings and queried by the LLM via tool calling. For the conceptual overview of how this fits into the GenAI architecture, see [Generative AI & LLM Configuration](#).

! STORAGE REQUIRED

Assets and all RAG Knowledge Base features require a storage backend to be configured. See [Storage Configuration](#) at the bottom of this page.

Layout

The interface uses a **resizable dual-panel layout**:

Left panel — file and folder listing with the action toolbar

Right panel — inline preview of the selected file

A **breadcrumb** at the top of the left panel shows the current folder path and allows navigation to any parent level. A **Root** button returns to the top-level folder instantly.

File Table

The file listing displays the following columns:

COLUMN	DESCRIPTION
Name	File or folder name
Size	File size in human-readable format
Type	MIME type or folder indicator
Last Modified	Date and time of the last modification
AI	Training status — a checkmark indicates the file has been indexed as embeddings, with a tooltip showing the training timestamp
Actions	Per-row download and delete buttons

File Management

Upload Files

Files are uploaded to the **current folder** via a file picker dialog (click **Upload Files**). Multiple files can be selected in one operation. Uploads are sent to:

```
POST /api/asset
```

After upload, an **asynchronous event automatically triggers individual AI indexing** for each uploaded file — no manual training step is needed for new uploads.

Create Folder

A dialog prompts for a folder name. Folders can be nested to any depth and are navigated via the breadcrumb.

Download

Each file has a dedicated download button that preserves the original filename.

Delete

Files and folders can be deleted via an inline button. A **toast notification** confirms completion. When a file is deleted, its **embeddings are automatically removed from the vector store**.

NO RENAME

Renaming files or folders is not currently supported. To rename a file, download it, delete the original, and re-upload with the new name.

Supported File Formats for Training

Apache Tika is the text-extraction engine used during AI Training. It supports a broad range of document types:

CATEGORY	FORMATS
Documents	PDF, DOCX, DOC, ODT, RTF, EPUB
Spreadsheets	XLSX, XLS, ODS, CSV
Presentations	PPTX, PPT, ODP
Web / Markup	HTML, XHTML, XML
Plain text	TXT, LOG, Markdown
Email	EML, MSG, MBOX
Images (with OCR)	PNG, JPEG, TIFF, BMP, GIF (requires Tesseract)

Files that Tika cannot extract text from (e.g. binary executables, ZIP archives without textual content) are **silently skipped** during training with no error counted.

Preview Panel

Selecting a file opens an inline preview in the right panel without leaving the page. Supported formats:

CATEGORY	FORMATS
Images	PNG, JPEG, GIF, WebP, SVG, BMP
PDFs	Rendered via iframe
Video	MP4, WebM, OGG (with player controls)
Audio	MP3, OGG, WAV, WebM (with player controls)
Text	TXT, CSV, HTML, CSS, JS, JSON, XML

Panel actions:

Maximise — opens fullscreen view (press Esc to close)

Download — downloads the file directly from the preview panel

Close — collapses the preview panel

The panel footer displays the file size, content type, modification date, and file extension.

AI Training (RAG)

The AI training features are only available when `ragEnabled=true` **and** an embedding model and embedding store are configured in **Administration** → **Global Settings** → **RAG Settings**.

Training Status per File

The **AI column** in the file table shows the indexing state of each file:

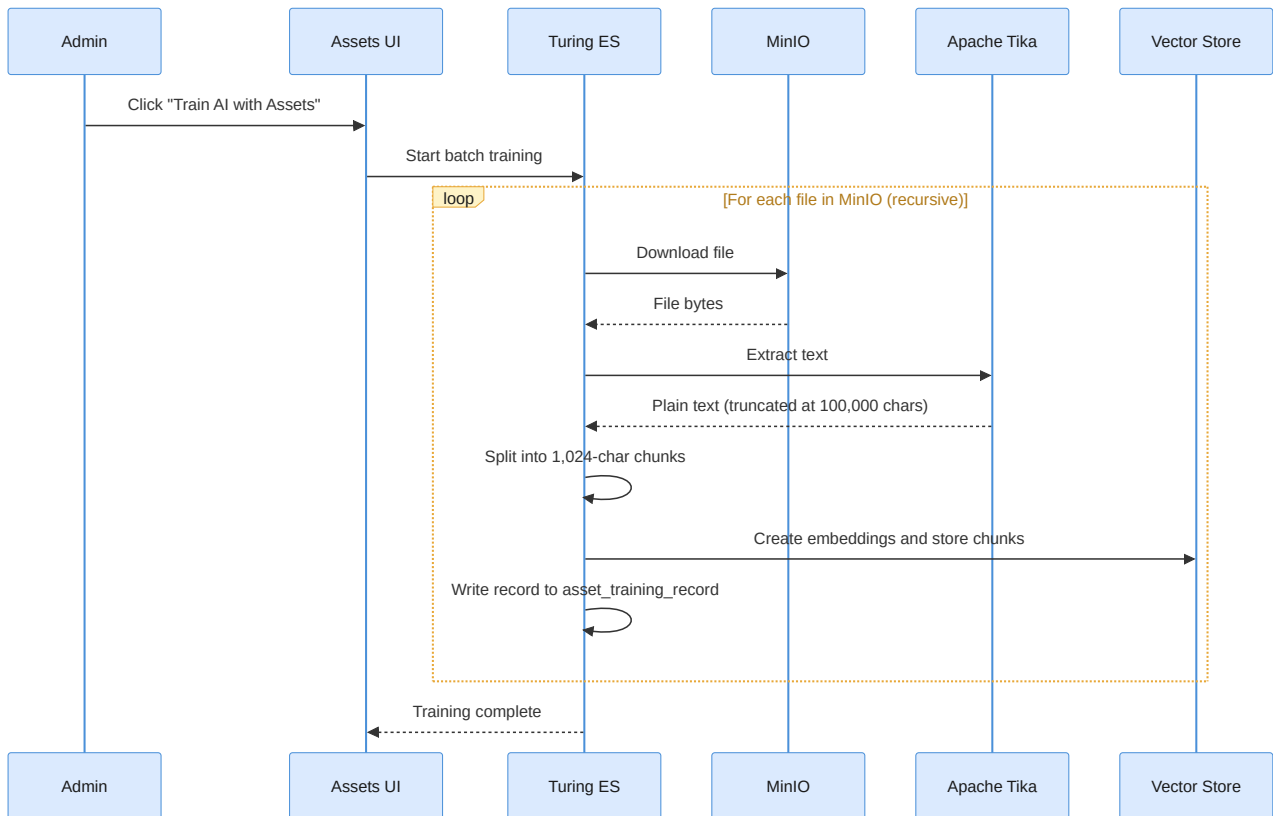
- ✅ **Checkmark** — file has been indexed; hover to see the training timestamp
- (empty) — file has not yet been indexed

Automatic Training on Upload

When a file is uploaded, Turing ES dispatches an **asynchronous event** that indexes the file individually without any user action required. Similarly, when a file is deleted, its embeddings are automatically purged from the vector store.

Batch Training

To index all existing files at once — useful after enabling RAG on an existing installation, or after changing the embedding model — use the **"Train AI with Assets"** button.



Batch training steps for each file:

Download file bytes from MinIO

Extract plain text via **Apache Tika** — supports PDF, DOCX, XLSX, PPTX, HTML, TXT, and images (with OCR)

Truncate text to **100,000 characters**

Split into **chunks of 1,024 characters**

Generate embeddings and store in the configured vector store

Write a record to `asset_training_record` with timestamp

Progress monitoring — while the batch is running, the UI polls every **3 seconds** and displays:

X / Y files processed, Z errors

Training states: `IDLE` → `RUNNING` → `COMPLETED` / `FAILED`

⚠ RE-TRAINING AFTER EMBEDDING MODEL CHANGE

If you change the Default Embedding Model in **Administration** → **Global Settings** → **RAG Settings**, all existing embeddings become invalid. Run "Train AI with Assets" again to re-index all files with the new model.

Embedding Metadata

Each chunk stored in the vector store carries the following metadata:

FIELD	VALUE
<code>source</code>	<code>"minio-asset"</code>
<code>objectName</code>	Full object path in MinIO
<code>objectPath</code>	Folder path within the bucket
<code>fileName</code>	Original filename
<code>contentType</code>	MIME type of the source file
<code>size</code>	File size in bytes

This metadata is used by AI Agents when returning search results, so the LLM can cite the source file and provide context about where the information came from.

Text Extraction Limits

There is **no hard file size limit** for uploads — MinIO accepts files of any size. However, during AI Training the extracted text is **truncated to 100,000 characters** before chunking. For very large documents this means only the first portion of the content is indexed. The truncation limit is a fixed internal constant.

How AI Agents Use the Knowledge Base

Once files are indexed, AI Agents can query the knowledge base via four built-in tools:

TOOL	DESCRIPTION
<code>search_knowledge_base</code>	Semantic similarity search across all indexed chunks
<code>knowledge_base_stats</code>	Returns total files, chunks, and storage size
<code>list_knowledge_base_files</code>	Lists all indexed files, with optional keyword filter
<code>get_file_from_knowledge_base</code>	Retrieves the full indexed content of a specific file

For details on configuring AI Agents and tools, see [AI Agents](#) and [Tool Calling](#).

Storage Configuration

A storage backend must be configured before Assets becomes available. Turing supports two storage backends: **MinIO** (recommended for production) and **Filesystem** (for simple deployments). Set `turing.storage.type` to enable one.

Option 1: MinIO (Recommended)

```
turing:
  storage:
    type: minio
    minio:
      endpoint: http://localhost:9000
      access-key: minioadmin
      secret-key: minioadmin
      bucket: turing-assets
```

The bucket (`turing-assets` by default) is **created automatically on startup** if it does not exist.

With Docker Compose, add the MinIO service alongside Turing ES:

```
minio:
  image: minio/minio
  ports:
    - "9000:9000"
    - "9001:9001"
  environment:
    MINIO_ROOT_USER: minioadmin
    MINIO_ROOT_PASSWORD: minioadmin
  command: server /data --console-address ":9001"
  volumes:
    - minio_data:/data
```

**TIP**

The MinIO web console is available at <http://localhost:9001> when running locally via Docker Compose. Use it to inspect buckets and verify that files are being stored correctly.

Option 2: Filesystem

```
turing:
  storage:
    type: filesystem
    filesystem:
      path: ./store/assets
```

Files are stored in the configured directory, which is created automatically if it does not exist. The filesystem backend includes **path traversal protection** — all paths are resolved against the base directory.

 NOTE

The filesystem backend is suitable for single-server deployments and development. For production environments with multiple instances or high availability requirements, use MinIO.

For the full property reference, see [Configuration Reference — Storage](#).

REST API Reference

All endpoints are under `/api/asset` and require authentication.

METHOD	ENDPOINT	DESCRIPTION
GET	<code>/api/asset?prefix=</code>	List files and folders in a prefix (folder)
POST	<code>/api/asset</code>	Upload one or more files (multipart/form-data)
GET	<code>/api/asset/download?objectName=</code>	Download a file (Content-Disposition: attachment)
GET	<code>/api/asset/preview?objectName=</code>	Preview a file inline (Content-Disposition: inline)
GET	<code>/api/asset/metadata?objectName=</code>	Retrieve file metadata (size, type, date)
POST	<code>/api/asset/folder?path=</code>	Create a new folder
DELETE	<code>/api/asset?objectName=</code>	Delete a file or folder
POST	<code>/api/asset/train</code>	Start batch AI Training
GET	<code>/api/asset/train/status</code>	Poll current training status
GET	<code>/api/asset/train/records?objectNames=</code>	Get training timestamps for specific files

Related Pages

PAGE	DESCRIPTION
Embedding Stores & Models	Configure vector database backends and embedding models used by AI Training
Tool Calling	All 27 native tools, including the four Knowledge Base tools
AI Agents	Compose agents that query the Knowledge Base via tool calling
Generative AI & LLM Configuration	Platform-wide GenAI settings and RAG architecture overview

Tool Calling

A **Tool Calling** is a function that the LLM can invoke autonomously during a conversation to retrieve information or perform an action. Instead of relying purely on training data, the LLM calls tools to fetch live data, search indexed content, execute code, and more — then incorporates the results into its reasoning.

Turing ES includes native tools organized into categories, plus support for external tools via **MCP Servers**. Tools are enabled per **AI Agent** — each agent selects only the tools it needs.

DSL Search — 6 tools

These tools provide **Elasticsearch-compatible Query DSL** access to any Semantic Navigation Site. Inspired by the **Elasticsearch MCP Server**, they allow the LLM to build and execute full DSL queries, aggregations, and analytics — automatically translated to the site's configured search engine (Elasticsearch, Solr, or Lucene).

TOOL	DESCRIPTION
<code>dsl_list_indices</code>	Lists all available SN Sites with engine type, locales, and status
<code>dsl_get_mappings</code>	Returns field mappings (types, facets, multi-valued) in Elasticsearch format
<code>dsl_search</code>	Executes a full Elasticsearch Query DSL search (40 query types, 35 aggregations)
<code>dsl_get_document</code>	Retrieves a document by ID with all stored fields
<code>dsl_suggest</code>	Autocomplete suggestions and spell-check corrections
<code>dsl_get_shards</code>	Shard/core information: document counts, engine type, endpoints, locales

How the LLM uses DSL tools

The LLM follows a pattern similar to how a developer uses the Elasticsearch API:

Discover → `dsl_list_indices("*")` to find available sites

Understand schema → `dsl_get_mappings("mySite")` to see fields and types

Search → `dsl_search("mySite", "en", queryBody)` with full DSL

Deep dive → `dsl_get_document("mySite", "en", "doc-123")` for full content

Correct typos → `dsl_suggest("mySite", "en", "enterprse")` for corrections

The `dsl_search` tool

This is the most powerful tool. The `queryBody` parameter accepts the complete Elasticsearch `_search` request body as a JSON string. The LLM constructs the query based on the user's intent:

Simple search:

```
{"query":{"match":{"title":"machine learning"}}, "size":5}
```

Filtered search with aggregations:

```
{
  "query":{"bool":{"must":[{"match":{"title":"security"}}], "filter":[{"term":
{"type":"article"}}]}},
  "size":10,
  "aggs":{"by_author":{"terms":{"field":"author", "size":5}}},
  "highlight":{"fields":{"title":{}, "body":{}}}
}
```

Date range with sorting:

```
{
  "query":{"range":{"date":{"gte":"2025-01-01"}}},
  "sort":[{"date":"desc"}],
  "size":20
}
```

Similar documents (More Like This):

```
{"query":{"more_like_this":{"fields":["title","body"],"like":"enterprise search platform","min_term_freq":1}}}
```

Facet value discovery (terms aggregation):

```
{"query":{"match_all":{"}},"size":0,"aggs":{"categories":{"terms":{"field":"category","size":50}}}}
```

For the complete DSL reference, see [DSL Query API](#) and [DSL Compatibility Matrix](#).

Prompt examples

"Search for documents about authentication in the Sample site" → `dsl_search` with `match` query

"What fields can I filter by on the Sample site?" → `dsl_get_mappings`

"Show me the full content of document ID abc-123" → `dsl_get_document`

"How many documents do we have per content type?" → `dsl_search` with `terms` aggregation

"Find documents similar to the article about Solr configuration" → `dsl_search` with `more_like_this`

"What articles were updated in the last 7 days about security?" → `dsl_search` with `range` + `sort`

"Compare the features of Product A and Product B" → `dsl_get_document` called for each product

"What categories are available?" → `dsl_search` with `size:0` and `terms` aggregation

"How many documents per locale does the site have?" → `dsl_get_shards`

RAG / Knowledge Base — 4 tools

These tools enable the LLM to query the **Knowledge Base** — files indexed from MinIO via the [Assets](#) section — using semantic similarity search.

TOOL	DESCRIPTION
<code>search_knowledge_base</code>	Searches for relevant documents by semantic similarity (top 10, threshold 0.7)
<code>knowledge_base_stats</code>	Returns statistics: total files, chunks, and storage size
<code>list_knowledge_base_files</code>	Lists all indexed files, with optional keyword filter
<code>get_file_from_knowledge_base</code>	Retrieves the full indexed content of a specific file

Prompt examples:

"What does our internal documentation say about deployment procedures?" → triggers

`search_knowledge_base`

"How many files are indexed in the knowledge base?" → triggers `knowledge_base_stats`

"Show me the full content of the file onboarding-guide.pdf" → triggers

`list_knowledge_base_files` then `get_file_from_knowledge_base`

Web Crawler — 2 tools

TOOL	DESCRIPTION
<code>fetch_webpage</code>	Fetches a web page by URL and returns its content as plain text
<code>extract_links</code>	Extracts all links from a web page, with optional keyword filter

Prompt examples:

"What does the Apache Solr documentation say about faceted search?" → triggers

`fetch_webpage`

"List all links on our company blog homepage" → triggers `extract_links`

Finance — 2 tools

TOOL	DESCRIPTION
<code>get_stock_quote</code>	Current price and market data for a stock ticker symbol
<code>search_ticker</code>	Looks up a ticker symbol by company name or keyword

Prompt examples:

"What is the current stock price of Apple?" → triggers `search_ticker` then `get_stock_quote`

"Show me the market data for MSFT" → triggers `get_stock_quote`

Weather — 1 tool

TOOL	DESCRIPTION
<code>get_weather</code>	Current weather and forecast for a city (1-7 day range)

Prompt examples:

"What's the weather forecast for São Paulo this week?" → triggers `get_weather`

Image Search — 1 tool

TOOL	DESCRIPTION
<code>search_images</code>	Searches the web for images and returns URLs and descriptions

Prompt examples:

"Find images of enterprise search architecture diagrams" → triggers `search_images`

DateTime — 1 tool

TOOL	DESCRIPTION
<code>get_current_time</code>	Returns the current date and time for a given IANA timezone

Prompt examples:

"What time is it right now in Tokyo?" → triggers `get_current_time`

Code Interpreter — 1 tool

TOOL	DESCRIPTION
<code>execute_python</code>	Executes Python code in a sandboxed environment and returns stdout/stderr

The Code Interpreter runs Python in an isolated sandbox directory with a **30-second execution timeout**. It supports:

Standard output capture

Matplotlib chart generation (with `Agg` backend — no display required)

Automatic `print()` wrapping for bare expressions

The Python executable path is configured in **Administration → Settings → Python Path**.

Prompt examples:

"Calculate the compound interest on \$10,000 at 5% for 10 years" → triggers `execute_python`

"Generate a bar chart showing monthly sales: Jan=100, Feb=150, Mar=120" → triggers `execute_python` with matplotlib

External Tools via MCP Servers

Beyond the native tools, AI Agents can access tools from any external server implementing the **Model Context Protocol (MCP)**. This covers company-internal systems, proprietary data APIs, and the growing ecosystem of public MCP servers.

See [MCP Servers](#) for configuration details.

Related Pages

PAGE	DESCRIPTION
AI Agents	How to compose agents with the tools they need
MCP Servers	Extend agents with external tools via MCP
DSL Query API	Full reference for the Elasticsearch-compatible DSL
DSL Compatibility Matrix	Engine compatibility for all DSL features
Assets	Knowledge Base files queried by RAG tools
Semantic Navigation	The search experience powering DSL tools

Custom Tools

*The 27 native tools cover most of what an agent needs to do. The 28th — and the 29th, and the 50th — is whatever your business actually does. **Custom Tools** is how you teach an agent to do it.*

A **Custom Tool** is a function you write that any **AI Agent** or **Chat Flow** can call as if it were native. You declare what it does, what arguments it accepts, and how the LLM should think about it. The body is a **Groovy script** — a friendly Java dialect that runs in a sandboxed JVM context.

Three things make Custom Tools different from "just write a microservice":

No deployment, no rebuild. You write a Custom Tool in the admin console, click Save, and it's available to every agent immediately.

First-class to the LLM. The tool's name, description, and parameter schema are exposed to the LLM exactly like a native tool. The model picks it on its own when relevant.

Vibe Coding. Tell the AI what you want — "*a tool that takes a CEP code and returns the address from BrasilAPI*" — and it writes the description, the LLM-facing prompt, the parameter schema, **and** the Groovy script. You review, save, ship.

Configure tools in **Administration → Custom Tools**.

When You Need a Custom Tool

Some questions don't have a native tool, an MCP server, or a stock answer:

USE CASE	WHY CUSTOM
Look up a Brazilian postal code (CEP)	Public API; one-line script; no need to spin up an MCP for it
Fetch a customer's account balance from your internal API	Specific to your stack; rotating per agent doesn't scale
Run a scoring formula your sales team uses	Proprietary logic; encoding it in a prompt is fragile
Translate a value through a lookup table you maintain	Hot data; pulling it in via Groovy is faster than an MCP
Compute working days between two dates respecting your country's holidays	Domain-specific; trivial in Groovy; would be expensive to externalize

A Custom Tool sits between "*too small for an MCP*" and "*too big for a prompt instruction*". That's a productive middle ground for most operational queries.

Anatomy of a Custom Tool

Every tool is a small bundle of seven decisions. Get the first three right and the LLM uses your tool well; get the script right and the tool *works*.

FIELD	REQUIRED	WHAT IT'S FOR
Title	Yes	The display name in the admin console (e.g., "CEP Lookup")
Description		A short admin-facing description. Helps your team know <i>when to attach this tool to an agent</i>
Icon		A Lucide icon for visual identity in the admin lists
LLM Description	Yes	The description the LLM sees. This is the most important field. It's how the model decides whether to call your tool. <i>"Returns the full street address (street, neighborhood, city, state) for a Brazilian postal code (CEP)."</i>
Parameters JSON		JSON-serialized list of <code>{name, type}</code> pairs declaring the arguments the LLM is allowed to pass. Types: <code>string</code> , <code>integer</code> , <code>number</code> , <code>boolean</code> .
Return Type	Yes	The JSON-Schema primitive of the value the script returns: <code>string</code> , <code>integer</code> , <code>number</code> , <code>boolean</code> , <code>object</code>
Groovy Script	Yes	The body. Receives parameters as named bindings, returns the value the LLM will see.
Enabled		Toggle to activate/deactivate without deleting

Two quiet but important fields back the Vibe Coding experience:

`descriptionMetaPrompt`, `llmDescriptionMetaPrompt`, `groovyMetaPrompt` — when you ask the AI to author a piece, the meta-prompt that produced it is stored alongside the result. You can revise *the prompt* later, not just the output.

The Groovy Sandbox

Custom Tools run in a Groovy script context — full Java/Groovy semantics, with one strong constraint: **the script returns a single value**, declared in `Return Type`.

A trivial example: a tool that adds two numbers.

Parameters JSON:

```
[
  { "name": "a", "type": "number" },
  { "name": "b", "type": "number" }
]
```

Return Type: `number`

Groovy Script:

```
return a + b
```

The names declared in Parameters JSON become bindings in the script. The LLM passes `a` and `b`; Groovy receives them; the result is the value of the last expression (or an explicit `return`).

A more realistic example: a CEP lookup against the public BrasilAPI.

Parameters JSON:

```
[
  { "name": "cep", "type": "string" }
]
```

Return Type: `object`

Groovy Script:

```
import groovy.json.JsonSlurper

def url = "https://brasilapi.com.br/api/cep/v2/${cep.replaceAll(/\\D/, '')}"
def conn = new URL(url).openConnection()
conn.requestMethod = 'GET'
conn.connect()

if (conn.responseCode != 200) {
    return [error: "CEP not found", code: conn.responseCode]
}

return new JsonSlurper().parseText(conn.inputStream.text)
```

When an LLM calls this tool, it gets back something like:

```
{
  "cep": "01310-100",
  "state": "SP",
  "city": "São Paulo",
  "neighborhood": "Bela Vista",
  "street": "Avenida Paulista",
  "service": "open-cep"
}
```

...and uses that to answer the user. The whole loop — user asks for an address from a postal code, model picks the tool, Groovy runs, model formats the answer — happens inside a single chat turn, no extra services.

SECURITY CAVEATS

The sandbox is a Groovy context, not a hardened VM. **Don't paste anything you wouldn't paste into a backend script you control.** Groovy can hit the network, read environment variables, and call any Java class on the classpath. In production, restrict who can author Custom Tools (the existing `CUSTOM_TOOL_*` permissions).

Vibe Coding: Let the AI Author the Tool

Writing a tool from scratch — even a trivial one — usually means writing four things: the LLM description, the parameter list, the script, and a quick test. The Vibe Coding chat does all four for you, then hands you a draft to review.

Click **Vibe Coding** on the Custom Tool form and describe what you want:

You: "A tool that takes a Brazilian CEP and returns the full address — street, neighborhood, city, state. Use BrasilAPI."

Assistant: "Done. Here's the draft: — LLM Description: 'Returns the full Brazilian address (street, neighborhood, city, state, full postal code) for a given CEP. Use this when the user asks about an address by postal code.' — Parameters: `cep` (string) — Return Type: object — Script: a Groovy block that calls `https://brasilapi.com.br/api/cep/v2/{cep}`, parses the JSON, returns the result; on 404 it returns `{error: 'CEP not found'}`. Want me to also strip non-digits from the input?"

You read the draft, accept it, save. Two minutes from idea to working tool.

The Vibe Coding chat is **incremental** — you can revise individual parts:

You: "Make the description more specific to addresses, and add a note that the script handles malformed CEP."

Assistant: "Updated. The LLM Description now says: 'Returns the full Brazilian address... Robust against formatted CEP inputs (handles "01310-100" and "01310100" identically). Returns an error object when the CEP isn't found.'"

The meta-prompts (`descriptionMetaPrompt`, `llmDescriptionMetaPrompt`, `groovyMetaPrompt`) are stored alongside each piece. The next time you Vibe-Code-revise, the assistant has full context of *what you asked for* and *what it generated*, so revisions stay coherent.

💡 THE LLM DESCRIPTION DESERVES THE MOST CARE

The LLM Description is the single field that determines whether agents actually use your tool. A vague description ("*BrasilAPI lookup*") means the model won't pick it when a user asks about an address. A specific description ("*Returns the full Brazilian address (street, neighborhood, city, state) for a postal code. Use when the user asks about an address by CEP, or wants to verify a CEP is valid.*") means the model picks it confidently. **Vibe Coding usually nails this on the first try because the assistant knows the LLM perspective from inside.** Read it back to yourself before saving.

Integrating Custom Tools

A Custom Tool is just an entry in the catalog until you wire it somewhere. Three places it can run:

In an AI Agent

Open an **AI Agent**, navigate to the Tools tab. Custom Tools appear in their own group at the bottom (after the eight native categories). Check the ones this agent should have access to.

When the agent runs, those Custom Tools are merged with the native ones into a single tool catalog the LLM sees. The LLM picks among them by name + description, exactly like native tools.

The same agent can mix:

Native tools (e.g., `search_knowledge_base`, `get_current_time`),
MCP server tools (e.g., a CRM lookup),
Custom Tools (e.g., your CEP lookup, your scoring formula).

The LLM doesn't distinguish — they all look like callable functions in the same catalog.

In a Chat Flow

A **Chat Flow** Tool node accepts `native`, `mcp`, or `custom` as its tool source. With `custom`, you point the node at one of your Custom Tools and pass arguments either statically (hard-coded values) or from collected flow variables.

Use this pattern when:

A flow needs to look up data deterministically *between questions* (not when the LLM thinks it's a good time).

You want auditability — the submission record will show that the tool ran with these inputs and got that result.

In the Direct LLM Chat

The direct **Chat** mode (no agent) doesn't expose Custom Tools by default — that mode is for *general-purpose assistance with optional toolboxes*. If you need a Custom Tool in direct chat, attach it to a thin "default agent" instead. This keeps the catalog manageable.

Three Real Custom Tools

1. Lead-Scoring Tool

Use case: A sales agent collects company size + role + timeline and needs to score the lead.

LLM Description: "Computes a lead score from 0 to 100 based on company size, role title, and decision timeline. Returns the score and a tier label (HOT / WARM / COLD)."

Parameters:

```
[
  { "name": "company_size", "type": "integer" },
  { "name": "role", "type": "string" },
  { "name": "timeline", "type": "string" }
]
```

Return Type: `object`

Groovy:

```
def sizeScore = company_size > 1000 ? 40 : company_size > 100 ? 25 : 10
def roleScore = role.toLowerCase() =~ /.*(ceo|cto|vp|director).*/ ? 30 : 15
def timelineScore = timeline == 'short_term' ? 30 : timeline == 'long_term' ? 15 : 5

def total = sizeScore + roleScore + timelineScore
def tier = total >= 70 ? 'HOT' : total >= 40 ? 'WARM' : 'COLD'

return [score: total, tier: tier]
```

Wire this into the demo-qualification **Chat Flow** right before End. The submission record now has the score; downstream automation routes accordingly.

2. Holiday-Aware Working Days Calculator

Use case: A support agent estimates SLA timing.

LLM Description: "Calculates the number of working days between two dates, excluding Brazilian national holidays. Useful for SLA estimates."

Parameters:

```
[
  { "name": "start_date", "type": "string" },
  { "name": "end_date", "type": "string" }
]
```

Return Type: `integer`

Groovy:

```
import java.time.LocalDate
import java.time.temporal.ChronoUnit
import java.time.DayOfWeek

def start = LocalDate.parse(start_date)
def end   = LocalDate.parse(end_date)

def holidays = [
    '2026-01-01', '2026-04-21', '2026-05-01',
    '2026-09-07', '2026-10-12', '2026-11-02',
    '2026-11-15', '2026-12-25'
].collect { LocalDate.parse(it) } as Set

def days = 0
def cursor = start
while (cursor.isBefore(end) || cursor.isEqual(end)) {
    if (cursor.dayOfWeek != DayOfWeek.SATURDAY
        && cursor.dayOfWeek != DayOfWeek.SUNDAY
        && !holidays.contains(cursor)) {
        days++
    }
    cursor = cursor.plusDays(1)
}
return days
```

The agent doesn't have to know about your country's calendar; the tool handles it.

3. Internal API Wrapper

Use case: Your support agent needs to look up a customer's tier from your internal API.

LLM Description: *"Looks up a customer's subscription tier (FREE / STARTER / PRO / ENTERPRISE) from their email. Use this before quoting features or pricing limits."*

Parameters:

```
[
  { "name": "email", "type": "string" }
]
```

Return Type: `object`

Groovy:

```
import groovy.json.JsonSlurper

def url = "https://internal-api.example.com/customers/by-email?
email=${URLEncoder.encode(email, 'UTF-8')}"
def conn = new URL(url).openConnection()
conn.setRequestProperty('Authorization', "Bearer
${System.getenv('INTERNAL_API_TOKEN')}")
conn.requestMethod = 'GET'

if (conn.responseCode == 404) return [tier: 'NONE', found: false]
if (conn.responseCode != 200) return [error: "API error
${conn.responseCode}"]

def data = new JsonSlurper().parseText(conn.inputStream.text)
return [tier: data.tier, since: data.created_at, found: true]
```

The agent is now grounded — it never quotes ENTERPRISE features to a STARTER customer.

SECRETS IN GROOVY

Use `System.getenv('YOUR_TOKEN')` for secrets, not literal strings. The Groovy text is logged when the tool runs (for debugging), and the script may be exported with the agent definition. Pull tokens from the JVM environment so they don't show up in logs or exports.

How the LLM Sees Your Tool

When an agent runs, the system builds a tool catalog the LLM uses to pick. Each Custom Tool produces a JSON-Schema function definition like this:

```
{
  "type": "function",
  "function": {
    "name": "cep_lookup",
    "description": "Returns the full Brazilian address (street, neighborhood, city, state) for a postal code. Use when the user asks about an address by CEP, or wants to verify a CEP is valid.",
    "parameters": {
      "type": "object",
      "properties": {
        "cep": { "type": "string" }
      },
      "required": ["cep"]
    }
  }
}
```

The model decides on its own when to call it. When it does, Turing ES:

- Validates the arguments against the parameter declaration.

- Executes the Groovy script with the bindings.

- Captures the return value.

- Feeds it back to the LLM in the next reasoning step.

If the script throws, the error message is fed back to the LLM as the tool result — the model usually recovers gracefully (*"I couldn't look up that CEP — could you double-check the digits?"*).

REST API

METHOD	ENDPOINT	DESCRIPTION
GET	/api/custom-tool	List all custom tools
GET	/api/custom-tool/{id}	Get a single tool
POST	/api/custom-tool	Create a tool
PUT	/api/custom-tool/{id}	Update a tool
DELETE	/api/custom-tool/{id}	Delete a tool
POST	/api/custom-tool/validate	Validate a script — returns syntax errors with line/column
POST	/api/custom-tool/ai-chat	Vibe Coding endpoint — describe a tool, get a draft back

The validate endpoint runs a fast Groovy parse without executing the script — a smoke test before save.

Permissions

PERMISSION	ALLOWS
CUSTOM_TOOL_VIEW	View custom tools
CUSTOM_TOOL_CREATE	Author new tools
CUSTOM_TOOL_EDIT	Edit existing tools
CUSTOM_TOOL_DELETE	Delete tools

All require `ROLE_ADMIN`. In production, restrict authoring to a small group — Groovy is full Java, and bad scripts can do real damage.

Testing & Iteration

The fastest dev loop:

Open the Custom Tool form, write or Vibe-Code a draft.

Click **Validate** to catch syntax errors instantly.

Save.

Attach the tool to a test **AI Agent** (or use a "sandbox" agent you keep around).

Open **Chat**, pick that agent, ask a question that should trigger the tool.

Watch the response. If the LLM didn't pick the tool, your **LLM Description** is too vague. If it picked it but with wrong arguments, your **Parameters JSON** doesn't match the script's bindings. If it picked it but the script errored, the script needs work.

Edit the tool. The agent picks up the change immediately — no restart.

For deeper diagnostics, **Observability** shows tool latency through the `turing.llm.calls` timer, and **Chat Analytics** shows whether the agent's goal-rate moves when you change the tool.

Common Patterns

One tool, multiple agents

A `cep_lookup` tool is useful to a sales agent (verifying a prospect's address), a support agent (validating a shipping address), and an onboarding coach (verifying signup data). Build it once; check it on three agents. No duplication.

Tools that wrap MCP servers

If you already have an MCP server, you usually don't need a Custom Tool that calls the same thing. Custom Tools shine when the work is *small enough that running an MCP is overkill*. The cutoff in practice: if the logic is more than 50 lines of Groovy or needs durable state, build an MCP. Otherwise, Custom Tool wins on operational simplicity.

Self-documenting business logic

A Groovy script reads top-to-bottom. When your finance team disputes how the lead score is computed, they can read the script. A custom tool can become *the* source of truth for a piece of business logic — readable, versionable, and callable.

Composability with Chat Flow

A complex business process is often: collect inputs → compute → call external system → format response. **Chat Flow** does the collection and the orchestration; **Custom Tools** do the compute and the external calls; **Native tools or MCP** do the formatting (e.g., open a ticket). The three together replace what would otherwise be a custom microservice.

Diagnostics

SYMPTOM	LIKELY CAUSE	FIX
LLM doesn't call the tool when it obviously should	LLM Description is vague; or the agent has too many tools and yours got drowned out	Sharpen the LLM Description with concrete trigger phrases ("Use when..."); or reduce the agent's tool count
LLM calls the tool with wrong argument names	Parameters JSON doesn't match script bindings	Names in Parameters JSON must exactly equal the bindings used in Groovy
Script throws at runtime	Network issue, parse error, NPE in the script	Use the Validate endpoint; add <code>try/catch</code> ; return an <code>[error: ...]</code> object instead of throwing
Tool works locally but fails in chat	Auth header missing in the script (e.g., environment variable not set on this server)	Verify env vars; or move the auth into the global config and reference it
Latency spike	The script makes a slow API call	Add a timeout: <code>conn.connectTimeout = 3000; conn.readTimeout = 3000</code>

Related Pages

PAGE	DESCRIPTION
Tool Calling	The 27 native tools — when to use a native tool vs. a custom one
MCP Servers	When to build an MCP instead of a Custom Tool
AI Agents	Agents are where Custom Tools get attached and used
Chat Flow	Use Custom Tools as deterministic Tool nodes in a flow
Observability	Watch tool latency and error rates in real time

MCP Servers

The **Model Context Protocol (MCP)** is an open standard that lets AI models call tools hosted on external servers — turning any API, database, or internal system into a capability that an AI Agent can use autonomously. If you are an administrator looking to extend your agents beyond the 27 built-in tools, MCP Servers let you connect to virtually any external service without writing custom code inside Turing ES.

An **MCP Server** extends the capabilities of an AI Agent by connecting it to any external server that implements the **Model Context Protocol (MCP)**. This allows Turing ES agents to use tools defined completely outside the platform — a company-internal knowledge system, a proprietary data API, a database query interface, or any of the growing ecosystem of public MCP servers.

MCP Servers are configured in **Administration → MCP Servers** and then selected per **AI Agent**.

Configuration Form

FIELD	DESCRIPTION
Name	Display name for the MCP server — shown when selecting it in an AI Agent
Transport type	<code>HTTP</code> or <code>stdio</code> — how Turing ES communicates with the server
Command	For <code>stdio</code> : the executable to launch the MCP server process (e.g., <code>npx</code>)
Arguments	For <code>stdio</code> : command-line arguments passed to the process (e.g., <code>@modelcontextprotocol/server-filesystem /data</code>)
URL	For <code>HTTP</code> : the MCP server endpoint URL
Execution mode	<code>Synchronous</code> or <code>Asynchronous</code>

Transport Types

stdio

Launches the MCP server as a **local subprocess**. The MCP client communicates with it via standard input/output streams. This is the standard mode for locally installed MCP servers.

```
# Example: filesystem MCP server
npx @modelcontextprotocol/server-filesystem /path/to/allowed/directory
```

Use stdio when the MCP server is installed locally (via npm, pip, or a system binary) and does not expose an HTTP endpoint.

HTTP

Connects to a **remote MCP server** over HTTP. This mode supports both public MCP services and internally hosted servers accessible over the network.

```
https://mcp.example.com/api/tools
```

Use HTTP when the MCP server is a hosted service or when it runs on a different machine or container.

Execution Modes

Synchronous

Waits for the MCP tool result before continuing the agent's reasoning chain. The LLM receives the tool result and can incorporate it into its next reasoning step before deciding whether to call another tool or generate a final response.

Use synchronous mode when the tool result is needed as context for subsequent steps.

Asynchronous

Dispatches the tool call without blocking. Useful for long-running operations or when the result does not need to be incorporated into the immediate reasoning chain.

MCP Server Examples

Internal ticketing system (stdio):

```
Name:      JIRA Internal
Transport: stdio
Command:   node
Arguments: /opt/mcp-jira/index.js --token <TOKEN>
Mode:      Synchronous
```

Company knowledge API (HTTP):

```
Name:      Internal Knowledge Base API
Transport: HTTP
URL:       https://internal-api.example.com/mcp
Mode:      Synchronous
```

Public MCP service (HTTP):

```
Name:      Brave Search
Transport: HTTP
URL:       https://api.search.brave.com/mcp
Mode:      Asynchronous
```

Assigning MCP Servers to Agents

Once configured, MCP servers appear in the **MCP Servers** multi-select field of the [AI Agent](#) configuration form. Each agent independently selects which MCP servers it can use. The tools exposed by each selected MCP server become available to the LLM during that agent's conversations.

Related Pages

PAGE	DESCRIPTION
AI Agents	Compose agents with native tools + MCP servers
Tool Calling	The 27 native tools built into Turing ES
Chat	Front-end chat interface where agents are used

AI Agents

Hire a salesperson. Hire a solutions engineer. Hire an onboarding coach. Hire them all in an afternoon, and have them on duty 24/7 by sundown.

An **AI Agent** in Turing ES is the specialist you don't have to recruit, train, or replace. It's a small, named bundle of:

- a **brain** — an **LLM Instance** you've already configured,
- a set of **hands** — **native tools** and **MCP servers** that let the agent actually *do* things (search your content, fetch live data, run code),
- a **voice** — a **Persona** that aligns the answers with your brand, your funnel stage, and your compliance posture.

Each agent appears as its own tab in the **Chat** interface. Visitors pick the specialist that matches their need; agents stay in their lane.

Configure agents in **Administration** → **AI Agents**.

What "Agent" Actually Means

The word "agent" is overloaded. In Turing ES, an Agent has two non-negotiable properties:

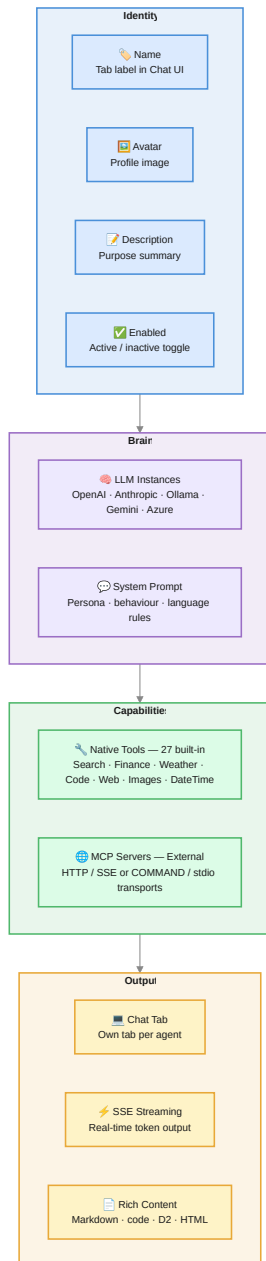
It's a composition, not a model. You're not picking a model from a marketplace and renaming it. You're combining decisions — *which* LLM, *which* tools, *which* voice — into a single, deployable unit. Change any of the three, and you've changed the agent.

It can act, not just answer. The defining feature of an agent over a plain chatbot is its ability to call tools — search a knowledge base, look up a stock quote, run a Python script, query an MCP server. The model doesn't just generate words; it *requests an action*, Turing ES executes it, and the model reasons over the result.

Together, these mean an agent is a **business process expressed as a conversation**. A user asks; the agent searches your documentation, fetches their account data, summarizes the answer, and books a follow-up — in one exchange.

Composition: The Four Layers

An AI Agent is built from four layers. Get all four right and the agent feels like a senior team member; get any one wrong and it feels like a generic chatbot.



1. Identity

FIELD	DESCRIPTION
Name	Display name shown as the tab label in the Chat interface. "Sales Concierge" beats "Agent #3".
Avatar	Profile image. Even a small one moves trust meters — visitors react to faces, even illustrated ones.
Description	One-line explanation of the agent's purpose. Shown in admin lists; hints visitors what to expect.
Enabled	Toggle to activate or hide the agent. Disabled agents do not appear in the Chat interface.

2. Brain

The **LLM Instance** is the agent's reasoning engine. You can attach **one or more** instances; at chat time the user (or front-end) picks which one to run.

Why multiple? Because the cost/quality tradeoff varies by question:

A trivial routing question doesn't need a frontier model. Attach a fast, cheap instance (e.g., Gemini Flash, GPT-4o-mini) and let users default to it.

A complex multi-step deal qualification benefits from a top-tier reasoner. Attach Claude Sonnet alongside, as an opt-in upgrade.

See [LLM Instances](#) for vendor support, default models, and provider-specific options.

3. Capabilities

This is the largest configuration surface. Two sets of capabilities can be attached:

Native tools — 27 first-party tools, grouped into 8 categories:

CATEGORY	TOOLS	WHEN TO ATTACH
Semantic Navigation	<code>list_sites</code> , <code>search_site</code> , <code>get_document_details</code> , <code>find_similar_documents</code> , <code>search_by_date_range</code> , <code>get_site_fields</code> , <code>get_valid_filter_values</code> , ... (15 total)	The agent should answer using your indexed enterprise content (CMS pages, products, internal documentation)
RAG / Knowledge Base	<code>search_knowledge_base</code> , <code>list_knowledge_base_files</code> , <code>get_file_content</code> , <code>knowledge_base_stats</code>	The agent should answer using files in Assets (PDFs, Word docs, manuals)
Web Crawler	<code>fetch_webpage</code> , <code>extract_links</code>	The agent should be able to read a public URL on demand (e.g., reference a customer's website)
Finance	<code>get_stock_quote</code> , <code>search_ticker</code>	Investor-facing or financial-research agents
Weather	<code>get_weather</code>	Travel, logistics, field-service agents
Image Search	<code>search_images</code>	Marketing, content-creation agents
Date / Time	<code>get_current_time</code>	Any agent that books, schedules, or references "now"
Code Interpreter	<code>execute_python</code>	Data analysis, chart generation, technical agents

MCP Servers — external tool servers the agent can call:

BADGE	TRANSPORT	TYPICAL USE
HTTP (blue)	SSE over HTTP	A REST API your team owns; a brand-context server; a CRM lookup
COMMAND (amber)	stdio	A local script or process — useful for filesystem or shell access on the host

See [Tool Calling](#) for the full native tool reference and [MCP Servers](#) for connection setup.

⚠ LEAN TOOL LISTS WIN

A common mistake: attaching every tool because "*it might be useful*". Every additional tool consumes prompt tokens to describe to the model, and forces the model to choose among more options. **Fewer, sharper tools produce more precise tool calls and shorter latencies.** Start with 3–5 tools per agent and add only what an actual user request demanded.

4. Voice

The **Persona** field attaches a [Persona](#) to the agent. The persona overlays:

- a system instruction (brand voice directive),
- tone, verbosity, language style,
- mandatory + forbidden vocabulary (enforced both pre- and post-LLM),
- optional few-shot examples retrieved from a vector store,
- optional live brand context from an MCP.

A persona is **optional**. An agent without one uses the LLM's default voice. For internal tools that's fine; for any customer-facing agent it's almost always wrong.

How an Agent Actually Runs

When a user sends a message, this loop runs end-to-end. Understanding it is the difference between debugging "the AI gave a weird answer" in five minutes vs. five hours.

Step 1 — Prompt assembly

Turing ES builds the prompt in this exact order:

Persona system instruction (if a persona is attached)

Style guidelines block — derived from the persona's tone + verbosity + language style

Mandatory / forbidden vocabulary lists (if any)

Brand context — fetched from the persona's MCP server, if configured

Agent system prompt — the agent's own purpose-specific instruction

Tool definitions — JSON schemas for every native tool and MCP tool the agent has access to

Few-shot examples — top-K Q/A pairs retrieved by similarity from the persona's few-shot store, if configured (only on the first user turn)

Conversation history — full message log so far

Current user message

Layers 1-6 don't change per turn after the first message. Spring AI, the underlying integration, handles the assembly; you don't write any of this manually.

Step 2 — LLM inference

The assembled prompt goes to the LLM Instance the user picked. The model receives it and decides one of two things:

Respond directly — when the answer doesn't need any tool. Common for chit-chat, restated questions, simple factual recall.

Request a tool call — when the answer requires action. The model emits a structured request: tool name + arguments.

This decision is the agent's "thinking step". Tool-capable models (Claude, GPT-4, Gemini) do this well; small or older models struggle and may hallucinate tool calls or skip them entirely.

Step 3 — Tool execution

If the model requested a tool, Turing ES:

- Validates the call (does the agent have access to that tool? are the arguments valid JSON?).

- Routes it to the correct executor — native tool implementation or MCP server.

- Captures the result (or the error).

- Logs the invocation: tool name, arguments, latency, response size.

Step 4 — Multi-step reasoning

The tool result is fed back to the model, which can:

Call another tool — chain steps. For example: `search_knowledge_base` → `get_file_content` → `execute_python` to chart the data.

Stop and answer — synthesize the results into a final response.

This loop runs until the model decides it has enough. Spring AI's

`internalToolExecutionEnabled(true)` setting handles the loop transparently — the orchestration is built in.

Step 5 — Streaming the response

The final response streams back to the chat UI **token by token** via Server-Sent Events. The user sees the answer as it's being generated, the same way they'd watch someone typing. This isn't cosmetic — perceived latency drops by half versus waiting for a complete response.

Step 6 — Post-LLM persona enforcement

Before the response leaves the chat executor, `TurPersonaToneValidator` scans it for forbidden vocabulary from the persona. Matches are masked. The user never sees them. Logs record the event so you know your prompt isn't holding in some scenarios.

Step 7 — Analytics emission

A **chat session event** is recorded by `TurChatAnalyticsService` (turn count, token usage, agent ID, persona ID, locale, started-at, completed-at). This is the foundation for **Chat Analytics** — the dashboard that tells you, days later, *which agents are converting and which aren't*.

Composing Agents Around the Funnel

The biggest leverage in agent design is **funnel mapping**: build one agent per stage, not one agent for everything. Each agent gets a tighter system prompt, a more focused tool list, and a persona that fits the moment.

The Discovery Agent

For visitors who arrived from a marketing campaign and don't yet know what they want.

LAYER	CONFIGURATION
Name	Discovery Concierge
LLM	A fast, cheap instance (Gemini Flash, GPT-4o-mini) — discovery is high-volume
Tools	<code>search_site</code> (your marketing pages), <code>find_similar_documents</code>
MCP Servers	Optional: a CRM-lookup MCP that personalizes if the user is logged in
Persona	<code>top-of-funnel-sales</code> — EXECUTIVE tone, PERSUASIVE, verbosity 2
System Prompt	"You greet first-time visitors. Ask one qualifying question per turn. Don't list features; ask about outcomes. After three turns, offer a 15-min call."

Conversion target: discovery → demo booking.

The Solutions Engineer Agent

For technical evaluators in active comparison.

LAYER	CONFIGURATION
Name	Solutions Engineer
LLM	A top-tier reasoner (Claude Sonnet, GPT-4) — evaluators ask hard questions
Tools	<code>search_knowledge_base</code> , <code>get_file_content</code> , <code>fetch_webpage</code> , <code>execute_python</code>
MCP Servers	Internal API documentation MCP
Persona	<code>solutions-engineer</code> — TECHNICAL tone, INSTRUCTIONAL, verbosity 4
System Prompt	"You answer integration and architecture questions. Lead with the architectural answer. Confirm assumptions before recommending. Cite docs when possible."

Conversion target: evaluation → technical buy-in.

The Onboarding Coach Agent

For customers who just signed up.

LAYER	CONFIGURATION
Name	Onboarding Coach
LLM	Mid-tier (Sonnet, GPT-4o) — needs to follow conversational state precisely
Tools	<code>search_knowledge_base</code> (product docs), <code>get_file_content</code>
MCP Servers	Optional: account-state MCP that knows where the user is in the wizard
Persona	<code>onboarding-coach</code> — CASUAL, INSTRUCTIONAL, verbosity 3
System Prompt	"You guide brand-new users through their first day. Show one next step. Wait for confirmation. Celebrate small wins."

Conversion target: signup → activation milestone.

The Internal IT Agent

For employees, on-premises, behind your firewall.

LAYER	CONFIGURATION
Name	IT Helpdesk
LLM	A local LLM via Ollama — keeps queries off third-party clouds
Tools	<code>search_knowledge_base</code> (IT runbook), <code>execute_python</code> , <code>get_current_time</code>
MCP Servers	Internal ticketing system (stdio MCP)
Persona	None — this one optimizes for precision, not voice
System Prompt	<i>"You answer IT questions for employees. Cite the runbook section. If the issue isn't covered, open a ticket via the MCP and reply with the ticket ID."</i>

Outcome target: first-call resolution rate.

Configuration Form

The agent form is split into four tabs.

Settings

FIELD	REQUIRED	DESCRIPTION
Name	Yes	Display name shown as the tab label in the Chat interface
Avatar		Profile image — supports upload and removal
Description		Brief explanation of the agent's purpose
System Prompt		Instructions sent as a system message before every conversation. Defines purpose, scope, and behavior.
Persona		Optional Persona — overlays voice, vocabulary, brand context, and few-shot examples
Enabled		Toggle to activate or deactivate the agent

⚠️ DEFAULT SYSTEM PROMPT

If the system prompt is left blank, the agent uses the built-in default:

"You are an AI assistant. Answer the user's questions using the tools available to you. If you have access to MCP server tools, use them when relevant to fulfill the user's request. If the user asks in a specific language, respond in that same language."

This is fine for prototypes. For any customer-facing agent, write a real system prompt.

LLM

Select one or more LLM Instances. The list shows each instance's title, description, vendor, and model name. The user (or the front-end) picks which one to use at chat time, from the agent's allowed set.

Tools

Select which of the **27 native tools** are available. Tools are grouped by category — each group has a select-all checkbox for quick configuration.

MCP Servers

Select which external MCP servers this agent can call. The list shows each server's title, description, and connection type.

REST API

Agent Management

METHOD	ENDPOINT	DESCRIPTION
GET	/api/ai-agent	List all agents (ordered by title)
GET	/api/ai-agent/structure	Empty structure template for a new agent
GET	/api/ai-agent/{id}	Get a specific agent
POST	/api/ai-agent	Create a new agent
PUT	/api/ai-agent/{id}	Update an existing agent
DELETE	/api/ai-agent/{id}	Delete an agent

Agent Chat

METHOD	ENDPOINT	DESCRIPTION
POST	/api/v2/ai-agent/{agentId}/chat	Stream chat response (SSE). Body: { <code>llmInstanceId</code> , <code>messages[]</code> }
GET	/api/v2/ai-agent/{agentId}/chat/context-info	Context window size for the chosen LLM. Query: <code>llmInstanceId</code>

Native Tools

METHOD	ENDPOINT	DESCRIPTION
GET	/api/native-tool	List all available tool groups, names, and descriptions

Caching

Agent definitions are cached at the repository layer:

`turAIAgentfindAll` — the full list

`turAIAgentfindById` — individual lookups

Entries are invalidated automatically on create / update / delete. This means agent changes propagate immediately; no app restart needed.

Diagnosing a Misbehaving Agent

SYMPTOM	LIKELY CAUSE	WHERE TO LOOK
Agent gives generic answers, doesn't cite your content	No tool was called; the model didn't know to use one	Add a sharper system prompt directive: <i>"Always start by calling <code>search_knowledge_base</code> before answering."</i>
Agent uses brand names from a competitor or wrong product	Forbidden vocabulary not configured	Add the wrong terms to the persona's forbidden list
Agent's tone changes mid-conversation	No persona attached, or weak system prompt	Attach a persona; the post-LLM validator and prompt-side directives stabilize voice
Agent calls tools but ignores the results	Model is too small or the tool result is too large	Try a stronger LLM Instance; or trim the tool's output (e.g., reduce <code>find_similar_documents</code> top-K)
Conversion rate dropping over weeks	Few-shot store is stale; brand-context MCP returning old prices	Chat Analytics drill-down → check goal-achievement rate per persona
Latency spikes occasionally	A tool is slow; or the LLM provider is rate-limiting	Observability dashboards (<code>turing.llm.calls</code> timer)

Related Pages

PAGE	DESCRIPTION
LLM Instances	Configure the model providers used as agent brains
Tool Calling	The 27 native tools, grouped and explained
MCP Servers	Connect agents to external tool servers
Personas	Give agents a brand voice
Chat	The interface where agents come to life
Chat Analytics	Measure which agents are converting
Observability	Watch latency, token usage, and tool reliability in real time

Personas

*Your customer doesn't want to talk to an LLM. They want to talk to **your company**.*

A **Persona** is the voice profile of your AI. It's the difference between an assistant that sounds like every other chatbot on the internet and one that sounds like the most senior, most on-brand, most consistent representative your team could possibly hire.

You wire a Persona once, and from that moment forward, every agent you attach it to:

speaks in **your tone**, not the model's default,
uses the words you want to be remembered by,
never uses the words that get you in trouble,
answers with the **vocabulary of customers who already converted**, retrieved live from a few-shot store,
pulls **brand facts in real time** from an MCP server, so promotions and product names are never out of date.

Configure Personas in **Administration → Personas**. They are reusable across any AI Agent.

Why Personas Exist

Pick any modern LLM and ask it to "answer like a sales rep". You'll get something that's helpful, polite, and completely indistinguishable from the next ten companies asking the model the same thing. The same model, asked the same way, will quote different prices, use different brand names, and forget your discount window.

That's a problem for two reasons:

Conversion — generic voice doesn't sell. Customers convert when they feel they're being talked to like adults, by a person who knows what they came for. A senior sales rep doesn't read scripts; they sound like *the company*. Personas give your AI that consistency.

Risk — the same generic voice can also say something you'd never authorize. A wrong product name. A discount that doesn't exist. A claim about a competitor. Personas put guardrails in place: vocabulary you require, vocabulary you forbid — enforced both **in the prompt** and **after the response** before it reaches the user.

A Persona is the cheapest, most repeatable way to make every AI Agent speak with one voice.

Anatomy of a Persona

A Persona is a small bundle of decisions. Each one nudges the LLM in a specific direction, and together they replace the model's default voice with yours.

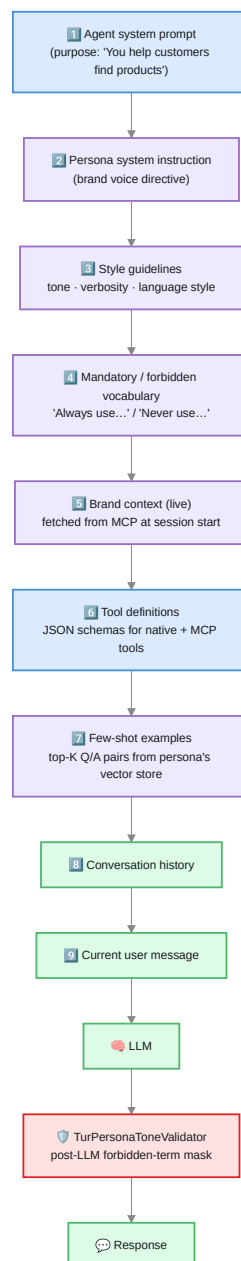
FIELD	WHAT IT DOES	WHY IT MATTERS FOR CONVERSION
Name	Internal identifier (not shown to the user)	Lets your team say "use the Sales Persona for that agent" without ambiguity
Description	One-line summary of when to use it	Helps non-technical admins pick the right persona
System Instruction	Free-text directive prepended to the agent's own system prompt	The narrative core — <i>"Speak as a senior account executive at YourCompany. Lead with outcomes, never features. Ask one question at a time."</i>
Tone	FORMAL · CASUAL · TECHNICAL · EXECUTIVE	Sets register. EXECUTIVE for C-level discovery; CASUAL for self-serve onboarding; TECHNICAL for developer relations; FORMAL for regulated industries.
Verbosity	1–5 scale	Lower is shorter. Use 2 for chat (mobile-friendly answers), 4 for advisory contexts where the user expects depth.
Language Style	NEUTRAL · DIRECT · NARRATIVE · PERSUASIVE · INSTRUCTIONAL	The shape of the answer. PERSUASIVE for sales. INSTRUCTIONAL for onboarding. NARRATIVE for long-form content.
Mandatory Terms	Pipe-separated list of words/phrases the LLM must use	Anchors brand language. "Customer Success ROI outcome" keeps the conversation in your vocabulary.
Forbidden Terms	Pipe-separated list of words/phrases that must never appear	Compliance and brand safety. Words like competitor names, deprecated product names, or sensitive claims.
Few-shot Store	Vector store with example Q/A pairs	Live retrieval of similar past conversations — the LLM learns <i>your</i> style by example, not by description
Brand Context MCP	An MCP server providing live brand facts	Real-time pricing, product catalogs, current promotions — pulled per conversation, never stale
Enabled	Toggle	Disable a persona to retire it without deleting; agents fall back to no-persona behavior

💡 PIPE-SEPARATED, NOT COMMA-SEPARATED

Both `mandatoryTerms` and `forbiddenTerms` use the pipe (`|`) separator, not commas. This lets a single term contain a comma — important for phrases like *"Tier 1, Tier 2 customers"*.

How a Persona Reaches the LLM

A Persona doesn't replace your AI Agent's system prompt — it **layers on top of it**. Here's exactly what happens at chat time:



Agent's own system prompt — the agent's purpose-specific instructions ("You help customers find products on our store"). Defined once on the agent.

Persona system instruction — the brand voice directive. Prepended to the agent's prompt.

Style guidelines block — a small structured paragraph derived from `tone` + `verbosity` + `languageStyle` (e.g., "Use an EXECUTIVE tone, verbosity 2 (concise), PERSUASIVE language.").

Mandatory / forbidden vocabulary — explicit "Always use:" and "Never use:" lists.

Brand context — if a `brandContextMcpServer` is wired, Turing ES calls it once at session start and injects the returned facts ("current product catalog", "active promotions") into the prompt.

Few-shot examples — if a `fewShotStore` is wired, the **user's first message** is embedded and the top-K most similar past Q/A pairs are retrieved and included before the user message. The LLM sees *real examples of how your company answers questions like this*.

Conversation history — the running message list.

Current user message — what they just typed.

The model never sees layers 1–6 again after the first turn — they're cached. Layers 7 and 8 evolve naturally with the conversation.

Post-LLM Tone Validation

The prompt-side guardrails (forbidden vocabulary in the system message) are necessary but not sufficient. LLMs occasionally drift, especially under tool-calling pressure or in long sessions. So Turing ES enforces forbidden terms **a second time, after the LLM responds**, before the response reaches the chat executor.

`TurPersonaToneValidator` scans every assistant response for matches against the persona's forbidden list. Matches are **masked** — replaced with a neutral placeholder — rather than the entire response being blocked. Logs record the match so you know when your prompt isn't holding.

Two-layer enforcement is deliberate:

LAYER	STOPS WHAT	LATENCY
Prompt-side	The model never <i>intends</i> to say it	Free (already in the system prompt)
Post-LLM	The model said it anyway	Microseconds — pure regex match

You don't need to choose. Both are on by default whenever a Persona is attached.

The Few-Shot Store: Teach by Example

The hardest part of a brand voice isn't the rules — it's the *feel*. You can describe FORMAL, but you can't fully describe "*the way Maria from sales answers a discount request*". So Turing ES lets you point a Persona at a **vector store of past Q/A pairs**, and at every conversation it retrieves the most relevant ones to use as few-shot examples.

How to populate the store:

Create a dedicated **Embedding Store** for the persona (e.g., a ChromaDB collection named `sales-persona-fewshot`).

Index real or curated Q/A pairs in it. Each document is a structured pair: a question your customers might ask, and an answer in the persona's voice. Quality > quantity. **20 well-written pairs beats 200 mediocre ones.**

Wire the store on the persona's **Few-shot Store** field.

At chat time:

The user's first message is embedded.

Turing ES retrieves the top similar Q/A pairs (default top 3, similarity ≥ 0.7).

Pairs are injected before the user's message as `Example Q:` / `Example A:` blocks.

The LLM sees concrete examples of how your company answers similar questions.

The result: a *PERSUASIVE* persona that doesn't just claim to be persuasive, but mirrors the cadence of conversations that already converted in your CRM.

CURATING Q/A PAIRS FROM REAL CONVERSATIONS

The **Chat Analytics** drill-down view exports finished sessions with full transcripts. Pick the ones the AI nailed — high goal-achievement rate, positive sentiment, business outcome confirmed — and use them as your seed Q/A set. Your few-shot store stays fresh because *real conversions feed it*.

Brand Context: Live Facts, Not Stale Prompts

Customers ask about pricing. About active promotions. About whether a feature is in beta. About a model that was just released yesterday.

If you bake those facts into the system prompt, two things happen:

The prompt grows past the model's optimal range.

Every time a price changes, marketing has to ask engineering to redeploy a prompt.

The **Brand Context MCP** field solves this. It points the persona at an MCP server you control. At every conversation, Turing ES calls a single MCP tool (e.g., `get_brand_context`) and injects the JSON response into the system prompt as a *"Current brand facts:"* block.

You get:

A **single source of truth** for prices, product names, promotions.

Marketing can update facts without anyone touching the persona.

Prompts stay short and readable — a key driver of LLM performance.

The same MCP can be reused across multiple personas (sales, support, partner).

A common architecture: marketing maintains a CMS, and an MCP server reads from that CMS. The persona pulls from the MCP. The CMS is the system of record; everything else is downstream.

Three Personas in Action

Here are three concrete personas, each tuned for a different point in the funnel.

1. The Top-of-Funnel Salesperson

When the visitor hasn't decided what they want yet.

FIELD	VALUE
Name	top-of-funnel-sales
Tone	EXECUTIVE
Verbosity	2
Language Style	PERSUASIVE
Mandatory Terms	outcome measurable within 90 days Customer Success
Forbidden Terms	cheap budget option competitor
System Instruction	"You are a senior account executive. Open with a one-sentence outcome the prospect can imagine. Ask exactly one qualifying question per turn. Never quote prices in the first 5 turns — instead, anchor on value."
Few-shot Store	sales-fewshot-tof (curated from conversations that converted to demo bookings)
Brand Context MCP	marketing-facts-mcp

The persona converts **discovery** into **booked demos**.

2. The Solutions Engineer

When the visitor is technical and evaluating.

FIELD	VALUE
Name	solutions-engineer
Tone	TECHNICAL
Verbosity	4
Language Style	INSTRUCTIONAL
Mandatory Terms	architecture reference API contract SLA
Forbidden Terms	magic black box
System Instruction	"You are a solutions engineer. Lead with the architectural answer. If the question is fuzzy, ask for the integration context (their stack, scale, latency budget) before answering."
Few-shot Store	engineering-fewshot (sourced from successful technical pre-sales calls)
Brand Context MCP	product-docs-mcp

The persona converts **technical evaluation** into **technical buy-in**.

3. The Onboarding Coach

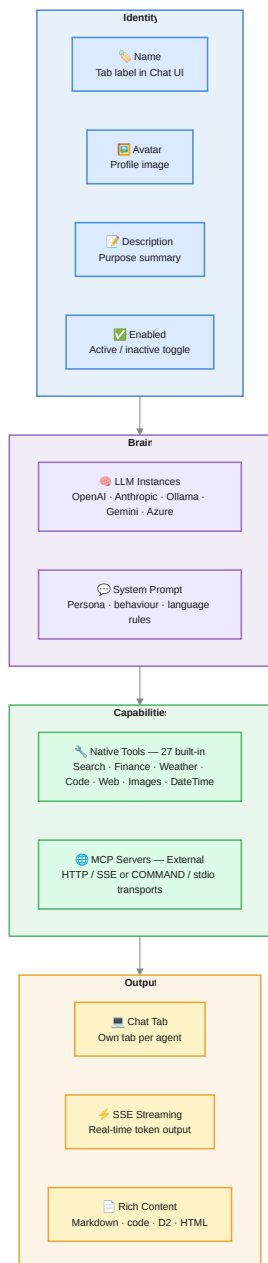
When the customer has just signed up and needs to get to value fast.

FIELD	VALUE
Name	onboarding-coach
Tone	CASUAL
Verbosity	3
Language Style	INSTRUCTIONAL
Mandatory Terms	you'll in 5 minutes first then
Forbidden Terms	simply just obvious
System Instruction	<i>"You are an onboarding coach. Always show one next step at a time. Confirm the user finished the previous step before moving on. Celebrate small wins."</i>
Few-shot Store	onboarding-fewshot (from sessions where a new customer reached the activation milestone)
Brand Context MCP	product-docs-mcp

The persona converts **activation** into **retention**.

Where Personas Fit in the Bigger Picture

A Persona is one of three layers that make an AI Agent come alive:



LAYER	WHAT IT PROVIDES
LLM Instance	The brain — the raw reasoning engine (LLM Instances)
Tools + MCP Servers	The hands — what the agent can <i>do</i> (Tool Calling, MCP Servers)
Persona	The voice — <i>how</i> it speaks

Without a Persona, an Agent still works — but its voice is the LLM's default voice. With a Persona, every agent that uses it sounds like the same coherent representative of your brand. Swap the LLM, swap the tools — the voice stays.

REST API

METHOD	ENDPOINT	DESCRIPTION
GET	/api/persona	List all personas
GET	/api/persona/{id}	Get a single persona
POST	/api/persona	Create a persona
PUT	/api/persona/{id}	Update a persona
DELETE	/api/persona/{id}	Delete a persona

When updating, the `mandatoryTerms` and `forbiddenTerms` arrive as pipe-joined strings from the form; the controller persists them verbatim. Validation happens at the LLM injection point (so an empty list is fine — it simply contributes no constraint).

How to Choose a Persona for an Agent

When you create or edit an **AI Agent**, the **Persona** field is a dropdown. The decision tree is short:

Is this agent customer-facing? If no, a Persona is optional (internal agents often just need precision, not voice). If yes, **always pick a persona** — even a minimal one. Customer-facing

without a persona means *the LLM's default voice is your company's voice*.

What part of the funnel? Pick a persona scoped to that stage (top of funnel, evaluation, onboarding, retention, escalation). Don't try to make one persona do all five.

Is the persona's few-shot store fresh? If the store was seeded six months ago and your product has shipped two major releases since, the few-shot answers will reference outdated features. Review and re-seed quarterly.

Common Pitfalls

PITFALL	SYMPTOM	FIX
Persona contradicts the agent's system prompt	Inconsistent answers; the LLM "fights itself"	The agent prompt says <i>what</i> the agent does; the persona says <i>how</i> it speaks. Keep them orthogonal.
Forbidden terms overlap with the agent's domain	Responses get masked frequently; user sees [redacted] placeholders	Forbidden lists are for marketing/compliance terms, not technical jargon. Move technical filters to the agent prompt.
Few-shot store mixed with general knowledge	Persona drifts toward whatever's in the store	The few-shot store should hold <i>only</i> curated Q/A pairs in the persona's voice. Use a separate Knowledge Base store for retrieval.
Brand context MCP returns 200KB of JSON	Token usage spikes; latency rises	The MCP should return the smallest possible relevant context — ideally under 1,000 tokens. Filter server-side, not client-side.
Single persona for the entire funnel	Onboarding coach also tries to upsell; sales rep also tries to teach Python	Personas are cheap. Make several, scoped tightly. The dropdown will keep them organized.

Related Pages

PAGE	DESCRIPTION
AI Agents	Where personas are attached
Chat	The interface where personas show their voice
Embedding Stores	Backend for the few-shot store
MCP Servers	Backend for live brand context
Chat Analytics	Where you discover whether the persona is actually converting

Chat

Every conversation is a chance to convert. The Chat interface is where Turing ES's reasoning, search, and brand voice meet your customer's question.

The **Chat** interface is the front door of every AI capability in Turing ES. It's organized into three modes, each tuned for a different shape of conversation:

MODE	WHEN IT SHINES	WHAT GROUNDS THE ANSWER
Chat (direct LLM)	The user wants raw assistance — <i>"summarize this PDF", "draft this email"</i>	The LLM's parametric knowledge + any tools the user enables
Semantic Navigation	The user is looking for something <i>inside your indexed content</i>	Strict — only your indexed sites and documents
AI Agent (<i>one tab per agent</i>)	The user is having a purpose-specific conversation (sales, onboarding, support)	The agent's tools + persona + your content

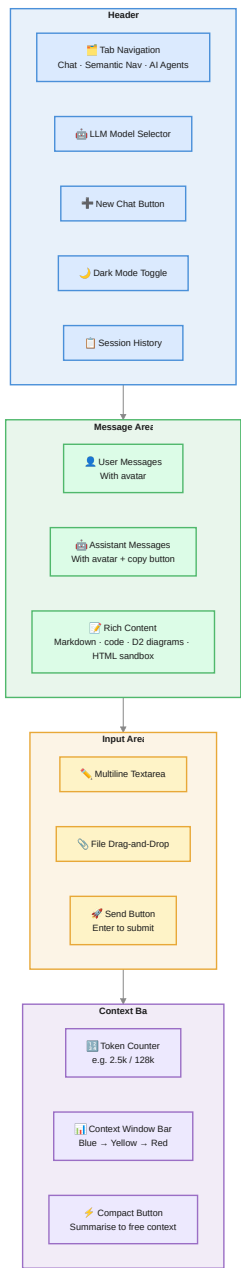
Every mode streams responses token by token, supports rich content (Markdown, code, charts, diagrams, HTML previews), and stores history in the user's browser — *not* on your servers.

ⓘ LLM REQUIRED

The Chat interface is only available when at least one LLM Instance is configured and enabled. See [LLM Instances](#) to set one up.

The Big Picture

The chat surface looks the same across modes, but each mode reaches a different backend and uses a different prompt strategy.



Header controls:

CONTROL	DESCRIPTION
Tab navigation	Switch between Chat, Semantic Navigation, and the dynamic AI Agent tabs
LLM model selector	Pick the LLM Instance to run this conversation against
New Chat	Start a fresh session (the current one is saved automatically)
Dark mode toggle	Switch between light and dark themes; code highlighting follows
Session history	Browse, restore, or delete previous conversations from this browser

Context Bar — sits below the message area:

INDICATOR	BEHAVIOUR
Token counter	Shows <code>current/max</code> (e.g., <code>2.5k/128k</code>); estimated client-side at ~4 chars per token
Progress bar	Visual fill — blue under 60%, yellow at 60–79%, red at 80%+
Compact	Compresses the conversation via the LLM to free context space

Mode 1 — Chat (Direct LLM)

A general-purpose conversation with the selected LLM. Use this when the question doesn't depend on your enterprise content — the user wants the model itself, sometimes augmented with a few opt-in tools.

Real conversations this mode handles well:

"Rewrite this paragraph in a more formal tone."

"Plot the rolling 30-day average of these numbers."

"What's the weather in Lisbon next Tuesday?"

"Find me an image of a Saturn V launch."

File Attachments

Drag-and-drop, or click the paperclip. Two paths, depending on the file:

FILE TYPE	HOW IT'S HANDLED
Documents (PDF, DOCX, XLSX, PPTX, HTML, TXT, ...)	Text extracted via Apache Tika and added to the prompt as context
Images (PNG, JPEG, WebP, GIF, ...)	Sent directly as media to vision-capable models (Claude Sonnet, GPT-4o, Gemini)

Attached files appear as badges on the message they're sent with. Multiple files per message are supported.

Streaming

Every response streams in real time over **Server-Sent Events (SSE)**. The user sees tokens arrive as the model produces them — no spinner, no wait. Perceived latency drops by roughly half compared to wait-then-show.

Tools You Can Enable

In direct LLM mode, the user toggles tools per conversation. The LLM decides when to invoke them.

TOOL	WHAT IT DOES	LATENCY TYPICAL
Code Interpreter	Runs Python in a sandbox (Matplotlib supported). 30s timeout. Files generated (charts, CSVs) come back as download links inline.	1-10s
Web Crawler	Fetches and extracts a public web page. Up to 12,000 chars of body text and 30 links.	1-3s
Image Search	Searches images via DuckDuckGo / Bing. Up to 8 results.	1-2s
Weather	1-7 day forecast for any location (Open-Meteo).	<1s
Finance	Stock quotes and historical prices via Yahoo Finance.	1-2s
Date / Time	Current date/time in any timezone.	<100ms
RAG Search	Searches the Knowledge Base by semantic similarity; lists files; reads file contents.	100-500ms

The model picks which tool to call based on the question. The user enables the menu of options; the LLM picks within it.

Mode 2 — Semantic Navigation

This is where the chat becomes a *grounded* conversation: the LLM's job is no longer to answer from its training data, but to **search your indexed sites and explain what it finds**.

Use this mode when:

The question is about your **products**, **internal documentation**, or **published content**.

You need answers that **always trace back to a real document**.

Hallucinations are unacceptable — the LLM is constrained to answer from indexed content.

The system prompt for this mode is built per request and includes:

The list of available SN Sites and their locales,

The facets configured on each site (so the model knows which filters it can use),

An instruction to ground every answer in search results.

Tools available:

TOOL	PURPOSE
<code>list_sites</code>	Enumerates available SN Sites and their locales
<code>get_site_fields</code>	Returns the indexed fields and facets for a specific site
<code>get_valid_filter_values</code>	Lists valid values for a given facet field — prevents the model from inventing filters
<code>search_site</code>	Performs a semantic search and returns results with snippets and metadata

Any **MCP Servers** configured globally are also available here, extending the tool set with external capabilities (e.g., a CRM lookup, a translation service).

 WHEN TO USE SN MODE VS. AN AGENT

SN mode is the right answer when the user is *looking for something inside your content* but doesn't need a specific persona or workflow. It's the AI version of "*search the site, but smart*".

Agent mode is the right answer when the conversation has a *purpose* — booking a demo, qualifying a lead, getting onboarded. Both can search the same content; the agent layers persona + workflow on top.

Mode 3 — AI Agents

Each **AI Agent** configured and enabled in **Administration** → **AI Agents** appears as its own tab in Chat. The visitor picks the specialist that fits their need.

WHAT'S PER-AGENT	SET IN
Name & Avatar	Agent → Settings tab
System prompt	Agent → Settings tab
LLM Instance	Agent → LLM tab
Native tool selection	Agent → Tools tab
MCP servers	Agent → MCP Servers tab
Persona <i>(brand voice)</i>	Agent → Settings (Persona dropdown)

The **Persona** is the voice layer — see [Personas](#) for full coverage. With a persona attached, every conversation in this tab speaks in your brand's tone, uses your mandatory vocabulary, avoids your forbidden vocabulary, and (if configured) draws on a **few-shot store of curated Q/A pairs** plus **live brand context from an MCP server**.

Use agents when you want consistency across thousands of conversations — *"every visitor who lands on the discovery agent should hear the same voice, regardless of LLM or time of day"*.

Picking a Mode for the Job

If you take one thing away from this page, take this:

THE USER IS TRYING TO...	USE THIS MODE
Generate, summarize, or transform content (no enterprise grounding needed)	Chat (direct LLM)
Find or understand something inside your indexed content	Semantic Navigation
Do a <i>purposeful</i> conversation (sell, support, onboard, qualify)	AI Agent with a persona
Investigate a specific past conversation and what went well or wrong	Open the Chat Analytics page (separate, not in Chat)

You don't have to pick one mode and stick with it. Most deployments use all three — direct chat for power users, SN chat for content exploration, and agents for customer-facing flows.

How a Message Travels End-to-End

What actually happens between *the user pressing Enter* and *the response appearing*?

Front-end captures the message + any attachments and POSTs to the appropriate endpoint:

Direct LLM → `/api/v2/llm/{instanceId}/chat`

SN chat → `/api/v2/llm/{instanceId}/semantic-chat`

Agent → `/api/v2/ai-agent/{agentId}/chat`

Spring AI assembles the prompt: agent system prompt → persona overlays (if any) → tool definitions → conversation history → current message + media.

Tool calling — if the LLM requests a tool, Spring AI executes it (native or MCP), returns the result, and the loop continues until the model decides to answer.

Streaming — the response flows back through `Flux<ChatResponse>`. The front-end consumes it as SSE, rendering each token as it arrives.

Persona post-validation — `TurPersonaToneValidator` redacts any forbidden terms before the response is finalized.

Analytics emission — `TurChatAnalyticsService.recordSessionStart` / `recordTurn` / `recordSessionEnd` record turn count, token usage, and outcome to the analytics store. See [Chat Analytics](#).

Browser stores the session in IndexedDB, attaches an auto-generated title (LLM-summarized from the first exchange), and updates the session sidebar.

If observability is enabled (Prometheus scraping `/actuator/prometheus`), every step also emits metrics — see [Observability](#).

Rich Content Rendering

Chat responses are rendered with full media-type awareness:

CONTENT TYPE	RENDERING
Markdown	GitHub Flavored — tables, strikethrough, task lists, inline code, blockquotes
Code blocks	Syntax highlighting via highlight.js with automatic light/dark theme switching
D2 diagrams	Rendered to SVG via WASM; falls back to a dev server in development
HTML	Sandboxed preview in an iframe; toggle between rendered view and source, with fullscreen
Generated files	Files from the Code Interpreter (charts, CSVs, processed data) appear as inline download links

Code blocks pick up the chat's dark/light theme automatically — no flash on switch.

Session History

Sessions are stored in the **browser's IndexedDB**. They never leave the user's machine.

What that means in practice:

Sessions are **per browser and per device** — clearing browser data removes them.

No authentication is required to access past sessions.

No server cost for session storage.

Session sidebar features:

FEATURE	DESCRIPTION
Auto-title	A short title is generated by the LLM from the first exchange; falls back to the first message text if generation fails
Model badge	Which LLM model ran the session
Message count	Number of messages
Timestamp	Date/time of the last message
Restore	Click to resume a previous session
Delete	Remove a session from local history

Sessions are saved automatically after each complete response.

❗ BROWSER-LOCAL VS. CHAT ANALYTICS

The IndexedDB **session history** is a *user convenience* — it's how a single user finds their own past conversations. The **Chat Analytics** store is *operator analytics* — anonymized session metadata + transcripts that ship to MongoDB or Redis for dashboarding. They serve different audiences and don't depend on each other.

Context Window Management

A context bar at the bottom of the input shows token usage in real time:

2.5k / 128k 

Tokens are estimated client-side at **~4 characters per token** (`Math.ceil(text.length / 4)`), counting the full message history. The bar's fill and color reflect how close you are to the model's context window.

Resolving the Context Window Size

When the front-end first loads a session, it figures out the model's window via a three-tier fallback:

Backend API — `GET /v2/llm/{instanceId}/chat/context-info` returns the configured limit. The front-end caches it.

LLM Instance configuration — the `contextWindow` field set on the instance.

Default — 128,000 tokens, used when neither of the above is available.

Progress Bar Colours

USAGE	COLOUR	MEANING
Below 60%	Blue	Plenty of room
60–79%	Yellow	Consider compacting
80%+	Red	Compact before the limit

Compact

The **Compact** button (lightning bolt) becomes available when the conversation has at least **4 messages** and the model isn't currently generating. Compacting:

Sends the full history to the same LLM with a summarisation prompt.

The LLM produces a concise summary preserving facts, decisions, and context.

The conversation in IndexedDB is replaced with a single `**[Context compacted]**` block + the summary.

The next user turn continues from the compacted state, with significantly reduced token usage.

Compacting works in all three modes.

WHEN TO COMPACT

When the bar turns **yellow**, compact proactively. By the time it's red, the model may have already started losing context from the earliest messages. The Compact tooltip shows how much context remains.

Files & Attachments Reference

CAPABILITY	DETAIL
Upload method	Drag-and-drop onto the chat window, or click the attachment button
Transfer format	Multipart form (files sent together with the message)
Document processing	Apache Tika extracts text from PDF, DOCX, XLSX, PPTX, HTML, TXT, and more
Image processing	Passed directly as media bytes to vision-capable models
Display	Shown as file badges on the sent message bubble

API Endpoints

METHOD	ENDPOINT	MODE
POST	/api/v2/llm/{instanceId}/chat	Direct LLM (SSE stream)
POST	/api/v2/llm/{instanceId}/semantic-chat	Semantic Navigation (SSE stream, tool-calling enabled)
POST	/api/v2/ai-agent/{agentId}/chat	Agent (SSE stream, full agent execution loop)
GET	/api/v2/llm/{instanceId}/chat/context-info	Context window size resolution

See [REST API Reference](#) → [GenAI API](#) for request/response shapes and examples.

Related Pages

PAGE	DESCRIPTION
AI Agents	The specialists that power the per-tab agent conversations
Personas	The voice layer applied to every customer-facing agent
LLM Instances	Configure and connect the model providers
Tool Calling	The 27 native tools and how they appear in chat
MCP Servers	Connect external tool servers to chat
Chat Analytics	Investigate, classify, and learn from past conversations
Observability	Prometheus + Grafana dashboards over chat traffic

Chat Flow

*Sometimes a conversation needs to go somewhere specific. A demo qualification has six questions in a precise order. An onboarding wizard has a checklist that can't be skipped. A support escalation collects four pieces of information before opening a ticket. **Chat Flow** is how you draw those journeys — visually, declaratively, and in five minutes.*

A **Chat Flow** is a graph of conversational steps attached to an **AI Agent**. Each step (a *node*) is a small contract — "ask this, validate that, store this answer in this variable, then go to the next step". The agent still uses the LLM to talk to the user; the flow steers the conversation while the LLM does the talking.

Three things make Chat Flow special:

Visual authoring. You drag nodes onto a canvas, draw edges between them, and the runtime walks the graph for you. No code.

Multiple guardrail strategies. You decide *how strictly* the flow is enforced — from a lightweight prompt-only nudge to a JSON-validating LLM judge.

Vibe Coding. A built-in AI authoring chat lets you describe the flow in plain English ("a five-step demo qualification flow that collects company name, role, current solution, decision timeline, and email") and watch it draw itself. You revise it the same way: "split the timeline question into two — short term and long term".

Configure flows in **Administration** → **AI Agents** → <your agent> → **Chat Flow**.

When You Need a Chat Flow

A free-form agent works beautifully when the conversation is open-ended. "Help me find a product" or "explain this concept" don't need scripted scaffolding — the LLM and your tools are enough.

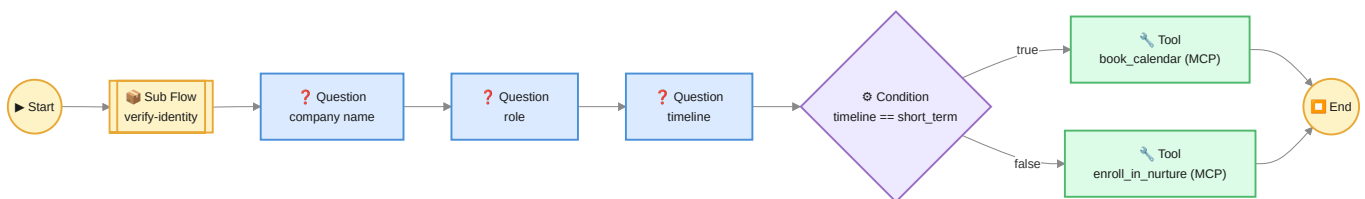
But there are conversations where you actually need to **collect specific information in a specific order**:

GOAL	NEEDS A FLOW BECAUSE...
Qualify a sales lead	Five questions you need answered, every time, in order
Open a support ticket	Four pieces of info required by your ticketing system
Walk a new user through onboarding	Each step depends on the previous one being confirmed
Run a survey or NPS conversation	Questions follow a strict structure for analysis later
Gather KYC / compliance information	Regulators expect a complete audit trail of <i>what was asked</i> and <i>what was answered</i>

Without a flow, you'd write a 1,500-token system prompt full of "first ask this, then ask that, never skip..." — and the LLM would still skip something halfway through. With a flow, the prompt is short, the order is enforced by the engine, and every collected value is stored in a typed variable.

The Five Block Types

A Chat Flow graph is built from **five node types**. Each one has a specific runtime behavior.



A typical demo-qualification graph: an identity sub-flow at the start, three Question nodes, a Condition, and one of two Tool nodes depending on the timeline.

1. Start

The single entry point of every flow. There is exactly one Start node per flow. It has no input edges and one output edge to the first real step.

The Start node is invisible to the user — the conversation simply begins at the node it points to.

2. Question

The bread and butter of a flow. A Question node:

Carries an **AI Instruction** — what the LLM should ask the user. Written in natural language: *"Ask the user what their current monthly active user count is. Frame it as a market-sizing question, not an audit."*

Optionally has a **Validation Rule** — what counts as a valid answer. *"Must be a number; if the user says 'a lot' or 'many', ask them to estimate."*

Stores the user's answer in an **Output Variable** — a named slot in the flow's state (e.g., `monthly_active_users`).

When a Question node is active, the runtime injects the AI Instruction into the system prompt so the LLM knows exactly what to ask. The LLM still controls the *wording*, the *register*, the *follow-up* — but the *intent* is locked.

3. Condition

Branches the flow based on collected variables. A Condition node has:

A **Condition Expression** evaluated against the flow's variables (e.g., `monthly_active_users > 10000`).

Two outgoing edges — one labelled *true*, one labelled *false*. The runtime follows whichever matches.

Use Condition nodes to skip questions that don't apply, route enterprise leads differently from self-serve, or fork a survey based on an earlier answer.

4. Tool

Calls a tool *as part of the flow*, not because the LLM asked for it. A Tool node has:

A **tool source** — `native` (one of the 27 **native tools**), `mcp` (a connected **MCP server**), or `custom` (a **Custom Tool** you authored).

The **tool identifier** — function name for native, server id for MCP, custom-tool id for custom.

Use Tool nodes when a step needs a deterministic action *between questions*. Examples:

After collecting an email, call a CRM lookup MCP to enrich with the company.

After a series of preferences, call `search_knowledge_base` to retrieve recommendations.

After collecting an issue description, call a custom Groovy tool that opens a Jira ticket and returns the ID.

The result of the tool call lands in a flow variable and the conversation continues.

5. Sub Flow

The composition primitive. A Sub Flow node points at *another* Chat Flow on the same agent and runs it inline. When the sub-flow reaches its End node, control returns to the parent flow's next node.

Use Sub Flows for shared building blocks:

A "*collect-contact-info*" sub-flow used by the demo qualification, the trial signup, and the support escalation.

A "*verify-identity*" sub-flow used at the start of any compliance-sensitive workflow.

A "*NPS-prompt*" sub-flow triggered at the end of a successful conversation.

Sub-flows have their own state row (`TurChatFlowState.parentStateId` records the call stack). Variables collected in the sub-flow are available in the parent when control returns.

End

Terminator. Like Start, exactly one per flow. When the runtime reaches End, it marks the flow's state as completed, persists a `TurChatFlowSubmission` row (full collected variables + outcome), and the conversation continues normally — the agent is back to free-form mode.

Guardrails: Three Strategies, Three Levels of Strictness

The same flow graph can be enforced **three different ways** at runtime. You pick per-flow which strategy fits the conversation. There is no "best" — there is a tradeoff between strictness, latency, and token cost.

Strategy 1 — **HEURISTIC** (default)

The fast lane. No extra LLM calls. The current node's instruction is injected as a firm contract in the system prompt; after the LLM replies, a lightweight Java-side heuristic decides whether the user gave a valid answer and the engine advances.

PROPERTY	VALUE
Extra LLM calls	0
Latency overhead	None
Token overhead	~50-150 tokens added to system prompt
Strictness	Low — the LLM may sometimes accept fuzzy answers
Use when	The flow is conversational, customers can rephrase, and the cost of a mis-route is low

Heuristic is the right starting point for most flows. Run it. Watch **Chat Analytics** for sessions where the goal wasn't achieved. Upgrade only if you see drift.

Strategy 2 — **LLM_JUDGE**

Two-call enforcement. After the chat LLM replies to the user, a *second* LLM call ("the judge") is made with a tiny structured prompt: *"Was the user's answer on-topic? Did they provide a valid value? Should we advance?"* The judge returns JSON like:

```
{
  "on_topic": true,
  "collected_value": "12500",
  "ready_to_advance": true
}
```

If `on_topic` is false, the engine substitutes a redirect message (*"Let's stay on track — could you share your monthly active users?"*) instead of advancing. If `collected_value` is present, it's stored in the node's output variable.

PROPERTY	VALUE
Extra LLM calls	1 per turn while a flow is active
Latency overhead	~200-500ms (judge call)
Token overhead	~200 tokens per turn
Strictness	High — drift is caught and corrected
Use when	The flow is a structured form (KYC, compliance), drift costs money or violates policy

The judge can use a smaller, cheaper model than the chat LLM — its job is classification, not generation.

Strategy 3 — `STRUCTURED_OUTPUT`

Single-call enforcement. Instead of two LLM calls, the chat LLM is instructed to reply with a **single JSON object** that bundles both the user-facing message *and* the verdict:

```
{
  "reply": "Got it — 12,500 monthly active users. Next, what's your role on the team?",
  "on_topic": true,
  "collected_value": "12500",
  "ready_to_advance": true,
  "abandoned": false
}
```

The engine extracts `reply` and shows it to the user; applies the verdict; advances. One call, full enforcement.

PROPERTY	VALUE
Extra LLM calls	0 (the chat call carries both)
Latency overhead	None — the judge piggybacks on the reply
Token overhead	~100 tokens (structured response is more verbose than free text)
Strictness	High — same as <code>LLM_JUDGE</code>
Use when	You want strict enforcement <i>and</i> low latency

Modern providers (OpenAI, Gemini, Ollama) follow JSON instructions reliably. Anthropic occasionally drifts to free text — when that happens, the engine falls back gracefully to heuristic mode for that turn.



PICKING A STRATEGY

Start with `HEURISTIC`. Run it for a week. Read 20 sessions in [Chat Analytics](#) drill-down — pay attention to ones that ended with `goalAchieved=NO` or `sentiment=FRUSTRATED`. If the LLM was drifting off-topic, upgrade to `STRUCTURED_OUTPUT`. Only escalate to `LLM_JUDGE` if `STRUCTURED_OUTPUT` falls back too often (provider-specific issue) or if you need the judge to use a different model from the chat LLM.

Trigger Modes: Auto-Routing or Manual

A flow doesn't run unless something *picks* it. Two ways:

Auto-trigger via the LLM router

When you set a **Trigger Description** on a flow (e.g., *"User wants to book a demo or qualify their team for a sales call"*), the engine includes it in a tiny LLM router prompt at the start of every chat turn. The router decides: does the user's message match any of this agent's flow descriptions? If yes, that flow becomes active.

TRIGGER MODE	BEHAVIOR
ONCE (default)	Auto-trigger at most once per conversation. After the flow reaches End, the router won't pick it again in this conversation.
ALWAYS	Re-trigger whenever the router decides the message matches — even after a previous run.

Leave Trigger Description blank to disable auto-trigger entirely. The flow can still be invoked manually.

Manual selection

Users (or the front-end) can explicitly start a flow by name. This bypasses the router and is useful when:

- You want a button on the front-end ("*Schedule a demo*") that always starts the qualification flow.

- A flow is sensitive enough that you don't want the router making the call (e.g., compliance KYC — only triggered by an authenticated admin).

Per-Conversation State

Every running flow has a `TurChatFlowState` row identified by `(conversationId, flowId)`. It tracks:

FIELD	WHAT IT HOLDS
<code>currentNodeId</code>	The node the user is currently on
<code>variablesJson</code>	All collected output variables, JSON-serialized
<code>parentStateId</code>	Set when this state is running a sub-flow — points to the parent state. The runtime walks back up when the sub-flow ends.
<code>updatedAt</code>	Last touched — useful for stale-flow cleanup

The state is updated **after** the assistant replies — so if the conversation crashes mid-turn, the state still reflects the last completed step.

When the End node is reached, the engine writes a `TurChatFlowSubmission` row: the full graph traversal, all collected variables, and the outcome. This is your audit log. You can list submissions by flow:

```
GET /api/ai-agent/{agentId}/chat-flow/{flowId}/submissions
```

A common pattern: a CRM integration listens for new submissions and creates a deal automatically, populated from the variables.

Vibe Coding: Author Your Flow With AI

Building a flow visually is fast. Building it by *describing it* is faster.

Click **Vibe Coding** on the flow editor and you're talking to an authoring assistant whose entire job is "build the chat flow this user wants". Two modes of interaction:

First-turn creation

You describe the flow in plain English — short or long, structured or rambling, your call. The assistant returns a complete graph: nodes positioned, edges connected, instructions written, output variables named.

You: "I need a five-step demo qualification: company name, role, current solution, decision timeline (under 3 months / 3-6 months / over 6 months), and email. After collecting all five, call our CRM via MCP to look up the company. End the flow."

Assistant: "Done. I've created six Question nodes for the data you listed, a Tool node calling your `crm-lookup` MCP server after the email, and an End node. The decision timeline question uses a Condition node so you can branch the follow-up depending on urgency. Take a look — I positioned everything left to right with the Start node on the far left."

The flow appears on the canvas. You can drag, edit, save.

What's happening under the hood:

The assistant is given a **rich system prompt** documenting every node type, every field, edge handle conventions, layout rules, and current guardrail/trigger policy.

It returns a structured `ChatFlowGeneration` JSON the engine post-processes: deterministic positioning, id normalization, connectivity validation. Common mistakes (missing Start, dangling edges, Condition without two branches) are auto-corrected.

Revision turns

Once a flow exists, you can keep talking:

You: "Split the timeline question into two — one for short-term plans (this quarter) and one for long-term (this year)."

Assistant: "Done. I split the timeline node into `short_term_plans` and `long_term_plans`, kept them in sequence after the role question, and adjusted the CRM lookup to fire after both are collected."

In revision turns the model sees the **full current state** plus a slimmer *edit-focused* prompt. It only changes what you asked for; the rest of your graph stays untouched.



WHEN VIBE CODING IS THE RIGHT MOVE VS. VISUAL EDITING

Vibe Coding wins when you're starting from scratch, when you're rearranging structurally ("*split this*", "*merge those*"), or when you don't yet know what blocks you'll need. **Visual editing wins** when you're tweaking copy on a single node, dragging to reorganize visually, or fine-tuning a validation rule. Use both — they edit the same graph.

The same Vibe Coding pattern is available for **Intents** and **Custom Tools**. The pattern is general — described in **AI Authoring (Vibe Coding)** — but Chat Flow is the most powerful expression of it because of the structural complexity of the graph.

A Real Example: Demo Qualification

Let's walk one flow end-to-end. Goal: qualify an inbound prospect, route hot leads to a calendar booking, route cold leads to a nurture sequence.

Graph (described)

```
Start
  → Question: company name      → output: company_name
  → Question: user's role       → output: role
  → Question: current solution  → output: current_solution
  → Question: timeline          → output: timeline (enum: short_term |
long_term | exploring)
  → Condition: timeline == short_term
    true → Tool: book_calendar (MCP) → End
    false → Tool: enroll_in_nurture (MCP) → End
```

What the user sees

The agent greets normally (free-form), sees the user message "*I'd like to learn more*", and the **router** matches the demo-qualification flow's trigger description. The flow becomes active. The agent's next message asks the company name. (The Question node's AI Instruction guided the wording; the LLM picked the actual phrasing in the user's locale.) After each answer, the engine validates with the chosen guardrail strategy and advances. Five answers in, the engine evaluates the Condition and routes to one of two Tool nodes. The Tool returns a result (a booked slot or a nurture confirmation), which the LLM phrases naturally into the final message.

End is reached; a `TurChatFlowSubmission` is written to the database; CRM gets notified.

What you see, days later

In **Chat Analytics**:

The conversation is one row, with `intentLabel = CONVERSION_INTENT`, `goalAchieved = YES`, `sentiment = POSITIVE`, `keyTerms = ["demo", "Q1", "team of 8"]`.

The submission record has the full variable map.

A scorecard panel groups submissions by `agentId` so you can compare which agent is converting.

That's the loop. Conversation → flow → submission → analytics → product decision → flow update → better conversation. Closed.

REST API

METHOD	ENDPOINT	DESCRIPTION
GET	/api/ai-agent/{agentId}/chat-flow	List all flows on an agent
GET	/api/ai-agent/{agentId}/chat-flow/{flowId}	Get a flow with its full graph JSON
POST	/api/ai-agent/{agentId}/chat-flow	Create a flow
PUT	/api/ai-agent/{agentId}/chat-flow/{flowId}	Update a flow's graph or settings
DELETE	/api/ai-agent/{agentId}/chat-flow/{flowId}	Delete a flow
GET	/api/ai-agent/{agentId}/chat-flow/{flowId}/submissions	List collected submissions
POST	/api/ai-agent/{agentId}/chat-flow/ai-chat	Vibe Coding endpoint — describe a flow, get a graph back

The Vibe Coding endpoint follows the standard **AI Authoring** shape: a chat over

`AiAuthoringRequest<ChatFlowGeneration>` / `AiAuthoringResponse<ChatFlowGeneration>`.

Configuration & Tuning

The flow engine has a few knobs:

PROPERTY	DEFAULT	PURPOSE
<code>turing.chat.flow.judge.model</code>	(default LLM)	Which LLM the <code>LLM_JUDGE</code> strategy uses for the judge call. Pick a small/cheap model — classification is its only job.
<code>turing.chat.flow.state.ttl-hours</code>	24	How long an in-flight <code>TurChatFlowState</code> is kept before being considered stale. Stale states are cleaned by a daily housekeeping job.

State is per-conversation — abandoning a conversation leaves a state row that's pruned by the housekeeping job after the TTL.

Common Patterns

Funnel-as-flow

One flow per funnel stage, attached to one agent each. Discovery → Qualification flow on the Sales agent. Activation → Onboarding flow on the Onboarding Coach agent. Renewal → Health-check flow on the CSM agent.

Reusable identity verification

A single `verify-identity` flow with three Question nodes (name, account email, last-four-digits). Every other compliance-sensitive flow starts with a Sub Flow node pointing at it. One source of truth; consistent behavior.

Tool nodes as side effects

Don't make the LLM ask the user *"would you like me to look up your account?"* — just put a Tool node after the relevant Question. The flow handles it deterministically. The LLM gets the result and weaves it into the next message.

Conditional branching for self-serve vs. enterprise

After the company-size question, branch:

`under_50_employees` → straight to the self-serve signup flow,

`over_50_employees` → into the enterprise qualification path with extra questions about procurement timelines.

One flow, two outcomes, no human intervention.

Diagnostics

SYMPTOM	LIKELY CAUSE	WHERE TO LOOK
Flow drifts off-topic mid-conversation	<code>HEURISTIC</code> strategy is too lenient for this flow	Switch to <code>STRUCTURED_OUTPUT</code> ; if the issue persists, switch to <code>LLM_JUDGE</code> with a stronger judge model
User messages bypass the flow router	The trigger description is too narrow; or the router LLM is too small	Broaden the Trigger Description with more user-language variations; or upgrade the default LLM
Variables aren't being collected	The Question node's Output Variable name doesn't match the validation rule's expected output	Check the Output Variable on the Question node; check the Tool node downstream for matching arg names
Sub-flow runs but never returns	The sub-flow has no End node, or there's a dangling edge before End	Open the sub-flow in the editor — Vibe Coding's auto-fix usually catches this; visual editing can introduce it
State accumulates indefinitely	TTL housekeeping isn't running	Verify <code>turing.chat.flow.state.ttl-hours</code> and that the housekeeping <code>@Scheduled</code> is firing

Related Pages

PAGE	DESCRIPTION
AI Agents	The agents that flows are attached to
Custom Tools	Build your own Tool nodes with Groovy + Vibe Coding
Tool Calling	Native tools available as Tool nodes
MCP Servers	External tools available as Tool nodes
Intent	The lightweight cousin: prompt suggestions for the chat home screen
Chat Analytics	Where flow outcomes show up

Intent

*The hardest part of any AI product isn't the AI — it's getting the user to ask the first question. **Intents** are how you put a hundred good first questions, in your customer's voice, on the chat home screen. And **Vibe Coding** is how you write all of them in five minutes.*

The **Intent** feature (</admin/intent/instance>) manages categorized prompt suggestions that appear in the Chat interface. Intents provide users with pre-built conversation starters organized by topic, making it easier to discover and use the AI capabilities of Turing ES.

Each intent represents a **category** (e.g., "Boost my work productivity") containing one or more **actions** — each action is a labeled prompt that is sent directly to the chat when clicked.

Think of intents as the *hook* on the home screen of every conversation. The customer didn't type a question yet — they're staring at the prompt box wondering what to ask. Intents say "*here's what people like you ask*", in their language. Conversion starts here.

Intent Listing

The listing page shows all configured intents as a grid of cards. Each card displays:

- Title** — the intent category name
- Description** — a brief explanation of the category
- Icon** — a Lucide icon representing the category

A **New Intent** button at the top opens the creation form. Clicking an existing card navigates to its detail page.

Creating or Editing an Intent

The intent form is divided into three sections.

General Information

FIELD	MAX LENGTH	DESCRIPTION
Title	150 chars	Category name displayed in the chat interface (required)
Description	500 chars	Brief explanation of what prompts this intent provides
Icon	—	Lucide icon picker — select an icon to represent this category

Actions

Actions are the individual prompts within an intent category. Each action has:

FIELD	DESCRIPTION
Label	Short action label shown to the user (e.g., "Create daily schedule") — required
Prompt	The full prompt text sent to the chat when the user clicks this action — required

Actions support **drag-and-drop reordering** — drag the handle on each action card to change the display order. Use the **Add Action** button to create new actions, and the trash icon to remove existing ones.

WRITE ACTIONS IN THE CUSTOMER'S VOICE

The label is what the user clicks. The prompt is what the LLM receives. They don't have to be the same. The label should sound like *the user asking themselves a question* ("Help me prepare for my next 1-on-1"); the prompt can be a more elaborate brief for the LLM ("Help me prepare for a 1-on-1 meeting with my manager. Suggest topics, questions to ask, and a way to bring up career growth."). The pairing is what makes intents feel like a friendly assistant rather than a search box.

Status

FIELD	DESCRIPTION
Enabled	Toggle to show or hide this intent in the chat interface. Disabled intents are not visible to users.



Vibe Coding: Author Intents With AI

Writing 4 categories × 6 actions × 2 languages = 48 prompts is exactly the kind of work the AI is best at. The Intent form has a **Vibe Coding** mode — you describe what you want; the assistant drafts the entire intent.

You: "Make me an intent for new product owners. Six actions covering the first 30 days on the job — backlog grooming, stakeholder mapping, OKRs, the first sprint demo, a retro plan, and a simple metric to track."

Assistant: "Done. Title: 'New Product Owner — First 30 Days'. Icon: graduation-cap. Six actions ready: — **Map your stakeholders** → 'Help me build a stakeholder map for my first month as PO. Roles, influence, what they need from me, and one question I should ask each of them.' — **Groom the backlog** → 'Walk me through grooming a backlog I just inherited. What's the right order to clean it?' — **Set OKRs for the first quarter** → ... Want me to also write a Portuguese version?"

The same pattern as **Chat Flow** and **Custom Tools**. Two interaction modes:

First-turn creation — describe the intent, get the full structure (title, description, icon, all actions).

Revision turns — "add an action about pricing", "make the third one shorter", "translate everything to Portuguese".

Vibe Coding is especially powerful here because intents are usually authored *en masse* — you're populating the home screen, not writing a one-off. Three minutes of prompting beats two hours of typing.

Pre-loaded Intents

Turing ES ships with four default intents:

INTENT	ICON	ACTIONS
Boost my work productivity	Briefcase	Create daily schedule, Draft a difficult email, Prepare meeting agenda
Learn something new	Graduation Cap	Explain complex topics, 30-day study plan, Test my knowledge
Create or improve content	Pen Tool	Social Media Ideas, Proofread & Enhance, Video Scripting
Health & Mindset	Heart Pulse	Quick Meditation Guide, Rapid Meal Ideas, At-home Workout

These can be edited, disabled, or deleted. New intents can be added at any time.

REST API

All endpoints require authentication via the **Key** header.

METHOD	ENDPOINT	DESCRIPTION
GET	/api/intent	List all intents (sorted by title)
GET	/api/intent/enabled	List enabled intents only (sorted by display order)
GET	/api/intent/{id}	Get a specific intent with its actions
POST	/api/intent	Create a new intent
PUT	/api/intent/{id}	Update an intent
DELETE	/api/intent/{id}	Delete an intent and all its actions

Request / Response Body

```
{
  "id": "intent-productivity",
  "title": "Boost my work productivity",
  "description": "Prompts to help you be more productive at work",
  "icon": "lucide:briefcase",
  "enabled": 1,
  "sortOrder": 0,
  "actions": [
    {
      "id": "action-1",
      "label": "Create daily schedule",
      "prompt": "Help me create an efficient daily schedule...",
      "sortOrder": 0
    },
    {
      "id": "action-2",
      "label": "Draft a difficult email",
      "prompt": "Help me draft a professional email...",
      "sortOrder": 1
    }
  ]
}
```

Example — list enabled intents:

```
curl "http://localhost:2700/api/intent/enabled" \
-H "Key: <YOUR_API_TOKEN>"
```

Example — create an intent:

```
curl -X POST "http://localhost:2700/api/intent" \
  -H "Key: <YOUR_API_TOKEN>" \
  -H "Content-Type: application/json" \
  -d '{
    "title": "Data Analysis",
    "description": "Prompts for analyzing and visualizing data",
    "icon": "lucide:bar-chart-3",
    "enabled": 1,
    "sortOrder": 4,
    "actions": [
      {
        "label": "Summarize a dataset",
        "prompt": "Analyze the following dataset and provide key statistics, trends, and insights.",
        "sortOrder": 0
      }
    ]
  }'
```

Permissions

Intent management uses role-based access control:

PERMISSION	DESCRIPTION
<code>INTENT_VIEW</code>	View intent categories
<code>INTENT_CREATE</code>	Create new intents
<code>INTENT_EDIT</code>	Edit existing intents
<code>INTENT_DELETE</code>	Delete intents

All permissions also require the `ROLE_ADMIN` role.

Related Pages

PAGE	DESCRIPTION
Chat	The chat interface where intents are displayed to users
AI Agents	Configure AI Agents that respond to chat prompts
Administration Guide	User, group, and role management

Chat Analytics

Every conversation your AI just had is a customer telling you what they want, what frustrated them, and whether you delivered. Chat Analytics is how you actually listen.

By the time a thousand customers have talked to your AI Agent, you have a thousand small data points about your product, your messaging, and your funnel. The question isn't whether the data is there — it's whether you can hear it.

Chat Analytics in Turing ES turns finished chat sessions into a queryable, AI-classified, dashboardable stream:

Every session emits a row to the analytics store the moment it ends.

A scheduled **AI-on-AI classifier** then reads each transcript and labels it with **intent**, **goal achievement**, **sentiment**, and **key terms**.

Grafana dashboards show **trends** (goal-rate over time, sentiment drift) and **comparisons** (which agent, which persona is dragging the mean).

A drill-down view in the React console lets you open *one* conversation and read it end to end — the metadata, the AI verdict, the full transcript.

You decide the storage backend with one config line:

```
turing:
  logging:
    engine: mongodb  # or "redis" or "none"
```

none means no analytics. **mongodb** is recommended for production volumes; **redis** is fine for dev or low traffic. Same API, same dashboards, same drill-down view.

Why It Exists

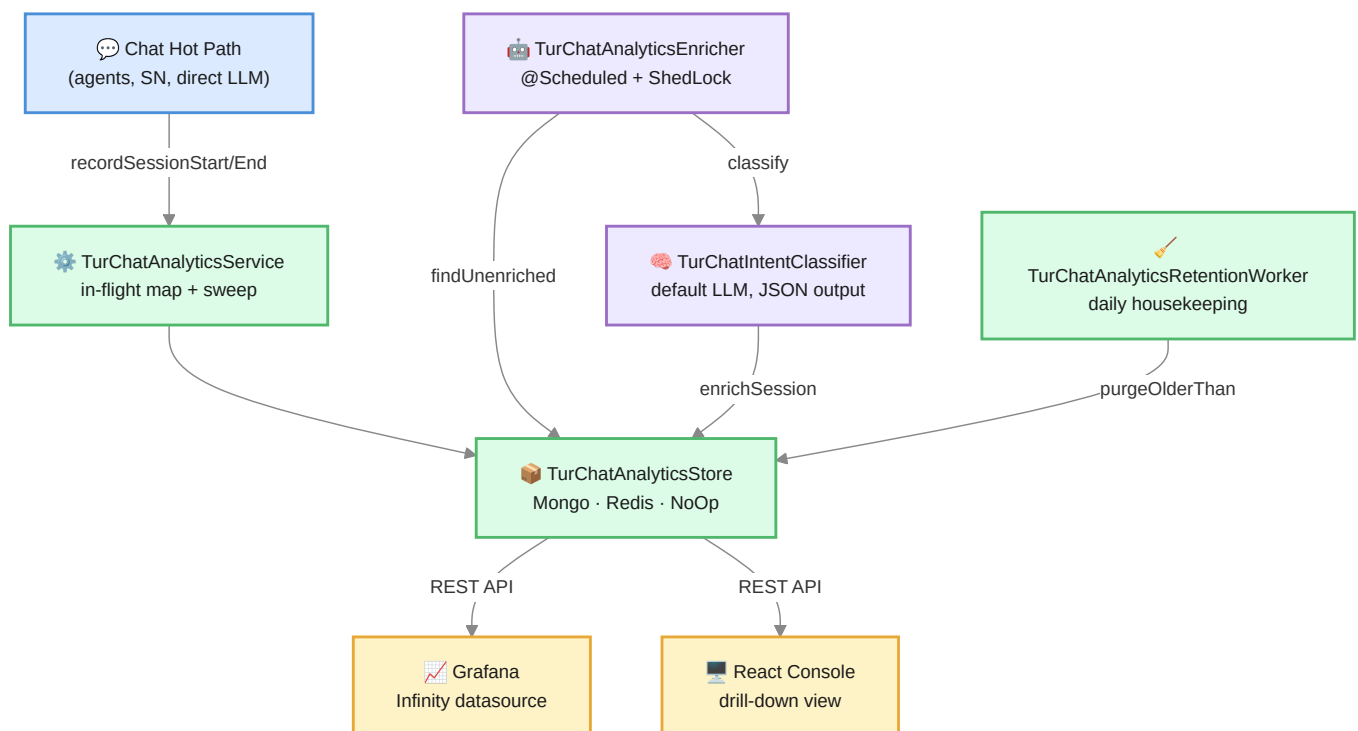
Three problems Chat Analytics solves that vanilla logging doesn't:

Aggregate analytics from semi-structured conversations. A log file tells you "*the request happened*". It doesn't tell you "*the customer asked about pricing, the agent answered well, and the customer left positive.*" Analytics needs structure on top of raw logs.

Decoupled from chat hot path. Recording analytics shouldn't slow down the chat. Turing ES emits session events asynchronously — the store doesn't block a single millisecond on the conversation.

Engine-agnostic dashboards. Mongo and Redis are very different stores. The Grafana dashboards, the REST API, and the React drill-down don't know which one you're using. Swap with a config change.

Architecture



There are five moving pieces:

PIECE	RESPONSIBILITY
<code>TurChatAnalyticsService</code>	Façade in front of the store; in-memory <code>inFlight</code> map of open sessions; periodic sweep prevents leaks
<code>TurChatAnalyticsStore</code>	Engine-agnostic interface implemented by three stores (MongoDB / Redis / NoOp)
<code>TurChatIntentClassifier</code>	The AI-on-AI labeler — calls the default LLM with the transcript, parses JSON, returns intent / goal / sentiment / key terms
<code>TurChatAnalyticsEnricher</code>	<code>@Scheduled</code> worker (every 5 min by default, ShedLock-protected) that picks unclassified terminal sessions and asks the classifier to label them
<code>TurChatAnalyticsRetentionWorker</code>	<code>@Scheduled</code> daily housekeeping — deletes sessions older than <code>retention-days</code> (off by default)

The chat hot path only ever talks to `TurChatAnalyticsService`. Everything else is downstream and asynchronous.

What Gets Recorded

Every chat session produces one row, identified by `conversationId`. The row is upserted twice:

Session start — when the user sends the first message.

Session end — when the conversation reaches a terminal state (the user closed the chat, the conversation timed out, or the executor explicitly ended).

Between start and end, **turns** are accumulated in memory only — they update the row at the end. This avoids a database write per turn.

COLUMN	WHEN SET	WHAT IT MEANS
<code>conversationId</code>	Start	Unique key. Matches the chat-memory <code>conversationId</code> so transcripts can be joined.
<code>agentId</code>	Start	Which AI Agent handled the conversation (or <code>null</code> for direct LLM / SN chat)
<code>personaId</code>	Start	Which persona was attached to the agent
<code>llmInstanceId</code>	Start	Which LLM ran the conversation
<code>embeddingModelId</code>	Start	Embedding model used (for RAG-augmented sessions)
<code>storeInstanceId</code>	Start	Vector store used for retrieval
<code>userId</code>	Start	User identifier when authenticated; <code>null</code> otherwise
<code>locale</code>	Start	Conversation locale
<code>firstUserMessage</code>	Start (write-once)	The opening question — useful as a fallback for classifiers when memory is disabled
<code>startedAt</code> / <code>completedAt</code>	Start / End	Wall-clock timestamps
<code>outcome</code>	End	<code>COMPLETED</code> · <code>ABANDONED</code> · <code>ERROR</code> · <code>IN_PROGRESS</code>
<code>totalTurns</code>	End	How many user-assistant exchanges happened
<code>totalTokensIn</code> / <code>totalTokensOut</code>	End	Aggregate token usage
<code>durationMs</code>	End	End-to-end conversation duration

The next set are populated *later* by the AI-on-AI enricher:

COLUMN	POPULATED BY	RANGE
<code>intentLabel</code>	Enricher	<code>SUPPORT</code> · <code>ONBOARDING</code> · <code>EXPLORATION</code> · <code>COMPLAINT</code> · <code>INFORMATION_SEEKING</code> · <code>CONVERSION_INTENT</code> · <code>OTHER</code> · <code>UNCLASSIFIED</code>
<code>intentConfidence</code>	Enricher	0.0–1.0 — the LLM's self-reported confidence
<code>goalSummary</code>	Enricher	One-sentence summary of what the user was trying to achieve
<code>goalAchieved</code>	Enricher	<code>YES</code> · <code>PARTIAL</code> · <code>NO</code> · <code>UNKNOWN</code>
<code>sentiment</code>	Enricher	<code>POSITIVE</code> · <code>NEUTRAL</code> · <code>NEGATIVE</code> · <code>FRUSTRATED</code> · <code>UNKNOWN</code>
<code>keyTerms</code>	Enricher	Up to 5 — the topic anchors
<code>processedAt</code>	Enricher	Watermark — sessions with this set are skipped on the next cycle

`UNCLASSIFIED` is the *honest* value when the classifier was off, the LLM returned junk JSON, or the transcript was empty. We mark `processedAt` anyway so the session isn't reprocessed forever.

The AI-on-AI Classifier

The classifier is a small service that does one thing: take a transcript, ask the default LLM to label it, parse the response.

The system prompt is fixed and structured:

```
You are an analyst that classifies a chat conversation between a user and an AI agent.
```

```
Read the transcript and infer:
```

- the user's INTENT (one of: SUPPORT, ONBOARDING, EXPLORATION, COMPLAINT, INFORMATION_SEEKING, CONVERSION_INTENT, OTHER),
- their GOAL summarised in one short sentence,
- whether that goal was ACHIEVED by the agent (YES, PARTIAL, NO, UNKNOWN),
- the user's SENTIMENT toward the agent (POSITIVE, NEUTRAL, NEGATIVE, FRUSTRATED, UNKNOWN; FRUSTRATED is reserved for repeated rephrasing or escalating tone),
- up to 5 KEY TERMS that capture the topic.

```
Respond with ONLY valid JSON, no Markdown fences, no commentary.
```

Two design decisions worth understanding:

Tolerant parsing. Some models wrap JSON in ````json` fences despite the instruction; the parser strips them. Missing fields collapse to `UNKNOWN`/empty rather than failing the entire classification. This is the difference between a working pipeline and one that silently drops data.

Default-LLM resolution. The classifier uses whatever LLM is set as the **Default LLM** in [Global Settings](#). One global decision; no per-session override. Picking a fast, cheap model here matters — this runs on every finished session.

The classifier records two telemetry signals on every call (see [Observability](#)):

METRIC	WHY IT MATTERS
<code>turing.chat.analytics.classify</code> (Timer)	Latency per classification — tracks model speed
<code>turing.chat.analytics.classifications</code> (Counter)	Outcome distribution — <code>classified</code> / <code>unclassified</code> / <code>error</code> / <code>skipped</code> / <code>empty_transcript</code> . If <code>error</code> or <code>unclassified</code> rises, you know something's wrong without reading logs.

The LLM call itself flows through the standard `TurLlmObservation`, so the enricher's LLM cost shows up alongside the rest of your chat traffic on the same Grafana panels — same `provider` / `model` tags, same Prometheus series.

The Enricher: Cluster-Safe, Backpressure-Aware

The enricher runs as a `@Scheduled` Spring service, gated by ShedLock so a multi-node deployment doesn't process the same session four times. Defaults:

TUNABLE	DEFAULT	PURPOSE
<code>turing.chat.analytics.enrich.cron</code>	every 5 minutes	How often to look for unenriched sessions
<code>turing.chat.analytics.enrich.batch-size</code>	20	How many to process per cycle. Keep small so a slow LLM doesn't starve the lock.
<code>turing.chat.analytics.enrich.transcript-messages</code>	30	How many of the most recent messages to feed the classifier

Each cycle does:

Skip if the analytics store is disabled, or if the default LLM is offline.

Pull up to `batch-size` sessions where `outcome != IN_PROGRESS && processedAt is missing`.

For each, read the transcript from `TurChatMemoryStore` and call `TurChatIntentClassifier.classify(...)`.

Write the enrichment columns + `processedAt` back to the store.

Log: `processed=N classified=X unclassified=Y elapsedMs=Z`.

The cycle is wrapped in a Micrometer Timer (`turing.chat.analytics.enrich.cycle`) so you can see in Grafana whether it's keeping up or falling behind.



BACKPRESSURE: HOW TO KNOW IF YOU'RE KEEPING UP

If the enricher is *not* keeping up, the cycle's `elapsedMs` will keep growing, **and** the count of unenriched sessions in Mongo/Redis will rise without bound. Two levers: increase `batch-size` (faster catch-up, but each cycle takes longer) or shorten the cron interval. There's no silver bullet — both burn LLM tokens.

Storage Backends

You pick **one** backend per deployment. They expose the same interface; switching between them is a config change.

MongoDB (recommended for production)

```
turing:
  logging:
    engine: mongodb
  mongodb:
    uri: mongodb://localhost:27017
    database: turing_analytics
    collection: chat_sessions
```

What you get:

Server-side aggregation (`$group` , `$dateTrunc`) — fast time-bucketed queries even with millions of sessions.

Secondary indexes auto-created at startup: `{startedAt: -1}` , `{processedAt: 1}` , `{agentId, startedAt}` , `{personaId, startedAt}` , `{outcome}` .

Idempotent upserts via `$set` / `$setOnInsert` — start fields are preserved across the start→end transitions.

Redis (development, low-volume)

```
turing:
  logging:
    engine: redis
  redis:
    uri: redis://localhost:6379
    key-prefix: turing:chat
```

What you get:

One `HASH` per `conversationId` , plus a sorted set indexed by `startedAt` for time-range scans.

All aggregation runs in-process — fine up to a few thousand sessions per day, slow beyond that. Same drill-down API and same dashboards as Mongo.

None (disabled)

```
turing:
  logging:
    engine: none
```

A no-op store is wired in. Calls compile and run, but write nothing and return empty lists. The Chat Analytics React page shows a friendly *"analytics is disabled"* card instead of an empty dashboard.

The In-Flight Map and Why It Has a Sweep

`TurChatAnalyticsService` keeps a `ConcurrentHashMap` of *currently active* sessions. Each entry holds the start timestamp + cumulative turn/token counters. Entries are removed when `recordSessionEnd(...)` fires.

But: what if a worker dies mid-session? What if the front-end disconnects and never sends an end event? Without protection, the map grows monotonically.

So:

A **periodic sweep** runs every hour (`turing.chat.analytics.inflight.sweep-ms`) and removes entries older than `turing.chat.analytics.inflight.max-age-hours` (default 4h — well past any realistic conversation).

A **gauge** (`turing.chat.analytics.inflight.size`) exposes the current count to Prometheus, so a slow leak is visible *before* it becomes memory pressure.

The sweep runs per-process — each node trims its own state. No cluster lock needed.

Retention

Customer transcripts are sensitive. They contain whatever the customer typed — questions, frustrations, sometimes PII. You probably don't want to keep them forever.

The **retention worker** runs daily (`turing.chat.analytics.retention.cron`, default `0 17 3 * *` — 03:17 UTC, off-peak) and deletes sessions older than `turing.chat.analytics.retention-days`.

Default is **0** — disabled. Operators **opt in explicitly**, because "*delete history*" is a one-way action. When you turn it on:

```
turing:
  chat:
    analytics:
      retention-days: 90
```

A daily log line tells you how many were removed:

```
[ChatRetention] removed 142 session(s) older than 2026-01-08T03:17:00Z (90 days)
```

Like the enricher, the retention worker is ShedLock-protected — only one node runs it per cycle.

REST API

All endpoints are read-only and live under `/api/system/chat-analytics`. Designed to be consumed by Grafana via the **Infinity datasource** — engine-agnostic JSON/HTTP.

Health

METHOD	ENDPOINT	RETURNS
GET	/health	{ "enabled": true, "engine": "mongodb" }

Used by Grafana provisioning and the React page to show a friendly disabled banner instead of empty panels.

Listing

METHOD	ENDPOINT	DESCRIPTION
GET	/sessions	Recent sessions in <code>[from, to]</code> . Filters: <code>agentId</code> , <code>personaId</code> , <code>outcome</code> , <code>intentLabel</code> , <code>goalAchieved</code> , <code>sentiment</code> , <code>limit</code>
GET	/aggregate	Counts grouped by a chosen dimension. <code>by</code> : <code>outcome</code> (default), <code>agentId</code> , <code>personaId</code> , <code>intentLabel</code> , <code>goalAchieved</code> , <code>sentiment</code> , <code>locale</code> . <code>limit</code> (default 20)

Trends

METHOD	ENDPOINT	DESCRIPTION
GET	/timeseries	Time-bucketed metric. <code>metric</code> : <code>sessions</code> , <code>goal_achievement_rate</code> , <code>negative_sentiment_rate</code> , <code>avg_duration_ms</code> , <code>avg_tokens_out</code> . <code>interval</code> : <code>hour</code> or <code>day</code>
GET	/scorecard	Per-dimension scorecard. <code>dimension</code> : <code>agentId</code> or <code>personaId</code> . Returns <code>sessions</code> , <code>goalAchievedRate</code> , <code>negativeRate</code> , <code>avgDurationMs</code> , <code>topIntent</code>

`goal_achievement_rate` weights `YES = 1`, `PARTIAL = 0.5`, `NO/UNKNOWN = 0`, divided by enriched sessions in the bucket. `negative_sentiment_rate` is the share of enriched sessions whose sentiment is `NEGATIVE` or `FRUSTRATED`.

Drill-down

METHOD	ENDPOINT	DESCRIPTION
GET	<code>/sessions/{id}</code>	Single session document — 200 with full row, 404 if not found
GET	<code>/sessions/{id}/transcript?limit=200</code>	Joins the session metadata with the full chat-memory transcript: <code>{ session, messages[] }</code>

The drill-down endpoints power the **Chat Analytics** page in the React console — list of sessions on the left, click a row, side panel opens with metadata + AI enrichment + full transcript.

Grafana Dashboard: Turing Chat Insights

Provisioned at `containers/grafana/dashboards/turing-chat-insights.json`. Fifteen panels organized in five rows. Engine-agnostic — calls the REST API via the Infinity datasource, so Mongo and Redis render identically.

Row 1 — At a glance

PANEL	WHAT IT SHOWS
Total chat sessions	Big number for the period
Outcome distribution	Donut: <code>COMPLETED</code> · <code>ABANDONED</code> · <code>ERROR</code> · <code>IN_PROGRESS</code>
Sessions by agent	Bar chart
Analytics store	Engine indicator (<code>mongodb</code> / <code>redis</code> / <code>none</code>)

Row 2 — Demographics

PANEL	WHAT IT SHOWS
Sessions by persona	Which personas are getting traffic
Sessions by locale	Which markets are talking

Row 3 — AI verdicts

PANEL	WHAT IT SHOWS
Sessions by intent	Donut color-coded by intent (SUPPORT red, CONVERSION_INTENT yellow, etc.)
Goal achievement	Horizontal bar — YES green / PARTIAL yellow / NO red
Sentiment distribution	Donut — POSITIVE green / NEGATIVE red / FRUSTRATED purple

Row 4 — Recent sessions

A wide table (100 rows) with every column, including the enrichment ones. Color-coded cells for **outcome**, **intent**, **goalAchieved**, **sentiment**. **intentConfidence** rendered as a percentage.

Row 5 — Trends and scorecards

PANEL	WHAT IT SHOWS
Goal-achievement rate over time	Smoothed line — drops trigger investigation
Negative + frustrated sentiment over time	The spike-detector for "something hurts"
Sessions per hour	Traffic shape
Scorecard by agent	Sessions, goal rate, negative rate, avg duration, top intent — sortable
Scorecard by persona	Same shape, dimension-shifted — separates <i>agent identity</i> from <i>persona configuration</i>

See [Observability](#) for the full Grafana setup, including the Infinity datasource provisioning and how the dashboards are loaded at container start.

The Investigation View (React Console)

`Console → Chat Analytics` is where individual conversations come to life. The page is two halves:

Left: the Recent sessions table with the same color-coded enrichment columns from Grafana — sortable, scrollable, last 100 sessions over the last 7 days.

Right (slide-out): click any row, a Sheet opens with three cards:

Session metadata — start/end timestamps, duration, agent, persona, LLM, locale, user ID, turn count, token totals.

AI enrichment — intent + confidence as badges, goal-achieved badge, sentiment badge, goal summary, key terms as chips. If the session hasn't been classified yet, a *"Awaiting enrichment — the scheduler runs every 5 minutes"* note.

Transcript — every message in role/timestamp/content blocks, scrollable.

The drill-down is pulled from the `/sessions/{id}/transcript` endpoint — one API call per session opened. Joins the analytics store with the chat-memory store on `conversationId`.

💡 FROM INVESTIGATION TO PRODUCT DECISION

The drill-down isn't just for QA — it's a feedback channel into your product. Open the sessions where `goalAchieved == NO` and `sentiment == FRUSTRATED`. Read the first user message. The patterns you'll find — *"users are looking for X but our agent doesn't know about X"* — are your roadmap. Conversely, sessions where `goalAchieved == YES` and `sentiment == POSITIVE` are gold for your **Persona few-shot store**.

How to Read the Data

The dashboard is full of numbers. Here's how to turn them into actions.

Conversion diagnostics

WHAT YOU SEE	WHAT IT LIKELY MEANS	WHAT TO DO
<code>goal_achievement_rate</code> falling week over week	Your prompts or persona drifted out of sync with what users actually need	Drill into recent failed sessions; update agent system prompt or persona examples
<code>negative_sentiment_rate</code> spiking on Mondays	Could be a deploy that broke something Sunday night	Cross-reference with deploys; check <code>outcome=ERROR</code> rate and <code>turing.llm.calls{status="error"}</code>
One agent has a goal rate 30% below the others	Agent-specific issue: bad prompt, wrong tools attached, or wrong LLM	Open the agent's scorecard row; click <i>Sessions by agent</i> → filter; look for failure pattern
One persona scores worse than another for the same agent	The persona's tone/forbidden vocabulary is fighting the agent	Scope the persona tighter (one persona per funnel stage, not one persona for everything)
<code>intentConfidence</code> consistently below 0.5	The classifier LLM is too small for the task, OR your conversations are genuinely ambiguous	Try a stronger default LLM in Global Settings; or accept the ambiguity and act on the trend, not single rows

Cost diagnostics

The enricher itself burns tokens on every session. Watch the same metric panel that tracks chat traffic:

`turing.llm.tokens{provider, model, direction}` — split by direction (`in` / `out`) and by provider.

The enricher's tokens flow through the same series. Compare daily totals; a sudden spike usually means the enricher is processing a backlog.

To make the enricher cheaper:

Pick a fast/cheap default LLM (GPT-4o-mini, Gemini Flash) — the classification task doesn't need a frontier model.

Lower `transcript-messages` — 30 is generous; 15 is often enough to classify intent accurately.

Configuration Reference

```
turing:
  logging:
    engine: mongodb          # "mongodb" | "redis" | "none"
    mongodb:
      uri: mongodb://localhost:27017
      database: turing_analytics
      collection: chat_sessions
    redis:
      uri: redis://localhost:6379
      key-prefix: turing:chat
  chat:
    analytics:
      enrich:
        cron: "0 */5 * * * *"      # every 5 minutes
        batch-size: 20             # sessions per cycle
        transcript-messages: 30     # most-recent messages fed to the
  classifier
    retention-days: 0             # 0 = disabled (default)
    retention:
      cron: "0 17 3 * * *"         # daily 03:17 UTC
    inflight:
      max-age-hours: 4             # eviction threshold for in-flight
  map
    sweep-ms: 3600000             # sweep cadence (1 hour)
```

Related Pages

PAGE	DESCRIPTION
AI Agents	The agents whose conversations are being analyzed
Personas	The voice layer — analytics is how you measure if the persona converts
Chat	The interface that produces the conversations
Observability	Prometheus + Grafana — the full metrics stack
Configuration Reference	All <code>turing.*</code> properties



Token Usage

If you are an administrator responsible for monitoring AI spend, this page is for you. Token Usage helps you track how much each LLM model is costing your organization by showing consumption broken down by model, day, and month — so you can identify heavy usage patterns, set budgets, and avoid unexpected bills.

The **Token Usage** page (</token-usage>) gives administrators a clear view of how many tokens are being consumed by the LLM integrations — broken down by model, day, and month. This is the primary tool for tracking AI costs and identifying heavy usage patterns.

! AVAILABILITY

This page only appears in the sidebar when **at least one LLM instance is enabled**. If no LLM is configured, the Generative AI section will not show Token Usage.

Period Selector

At the top of the page, a month navigator lets you move forward and backward through calendar months. All cards and tables update instantly when the period changes.

◀ January 2025 ▶

Summary Cards

Four cards provide an at-a-glance snapshot of the selected month:

CARD	DESCRIPTION
Total Requests	Number of LLM API calls made during the period
Input Tokens	Total tokens sent to the model (prompts, context, system messages)
Output Tokens	Total tokens received from the model (generated responses)
Total Tokens	Sum of input + output tokens

Values are displayed in a human-readable format — for example `1.2M`, `45K`, or `8.3K` — rather than raw numbers.

Summary by Model

A monthly aggregation table grouping consumption by LLM instance:

COLUMN	DESCRIPTION
Instance	The LLM instance name as configured in Administration
Vendor	Provider (e.g., OpenAI, Anthropic, Google, Azure, Ollama)
Model	Specific model identifier (e.g., <code>gpt-4o</code> , <code>claude-3-5-sonnet</code>)
Requests	Total requests to this instance in the period
Input	Total input tokens for this instance
Output	Total output tokens for this instance
Total	Combined token count for this instance

Use this table to compare token consumption across different providers and models, and to identify which instances account for the largest share of usage.

Daily Breakdown

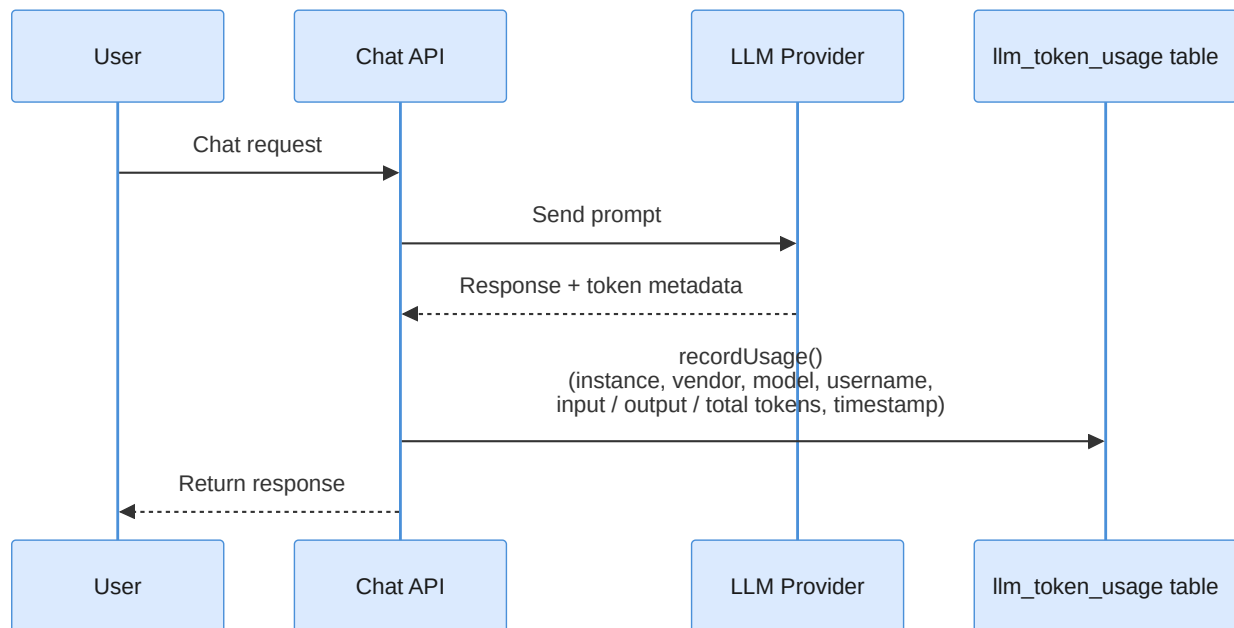
A day-by-day table showing consumption per model across the selected month:

COLUMN	DESCRIPTION
Date	Calendar day
Instance	LLM instance name
Vendor	Provider
Model	Model identifier
Input Tokens	Tokens sent that day
Output Tokens	Tokens received that day
Total Tokens	Combined token count for that day
Requests	Number of requests on that day

Days with no LLM activity do not appear in the table. This breakdown helps spot daily spikes — for example, a batch re-indexing job or a high-traffic event.

How Token Recording Works

Turing ES records token usage automatically on every LLM call — no extra configuration is needed.



What gets recorded on each call:

FIELD	DESCRIPTION
Instance	Which LLM instance handled the request
Vendor	Provider name
Model	Model identifier from the response metadata
Username	The authenticated user who triggered the call
Input Tokens	Extracted from the LLM response metadata
Output Tokens	Extracted from the LLM response metadata
Total Tokens	Input + output
Timestamp	Date and time of the request

i NOTE

Responses where all token counts are zero are **not recorded**. This filters out failed or incomplete LLM calls that would skew usage statistics.

API Endpoint

Token usage data is also available via the REST API. See [REST API Reference → Token Usage API](#) for endpoint details and examples.

Administration Guide

The **Administration** section (`/admin-settings`) manages users, access control, API credentials, global configuration, and system diagnostics. It is accessible from the main sidebar.

! KEYCLOAK MODE

When Keycloak is enabled (`turing.keycloak=true`), the **Users**, **Groups**, and **Roles** subsections are hidden — identity and access management is fully delegated to Keycloak. See [Security & Keycloak](#).

Login

When accessing Turing ES, a login page is displayed. The default username is `admin`. The password is defined at first startup via the `TURING_ADMIN_PASSWORD` environment variable — if not set, Turing ES will not create the admin account with a default password.

Set the environment variable before starting Turing ES for the first time:

Windows

```
set TURING_ADMIN_PASSWORD=your_password
```

Linux / macOS

```
export TURING_ADMIN_PASSWORD=your_password
```

Users

The Users page (`/admin/users`) lists all local user accounts. Click a user to edit, or use the creation button to add a new account.

Account Information:

FIELD	DESCRIPTION
Avatar	Profile picture using DiceBear — click to pick a style, or remove to use initials
Username	Unique login identifier (read-only for existing users)
First Name	User's first name (required)
Last Name	User's last name (required)

Contact:

FIELD	DESCRIPTION
Email	User's email address (required)

Groups & Roles (tabbed view):

TAB	DESCRIPTION
Groups	Search and assign available groups. Remove the user from groups using the remove button.
Roles	Read-only view of roles inherited from the user's assigned groups

Security:

FIELD	DESCRIPTION
Password	Required for new users. Leave blank when editing to keep the current password.

Groups

The Groups page (</admin/groups>) organises users into groups for role-based access control.

Group Details:

FIELD	DESCRIPTION
Name	Unique group identifier (required)
Description	Purpose or scope of this group

Members & Roles (tabbed view):

TAB	DESCRIPTION
Users	Search and add members. Remove users from the group using the remove button.
Roles	Search and assign roles. Remove roles using the remove button.

Roles

The Roles page (</admin/roles>) defines permissions that are assigned to groups.

Role Details:

FIELD	DESCRIPTION
Name	Unique role identifier (required)
Description	What this role permits

Privilege Matrix

Each role has a **privilege matrix** — an interactive table where you toggle individual permissions per resource category. The matrix is organized into two sections:

Generative AI:

CATEGORY	RESOURCES COVERED
LLM	Language model instances
EMBEDDING	Embedding model instances
STORE	Embedding store instances
AI_AGENT	AI Agent configurations
INTENT	Intent categories and actions

Enterprise Search:

CATEGORY	RESOURCES COVERED
SE	Search engine instances and cores
SN	Semantic Navigation sites and fields

Actions per category:

ACTION	DESCRIPTION
VIEW	Read access to the resource
CREATE	Create new resources
EDIT	Modify existing resources
DELETE	Remove resources
ALL	Bulk toggle — enables or disables all four actions at once



TIP

The `ROLE_ADMIN` role has full access to all resources regardless of the privilege matrix. The matrix applies to non-admin roles only.

API Tokens

The API Tokens page (`/admin/tokens`) manages tokens used to authenticate REST API requests. Every token is passed in the `Key` request header.

Form fields:

FIELD	DESCRIPTION
Title	A human-readable name for the token (required)
Description	Purpose or owner of this token

Token display (existing tokens only):

FIELD	DESCRIPTION
API Token	Read-only, monospace display with copy-to-clipboard button

! INFO

The token value is generated automatically on creation and displayed once. It cannot be retrieved again — store it securely.

Using the token in API requests:

```
curl "http://localhost:2700/api/sn/Sample/search?q=cloud&_setlocale=en_US" \  
-H "Key: <YOUR_API_TOKEN>"
```

Global Settings

The Global Settings page (`/admin/settings`) is the central configuration panel for defaults and external service integrations. Divided into four sections.

General

FIELD	DESCRIPTION
Decimal Separator	Choose between period (<code>.</code>) and comma (<code>,</code>) for numeric display. Preview shows formatted examples (e.g., <code>1,500.75</code> vs <code>1.500,75</code>).
Python Executable	Absolute path to the Python 3 binary used by the Code Interpreter tool. When left blank, Turing searches standard OS locations automatically.

LLM Settings

FIELD	DESCRIPTION
Default LLM Instance	Dropdown of enabled LLM instances — the default used when no site-level instance is configured. Select "None" to require explicit per-site assignment.
LLM Cache	Toggle to enable caching of LLM responses
Cache Duration	How long cached responses are retained (duration input with configurable time unit) — visible when cache is enabled
Regenerate Cache	Toggle to force fresh LLM calls instead of serving cached responses — visible when cache is enabled

WARNING

Caching LLM responses improves performance but may return stale answers if the underlying content changes frequently. Tune the duration to match your content update cadence.

RAG Settings

FIELD	DESCRIPTION
Enable RAG Globally	Master switch for Retrieval-Augmented Generation across all sites
Default Embedding Model	Dropdown of enabled embedding models — required when RAG is enabled
Default Embedding Store	Dropdown of enabled embedding stores (ChromaDB, PgVector, or Milvus) — required when RAG is enabled

WARNING

Changing the Default Embedding Model invalidates all existing embeddings. All indexed content must be re-indexed after changing this setting.

NOTE

The RAG Settings section is only visible if an embedding store is configured and available. If MinIO is not configured, the Knowledge Base and related RAG options will not appear.

Email Settings

Used by Turing ES to send notifications and test email connectivity.

FIELD	DESCRIPTION
Provider	Email service provider (currently Brevo)
API Key	API key for the email provider
Sender Email	The From email address
Sender Name	The display name shown to recipients
Recipient Email	Default destination for test and automated emails
Send Test Email	Button to send a test message and verify configuration

System Information

The System Information page (</admin/system-info>) is a diagnostic panel to monitor the health of the Turing ES instance. It contains two tabs.

Overview

Application:

ITEM	DESCRIPTION
Version	Current Turing ES build version
Java Version	JVM version in use
Java Vendor	JVM vendor (e.g., Eclipse Adoptium)
OS	Operating system name, version, and architecture

Database:

ITEM	DESCRIPTION
Status	UP (green) or DOWN (red) — connection health
Product	Database product name (e.g., H2, MariaDB)
Version	Database server version
Driver	JDBC driver name and version
URL	JDBC connection URL

Physical Memory (RAM):

ITEM	DESCRIPTION
RAM Usage	Percentage used with progress bar
Total RAM	Total physical memory installed
Free RAM	Available physical memory
Used RAM	Memory currently in use

Swap / Pagefile (if available):

ITEM	DESCRIPTION
Swap Usage	Percentage used with progress bar
Total Swap	Total swap space configured
Free Swap	Available swap space

Memory (JVM Heap):

ITEM	DESCRIPTION
Heap Usage	Percentage used with progress bar
Total Allocated	JVM heap currently allocated
Free (Allocated)	Free space within the allocated heap
Used Memory	Heap memory currently in use
Max Heap	Maximum heap size configured

Disk Space:

ITEM	DESCRIPTION
Disk Usage	Percentage used with progress bar
Total Space	Total disk capacity
Available Space	Free disk space
Used Space	Disk space currently in use

MongoDB (shown only when `turing.mongodb.enabled: true`):

ITEM	DESCRIPTION
Status	UP (green) or DOWN (red)
Version	MongoDB server version
Endpoint	Connection URI

MinIO (shown only when `turing.minio.enabled: true`):

ITEM	DESCRIPTION
Status	UP (green) or DOWN (red)
Version	MinIO server version
Endpoint	MinIO server URL

System Variables

A searchable table of all JVM system properties active at runtime. The search input filters properties by name or value in real time. A counter shows the number of visible entries out of the total.

Useful for verifying configuration overrides, checking classpath entries, or confirming environment variable values at deployment.

User Account

The User Account page (`/account`) allows the currently logged-in user to manage their own profile without requiring admin privileges.

Profile fields:

FIELD	DESCRIPTION
Avatar	DiceBear avatar picker — choose a style or remove to use initials
First Name	User's first name
Last Name	User's last name
Email	User's email address
Username	Read-only — cannot be changed

Change Password:

FIELD	DESCRIPTION
New Password	Leave blank to keep the current password
Confirm Password	Must match the new password

REST API

All administration endpoints require authentication via the `Key` header. For the full endpoint reference, see [REST API Reference](#).

Users

METHOD	ENDPOINT	DESCRIPTION
GET	/api/v2/user	List all users
GET	/api/v2/user/{username}	Get user by username
GET	/api/v2/user/current	Get the currently authenticated user
POST	/api/v2/user	Create a new user
PUT	/api/v2/user/{username}	Update a user
DELETE	/api/v2/user/{username}	Delete a user
PUT	/api/v2/user/{username}/avatar	Update user avatar
PUT	/api/v2/user/{username}/password	Change user password

Groups

METHOD	ENDPOINT	DESCRIPTION
GET	/api/v2/group	List all groups
GET	/api/v2/group/{id}	Get a group
POST	/api/v2/group	Create a group
PUT	/api/v2/group/{id}	Update a group
DELETE	/api/v2/group/{id}	Delete a group

Roles

METHOD	ENDPOINT	DESCRIPTION
GET	/api/v2/role	List all roles
GET	/api/v2/role/{id}	Get a role
POST	/api/v2/role	Create a role
PUT	/api/v2/role/{id}	Update a role
DELETE	/api/v2/role/{id}	Delete a role
GET	/api/v2/privilege	List all available privileges

API Tokens

METHOD	ENDPOINT	DESCRIPTION
GET	/api/dev/token	List all tokens
GET	/api/dev/token/{id}	Get a token
POST	/api/dev/token	Create a token (auto-generates value)
PUT	/api/dev/token/{id}	Update token metadata
DELETE	/api/dev/token/{id}	Delete a token

Global Settings

METHOD	ENDPOINT	DESCRIPTION
GET	/api/system/global-settings	Get all settings
PUT	/api/system/global-settings	Update settings
POST	/api/system/global-settings/email/test	Send test email

System Information

METHOD	ENDPOINT	DESCRIPTION
GET	/api/system/info	System overview (version, database, memory, disk, external services)
GET	/api/system/info/variables	JVM system properties

Related Pages

PAGE	DESCRIPTION
Security & Authentication	Authentication mechanisms and API token usage
Security & Keycloak	Keycloak OAuth2/OIDC integration
Configuration Reference	Full <code>application.yaml</code> property reference
Logging	Server, indexing, and AEM log viewer
REST API Reference	Complete API endpoint reference

Logging

The **Logging** console (`/admin/logging/instance`) provides centralized visibility into Turing ES operations. It is accessible from the sidebar and requires MongoDB to be enabled.

! MONGODB REQUIRED

The Logging console requires `turing.mongodb.enabled: true`. Logs are stored in MongoDB collections and automatically purged after the configured retention period. See [Configuration Reference](#) for setup details.

Log Categories

The logging landing page displays three cards, each linking to a dedicated log viewer:

CATEGORY	PATH	DESCRIPTION
Server	<code>/admin/logging/instance/server</code>	General application logs — startup, runtime errors, request processing, and framework messages
Indexing	<code>/admin/logging/instance/indexing</code>	Content indexing pipeline events — tracks every document from connector receipt through queue processing to final index commit
AEM	<code>/admin/logging/instance/aem</code>	Adobe Experience Manager connector logs — authentication, content retrieval, and AEM-specific processing

Server Logging

Displays general application logs captured from the Turing ES runtime.

Filters:

FILTER	DESCRIPTION
Level	INFO , WARN , ERROR , DEBUG , TRACE — or All
Date From / Date To	Restrict results to a time window
Search	Free-text search across log messages and stack traces

Result columns:

COLUMN	DESCRIPTION
Date	Full timestamp with relative time (e.g., "2 hours ago")
Node	Cluster node name (useful in multi-node deployments)
Level	Color-coded badge — blue (INFO), yellow (WARN), red (ERROR), purple (DEBUG), slate (TRACE)
Logger	Java class that produced the log entry
Message	Log message text with syntax highlighting

Stack trace viewer: Clicking the alert icon on an error row opens a resizable side panel (60/40 split) displaying the full stack trace with syntax highlighting for Java packages, severity levels, and Spring components.

Indexing Logging

Tracks every document through the indexing pipeline — from connector receipt to final index commit. This is the primary troubleshooting view for content indexing issues.

Filters:

FILTER	DESCRIPTION
Status	Any of the indexing statuses listed below
Result Status	SUCCESS or ERROR
Date From / Date To	Restrict results to a time window
Content ID	Filter by document ID (supports partial match)
URL	Filter by document URL (supports partial match)

Result columns:

COLUMN	DESCRIPTION
Date	Timestamp with relative time
Source	Source system that sent the document
Status	Processing status with icon and color coding (see table below)
Result Status	SUCCESS (green) or ERROR (red)
URL	Document URL (clickable)
Environment	AUTHOR (amber) or PUBLISHING (emerald)
Locale	Language code badge
Sites	Target SN Sites displayed as colored badges

Indexing Status Values

STATUS	ICON	DESCRIPTION
PREPARE_INDEX	Search	Preparing to index the document
PREPARE_UNCHANGED	Clock	No changes detected since last indexing
PREPARE_REINDEX	Refresh	Preparing a reindexation
PREPARE_FORCED_REINDEX	Zap	Forced reindexation triggered
RECEIVED_AND_SENT_TO_TURING	Send	Document received by the connector and forwarded to Turing ES
SENT_TO_QUEUE	Arrow	Document placed in the Artemis processing queue
RECEIVED_FROM_QUEUE	Database	Document consumed from the queue by the indexing pipeline
INDEXED	Check	Document successfully indexed
FINISHED	Check	Operation finished
DEINDEXED	Ban	Document removed from the index
NOT_PROCESSED	Clock	Document skipped
IGNORED	Ban	Document ignored due to an indexing rule or connector decision

AEM Logging

Displays logs specific to the Adobe Experience Manager integration. The filter and column layout is identical to [Server Logging](#) — level, date range, and text search are available.

Use this view to diagnose AEM authentication failures, content retrieval errors, or replication issues.

Common Features

All three log viewers share the following capabilities.

Auto-Refresh

A dropdown in the page header configures automatic data refresh:

INTERVAL	VALUE
Off	No auto-refresh
1 second	Real-time monitoring
5 seconds	Active troubleshooting
10 seconds	Default monitoring
30 seconds	Background monitoring
1 minute	Low-frequency check
5 minutes	Periodic review

The last sync timestamp is displayed next to the refresh indicator.

Pagination

CONTROL	DESCRIPTION
Page navigation	First, Previous, Next, Last page buttons
Rows per page	Configurable: 10, 20, 30, 40, 50, or 100 rows
Sort order	Ascending or descending by date

Syntax Highlighting

Log messages are rendered with syntax highlighting for:

Severity keywords — ERROR/FATAL (red), WARN (yellow), INFO (blue), DEBUG (purple)

Spring components — DispatcherServlet, ContextLoaderListener, Servlet, Filter (indigo)

Java packages and classes — amber italic

URLs and file paths — blue with underline

IP addresses — cyan

Quoted strings — sky blue monospace

REST API

The logging data is available via REST endpoints. Authentication required.

Server Logs

```
GET /api/logging
```

PARAMETER	DEFAULT	DESCRIPTION
page	0	Page number
pageSize	100	Records per page
level		Log level filter (INFO , WARN , ERROR , DEBUG , TRACE)
dateFrom		Start date (yyyy-MM-dd)
dateTo		End date (yyyy-MM-dd)
search		Text search in message or stack trace
sort	asc	Sort order (asc or desc)

Example:

```
curl "http://localhost:2700/api/logging?level=ERROR&pageSize=50&sort=desc" \
-H "Key: <YOUR_API_TOKEN>"
```

Indexing Logs

```
GET /api/logging/indexing
```

PARAMETER	DEFAULT	DESCRIPTION
<code>page</code>	<code>0</code>	Page number
<code>pageSize</code>	<code>100</code>	Records per page
<code>dateFrom</code>		Start date (<code>yyyy-MM-dd</code>)
<code>dateTo</code>		End date (<code>yyyy-MM-dd</code>)
<code>status</code>		Indexing status filter (see status table above)
<code>contentId</code>		Content ID filter (partial match)
<code>resultStatus</code>		<code>SUCCESS</code> or <code>ERROR</code>
<code>url</code>		URL filter (partial match)
<code>sort</code>	<code>asc</code>	Sort order (<code>asc</code> or <code>desc</code>)

Example:

```
curl "http://localhost:2700/api/logging/indexing?
status=INDEXED&resultStatus=ERROR&sort=desc" \
-H "Key: <YOUR_API_TOKEN>"
```

AEM Logs

```
GET /api/logging/aem
```

Same parameters as [Server Logs](#).

Response Structure

All three endpoints return a paginated response:

```
{
  "content": [ ... ],
  "page": 0,
  "pageSize": 100,
  "totalElements": 1250,
  "totalPages": 13
}
```

Configuration

Logging is configured in `application.yaml`. See [Configuration Reference](#) for the full property list.

MongoDB (application logs):

```
turing:
  mongodb:
    enabled: true
    uri: mongodb://localhost:27017
    logging:
      database: turingLog
      collection:
        server: server
        aem: aem
      indexing: indexing
    purge:
      days: 30
```


Local file logging (Logback):

```
logging:
  file:
    name: store/logs/turing.log
  logback:
    rollingpolicy:
      max-file-size: 25MB
      max-history: 10
  level:
    com.viglet: INFO
    org.springframework: INFO
    dev.langchain4j: INFO
```

Related Pages

PAGE	DESCRIPTION
Configuration Reference	Full property reference for MongoDB and Logback settings
Integration	Integration monitoring dashboard (per-connector indexing events)
Architecture Overview	End-to-end indexing flow

Observability — Prometheus & Grafana

You can't fix what you can't see. And in production, "I'll add some logs and check tomorrow" is what tomorrow's incident is built on.

Turing ES ships with a complete observability stack designed to be **flipped on at install time and left alone**:

Spring Boot Actuator exposes a Prometheus endpoint at `/actuator/prometheus`.

Prometheus scrapes it every 5 seconds and stores the time series.

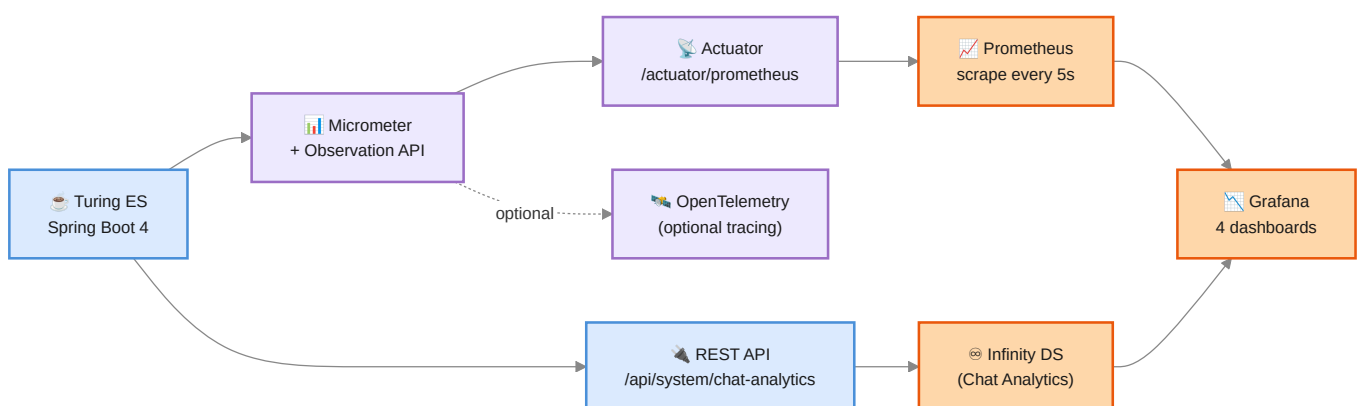
Grafana renders four pre-built dashboards over the metrics, plus the **Chat Analytics** dashboard that reads from the analytics REST API.

Micrometer emits stable metric names so dashboards survive upgrades.

Observation API bridges metrics ↔ tracing — every LLM call, every search, every snapshot lookup is both a timer and a span.

You don't have to write a single PromQL query to get value. The default dashboards cover 90% of operational questions out of the box. The other 10% — when something genuinely odd is happening — is where knowing the metric names pays off.

The Stack at a Glance



COMPONENT	ROLE	WHERE IT RUNS
Spring Boot Actuator	Exposes the metrics endpoint	Inside Turing ES, port 2700
Micrometer	Metrics library — counters, timers, gauges	Embedded in Turing ES
Observation API	Unified abstraction — one observation, two outputs (metric + span)	Embedded in Turing ES
Prometheus	Pull-based scraper, time-series database	<code>containers/prometheus/</code> (port 9090)
Grafana	Visualization, dashboards, alerts	<code>containers/grafana/</code> (port 3000)

Both Prometheus and Grafana are provisioned as Docker images in the `containers/` directory. A `docker compose up` brings them up alongside Turing ES.

! YOU DON'T HAVE TO USE THEM

Observability is **optional**. Turing ES runs perfectly without Prometheus or Grafana — Actuator just exposes the endpoint, and nothing scrapes it. But once you've seen a real incident through these dashboards, you'll wonder how you operated anything without them.

What Gets Measured

Stable metric names are the contract between Turing ES and your dashboards. They're documented in `TurMeterNames.java` and treated as a public API — they don't change between minor versions without a migration note.

LLM Calls

Every call to a language model — chat, streaming, embedding — flows through `TurLlmObservation`. Both the AI Agent loop and the Chat Analytics enricher use it.

METRIC	TYPE	TAGS	WHAT YOU LEARN
<code>turing.llm.calls</code>	Timer	<code>provider</code> , <code>operation</code> , <code>status</code>	Latency per call, error rate per provider
<code>turing.llm.tokens</code>	Counter	<code>provider</code> , <code>model</code> , <code>direction</code>	Token usage by provider/model, split into <code>in</code> and <code>out</code>

`provider` is the LLM vendor (`anthropic`, `openai`, `gemini`, `azure_openai`, `ollama`).
`operation` is `chat`, `stream`, or `embed`. `status` is `success` or `error`.

The same series carries traffic from chat (`/api/v2/llm/.../chat`) **and** from the Chat Analytics enricher. To split them, filter by `model` (the enricher uses whichever model your default LLM is set to).

Search

Every call to a search engine plugin (Solr, Elasticsearch, Lucene) is observed.

METRIC	TYPE	TAGS	WHAT YOU LEARN
<code>turing.search.calls</code>	Timer	<code>engine</code> , <code>operation</code> , <code>status</code>	Engine-side latency and error rate
<code>turing.search.pipeline</code>	Timer	<code>stage</code> , <code>status</code>	Java-side pipeline stages: <code>search.total</code> , <code>snapshot.build</code> , <code>search.response</code> , <code>result.process</code> , <code>widget.build</code>
<code>turing.search.snapshot.cache</code>	Counter	<code>outcome</code> (<code>hit</code> / <code>miss</code> / <code>evict</code>)	Snapshot cache hit ratio — drives latency dramatically

The split between `turing.search.calls` and `turing.search.pipeline` is deliberate: the first is the *backend* engine call (mostly Solr time); the second is everything Turing ES does *around* the call (snapshot lookup, result mapping, facet processing, widget assembly). When latency spikes, you'll know whether to blame the engine or the orchestration.

Chat Analytics

The AI-on-AI pipeline emits its own observability so you can see whether it's keeping up.

METRIC	TYPE	TAGS	WHAT YOU LEARN
<code>turing.chat.analytics.enrich.cycle</code>	Timer	<code>status</code> (<code>success</code> / <code>empty</code> / <code>skipped</code>)	How long each scheduled cycle takes; whether cycles are getting skipped (e.g., LLM offline)
<code>turing.chat.analytics.classify</code>	Timer	<code>outcome</code> (<code>classified</code> / <code>unclassified</code> / <code>error</code> / <code>skipped</code> / <code>empty_transcript</code>)	Per-session classification latency split by outcome
<code>turing.chat.analytics.classifications</code>	Counter	<code>outcome</code>	Cumulative count by outcome — <i>"how many sessions get classified vs. how many fail"</i>
<code>turing.chat.analytics.inflight.size</code>	Gauge	—	Current size of the in-memory in-flight map. Slow growth here is a leak.

If the gauge starts trending up and never comes down, the periodic sweep isn't keeping pace — investigate by looking at agent flows that don't end cleanly.

How to Bring the Stack Up

Local Development

```
cd containers
docker compose up -d prometheus grafana
```

This starts both containers with the provided `prometheus.yml` and Grafana provisioning.

`containers/prometheus/prometheus.yml` points at the host machine:

```
scrape_configs:
  - job_name: turing-es
    metrics_path: /actuator/prometheus
    scheme: http
    static_configs:
      - targets:
          - host.docker.internal:2700
        labels:
          application: turing-es
          instance: turing-dev
```

`host.docker.internal` resolves to the Docker host on Windows and macOS. On Linux, ensure `extra_hosts: ["host.docker.internal:host-gateway"]` is added to the Prometheus service in `docker-compose.yml`.

Once running:

Prometheus UI → `http://localhost:9090`

Grafana → `http://localhost:3000` (default admin/admin on first login)

Production

For production, Prometheus runs as part of your existing observability stack (often Kubernetes-deployed), scraping Turing ES via the same `/actuator/prometheus` endpoint. The dashboards in `containers/grafana/dashboards/` are JSON files — drop them into your existing Grafana instance via *Configuration* → *Dashboards* → *Import JSON*.

The Infinity datasource (used by the Chat Analytics dashboard) reads from Turing ES's REST API directly — no Prometheus needed for that one. It needs:

Allow-list for the URL `http://<turing-host>:2700/api/system/chat-analytics/*`

A datasource named `turing-chat-analytics` so the dashboards find it

Grafana provisioning files in `containers/grafana/provisioning/datasources/` show the expected configuration; copy them into your environment.

The Pre-Built Dashboards

Four dashboards ship in `containers/grafana/dashboards/`. Each tells a different story.

1. `turing-search` — Operational Search Health

The on-call dashboard. If pages are loading slowly, this is where you go.

Search latency p95 / p99 by site

Engine call latency broken down by `engine` (Solr, Elasticsearch, Lucene) and `operation`

Pipeline stage breakdown — `snapshot.build` vs. `search.response` vs. `result.process` vs. `widget.build`

Snapshot cache hit ratio — the single biggest lever on search latency

Error rate — `turing.search.calls{status="error"}`

A high `snapshot.build` time with low `search.response` time means "*Solr is fast, but we're rebuilding a snapshot too often*" — increase TTL or pre-warm.

2. `turing-search-insights` — Product Search Quality

The PM dashboard. Used to understand what users are looking for.

Top search terms over time

Zero-result rate — what users searched for but found nothing

CTR per term (click-through rate from results)

Result-position click distribution — are users clicking position 1, or scrolling to find what they want?

3. `turing-search-term-detail` — Drill-Down

Linked from the term-list dashboard. Pick a term, see the full picture: query volume, click-through rate, top clicked documents, time-of-day distribution.

4. `turing-chat-insights` — Chat Analytics

The full Chat Analytics dashboard. Reads from `/api/system/chat-analytics/**` via the Infinity datasource (engine-agnostic). See [Chat Analytics](#) for the panel-by-panel breakdown.

Useful Queries

"Are we healthy right now?"

```
# Error rate across all LLM providers, last 5 min
sum(rate(turing_llm_calls_seconds_count{status="error"}[5m]))
/ sum(rate(turing_llm_calls_seconds_count[5m]))
```

If this stays below 1%, you're healthy. Spikes usually correlate with provider rate-limiting or transient network issues.

"What's our search p95?"

```
# Search engine call p95, by site
histogram_quantile(0.95,
  sum by (le, engine) (rate(turing_search_calls_seconds_bucket[5m])))
```

For pipeline stages (Java-side):

```
histogram_quantile(0.95,
  sum by (le, stage) (rate(turing_search_pipeline_seconds_bucket[5m])))
```

If the engine number is fast but the pipeline is slow, the bottleneck is in Turing ES's orchestration (often `snapshot.build`).

"What's our cache hit ratio?"

```
rate(turing_search_snapshot_cache_total{outcome="hit"}[5m])
/ on() rate(turing_search_snapshot_cache_total{outcome=~"hit|miss"}[5m])
```

Below 80% means snapshots are evicting too often or TTL is too short.

"Is the chat-analytics enricher keeping up?"

```
# Cycles where status != "success" – the enricher skipped or failed
rate(turing_chat_analytics_enrich_cycle_seconds_count{status!="success"}[15m])
```

If this rises, the enricher is being skipped (LLM offline) or erroring. Cross-reference with `turing.chat.analytics.classifications{outcome="error"}`.

"How much are the AI Agents costing me, per provider?"

```
# Output tokens per second, last 1h, by provider
sum by (provider) (rate(turing_llm_tokens_total{direction="out"}[1h]))
```

Multiply by your provider's per-token cost to get a real-time burn rate. Combined with the `model` tag, you can see which model is dominating cost.

"Is the in-flight map leaking?"

```
# Gauge – should oscillate, not climb
turing_chat_analytics_inflight_size
```

A monotonically rising line = leak. The hourly sweep should keep this stable.

Tracing (Optional)

Micrometer's Observation API produces both metrics **and** spans. If you wire an OpenTelemetry exporter — Jaeger, Tempo, Honeycomb, Datadog — every observation becomes a trace span automatically.

Enable in `application.yaml`:

```
management:
  tracing:
    sampling:
      probability: 1.0
  otlp:
    tracing:
      endpoint: http://localhost:4318/v1/traces
```

Sampling at 1.0 (every request) is fine for low-traffic deployments and during incident debugging. For high-traffic production, drop to 0.05–0.1.

When tracing is on, every LLM call, search, and snapshot lookup gets a span with the same low-cardinality tags as the metrics. You can pivot between *"how slow is this in aggregate"* (Grafana) and *"why was this specific request slow"* (Jaeger) without changing instrumentation.

When to Add Custom Metrics

The shipped metrics cover the common operational questions. You might want to add custom ones when:

- A specific business KPI matters (e.g., conversion rate from a specific agent — you'd derive this from Chat Analytics, not Prometheus).

- A subsystem has its own SLO (e.g., a particular MCP server you depend on).

Pattern to follow (`io.micrometer.core.instrument`):

```
@Service
public class MyService {
    private final MeterRegistry meterRegistry;

    public MyService(@Autowired(required = false) MeterRegistry meterRegistry)
    {
        this.meterRegistry = meterRegistry;
    }

    public void doWork() {
        Timer.Sample sample = meterRegistry == null ? null :
        Timer.start(meterRegistry);
        try {
            // ...
            recordOutcome(sample, "success");
        } catch (Exception e) {
            recordOutcome(sample, "error");
            throw e;
        }
    }

    private void recordOutcome(Timer.Sample sample, String status) {
        if (meterRegistry == null || sample == null) return;
        sample.stop(Timer.builder("my.service.work")
            .tag("status", status)
            .register(meterRegistry));
    }
}
```

The `meterRegistry == null` check is important — it makes the metric optional, so unit tests run without an observation registry.

Add the metric name to `TurMeterNames.java` so it's part of the public observability contract.

Where Logs Fit

Metrics tell you *something is wrong*. Logs tell you *what is wrong*.

Turing ES uses standard SLF4J logging. In production, three options:

APPROACH	WHEN
Standard output → log aggregator (Loki, ELK, Datadog)	Recommended. Your existing log stack already handles this; nothing Turing-specific.
MongoDB Logback appender	When operators need to read logs from the admin console without server access. See Logging .
Both	Common in production. Stdout for the platform team, MongoDB for product/support.

Metrics + logs + traces is the classic three-pillar observability story. Turing ES gives you all three out of the box; the only thing you have to wire is the destination.

Common Operational Patterns

Latency spike, no error spike

Almost always a downstream provider issue. Check `turing.llm.calls{provider, status}` — if `success` count drops and latency rises, the provider is rate-limiting or the network is slow. The chat analytics enricher will also slow because it shares the LLM stack — watch `turing.chat.analytics.classify` for confirmation.

Token cost climbing without traffic climbing

Probably a model upgrade. Confirm by comparing yesterday's `rate(turing_llm_tokens_total{direction="out"}[1d])` to last week's, split by `model`. A new model might be more verbose by default.

Search p99 high, p50 fine

A subset of queries is slow. Check `turing.search.pipeline` p99 vs. p50 by `stage`. If `snapshot.build` is the culprit, a specific site has cold-cached snapshots — pre-warm or raise TTL.

Chat goal-rate dropping but volume rising

This is **the** signal of a quality regression that won't show in basic metrics. You're getting more conversations, but a lower share are achieving their goal. Either:

Marketing pulled in lower-intent traffic (look at the `intentLabel` distribution — `EXPLORATION` rising is a good sign here),

Or your agent recently changed and started failing — drill into Chat Analytics → scorecard by agent → find the dragger.

Mongo growing, sessions not classified

Look at `turing.chat.analytics.classifications{outcome}` over time:

If `error` is rising → something's wrong with the classifier (default LLM offline, prompt too long, etc.).

If `unclassified` is rising → the model's JSON parsing is failing — check `intentConfidence` averages. Consider a stronger default LLM.

If everything's `skipped` → the default LLM in Global Settings is disabled.

Configuration Reference

```
# Spring Boot Actuator – Prometheus endpoint
management:
  endpoints:
    web:
      exposure:
        include: health,info,prometheus
  metrics:
    tags:
      application: turing-es
      instance: ${HOSTNAME}
  tracing:
    sampling:
      probability: 1.0           # 1.0 in dev, 0.05 in production
  otel:
    tracing:
      endpoint: http://localhost:4318/v1/traces # optional – Jaeger / Tempo
/ etc.
```

The Actuator endpoint is exposed unauthenticated by default in dev. In production, secure it (Spring Security, IP allowlist, ingress restriction) or move it to a separate management port.

Related Pages

PAGE	DESCRIPTION
Architecture Overview	Where the observability layer fits in the system
Chat Analytics	The companion analytics for chat conversations
Logging	The third pillar — application logs
Configuration Reference	All <code>management.*</code> and <code>turing.*</code> properties

Developer Guide

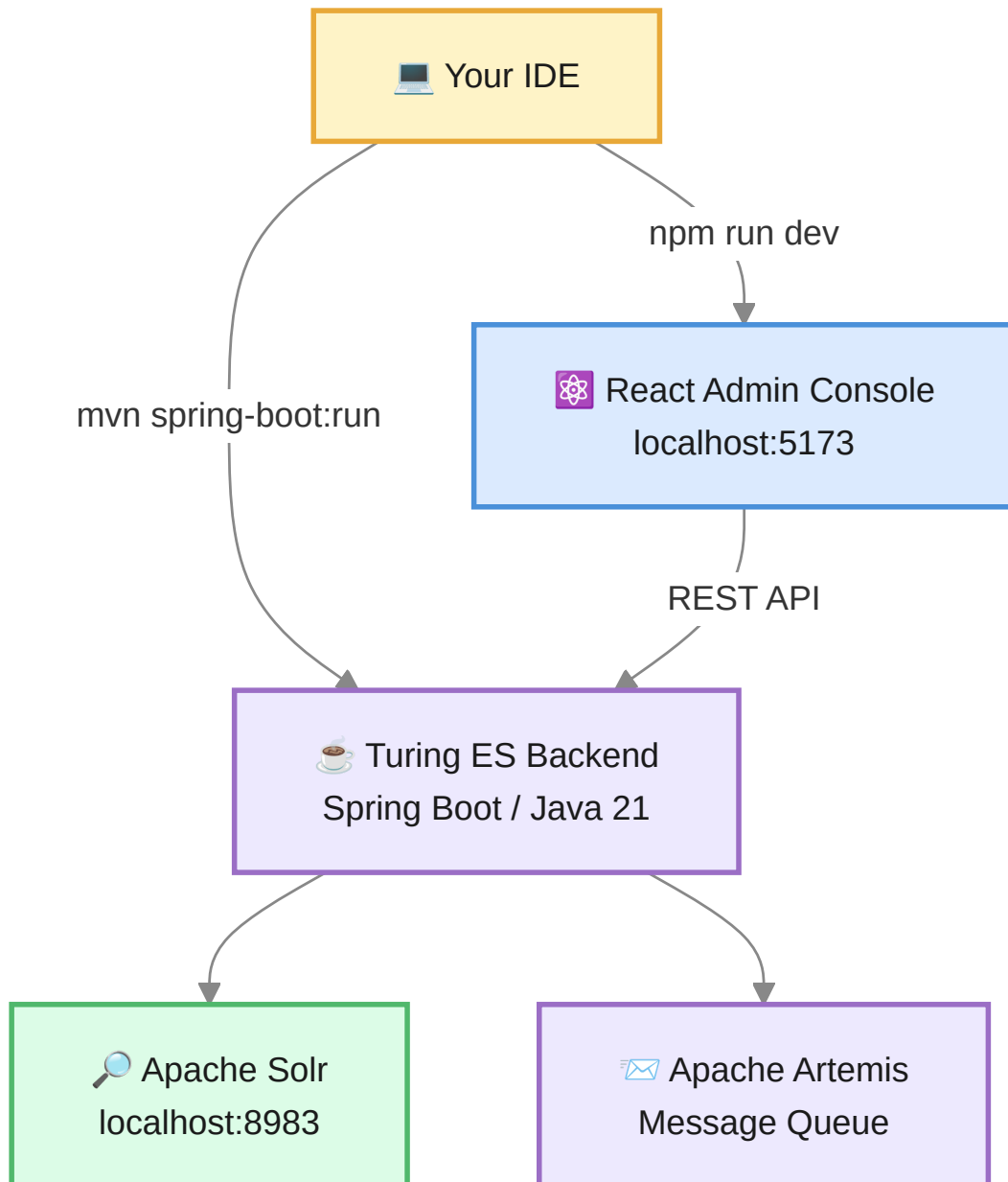
Whether you're **building a search experience** on top of Turing ES or **contributing to the project itself**, this guide has everything you need to get up and running quickly.

Turing ES is a fully open-source enterprise search platform with semantic navigation and GenAI capabilities. The source code lives at github.com/openviglet/turing and all contributions are welcome.

Tech Stack

Understanding the stack helps you navigate the codebase and decide where to plug in.

LAYER	TECHNOLOGY
Backend	Java 21 · Spring Boot · Spring AI
Search Engine	Apache Solr (primary) · Elasticsearch · Lucene
Message Queue	Apache Artemis
Database	H2 (dev) · PostgreSQL / MySQL (prod)
Frontend	React · TypeScript · shadcn/ui · Vite
AI / GenAI	Spring AI · ChromaDB · PgVector · Milvus
Build	Maven (backend) · npm (frontend)
CI/CD	GitHub Actions



Setting Up Your Dev Environment

Prerequisites

Before you begin, make sure you have these installed:

Java 21 (Temurin recommended)

Maven 3.9+

Node.js 20+ and npm

Git

Docker Desktop (for running Solr and other external services locally)

Clone the Repository

```
git clone https://github.com/openviglet/turing-ce.git
cd turing
```

Start Services with Docker Compose

Apache Solr is the only external service required to run Turing ES. Apache Artemis runs **embedded** inside the application and requires no separate installation. The easiest way to start Solr locally is with Docker Compose:

```
docker-compose up -d
```

This starts:

Apache Solr at <http://localhost:8983>



TIP

Wait for Solr to be fully ready before starting the backend. You can check at

<http://localhost:8983/solr/#/>.

Running Turing ES

Backend (Spring Boot)

Run the full backend including the bundled React console:

```
cd turing
mvn spring-boot:run -pl turing-app
```

Or skip the npm build step if you haven't changed the frontend:

```
mvn spring-boot:run -pl turing-app -Dskip.npm
```

The backend starts at `http://localhost:2700`.

Frontend — React Admin Console

For active frontend development, run the React dev server separately. First start the backend in headless mode:

```
mvn spring-boot:run -pl turing-app -Dskip.npm -Dspring-boot.run.profiles=dev-ui
```

Then launch the React app:

```
cd turing/turing-ui  
npm install  
npm run dev
```

The Vite dev server starts at `http://localhost:5173` with hot-reload enabled.

Production Build

```
cd turing  
mvn clean install  
mvn package -pl turing-app
```

The resulting JAR in `turing-app/target/` bundles both the backend and the compiled React assets.

Development URLs

SERVICE	URL	NOTES
Admin Console	<code>http://localhost:2700</code>	Backend-served
React Dev Server	<code>http://localhost:5173</code>	Vite hot-reload
SN Search Sample	<code>http://localhost:2700/sn/Sample</code>	
Swagger UI	<code>http://localhost:2700/swagger-ui.html</code>	Interactive API docs
OpenAPI Spec	<code>http://localhost:2700/v3/api-docs</code>	JSON spec
Solr	<code>http://localhost:8983</code>	Docker Compose

ⓘ DEFAULT CREDENTIALS

On first startup, set the admin password via the environment variable

`TURING_ADMIN_PASSWORD`. See the [Installation Guide](#) for details.

Java SDK

The Turing Java SDK lets you integrate semantic search into any JVM application.

Run the Sample

```
cd turing-java-sdk
mvn package
java -cp build/libs/turing-java-sdk-all.jar
com.viglet.turing.client.sn.sample.TurSNClientSample
```

Add to Your Project via Maven Central

Add to your `pom.xml`:

```
<dependency>  
  <groupId>com.viglet.turing</groupId>  
  <artifactId>turing-java-sdk</artifactId>  
  <version>2026.2.4</version>  
</dependency>
```

The SDK and `turing-commons` are published to [Maven Central](#). No custom repository configuration is needed.

Build the SDK

```
cd turing-java-sdk  
mvn package
```

Database Migrations with Liquibase

Turing ES uses **Liquibase** (v5.0.2) for database schema management. All schema changes — tables, columns, indexes, foreign keys — are tracked as versioned changelogs and applied automatically on startup. This ensures every environment (dev, staging, production) runs the exact same schema, regardless of the database platform.

How it works

When Turing ES starts, Spring Boot runs Liquibase before the application context is fully initialized. Liquibase reads the master changelog, compares it against the **DATABASECHANGELOG** tracking table, and applies any pending changesets. This means:

First startup — all migrations run, creating the full schema from scratch

Upgrades — only new changesets (added since the last startup) are applied

Rollbacks — changesets already applied are skipped (**DATABASECHANGELOG** tracks what has run)

Configuration

Liquibase is configured in **application.yaml**:

```
spring:
  liquibase:
    change-log: classpath:db/changelog/db.changelog-master.yaml
    enabled: true
    default-schema: PUBLIC
```

Changelog structure

All changelogs are in **YAML format** under **turing-app/src/main/resources/db/changelog/**:

```
db/changelog/
├─ db.changelog-master.yaml      ← master file, includes all others
├─ v2025.3_baseline.yaml        ← initial schema (full baseline)
├─ v2026.1_update.yaml          ← major version update
├─ v2026.1_type_enum_fix.yaml   ← type/enum corrections
├─ v2026.1.11.yaml             ← incremental patch
├─ v2026.1.11.1.yaml            ← incremental patch
├─ v2026.1.12.yaml             ← incremental patch
├─ v2026.1.14.0.yaml            ← incremental patch
├─ v2026.1.14.1.yaml            ← ...
├─ ...
└─ v2026.1.14.8.yaml
```

The **master changelog** (`db.changelog-master.yaml`) includes all files in sequential order:

```
databaseChangeLog:
- include:
  file: db/changelog/v2025.3_baseline.yaml
- include:
  file: db/changelog/v2026.1_update.yaml
- include:
  file: db/changelog/v2026.1_type_enum_fix.yaml
- include:
  file: db/changelog/v2026.1.11.yaml
# ... incremental patches follow
```

Naming convention

PATTERN	EXAMPLE	USAGE
<code>v{version}_baseline.yaml</code>	<code>v2025.3_baseline.yaml</code>	Full schema baseline for a major version
<code>v{version}_update.yaml</code>	<code>v2026.1_update.yaml</code>	Generated diff between two versions
<code>v{version}.yaml</code>	<code>v2026.1.12.yaml</code>	Incremental patch for a specific release

Writing a new migration

When adding a new table, column, or index, create a new changelog file following the naming convention and add it to the master changelog.

Example — adding a column to an existing table:

```
databaseChangeLog:
  - changeSet:
      id: v2026.1.15-01-add-priority-to-spotlight
      author: viglet-team
      preConditions:
        - onFail: MARK_RAN
        - tableExists:
            tableName: sn_site_spotlight
        - not:
            - columnExists:
                tableName: sn_site_spotlight
                columnName: priority
      changes:
        - addColumn:
            tableName: sn_site_spotlight
            columns:
              - column:
                  name: priority
                  type: INT
                  defaultValueNumeric: 0
```

Key rules:

Always use preconditions with `onFail: MARK_RAN` to make changesets idempotent — safe to re-run if the change was already applied manually

Changeset IDs use the format `v{version}-{sequence}-{description}` (e.g., `v2026.1.14-01-add-default-label-to-custom-facet`)

Authors use `viglet-team` for manual migrations or `{name} (generated)` for auto-generated diffs

Never modify an already-released changeset — always create a new one

Generating a diff changelog

The Liquibase Maven plugin can auto-generate a diff between two database versions:


```
cd turing-app
mvn liquibase:diff
```

This compares the reference database (current code) against the target database (previous version) and writes a changelog to `src/main/resources/db/changelog/v2026.1_update.yaml`. The diff tracks: tables, columns, indexes, foreign keys, and primary keys.

The plugin configuration is in `turing-app/pom.xml` and uses `liquibase.properties` for connection settings.

Database platform support

Liquibase abstracts SQL differences across platforms. The same changelogs work on all supported databases:

DATABASE	DRIVER	NOTES
H2	<code>org.h2.Driver</code>	Default for development — embedded, zero setup
MariaDB / MySQL	<code>org.mariadb.jdbc.Driver</code>	Production recommended
PostgreSQL	<code>org.postgresql.Driver</code>	Production recommended
Oracle	<code>oracle.jdbc.OracleDriver</code>	Enterprise environments
SQL Server	<code>com.microsoft.sqlserver.jdbc.SQLServerDriver</code>	Enterprise environments

Liquibase handles type mappings (e.g., `VARCHAR(255)` on MySQL vs `VARCHAR2(255)` on Oracle) transparently.

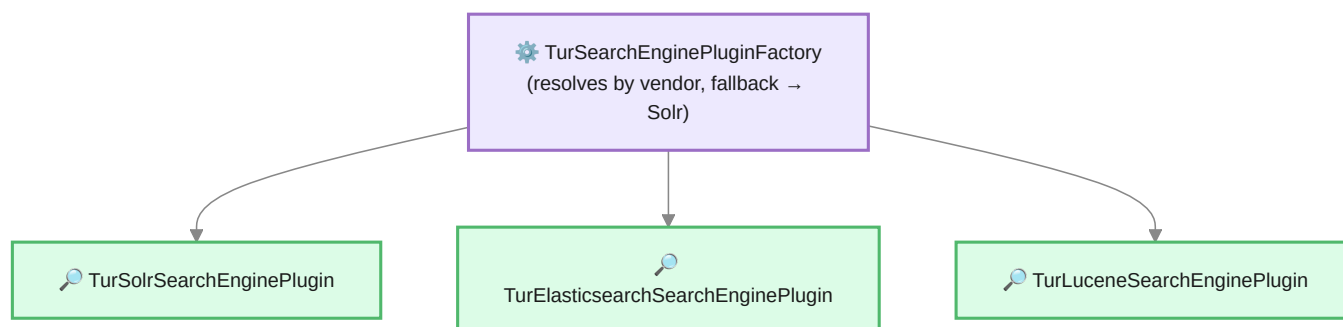
Code Quality

Turing ES maintains high code quality standards. You can check the project health at any time:

TOOL	LINK
SonarCloud	sonarcloud.io/organizations/viglet-turing
GitHub Actions	openviglet/turing/actions
GitHub Security	openviglet/turing/security
Codecov	app.codecov.io/gh/openviglet/turing

Search Engine Plugin Architecture

Turing ES uses a **plugin architecture** to support multiple search backends behind a unified interface. This abstraction allows the same application code to work with Apache Solr, Elasticsearch, or Lucene — the active plugin is resolved at runtime based on the vendor configured per Search Engine instance.



The **TurSearchEnginePlugin** interface

All backend operations are defined in a single interface with four categories:

Search

METHOD	DESCRIPTION
<code>retrieveSearchResults()</code>	Execute a full-text query and return results with facets
<code>retrieveFacetResults()</code>	Return facet counts for a query

Index management

METHOD	DESCRIPTION
<code>createIndex()</code>	Create a new core / index
<code>deleteIndex()</code>	Delete a core / index
<code>clearIndex()</code>	Remove all documents from a core without deleting it
<code>listIndexes()</code>	List all cores / indices on the backend

Schema management

METHOD	DESCRIPTION
<code>addOrUpdateField()</code>	Add or update a field definition in the schema
<code>deleteField()</code>	Remove a field from the schema
<code>fieldExists()</code>	Check whether a field exists in the schema

Documents & monitoring

METHOD	DESCRIPTION
<code>indexDocument()</code>	Index a single document
<code>deIndex()</code>	Remove a document from the index
<code>commit()</code>	Flush pending writes (for backends that require explicit commits)
<code>getDocumentTotal()</code>	Return the total number of indexed documents
<code>getSystemInfo()</code>	Return backend version, OS, JVM, and memory information

Implementing a new backend

To add support for a new search engine:

Create a class implementing `TurSearchEnginePlugin`

Implement all methods in the four categories above

Register the plugin in `TurSearchEnginePluginFactory` with a vendor identifier

The factory resolves the correct plugin based on the `vendor` field of the `TurSEInstance` entity

❗ FALLBACK BEHAVIOUR

If a vendor is unrecognised or unavailable, `TurSearchEnginePluginFactory` falls back to the **Solr** plugin. Apache Solr is the primary supported backend with the most complete feature set.

REST API

Turing ES exposes a REST API for integrating search and AI capabilities into any application. All endpoints use **JSON**. Authentication uses the `Key` header with an API Token created in **Administration → API Tokens**.

For the full endpoint reference — search, autocomplete, spell check, latest searches, GenAI chat, and token usage — see **REST API Reference**.

For interactive exploration, use the built-in Swagger UI at `http://localhost:2700/swagger-ui.html` or the OpenAPI spec at `http://localhost:2700/v3/api-docs`.

Contributing

We'd love your help making Turing ES better. Here's how to get involved:

Fork the `openviglet/turing` repository.

Create a branch for your feature or fix: `git checkout -b feature/my-improvement`

Commit your changes with clear, descriptive messages.

Open a Pull Request — describe what you changed and why.

For larger contributions, open an issue first to discuss the approach before writing code.



TIP

Check the open [GitHub Issues](#) for good first issues tagged with `good first issue` or `help wanted`.

React SDK

The **@viglet/turing-react-sdk** provides headless hooks and unstyled UI components for building custom search experiences on top of the Viglet Turing ES Semantic Navigation API.

Headless architecture — zero built-in styles, full render control via render props

Two modes — manual (programmatic) or URL-synced (URL as source of truth)

TypeScript-first — full type safety for all hooks, components, and API responses

React 18 & 19 compatible

Storybook — Interactive demos and live examples at turing.viglet.org/react-sdk

Installation

```
npm install @viglet/turing-react-sdk
```

Peer dependencies: `react` (^18 or ^19) and `axios` (^1.0).

Quick Start

```
import { TuringProvider, useTuringSearch } from "@viglet/turing-react-sdk";

function App() {
  return (
    <TuringProvider config={{ site: "my-site", locale: "en_US" }}>
      <SearchPage />
    </TuringProvider>
  );
}

function SearchPage() {
  const { documents, searchQuery, status } = useTuringSearch();

  return (
    <div>
      <input
        placeholder="Search..."
        onKeyDown={(e) => {
          if (e.key === "Enter") searchQuery(e.currentTarget.value);
        }}
      />
      {status === "loading" && <p>Loading...</p>}
      {documents.map((doc) => (
        <div key={doc.url}>
          <h3>{doc.title}</h3>
          <p>{doc.description}</p>
        </div>
      ))}
    </div>
  );
}
```

TuringProvider

The root context provider that makes Turing config and search state available to all hooks.

```
import { TuringProvider } from "@viglet/turing-react-sdk";
```

Props

PROP	TYPE	REQUIRED	DESCRIPTION
<code>config</code>	<code>TuringConfig</code>	Yes	Site configuration
<code>urlSync</code>	<code>{ searchParams, setSearchParams }</code>	No	Enable URL synchronization
<code>children</code>	<code>ReactNode</code>	Yes	Child components

TuringConfig

```
{
  site: string;    // Turing site name (required)
  locale?: string; // Default locale (e.g., "en_US")
  sort?: string;   // Default sort (e.g., "relevance")
}
```

URL Sync Mode

When `urlSync` is provided, the URL becomes the single source of truth. Changes to the URL trigger automatic searches, and searches update the URL automatically.

```
import { useSearchParams } from "react-router-dom";

function App() {
  const [searchParams, setSearchParams] = useSearchParams();
  return (
    <TuringProvider
      config={{ site: "my-site", locale: "pt_BR" }}
      urlSync={{ searchParams, setSearchParams }}
    >
      <SearchPage />
    </TuringProvider>
  );
}
```


Hooks

useTuringSearch

Core search hook for **manual (non-URL)** mode. Manages the full lifecycle: query, results, pagination, facets, locale, and sort.

```
const {
  status,           // "idle" | "loading" | "success" | "error"
  data,             // Raw TurSearchResponse
  chat,             // AI chat response (null for wildcard queries)
  documents,        // Resolved documents with mapped default fields
  groups,           // Resolved groups (when group param used)
  error,            // Error message
  params,           // Current search params
  search,           // Execute with given params
  searchQuery,      // New text query (resets page to 1)
  navigate,         // Navigate via Turing href (facet, pagination)
  changeLocale,     // Change locale
  changeSort,       // Change sort
  goToPage,         // Go to specific page
} = useTuringSearch(initialParams?);
```

useTuringAutoComplete

Autocomplete hook that calls the Turing `/ac` endpoint with built-in debouncing.

```
const {
  suggestions, // string[] – completion suggestions
  isLoading,   // Fetching state
  fetch,       // Trigger autocomplete (min 2 chars)
  clear,       // Clear suggestions
} = useTuringAutoComplete(debounceMs?: number); // default: 300
```

useTuringFacets

Provides facet data with toggle/clear actions. Requires `urlSync` mode.

```
const {
  facetGroups, // Groups with toggle/clear actions
  hasSelected, // Any facet selected?
  clearAll,    // Remove all selections
  raw,        // Raw API facet data
} = useTuringFacets();
```

Each `facetGroup` contains:

```
{
  name: string;      // Group identifier
  label: string;     // Localized label
  multivalued: boolean;
  hasSelected: boolean;
  clear: () => void;  // Clear group selections
  facets: [{
    count: number;
    label: string;
    selected: boolean;
    toggle: () => void; // Toggle this facet
  }];
}
```

useTuringPagination

Provides pagination data with navigation helpers. Requires `urlSync` mode.

```
const {
  pages,          // All pages with action helpers
  currentPage,    // Current page number
  pageCount,      // Total pages
  totalCount,     // Total results
  hasPages,       // Multiple pages?
} = useTuringPagination();
```

Each `page` contains:

```

{
  page: number;
  label: string;           // "1", "2", "Next", etc.
  type: string;           // PAGE | CURRENT | FIRST | PREVIOUS | NEXT | LAST |
  ELLIPSIS
  isCurrent: boolean;
  isEllipsis: boolean;
  select: () => void; // Navigate to this page
}

```

useTuringDocument

Typed, normalized access to document custom fields, eliminating `Array.isArray()` checks.

```

const {
  getString, // (field, fallback?) => string
  getArray,  // (field) => string[]
  getNumber, // (field, fallback?) => number
  getBoolean, // (field) => boolean
  getDate,   // (field) => Date | null
  getRaw,    // <T>(field) => T | undefined
  fields,    // Direct field access
} = useTuringDocument(document);

```

useTuringTabs

Manages tab-based search with filter switching and grouping. Keeps URLs clean using implicit params.

```
const {
  activeIndex, // Active tab index
  activeTab,   // Active tab definition
  tabs,        // All tabs with select() action
  isGrouped,   // Active tab uses grouping?
  documents,   // Docs for active tab
  groups,      // Groups (if grouped)
} = useTuringTabs({
  attribute: "templateName",
  items: [
    { label: "All" },
    { label: "Courses", filter: "templateName:cursos", group: "category" },
    { label: "News", filter: "templateName:noticias", rows: 5 },
  ],
});
```

useTuringSortOptions

Fetches available sort options (built-in + custom from Turing admin console).

```
const {
  sortOptions, // { value: string, label: string }[]
  isLoading,
  error,
  refresh,     // Re-fetch options
} = useTuringSortOptions();
```

useTuringSearchHistory

Persists search terms in IndexedDB, scoped per site + domain.

```
const {
  history, // Recent terms (newest first, deduplicated)
  save,    // Save a term
  remove,  // Remove a term
  clear,   // Clear all for current site
} = useTuringSearchHistory(maxItems?: number); // default: 50
```

useTuringUrlSearch

Returns the shared search store from TuringProvider when `urlSync` is configured.

```
const store = useTuringUrlSearch();  
// Same shape as TuringProvider's internal store:  
// status, data, documents, groups, params, inputValue,  
// submitSearch, navigate, setLocale, setSort, showAll, etc.
```

UI Components

All components are **unstyled** (headless). You control the look via render props and CSS.

TuringSearchBar

Simple search bar with customizable input/button via render props.

```
<TuringSearchBar  
  onSearch={(query) => console.log(query)}  
  placeholder="Search..."  
  defaultQuery=""  
  renderInput={(props) => <input {...props} className="my-input" />}  
  renderButton={({ onClick }) => <button onClick={onClick}>Go</button>}  
/>
```

PROP	TYPE	DESCRIPTION
<code>onSearch</code>	<code>(query: string) => void</code>	Callback on Enter or button click
<code>defaultQuery</code>	<code>string</code>	Pre-fill value
<code>placeholder</code>	<code>string</code>	Input placeholder
<code>renderInput</code>	<code>(props) => ReactNode</code>	Custom input renderer
<code>renderButton</code>	<code>(props) => ReactNode</code>	Custom button renderer

TuringSearchField

Advanced compound component with autocomplete, search history, and dropdown.

```
<TuringSearchField turing={urlSearchStore} debounce={300} maxHistory={8}>
  <TuringSearchField.Input placeholder="Search..." className="my-input" />
  <TuringSearchField.Button className="my-
btn">Search</TuringSearchField.Button>
  <TuringSearchField.Dropdown
    className="my-dropdown"
    historyLabel="Recent searches"
    clearAllLabel="Clear all"
    renderSuggestion={({term, onClick}) => (
      <div onClick={onClick}>{term}</div>
    )}
    renderHistory={({term, onClick, onRemove}) => (
      <div>
        <span onClick={onClick}>{term}</span>
        <button onClick={onRemove}>x</button>
      </div>
    )}
  />
</TuringSearchField>
```

TuringResultList

Renders search results using the slot/render-prop pattern.

```
<TuringResultList
  documents={documents}
  isLoading={status === "loading"}
  itemComponent={({ document, index }) => (
    <div key={document.url}>
      <h3>{document.title}</h3>
      <p>{document.description}</p>
    </div>
  )}
  emptyComponent={() => <p>No results found.</p>}
  loadingComponent={() => <p>Loading...</p>}
/>
```

PROP	TYPE	DESCRIPTION
<code>documents</code>	<code>ResolvedDocument[]</code>	Documents to render
<code>itemComponent</code>	<code>(props) => ReactNode</code>	Item renderer (required)
<code>emptyComponent</code>	<code>() => ReactNode</code>	No-results state
<code>loadingComponent</code>	<code>() => ReactNode</code>	Loading state
<code>isLoading</code>	<code>boolean</code>	Show loading component

TuringPagination

Renders pagination items with navigation actions.

```
<TuringPagination
  items={data.pagination}
  onNavigate={(href) => navigate(href)}
  itemComponent={({ item, isCurrent, isEllipsis, onClick, label }) => (
    <button
      onClick={onClick}
      disabled={isCurrent}
      style={{ fontWeight: isCurrent ? "bold" : "normal" }}
    >
      {isEllipsis ? "..." : label}
    </button>
  )}
/>
```

Data Types

ResolvedDocument

Documents returned by hooks have default fields already mapped:

```
{
  url: string;           // Document URL
  title: string;         // Document title
  description: string;   // Document description
  date: string;          // Document date
  image: string;         // Image URL
  text: string;          // Full text
  raw: TurDocument;     // Full document with all custom fields
}
```

SearchParams

```
{
  q: string;             // Query string
  p?: string;            // Page number
  _setlocale?: string;   // Locale
  sort?: string;         // Sort field
  fq?: string[];         // Facet queries (filters)
  group?: string;        // Group field name
  rows?: string;         // Results per page
}
```


Full Example: Search Page with Facets

```
import {
  TuringProvider,
  useTuringUrlSearch,
  useTuringFacets,
  useTuringPagination,
  TuringResultList,
  TuringSearchBar,
  TuringPagination,
} from "@viglet/turing-react-sdk";
import { useSearchParams } from "react-router-dom";

function App() {
  const [searchParams, setSearchParams] = useSearchParams();
  return (
    <TuringProvider
      config={{ site: "my-site", locale: "en_US" }}
      urlSync={{ searchParams, setSearchParams }}
    >
      <SearchPage />
    </TuringProvider>
  );
}

function SearchPage() {
  const store = useTuringUrlSearch();
  const { facetGroups, clearAll } = useTuringFacets();
  const { pages } = useTuringPagination();

  return (
    <div>
      <TuringSearchBar onSearch={(q) => store.searchQuery(q)} />

      <aside>
        {facetGroups.map((group) => (
          <div key={group.name}>
            <h4>{group.label}</h4>
            {group.facets.map((facet) => (
              <label key={facet.label}>
                <input
                  type="checkbox"
                  checked={facet.selected}
                  onChange={facet.toggle}
                />
                {facet.label} ({facet.count})
              </label>
            ))}
          </div>
        ))}
      </aside>
    </div>
  );
}
```

```
        </div>
      )})
    </aside>

    <main>
      <TuringResultList
        documents={store.documents}
        isLoading={store.status === "loading"}
        itemComponent={({ document }) => (
          <article>
            <h3><a href={document.url}>{document.title}</a></h3>
            <p>{document.description}</p>
          </article>
        )}
        emptyComponent={() => <p>No results found.</p>}
      />

      <TuringPagination
        items={store.data?.pagination ?? []}
        onNavigate={store.navigate}
      />
    </main>
  </div>
);
}
```

Links

[GitHub Repository](#)

[npm Package](#)

[Storybook \(Live Demos\)](#)

[REST API Reference](#)

REST API Reference

Turing ES exposes a REST API for integrating search and AI capabilities into any application. All endpoints use **JSON**. Public endpoints (search, chat, autocomplete) require no authentication. Administrative endpoints require a **Key** header.

For authentication details, see [Authentication](#). For the interactive API explorer, visit <http://localhost:2700/swagger-ui.html>.

Authentication

Protected endpoints require an **API Token** in the **Key** request header:

```
Key: <YOUR_API_TOKEN>
```

Create tokens in **Administration** → **API Tokens**. See [Authentication](#) for step-by-step instructions.

Example — authenticated API call:

```
curl "http://localhost:2700/api/sn/Sample/search?q=cloud&_setlocale=en_US" \
-H "Key: <YOUR_API_TOKEN>" \
-H "Accept: application/json"
```

Public Endpoints (no authentication required)

Certain endpoints are publicly accessible, allowing client applications to perform searches and chat without managing sessions:

ENDPOINT	PURPOSE
GET /api/sn/*/search	Semantic Navigation search
GET /api/sn/*/chat	GenAI chat on an SN Site
GET /api/sn/*/ac	Autocomplete
POST /api/genai/chat	Direct GenAI chat
POST /api/ocr/**	OCR text extraction
POST /api/v2/integration/**	External integration endpoints
GET /api/v2/guest/**	Guest access endpoints
POST /graphql	GraphQL queries
GET /api/login	Login endpoint

All other endpoints require authentication — including the full administration API, user management, site configuration, and AI Agent management.

OpenAPI & Swagger

Explore and test every endpoint interactively:

INTERFACE	URL
Swagger UI	http://localhost:2700/swagger-ui.html
OpenAPI 3.0 spec	http://localhost:2700/v3/api-docs

Semantic Navigation API

Search

The core search endpoint. Returns results, facets, spotlights, spell-check suggestions, and pagination.

```
GET http://localhost:2700/api/sn/{siteName}/search
POST http://localhost:2700/api/sn/{siteName}/search
```

Query Parameters (GET):

PARAMETER	REQUIRED	DESCRIPTION
<code>q</code>	✓	Search query
<code>_setlocale</code>	✓	Locale code (e.g., <code>en_US</code> , <code>pt_BR</code>)
<code>p</code>		Page number (default: <code>1</code>)
<code>rows</code>		Results per page (overrides site default)
<code>sort</code>		Sort field and direction (e.g., <code>date desc</code>)
<code>fq[]</code>		Filter query — apply a facet filter (e.g., <code>fq[]=type:news</code>)
<code>group</code>		Group results by a field value

Example (GET):

```
curl "http://localhost:2700/api/sn/Sample/search?
q=enterprise+search&p=1&_setlocale=en_US&rows=10" \
-H "Accept: application/json"
```

Search Response Structure

The search response is a **self-describing navigational JSON**: every link a user might follow — apply a filter, remove a filter, paginate, filter by metadata — is pre-built and ready to use. The front-end does not need to construct query strings or manage filter state.

```
{
  "pagination": [
    { "type": "CURRENT", "text": "1", "href": "...", "page": 1 },
    { "type": "NEXT", "text": "Next", "href": "...", "page": 2 }
  ],
  "queryContext": {
    "count": 42,
    "index": "Sample",
    "limit": 10,
    "offset": 0,
    "page": 1,
    "pageCount": 5,
    "pageEnd": 10,
    "pageStart": 1,
    "responseTime": 23,
    "query": { ... },
    "defaultFields": { ... },
    "facetType": "AND",
    "facetItemType": "AND"
  },
  "results": {
    "document": [
      {
        "source": "TURING",
        "elevate": false,
        "fields": { "title": "...", "url": "...", ... },
        "metadata": [
          { "name": "type", "values": [ { "label": "article", "link": "..." }
        ] }
      ]
    ]
  },
  "widget": {
    "facet": [ ... ],
    "secondaryFacet": [ ... ],
    "facetToRemove": { ... },
    "similar": [ ... ],
    "spellCheck": { ... },
    "spotlights": [ ... ],
    "locales": [ ... ],
    "cleanUpFacets": "...",
    "selectedFilterQueries": [ ... ]
  },
  "groups": [ ... ]
}
```

Top-level sections:

SECTION	DESCRIPTION
pagination	Pre-built page links (<code>FIRST</code> , <code>PREVIOUS</code> , <code>CURRENT</code> , <code>NEXT</code> , <code>LAST</code>) — the front-end follows them directly
queryContext	Result count, page info, response time, active facet operators, and default field mappings
results.document	Array of matched documents. Each contains <code>fields</code> (all indexed values), <code>metadata</code> (facet values with pre-built filter links), <code>source</code> , and <code>elevate</code> flag
widget.facet	Primary facet groups with counts and pre-built filter/clear links per value. Enabled by <code>Facet = true</code>
widget.secondaryFacet	Separate facet groups for fields promoted to Secondary Facet — independent from <code>facet</code> , for different UI treatment (e.g., tabs)
widget.similar	More Like This results. Enabled by <code>MLT = true</code>
widget.spellCheck	Spelling correction with <code>original</code> , <code>corrected</code> text, and <code>usingCorrectedText</code> flag. Enabled by <code>Spell Check = true</code>
widget.spotlights	Curated documents injected at configured positions when the query matches a spotlight term
widget.locales	All configured locales with pre-built links — use as valid values for <code>_setlocale</code>
widget.cleanUpFacets	Pre-built link to clear all active facet filters
groups	Grouped results when the <code>group</code> parameter is used

Self-describing navigation operates at three levels:

Facet navigation — Each facet value carries pre-built links for applying, removing, or clearing filters

Document metadata navigation — Each result's metadata values carry filter links (e.g., click a `type:article` tag to filter all articles)

Pagination — Next/previous/specific page links are included; no offset calculation needed

A Turing ES search UI can be built as a **pure rendering layer** — render results, facets, tags, pagination, and locale switchers directly from the response. Adding a new facet to the SN Site configuration propagates automatically, with no client code changes.

Locale selection:

```
GET /api/sn/{siteName}/search?q=annual+report&_setlocale=pt_BR
```

POST Body Parameters

The POST endpoint accepts the same query parameters plus additional fields in the JSON body:

FIELD	TYPE	DESCRIPTION
<code>userId</code>	<code>string</code>	User identifier (for metrics and latest searches)
<code>query</code>	<code>string</code>	Search query (alternative to <code>q</code> query param)
<code>locale</code>	<code>string</code>	Locale code (alternative to <code>_setlocale</code> query param)
<code>page</code>	<code>integer</code>	Page number
<code>rows</code>	<code>integer</code>	Results per page
<code>sort</code>	<code>string</code>	Sort field and direction
<code>group</code>	<code>string</code>	Group results by field
<code>fq</code>	<code>string[]</code>	Filter queries
<code>fqAnd</code>	<code>string[]</code>	AND filter queries
<code>fqOr</code>	<code>string[]</code>	OR filter queries
<code>fqOperator</code>	<code>string</code>	Filter query operator between facets
<code>fqItemOperator</code>	<code>string</code>	Filter query operator within facet values
<code>fieldList</code>	<code>string[]</code>	Restrict which fields are returned
<code>disableAutoComplete</code>	<code>boolean</code>	Disable autocomplete suggestions (default: <code>false</code>)
<code>populateMetrics</code>	<code>boolean</code>	Enable search metrics recording (default: <code>true</code>)
<code>targetingRules</code>	<code>string[]</code>	Targeting rules — see Targeting Rules

FIELD	TYPE	DESCRIPTION
<code>targetingRulesWithConditions</code>	map	Targeting rules with conditions
<code>targetingRulesWithConditionsAND</code>	map	Targeting rules with AND conditions
<code>targetingRulesWithConditionsOR</code>	map	Targeting rules with OR conditions

Example (POST with targeting rules):

```
curl -X POST "http://localhost:2700/api/sn/Sample/search" \
-H "Content-Type: application/json" \
-d '{
  "query": "benefits",
  "locale": "en_US",
  "rows": 10,
  "targetingRules": ["department:HR", "department:Finance"]
}'
```

For the full Targeting Rules reference — rule types, Solr query generation, fallback clause, and practical examples — see [Semantic Navigation → Targeting Rules](#).

Auto Complete

Returns suggestions for the given prefix. Ideal for search-as-you-type UIs.

```
GET http://localhost:2700/api/sn/{siteName}/ac
```

PARAMETER	REQUIRED	DESCRIPTION
q	✓	Prefix to complete
_setlocale	✓	Locale
rows		Maximum number of suggestions

Example:

```
curl "http://localhost:2700/api/sn/Sample/ac?q=enter&_setlocale=en_US"
```

Response:

```
["enterprise", "enterprise search", "enterprise AI", "entries"]
```

Latest Searches

Returns the most recent search terms for a given user — useful for personalised search history UIs.

```
POST http://localhost:2700/api/sn/{siteName}/search/latest
```

PARAMETER	LOCATION	DESCRIPTION
q	Query	Current query
rows	Query	Max results (default: 5)
_setlocale	Query	Locale
userId	Body (JSON)	User identifier

Example:

```
curl -X POST "http://localhost:2700/api/sn/Sample/search/latest?
q=cloud&rows=5&_setlocale=en_US" \
-H "Key: <API_TOKEN>" \
-H "Content-Type: application/json" \
-d '{ "userId": "user123" }'
```

Response:

```
["cloud computing", "cloud storage", "cloud security"]
```

Search Locales

Lists all configured locales for a Semantic Navigation Site.

```
GET http://localhost:2700/api/sn/{siteName}/search/locales
```

Response:

```
[
  { "locale": "en_US", "link": "/api/sn/Sample/search?_setlocale=en_US" },
  { "locale": "pt_BR", "link": "/api/sn/Sample/search?_setlocale=pt_BR" }
]
```

Spell Check

Corrects a query against the site's indexed vocabulary.

```
GET http://localhost:2700/api/sn/{siteName}/{locale}/spell-check
```

PARAMETER	REQUIRED	DESCRIPTION
 q		Text to check

Example:

```
curl "http://localhost:2700/api/sn/Sample/en_US/spell-check?q=entirprise"
```

GenAI API

STREAMING RESPONSES

Both chat endpoints return **Server-Sent Events (SSE)** streams — content is delivered progressively as the model generates it. Clients should consume the response as an event stream rather than waiting for a single JSON payload.

RAG Chat (SN Site)

Sends a question to the GenAI engine of a Semantic Navigation Site. Returns a streaming SSE response grounded in the site's indexed content.

```
GET http://localhost:2700/api/sn/{siteName}/chat?q={question}
```

Example:

```
curl "http://localhost:2700/api/sn/Sample/chat?q=What+are+the+main+features?" \
  -H "Key: <YOUR_API_TOKEN>"
```

AI Agent Chat

Sends a message to a specific AI Agent. Returns a streaming SSE response.

```
POST http://localhost:2700/api/v2/llm/agent/{agentId}/chat
```

Example:

```
curl -X POST "http://localhost:2700/api/v2/llm/agent/my-agent/chat" \  
-H "Key: <YOUR_API_TOKEN>" \  
-H "Content-Type: application/json" \  
-d '{ "message": "Summarise the latest quarterly report" }'
```

See [Chat](#) for the full chat interface documentation.

Token Usage API

Returns token consumption statistics for a given month.

```
GET http://localhost:2700/api/v2/llm/token-usage?month=YYYY-MM
```

Example:

```
curl "http://localhost:2700/api/v2/llm/token-usage?month=2025-01" \  
-H "Key: <YOUR_API_TOKEN>"
```

Authentication required. See [Token Usage](#) for details.

Integration API

The Integration API provides a **reverse-proxy** endpoint that forwards requests to a configured external integration instance (e.g., an AEM connector or Web Crawler). All HTTP methods are supported. The proxy validates the target host and path to prevent SSRF attacks.

Proxy Endpoint

```
GET|POST|PUT|DELETE  
http://localhost:2700/api/v2/integration/{integrationId}/**
```

The `{integrationId}` maps to an Integration Instance configured in Administration. The remainder of the path (`**`) is forwarded to the instance's endpoint URL.

Example:

```
curl -X POST "http://localhost:2700/api/v2/integration/my-aem-instance/index"  
\br/>-H "Content-Type: application/json" \  
-d '{ "url": "https://example.com/content/page.html" }'
```

This endpoint is **public** (no authentication required).

Integration Instance CRUD (Admin)

Manage integration instances. Authentication required.

METHOD	ENDPOINT	DESCRIPTION
GET	/api/integration	List all integration instances
GET	/api/integration/{id}	Get an integration instance
POST	/api/integration	Create an integration instance
PUT	/api/integration/{id}	Update an integration instance
DELETE	/api/integration/{id}	Delete an integration instance
GET	/api/integration/vendor	List available integration vendors

Assets API

Manage files and the RAG Knowledge Base. All endpoints require authentication.

METHOD	ENDPOINT	DESCRIPTION
GET	/api/asset?prefix=	List files and folders in a prefix (folder)
POST	/api/asset	Upload one or more files (<code>multipart/form-data</code>)
GET	/api/asset/download?objectName=	Download a file (<code>Content-Disposition: attachment</code>)
GET	/api/asset/preview?objectName=	Preview a file inline (<code>Content-Disposition: inline</code>)
GET	/api/asset/metadata?objectName=	Retrieve file metadata (size, type, date)
POST	/api/asset/folder?path=	Create a new folder
DELETE	/api/asset?objectName=	Delete a file or folder
POST	/api/asset/train	Start batch AI Training
GET	/api/asset/train/status	Poll current training status
GET	/api/asset/train/records?objectNames=	Get training timestamps for specific files

After upload, an asynchronous event triggers AI indexing for each file. When a file is deleted, its embeddings are automatically removed from the vector store.

For full details, see [Assets](#).

AI Agent API

Agent Management

Manage AI Agents. Authentication required.

METHOD	ENDPOINT	DESCRIPTION
GET	/api/ai-agent	List all agents (ordered by title)
GET	/api/ai-agent/structure	Get empty structure template for a new agent
GET	/api/ai-agent/{id}	Get a specific agent
POST	/api/ai-agent	Create a new agent
PUT	/api/ai-agent/{id}	Update an existing agent
DELETE	/api/ai-agent/{id}	Delete an agent

Agent Chat

METHOD	ENDPOINT	DESCRIPTION
POST	/api/v2/ai-agent/{agentId}/chat	Stream chat response (SSE). Request body: { <code>llmInstanceId</code> , <code>messages[]</code> }
GET	/api/v2/ai-agent/{agentId}/chat/context-info	Get LLM context window size. Query param: <code>llmInstanceId</code>

Native Tools

METHOD	ENDPOINT	DESCRIPTION
GET	/api/native-tool	List all available tool groups with tool names and descriptions

For full details, see [AI Agents](#).

Administration API

The following endpoints manage SN Sites, fields, and spotlights. All require authentication via the `Key` header. For full details, explore the Swagger UI at <http://localhost:2700/swagger-ui.html>.

SN Site Management

METHOD	ENDPOINT	DESCRIPTION
GET	/api/sn	List all SN Sites
GET	/api/sn/{id}	Get an SN Site
POST	/api/sn	Create an SN Site
PUT	/api/sn/{id}	Update an SN Site
DELETE	/api/sn/{id}	Delete an SN Site (removes index cores)
GET	/api/sn/{id}/monitoring	Site monitoring status (document count, queue size)
GET	/api/sn/{id}/export	Export an SN Site configuration
GET	/api/sn/export	Export all SN Site configurations

Example — list sites:

```
curl "http://localhost:2700/api/sn" \  
  -H "Key: <YOUR_API_TOKEN>"
```

Field Management

METHOD	ENDPOINT	DESCRIPTION
GET	/api/sn/{snSiteId}/field	List fields for a site
GET	/api/sn/{snSiteId}/field/{id}	Get a field
POST	/api/sn/{snSiteId}/field	Create a field
PUT	/api/sn/{snSiteId}/field/{id}	Update a field
DELETE	/api/sn/{snSiteId}/field/{id}	Delete a field

Spotlight Management

METHOD	ENDPOINT	DESCRIPTION
GET	/api/sn/{snSiteId}/spotlight	List spotlights for a site
GET	/api/sn/{snSiteId}/spotlight/{id}	Get a spotlight
POST	/api/sn/{snSiteId}/spotlight	Create a spotlight
PUT	/api/sn/{snSiteId}/spotlight/{id}	Update a spotlight
DELETE	/api/sn/{snSiteId}/spotlight/{id}	Delete a spotlight

Intent Management

METHOD	ENDPOINT	DESCRIPTION
GET	/api/intent	List all intents
GET	/api/intent/enabled	List enabled intents (sorted by display order)
GET	/api/intent/{id}	Get an intent with its actions
POST	/api/intent	Create an intent
PUT	/api/intent/{id}	Update an intent
DELETE	/api/intent/{id}	Delete an intent

For full details, see [Intent](#).

Import / Export

METHOD	ENDPOINT	DESCRIPTION
POST	/api/import	Import SN Site configurations from a ZIP file
POST	/api/sn/import	Import content (JSON job items)
POST	/api/sn/import/zip	Import content from a ZIP with file attachments

For full details, see [Import & Export](#).

Logging

METHOD	ENDPOINT	DESCRIPTION
GET	/api/logging	Server logs (paginated, filterable)
GET	/api/logging/indexing	Indexing pipeline logs
GET	/api/logging/aem	AEM connector logs

For full details, see [Logging](#).

GraphQL API

Turing ES exposes a GraphQL endpoint that provides the same Semantic Navigation search capabilities as the REST API using GraphQL queries. The endpoint is **public** (no authentication required).

```
POST http://localhost:2700/graphql
```

QUERY	DESCRIPTION
<code>siteNames</code>	Returns all configured SN Site names as a string array
<code>siteSearch(siteName, searchParams, locale)</code>	Performs a search against an SN Site

An interactive GraphQL IDE is available at `/graphiql` with schema introspection, auto-completion, and query validation.

Example:

```
curl -X POST "http://localhost:2700/graphql" \
  -H "Content-Type: application/json" \
  -d '{
    "query": "query { siteSearch(siteName: SAMPLE, searchParams: { q:
  \"enterprise search\", rows: 10 }, locale: \"en_US\") { queryContext { count }
  results { document { fields { title url } } } } }"
  }'
```

For the full schema reference, query examples, dynamic fields, filter operators, and response types, see **GraphQL API**.

OCR API

Extracts text from documents via OCR. Two endpoints are available — one for file uploads and one for URLs. Both are **public** (no authentication required).

Extract Text from File

```
POST http://localhost:2700/api/ocr/file
```

Accepts a `multipart/form-data` request with a `file` parameter.

Example:

```
curl -X POST "http://localhost:2700/api/ocr/file" \  
-F "file=@/path/to/document.pdf"
```

Extract Text from URL

```
POST http://localhost:2700/api/ocr/url
```

Accepts a JSON body with a `url` field pointing to a remote document.

Example:

```
curl -X POST "http://localhost:2700/api/ocr/url" \  
-H "Content-Type: application/json" \  
-d '{ "url": "https://example.com/report.pdf" }'
```

Both endpoints return a JSON object containing the extracted text and file metadata.

Related Pages

PAGE	DESCRIPTION
Authentication	How to create and use API Tokens
Semantic Navigation	SN Site configuration and the search response structure
Chat	Front-end chat interface and GenAI API
AI Agents	Agent composition, tool selection, and MCP Servers
Assets	File management and RAG Knowledge Base
Intent	Chat prompt suggestions API
Import & Export	Site configuration and content import
Logging	Application and indexing log API
Token Usage	Token consumption reporting

GraphQL API

Turing ES exposes a GraphQL endpoint that provides the same Semantic Navigation search capabilities as the REST API but using GraphQL queries. The endpoint is **public** — no authentication required.

```
POST http://localhost:2700/graphql
```

GraphiQL Explorer

An interactive GraphQL IDE is available in the admin console and at `/graphiql`. It provides:

Schema introspection — browse all types, queries, and fields

Auto-completion — press Ctrl+Space for field suggestions

Query validation — syntax and type errors are highlighted in real time

Query history — previously executed queries are saved locally

The explorer synchronizes its theme (light/dark) with the Turing ES console.

Queries

siteNames

Returns all configured Semantic Navigation site names as a string array, sorted alphabetically.

```
query {  
  siteNames  
}
```

Response:

```
{
  "data": {
    "siteNames": ["Customer Portal", "Knowledge Base", "Sample"]
  }
}
```

siteSearch

Performs a search against a Semantic Navigation site. This is the primary query — it mirrors the full capability of the REST search endpoint.

```
query {
  siteSearch(
    siteName: TurSNSiteName!
    searchParams: SearchParamsInput!
    locale: String!
  ): SiteSearchResult
}
```

ARGUMENT	TYPE	DESCRIPTION
siteName	TurSNSiteName!	Site name as a dynamic enum value (see Site Name Enum)
searchParams	SearchParamsInput!	Search parameters (see SearchParamsInput)
locale	String!	Locale code (e.g., en_US , pt_BR). Falls back to en if not supported by the site.

Site Name Enum

Site names are exposed as a **dynamic GraphQL enum** (`TurSNSiteName`) that is generated at runtime from the configured sites in the database.

Conversion rules:

SITE NAME	ENUM VALUE	RULE APPLIED
<code>Sample</code>	<code>SAMPLE</code>	Uppercase
<code>My Sample Site</code>	<code>MY_SAMPLE_SITE</code>	Spaces → underscores
<code>Site 2.0!</code>	<code>SITE_2_0_</code>	Special characters → underscores
<code>123 Numbers</code>	<code>SITE_123_NUMBERS</code>	Numeric prefix → <code>SITE_</code> prepended
<code>__internal</code>	<code>SITE___INTERNAL</code>	Double underscore prefix → <code>SITE_</code> prepended

Duplicate handling: If two sites produce the same enum value, the second one gets a `_2` suffix, the third `_3`, and so on.

The enum always includes `UNKNOWN` as the first value. Use the `siteNames` query to discover available site names, then convert to the enum format for `siteSearch`.

SearchParamsInput

FIELD	TYPE	DEFAULT	DESCRIPTION
<code>q</code>	<code>String</code>	<code>*</code>	Search query text
<code>rows</code>	<code>Int</code>	Site default	Results per page (use <code>-1</code> for site default)
<code>p</code>	<code>Int</code>	<code>1</code>	Page number
<code>sort</code>	<code>String</code>	<code>relevanc e</code>	Sort field and direction (e.g., <code>relevance</code> , <code>date desc</code>)
<code>group</code>	<code>String</code>	—	Group results by a field value
<code>nfpr</code>	<code>Int</code>	<code>1</code>	Number of facets per row in grouped results
<code>fq</code>	<code>[String]</code>	—	Filter queries (e.g., <code>["type:article"]</code>)
<code>fqAnd</code>	<code>[String]</code>	—	AND filter queries
<code>fqOr</code>	<code>[String]</code>	—	OR filter queries
<code>fqOp</code>	<code>String</code>	<code>NONE</code>	Filter operator between facets: <code>AND</code> , <code>OR</code> , or <code>NONE</code>
<code>fqiOp</code>	<code>String</code>	<code>NONE</code>	Filter operator within facet values: <code>AND</code> , <code>OR</code> , or <code>NONE</code>
<code>locale</code>	<code>String</code>	—	Override locale (takes precedence over the top-level <code>locale</code> argument)
<code>fl</code>	<code>[String]</code>	—	Field list — restrict which document fields are returned

Filter Operators

The `fqOp` and `fqiOp` parameters control how multiple filter queries are combined:

VALUE	BEHAVIOR	EXAMPLE
NONE	No filter logic applied	Default — filters are passed as-is
AND	All filters must match	<code>fq: ["type:article", "dept:HR"]</code> with <code>fqOp: "AND"</code> → documents must be type:article AND dept:HR
OR	Any filter may match	<code>fqOr: ["type:article", "type:news"]</code> with <code>fqOp: "OR"</code> → documents matching either type

Invalid operator values fall back to `NONE`.

Response Schema

SiteSearchResult

The top-level response object:

FIELD	TYPE	DESCRIPTION
<code>pagination</code>	<code>[SearchPagination]</code>	Pre-built page links
<code>queryContext</code>	<code>SearchQueryContext</code>	Result metadata and statistics
<code>results</code>	<code>SearchResults</code>	Document list with facets
<code>groups</code>	<code>[SearchGroup]</code>	Grouped results (when <code>group</code> parameter is used)
<code>widget</code>	<code>SearchWidget</code>	Widget HTML content

SearchQueryContext

FIELD	TYPE	DESCRIPTION
<code>count</code>	<code>Int</code>	Total number of matching documents
<code>index</code>	<code>String</code>	Site name
<code>limit</code>	<code>Int</code>	Results per page
<code>offset</code>	<code>Int</code>	Result offset
<code>page</code>	<code>Int</code>	Current page number
<code>pageCount</code>	<code>Int</code>	Total number of pages
<code>pageEnd</code>	<code>Int</code>	Last result index on this page
<code>pageStart</code>	<code>Int</code>	First result index on this page
<code>responseTime</code>	<code>Int</code>	Query execution time in milliseconds
<code>query</code>	<code>SearchQuery</code>	Original query parameters (<code>q</code> , <code>sort</code> , <code>p</code> , <code>rows</code> , <code>group</code> , <code>nfpr</code>)
<code>defaultFields</code>	<code>SearchDefaultFields</code>	Default field mappings (<code>title</code> , <code>text</code> , <code>url</code> , <code>date</code> , <code>description</code> , <code>image</code>)
<code>facetType</code>	<code>String</code>	Facet operator type
<code>facetItemType</code>	<code>String</code>	Facet item operator type

SearchResults

FIELD	TYPE	DESCRIPTION
<code>numFound</code>	<code>Int</code>	Total matching documents
<code>start</code>	<code>Int</code>	Result offset
<code>document</code>	<code>[SearchDocument]</code>	Array of matching documents
<code>facet</code>	<code>[SearchFacet]</code>	Facet groups with counts

SearchDocument

FIELD	TYPE	DESCRIPTION
<code>source</code>	<code>String</code>	Source system (e.g., <code>TURING</code>)
<code>fields</code>	<code>SearchDocumentFields</code>	Document field values (see Document Fields)
<code>metadata</code>	<code>[SearchDocumentMetadata]</code>	Key-value metadata pairs with <code>name</code> and <code>value</code>

SearchFacet

FIELD	TYPE	DESCRIPTION
<code>name</code>	<code>String</code>	Facet field name
<code>label</code>	<code>[SearchFacetLabel]</code>	Localized labels (<code>lang</code> , <code>text</code>)
<code>facets</code>	<code>[SearchFacetItem]</code>	Individual facet values

SearchFacetItem

FIELD	TYPE	DESCRIPTION
<code>label</code>	<code>String</code>	Facet value label
<code>count</code>	<code>Int</code>	Number of documents with this value
<code>link</code>	<code>String</code>	Pre-built filter link

SearchPagination

FIELD	TYPE	DESCRIPTION
<code>text</code>	<code>String</code>	Display text (e.g., "1", "Next")
<code>href</code>	<code>String</code>	Pre-built page link
<code>page</code>	<code>Int</code>	Page number
<code>current</code>	<code>Boolean</code>	Whether this is the current page

SearchGroup

FIELD	TYPE	DESCRIPTION
<code>name</code>	<code>String</code>	Group value (e.g., category name)
<code>count</code>	<code>Int</code>	Number of documents in this group
<code>page</code>	<code>Int</code>	Current page within the group
<code>pageCount</code>	<code>Int</code>	Total pages in the group
<code>pageEnd</code>	<code>Int</code>	Last result index
<code>pageStart</code>	<code>Int</code>	First result index
<code>limit</code>	<code>Int</code>	Results per page
<code>results</code>	<code>SearchResults</code>	Documents within this group
<code>pagination</code>	<code>[SearchPagination]</code>	Pagination for this group

SearchWidget

FIELD	TYPE	DESCRIPTION
<code>id</code>	<code>String</code>	Widget identifier
<code>name</code>	<code>String</code>	Widget name
<code>description</code>	<code>String</code>	Widget description
<code>html</code>	<code>String</code>	Rendered HTML content

Document Fields

Static Fields

Every document exposes these standard fields in `SearchDocumentFields`:

FIELD	TYPE	DESCRIPTION
<code>id</code>	<code>String</code>	Document unique identifier
<code>title</code>	<code>String</code>	Document title
<code>text</code>	<code>String</code>	Full text content
<code>url</code>	<code>String</code>	Document URL
<code>date</code>	<code>String</code>	Publication or modification date
<code>description</code>	<code>String</code>	Short description or summary
<code>image</code>	<code>String</code>	Image URL

Dynamic Fields

The GraphQL schema is **dynamically extended** at runtime with custom fields from your SN Site configuration. If your site defines fields like `category`, `author`, or `department`, they appear as additional fields on `SearchDocumentFields` with the `JSON` scalar type.

How it works:

At startup, Turing reads all field extensions from every configured SN Site

Fields with valid GraphQL names (matching `^[_A-Za-z]\w*$`, no `__` prefix) are collected

Static fields (`id`, `title`, `text`, `url`, `date`, `description`, `image`) are excluded to avoid conflicts

The schema is extended:

```
extend type SearchDocumentFields { category: JSON, author: JSON, ... }
```

Dynamic fields use the `JSON` scalar type, which can hold strings, numbers, arrays, or nested objects — matching the flexible nature of indexed document attributes.



FIELD LIST PROJECTION

Use the `fl` parameter in `SearchParamsInput` to restrict which fields are returned. This reduces response size and can improve performance when you only need specific fields.

```
searchParams: { q: "report", fl: ["title", "url", "category"] }
```

Locale Resolution

The locale for a search is resolved with the following precedence:

`searchParams.locale` — if provided in the input, takes highest priority

Top-level `locale` argument — the required `locale` parameter on `siteSearch`

Default — falls back to `en` if neither is provided or the locale is not supported by the site

If the resolved locale is not configured on the target SN Site, the search falls back to the site's first available locale.

Examples

Basic Search

```
query BasicSearch {
  siteSearch(
    siteName: SAMPLE
    searchParams: {
      q: "technology"
      rows: 10
      p: 1
      sort: "relevance"
    }
    locale: "en"
  ) {
    queryContext {
      count
      page
      pageCount
      responseTime
    }
    results {
      numFound
      start
      document {
        fields { title, text, url, date }
        source
      }
    }
    pagination { text, href, page, current }
  }
}
```

Advanced Search with Filters and Facets

```
query AdvancedSearch {  
  siteSearch(  
    siteName: SAMPLE  
    searchParams: {  
      q: "annual report"  
      rows: 10  
      p: 1  
      fq: ["type:article", "department:Finance"]  
      fqOp: "AND"  
      fl: ["title", "url", "date"]  
    }  
    locale: "en_US"  
  ) {  
    queryContext { count, responseTime }  
    results {  
      numFound  
      document {  
        fields { title, url, date }  
        metadata { name, value }  
      }  
      facet {  
        name  
        label { lang, text }  
        facets { label, count, link }  
      }  
    }  
  }  
}
```

Grouped Search

```
query GroupedSearch {
  siteSearch(
    siteName: SAMPLE
    searchParams: {
      q: "*"
      group: "category"
      rows: 5
    }
    locale: "en_US"
  ) {
    groups {
      name
      count
      page
      pageCount
      results {
        document {
          fields { title, url }
        }
      }
      pagination { text, page, current }
    }
  }
}
```

Search with OR Filters

```
query OrFilterSearch {
  siteSearch(
    siteName: SAMPLE
    searchParams: {
      q: "benefits"
      fq0r: ["type:article", "type:news"]
      fq0p: "OR"
    }
    locale: "en_US"
  ) {
    queryContext { count }
    results {
      document {
        fields { title, url }
      }
    }
  }
}
```

Minimal Search

```
query SimpleSearch {
  siteSearch(
    siteName: SAMPLE
    searchParams: { q: "search" }
    locale: "en"
  ) {
    results {
      document {
        fields { title, url }
      }
    }
  }
}
```

cURL Example

```
curl -X POST "http://localhost:2700/graphql" \  
  -H "Content-Type: application/json" \  
  -d '{  
    "query": "query { siteSearch(siteName: SAMPLE, searchParams: { q:  
\"enterprise search\", rows: 10 }, locale: \"en_US\") { queryContext { count }  
results { document { fields { title url } } } } }\"  
  }'
```

Configuration

GraphQL is configured in `application.yaml`:

```
spring:  
  graphql:  
    graphiql:  
      enabled: true  
      path: /graphiql  
    http.path: /graphql  
  cors:  
    allowed-origins: ${turing.allowedOrigins}  
    allowed-methods: GET,POST  
    allowed-headers: "*"'
```

PROPERTY	DEFAULT	DESCRIPTION
<code>spring.graphql.http.path</code>	<code>/graphql</code>	GraphQL endpoint path
<code>spring.graphql.graphiql.enabled</code>	<code>true</code>	Enable GraphiQL browser IDE
<code>spring.graphql.graphiql.path</code>	<code>/graphiql</code>	GraphiQL IDE path
<code>spring.graphql.cors.allowed-origins</code>	<code>\${turing.allowedOrigins}</code>	CORS allowed origins
<code>spring.graphql.cors.allowed-methods</code>	<code>GET,POST</code>	Allowed HTTP methods

For the full configuration reference, see [Configuration Reference → GraphQL](#).

Error Handling

SCENARIO	BEHAVIOR
Invalid site name	Returns <code>null</code> for <code>siteSearch</code> — the site is not found
Unsupported locale	Falls back to the site's first available locale
Invalid filter operator	Falls back to <code>NONE</code>
Invalid field in <code>fl</code>	Field is ignored — only valid fields are returned
Empty query (<code>q: ""</code>)	Treated as <code>q: "*" (match all)</code>

GraphQL validation errors (missing required fields, wrong types) are returned as standard GraphQL error responses before the query reaches the resolver.

Related Pages

PAGE	DESCRIPTION
REST API Reference	REST search endpoint with the same capabilities
Semantic Navigation	SN Site configuration — fields, facets, and search behavior
Configuration Reference	Full <code>application.yaml</code> property reference

Authentication

Turing ES supports three authentication modes:

MODE	BEST FOR	EXTERNAL DEPENDENCY
Native (default)	Development, simple setups	None
Social Login (OAuth2)	Small teams, GitHub/Google/Microsoft accounts	Provider OAuth App
Keycloak SSO	Enterprise production with LDAP/AD, MFA, RBAC	Keycloak server

This page covers **native authentication**. For social login with Google, GitHub, or Microsoft, see [Social Login \(OAuth2\)](#). For the full Keycloak production setup, see [Security & Keycloak](#).

Native Authentication (default)

When Keycloak is not enabled, Turing ES uses its own user store:

Admin Console

Users log in via a form at <http://localhost:2700/login> and receive a **Java HTTP session**. Sessions are maintained server-side and expire after inactivity. No special configuration is required — this mode works out of the box.

Users, groups, and roles are managed in **Administration → Users, Groups, and Roles**.

REST API — API Key

All REST API requests authenticate via an **API Key** passed in the [Key](#) request header. API Tokens are not tied to user sessions — they are created in **Administration → API Tokens** and remain valid until explicitly deleted.

Creating an API Token:

Sign in to the Administration Console.

Navigate to **Administration → API Tokens**.

Click **New**, fill in a name and description.

Copy the generated token immediately — it will not be shown again.

For the full endpoint reference, authentication examples, and the list of public endpoints that require no authentication, see [REST API Reference → Authentication](#).

TOKENS ARE ENCRYPTED AT REST

API Token values are encrypted at rest in the database. The encryption key is set via `turing.ai.crypto.key` in `application.yaml`. Change this key in production — see [Configuration Reference](#).

Switching to Social Login (OAuth2)

For teams using Google, GitHub, or Microsoft, you can enable social login buttons by adding the provider configuration in `application.yaml`:

```
spring:
  security:
    oauth2:
      client:
        registration:
          github:
            client-id: YOUR_CLIENT_ID
            client-secret: YOUR_CLIENT_SECRET
```

The login page automatically shows buttons only for configured providers. See [Social Login \(OAuth2\)](#) for complete setup instructions for each provider.

Switching to Keycloak SSO

For production environments with SSO requirements, Turing ES can delegate authentication to Keycloak via OAuth2 / OpenID Connect. Enable it with:

```
-Dturing.keycloak=true
```

See [Security & Keycloak](#) for the full 6-step production setup — database, Keycloak installation, realm configuration, SSL certificates, JVM properties, and Apache reverse proxy.

Related Pages

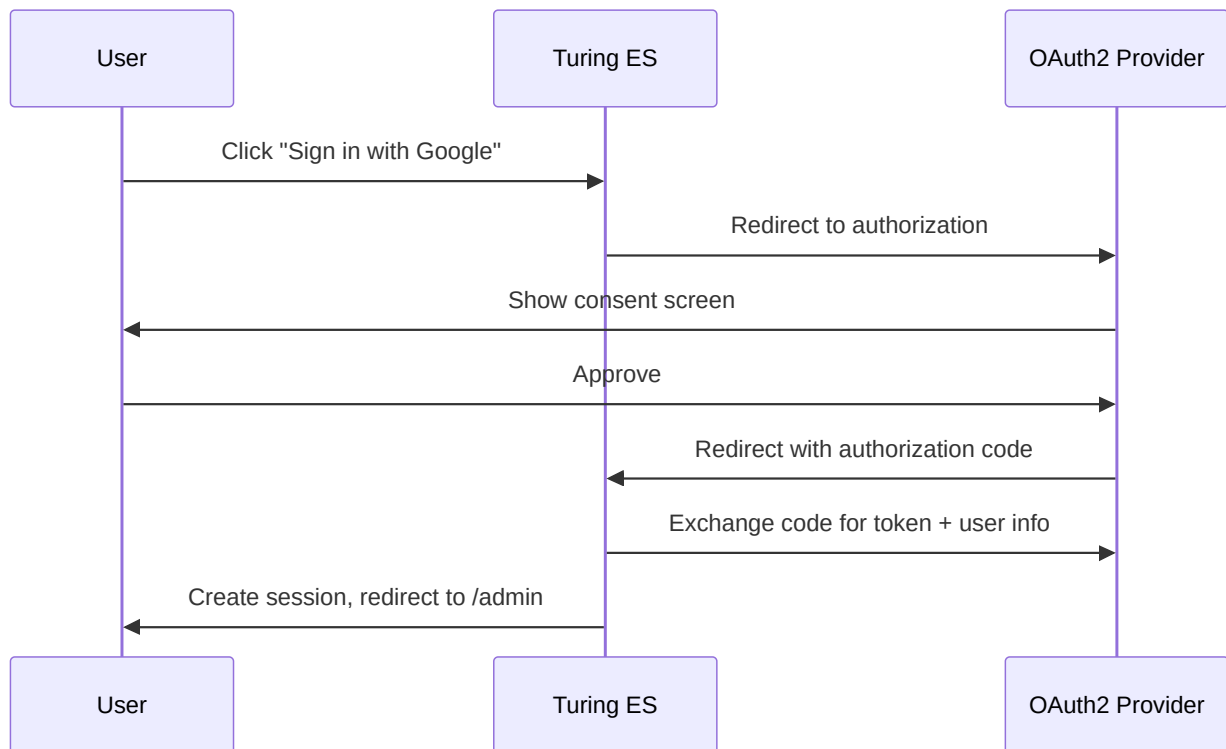
PAGE	DESCRIPTION
Social Login (OAuth2)	Google, GitHub, and Microsoft social login
Security & Keycloak	Full production setup with Keycloak OAuth2/OIDC
Administration Guide	Managing users, groups, roles, and API tokens
REST API	Full REST API reference including authentication examples
Configuration Reference	<code>turing.ai.crypto.key</code> and other security-related settings

Social Login (OAuth2)

Turing ES supports **social login** via OAuth2 with Google, GitHub, and Microsoft. When configured, users can sign in with their existing accounts from these providers — no separate password required.

This is ideal for development environments, small teams, and scenarios where a full Keycloak deployment is not needed. For enterprise SSO with LDAP/AD federation, MFA, and centralized user management, see [Security & Keycloak](#).

How It Works



When a user signs in via a social provider for the first time, Turing ES **automatically creates a local user** with the profile information from the provider (name, email, avatar). Subsequent logins update the profile data.

Prerequisites

REQUIREMENT	DETAILS
<code>turing.authentication.thirdparty</code>	Must be <code>true</code> (default)
Provider OAuth App	Create an OAuth App/Client in the provider's console
Redirect URI	<code>http://<host>: <port>/login/oauth2/code/<provider></code>

Configuration

Add the provider registration under `spring.security.oauth2.client` in your `application.yaml`:

Google

Go to [Google Cloud Console](#)

Create an **OAuth 2.0 Client ID** (Web application)

Add the authorized redirect URI: `http://localhost:2700/login/oauth2/code/google`

```
spring:
  security:
    oauth2:
      client:
        registration:
          google:
            client-id: YOUR_GOOGLE_CLIENT_ID
            client-secret: YOUR_GOOGLE_CLIENT_SECRET
```

💡 SCOPES

Spring Security auto-configures the Google provider with `openid`, `profile`, and `email` scopes — no need to set them manually.

GitHub

Go to [GitHub Developer Settings > OAuth Apps](#)

Click **New OAuth App**

Set **Homepage URL** to `http://localhost:2700`

Set **Authorization callback URL** to `http://localhost:2700/login/oauth2/code/github`

```
spring:
  security:
    oauth2:
      client:
        registration:
          github:
            client-id: YOUR_GITHUB_CLIENT_ID
            client-secret: YOUR_GITHUB_CLIENT_SECRET
```

GITHUB SPECIFICS

GitHub does not provide a separate first/last name — Turing splits the `name` attribute automatically.

Email may be `null` if the user's GitHub email is private. The email field is hidden in the profile page when empty.

The username is the GitHub `login` handle.

Microsoft (Entra ID / Azure AD)

Go to [Azure Portal > App registrations](#)

Register a new application

Under **Authentication**, add a **Web** redirect URI:

`http://localhost:2700/login/oauth2/code/microsoft`

Under **Certificates & secrets**, create a new client secret

```
spring:
  security:
    oauth2:
      client:
        registration:
          microsoft:
            client-id: YOUR_MICROSOFT_CLIENT_ID
            client-secret: YOUR_MICROSOFT_CLIENT_SECRET
            scope: openid,profile,email
            authorization-grant-type: authorization_code
            redirect-uri: "{baseUrl}/login/oauth2/code/{registrationId}"
        provider:
          microsoft:
            issuer-uri: https://login.microsoftonline.com/common/v2.0
```

Multiple Providers

You can configure any combination of providers simultaneously. The login page **automatically shows only the buttons for configured providers**:

```
spring:
  security:
    oauth2:
      client:
        registration:
          google:
            client-id: ...
            client-secret: ...
          github:
            client-id: ...
            client-secret: ...
          microsoft:
            client-id: ...
            client-secret: ...
        provider:
          microsoft:
            issuer-uri: https://login.microsoftonline.com/common/v2.0
```

If only GitHub is configured, only the GitHub button appears. If none are configured, the social login section is hidden entirely.

Redirect URIs Reference

PROVIDER	REDIRECT URI
Google	<code>http://<host>:<port>/login/oauth2/code/google</code>
GitHub	<code>http://<host>:<port>/login/oauth2/code/github</code>
Microsoft	<code>http://<host>:<port>/login/oauth2/code/microsoft</code>

The pattern is always: `{baseUrl}/login/oauth2/code/{registrationId}`

For production with HTTPS: `https://turing.example.com/login/oauth2/code/google`

User Profile Behavior

When logged in via a social provider:

Profile fields are read-only — name, email, and username are managed by the provider

Avatar can still be customized in Turing

Password section is hidden — authentication is handled by the provider

The provider name is displayed on the profile page (e.g., "github", "google")

Configuration Properties

PROPERTY	DEFAULT	DESCRIPTION
<code>turing.authentication.thirdparty</code>	<code>true</code>	Enable third-party OAuth2 login buttons
<code>spring.security.oauth2.client.registration.<provider>.client-id</code>	—	OAuth2 Client ID from the provider
<code>spring.security.oauth2.client.registration.<provider>.client-secret</code>	—	OAuth2 Client Secret from the provider

Troubleshooting

404 on provider authorization page

The Client ID is likely incorrect. Double-check the value in `application.yaml` matches the one in the provider's console.

User profile shows empty name

The provider attributes may use different field names. Turing handles Google (`given_name`, `family_name`), GitHub (`login`, `name`), and Microsoft (`given_name`, `family_name`) automatically. For other providers, the `name` attribute is split into first/last name.

Email is empty (GitHub)

GitHub may not expose the user's email if it is set to private. This is by design — the email field is hidden in the profile page when empty.

Login redirects to wrong page

The default success URL is `/admin`. If using a different frontend path, ensure the Spring Security configuration matches.

Related Pages

PAGE	DESCRIPTION
Authentication	Native authentication and API Keys
Security & Keycloak	Full Keycloak production setup with SSO
Configuration Reference	All configuration properties

Security & Keycloak

This page covers the full **Keycloak OAuth2 / OpenID Connect** production setup for Turing ES, enabling SSO integration with corporate identity providers and centralized user management.

For day-to-day authentication — native session-based login and REST API Key usage — see **Authentication**.

Why Use Keycloak with Turing ES?

The primary reason to integrate Keycloak is to connect Turing ES to your organization's **existing Single Sign-On (SSO) infrastructure**. Instead of managing a separate set of credentials just for Turing ES, users sign in with the same corporate identity they already use for email, cloud tools, and internal applications — no additional password, no extra login step.

Keycloak acts as an **identity broker** between Turing ES and your identity provider. Turing ES never sees or stores user passwords — it receives a signed JWT token from Keycloak confirming who the user is and what roles they have.

Supported Identity Providers

Because Keycloak supports the **SAML 2.0**, **OAuth 2.0**, and **OpenID Connect** standards, it can federate identity from virtually any enterprise identity provider:

IDENTITY PROVIDER	PROTOCOL	USE CASE
Microsoft Entra ID (Azure AD)	SAML 2.0 / OIDC	Microsoft 365, Azure, corporate Windows environments
Google Workspace	OIDC	Organizations on Google Workspace / Gmail
Okta	SAML 2.0 / OIDC	Multi-cloud enterprises, workforce identity
Auth0	OIDC	Developer-focused platforms, B2C applications
Ping Identity	SAML 2.0 / OIDC	Large enterprises, government, regulated industries
OneLogin	SAML 2.0 / OIDC	HR-integrated identity management
ADFS (Active Directory Federation Services)	SAML 2.0	On-premise Windows Server / Active Directory
LDAP / Active Directory	LDAP	Direct directory sync without SAML — Keycloak federates LDAP users natively
GitHub / GitLab	OAuth 2.0	Developer-focused organizations
Any SAML 2.0 IdP	SAML 2.0	Shibboleth, SimpleSAMLphp, custom SAML providers
Any OIDC provider	OIDC	Any standards-compliant OpenID Connect provider

What This Enables

Single Sign-On (SSO): Users click "Sign in" on Turing ES, are redirected to their corporate login (e.g., Microsoft 365 login page), authenticate once, and are immediately signed in to Turing ES. If they're already signed in to another corporate application, they skip the login form entirely — the SSO session is shared.

Single Sign-Out: Logging out of Turing ES triggers a Keycloak OIDC logout that propagates to the corporate IdP. The user is signed out of Turing ES and optionally out of all applications sharing the same SSO session.

Centralized User Management: User accounts, groups, and roles are managed in Keycloak (or synchronized from the corporate directory). No need to create or maintain users in Turing ES manually — Keycloak provides the identity, roles, and group memberships via JWT claims.

Multi-Factor Authentication (MFA): Keycloak supports TOTP (Google Authenticator, Microsoft Authenticator), WebAuthn/FIDO2 (hardware security keys, biometrics), SMS OTP, and email verification. MFA policies are configured in Keycloak and enforced transparently — Turing ES requires no changes.

Role-Based Access Control (RBAC): Keycloak roles and realm roles are mapped to Turing ES permissions. Define roles like `turing-admin`, `turing-editor`, `turing-viewer` in Keycloak, assign them to users or groups, and Turing ES enforces access accordingly.

Social Login: For public-facing or partner-facing deployments, Keycloak can enable login via Google, GitHub, Facebook, Apple, Twitter, and other social providers alongside or instead of corporate identity.

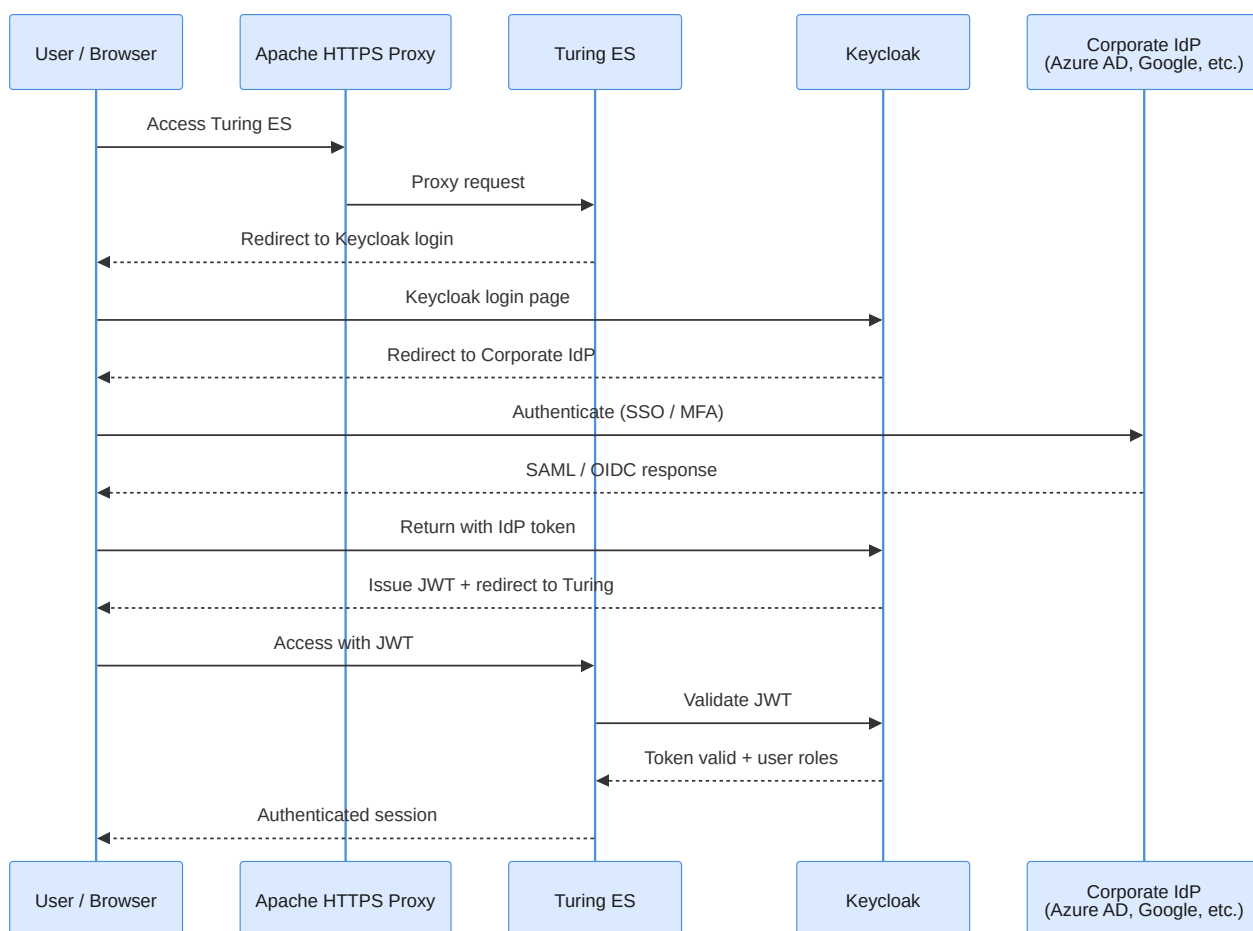
User Federation: Keycloak can synchronize users from **LDAP** or **Active Directory** without requiring SAML. Users are imported (or federated on-demand) from the directory, with password validation delegated to the directory server. This is the simplest path for organizations running on-premise AD without ADFS.

Token-Based API Access: When Keycloak is enabled, REST API clients can authenticate using **OAuth2 access tokens** in addition to API Keys. Obtain a token via Keycloak's token endpoint and pass it as a Bearer token — useful for service-to-service integrations where API Keys are not appropriate.

Audit and Compliance: Keycloak logs all authentication events — successful logins, failed attempts, token refreshes, logouts. These events can be exported to SIEM systems for compliance and security monitoring.

Architecture

When `turing.keycloak=true` is set in the JVM properties, Turing ES delegates all authentication to Keycloak using the Authorization Code flow (OAuth2) and validates access tokens as JWTs (OpenID Connect). Keycloak can be dedicated to Turing ES or shared with other applications — in both cases, Turing ES registers as a **client** within a Keycloak **realm**.



Full Production Setup: Turing + Keycloak + Apache HTTP

The recommended production topology runs Keycloak and Turing ES on the same host, with Apache HTTP Server as the HTTPS reverse proxy in front. All three services are reachable through a single public hostname and port (443), with Apache routing by path prefix.

```
Browser / Client
  |   HTTPS :443
  ▼
Apache HTTP Server (reverse proxy)
  ├── /kc/   → Keycloak   (https://localhost:8443)
  ├── /solr/ → Solr       (http://localhost:8983)
  └── /      → Turing ES  (https://localhost:2700)
```

Step 1 — Database Setup

Both Turing ES and Keycloak require their own databases. Create them before starting either service.

```
-- Turing ES database and user
CREATE USER 'turing'@'%' IDENTIFIED BY '<turing-db-password>';
CREATE DATABASE IF NOT EXISTS turing;
GRANT ALL PRIVILEGES ON turing.* TO 'turing'@'%' WITH GRANT OPTION;

-- Keycloak database and user
CREATE USER 'keycloak'@'%' IDENTIFIED BY '<keycloak-db-password>';
CREATE DATABASE IF NOT EXISTS keycloak;
GRANT ALL PRIVILEGES ON keycloak.* TO 'keycloak'@'%' WITH GRANT OPTION;
```

Tested with MariaDB and MySQL. When using MariaDB, the JDBC connection URL should include `allowPublicKeyRetrieval=true&useSSL=false` if SSL is not configured between the application and the database server.

💡 USE STRONG, UNIQUE PASSWORDS

Replace all placeholder passwords (`<turing-db-password>`, `<keycloak-db-password>`) with strong, randomly generated values. Never reuse passwords between services.

Step 2 — Keycloak Installation and Build

Download and extract the Keycloak distribution. Before starting Keycloak for the first time, run the build step to configure the HTTP relative path. This is required when Keycloak runs behind a reverse proxy at a path prefix (e.g., `/kc/`).

```
./kc.sh build --http-relative-path=/kc
```

Generate a keystore for Keycloak's HTTPS listener:

```
keytool -genkeypair \  
-alias localhost \  
-keyalg RSA \  
-keysize 2048 \  
-validity 365 \  
-keystore server.keystore \  
-dname "CN=<your-hostname>, O=<your-org>, C=<country-code>" \  
-keypass <keystore-password> \  
-storepass <keystore-password>
```

Start Keycloak in production mode:

```
./kc.sh start \  
--optimized \  
--hostname-strict=false \  
--https-key-store-password=<keystore-password>
```

Keycloak will be available at `https://localhost:8443/kc/`.

Running Keycloak as a Linux service

Create a systemd unit file at `/etc/systemd/system/keycloak.service`:

```
[Unit]
Description=Keycloak
After=syslog.target network.target

[Service]
User=viglet
EnvironmentFile=/appl/viglet/keycloak/env
ExecStart=/appl/viglet/keycloak/bin/kc.sh start --optimized --hostname-strict=false --https-key-store-password=<keystore-password>

[Install]
WantedBy=multi-user.target
```

Enable and start the service:

```
systemctl daemon-reload
systemctl enable keycloak
systemctl start keycloak
```


Step 3 — Keycloak Realm and Client Configuration

After starting Keycloak, open the admin console and perform the following setup:

Create a Realm

A realm is an isolated namespace for users, clients, and roles. Create a dedicated realm for Turing ES (e.g., `turing` or a name matching your organization). All subsequent configuration is done within this realm.

Create a Client

Within the realm, create a new client for Turing ES:

FIELD	VALUE
Client ID	<code>turing-app</code> (or your preferred identifier)
Client Protocol	<code>openid-connect</code>
Access Type	<code>confidential</code>
Standard Flow	Enabled
Direct Access Grants	Disabled (not needed)
Root URL	<code>https://<your-hostname></code>
Valid Redirect URIs	<code>https://<your-hostname>/login/oauth2/code/turing-app</code>
Web Origins	<code>https://<your-hostname></code>

After saving, go to the **Credentials** tab and copy the generated **Client Secret**.

Configure Scopes

Ensure the client requests the following scopes: `openid`, `profile`, `roles`. These are needed for Turing ES to receive the user's identity and role information from the JWT.

Step 4 — SSL Certificate for Turing ES

Turing ES can terminate HTTPS directly via its embedded Tomcat server. Generate a self-signed certificate (replace with a CA-signed certificate for production):

```
# Generate self-signed certificate and key
openssl req -x509 -nodes -newkey rsa:2048 \
  -keyout <hostname>.key \
  -out <hostname>.crt \
  -days 365

# Package as PKCS12 for Java
openssl pkcs12 -export \
  -out turing.p12 \
  -inkey <hostname>.key \
  -in <hostname>.crt \
  -name turing

# Alternatively, generate directly with keytool
keytool -genkey \
  -alias turing \
  -storetype PKCS12 \
  -keyalg RSA \
  -keysize 2048 \
  -keystore turing.p12 \
  -validity 3650
```

Place `turing.p12` in a secure path on the server (e.g., `/appl/viglet/turing/server/cert/`).

Import the Apache certificate into the Java truststore

Turing ES calls Keycloak over HTTPS to validate tokens. If Keycloak's certificate is self-signed, Turing ES will fail the SSL handshake unless you import the Apache (or Keycloak) certificate into the Java truststore:

```
keytool -import \
  -trustcacerts \
  -alias turing-apache \
  -file /path/to/<hostname>.crt \
  -keystore $JAVA_HOME/lib/security/cacerts
```

The default truststore password is `changeit`.

Step 5 — Turing ES JVM Configuration

All Turing ES configuration is passed via JVM system properties at startup. Set `JAVA_OPTS` before launching the JAR:

```
JAVA_OPTS="\
  -Xmx1g -Xms1g \
  \
  -Dspring.datasource.url=jdbc:mariadb://localhost:3306/turing?
allowPublicKeyRetrieval=true&useSSL=false \
  -Dspring.datasource.username=turing \
  -Dspring.datasource.password=<turing-db-password> \
  -Dspring.datasource.driver-class-name=org.mariadb.jdbc.Driver \
  -
Dspring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MariaDB103Diale
ct \
  \
  -Dturing.keycloak=true \
  -Dspring.security.oauth2.client.registration.keycloak.client-id=turing-app \
  -Dspring.security.oauth2.client.registration.keycloak.client-secret=<client-
secret> \
  -Dspring.security.oauth2.client.registration.keycloak.scope=openid \
  -Dspring.security.oauth2.client.registration.keycloak.authorization-grant-
type=authorization_code \
  -Dspring.security.oauth2.client.registration.keycloak.redirect-
uri=https://<your-hostname>/login/oauth2/code/turing-app \
  -Dspring.security.oauth2.client.provider.keycloak.issuer-uri=https://<your-
hostname>/kc/realms/<your-realm> \
  -Dspring.security.oauth2.resourceserver.jwt.issuer-uri=https://<your-
hostname>/kc/realms/<your-realm> \
  \
  -Dserver.ssl.enabled=true \
  -Dserver.ssl.key-store=/appl/viglet/turing/server/cert/turing.p12 \
  -Dserver.ssl.key-store-password=<keystore-password> \
  -Dserver.ssl.keyStoreType=PKCS12 \
  -Dserver.ssl.keyAlias=turing \
  \
  -Dturing.url=https://<your-hostname>"

java $JAVA_OPTS -jar viglet-turing.jar
```

JVM property reference

PROPERTY	DESCRIPTION
<code>turing.keycloak</code>	Set to <code>true</code> to enable Keycloak authentication. Default: <code>false</code> (native session + API Key)
<code>spring.security.oauth2.client.registration.keycloak.client-id</code>	Client ID registered in the Keycloak realm
<code>spring.security.oauth2.client.registration.keycloak.client-secret</code>	Client secret from the Keycloak Credentials tab
<code>spring.security.oauth2.client.registration.keycloak.scope</code>	OAuth2 scopes to request. Minimum: <code>openid</code>
<code>spring.security.oauth2.client.registration.keycloak.authorization-grant-type</code>	Must be <code>authorization_code</code>
<code>spring.security.oauth2.client.registration.keycloak.redirect-uri</code>	Must match the redirect URI registered in Keycloak. Pattern: <code>https://<your-hostname>/login/oauth2/code/<client-id></code>
<code>spring.security.oauth2.client.provider.keycloak.issuer-uri</code>	Keycloak realm issuer URI: <code>https://<hostname>/kc/realms/<realm-name></code>
<code>spring.security.oauth2.resourceserver.jwt.issuer-uri</code>	Same as issuer-uri; used for JWT validation
<code>server.ssl.enabled</code>	Enables HTTPS on the embedded Tomcat server
<code>server.ssl.key-store</code>	Path to the PKCS12 keystore file
<code>server.ssl.key-store-password</code>	Password for the keystore
<code>server.ssl.keyStoreType</code>	Must be <code>PKCS12</code>
<code>server.ssl.keyAlias</code>	Alias of the certificate entry inside the keystore

PROPERTY	DESCRIPTION
<code>turing.url</code>	Public base URL of Turing ES (used for OAuth2 redirect URI construction and post-logout redirect)

Step 6 — Apache HTTP Server as Reverse Proxy

Build Apache HTTP Server with the required modules, or install a distribution package that includes `mod_ssl`, `mod_proxy`, and `mod_proxy_http`.

Compile from source (if needed):

```
apt install autoconf libtool-bin make libpcre* libssl-dev libnghttp2-dev
libexpat1-dev

./buildconf && ./configure \
  --prefix=/appl/apache/httpd \
  --enable-mods-shared=all \
  --enable-so \
  --enable-ssl=shared \
  --enable-http2=shared \
  --enable-proxy=shared \
  --enable-proxy-http=shared \
  --enable-proxy-ajp=shared \
  --enable-proxy-balancer=shared \
  --with-mpm=worker && \
make && make install
```

VirtualHost configuration for HTTPS reverse proxy:

```
<VirtualHost _default_:443>
  DocumentRoot "/appl/apache/httpd/htdocs"
  ServerName <your-hostname>:443

  # SSL
  SSLEngine on
  SSLCertificateFile    "/path/to/<hostname>.cert"
  SSLCertificateKeyFile "/path/to/<hostname>.key"

  # Proxy settings
  HostnameLookups Off
  UseCanonicalName Off
  ProxyPreserveHost On

  # Required headers for applications behind a proxy
  RequestHeader set X-Forwarded-Proto "https"
  RequestHeader set X-Forwarded-Port "443"

  # Allow SSL proxying to backend services with self-signed certs
  SSLProxyEngine on
  SSLProxyCheckPeerName off

  # Route /solr/ to Solr (plain HTTP, internal only)
  ProxyPass          "/solr/" "http://localhost:8983/solr/"
  ProxyPassReverse   "/solr/" "http://localhost:8983/solr/"

  # Route /kc/ to Keycloak (HTTPS, internal)
  ProxyPass          "/kc/" "https://localhost:8443/kc/"
  ProxyPassReverse   "/kc/" "https://localhost:8443/kc/"

  # Route everything else to Turing ES (HTTPS, internal)
  ProxyPass          "/" "https://localhost:2700/"
  ProxyPassReverse   "/" "https://localhost:2700/"
</VirtualHost>
```

Key configuration notes:

`ProxyPreserveHost On` — passes the original `Host` header to the backend. Required for Keycloak and Turing ES to generate correct redirect URIs.

`RequestHeader set X-Forwarded-Proto "https"` — tells backend services that the original request was HTTPS, even though the internal connection may be unencrypted (in configurations where backends do not use SSL).

`SSLProxyCheckPeerName off` — disables hostname verification for the backend SSL connections. Required when using self-signed certificates on Keycloak and Turing ES. In production with CA-signed certificates, this should be set to `on`.

`/kc/` routes to Keycloak HTTPS because Keycloak was started with HTTPS. The path prefix matches the `--http-relative-path=/kc` configured in Step 2.

Logout flow

When Keycloak is enabled, logging out of Turing ES triggers a full OIDC logout. After the local session is cleared, the user is redirected to Keycloak's logout endpoint:

```
https://<your-hostname>/kc/realms/<your-realm>/protocol/openid-connect/logout
?client_id=turing-app
&post_logout_redirect_uri=https://<your-hostname>
```

Keycloak invalidates the SSO session and redirects the user back to the Turing ES root URL.

Using an Existing Keycloak Instance

If your organization already runs a Keycloak instance, you do not need to install a dedicated one. Simply:

- Create a new realm (or use an existing realm, if your organization's policies allow it)

- Register Turing ES as a new client within that realm following the configuration in Step 3

- Configure the JVM properties in Step 5 to point to the existing Keycloak's issuer URI

The `issuer-uri` format is always `https://<keycloak-hostname>/kc/realms/<realm-name>` — adjust the path prefix if your existing Keycloak uses a different one (or no prefix).

Security Checklist

Before going to production, verify the following:

- ☐ Keycloak is only accessible via the Apache reverse proxy (port 8443 is not exposed externally)
 - ☐ Solr is only accessible via the reverse proxy or not exposed externally at all
 - ☐ Turing ES port (2700) is not exposed externally — all traffic goes through Apache
 - ☐ All self-signed certificates are replaced with CA-signed certificates (or certificates from an internal PKI)
 - ☐ `SSLProxyCheckPeerName` is set to `on` if using CA-signed backend certificates
 - ☐ Keycloak admin credentials have been changed from defaults
 - ☐ Database users have minimum required permissions (no superuser)
 - ☐ JVM properties containing secrets (`client-secret` , `key-store-password` , `datasource.password`) are not logged or exposed in process listings — consider using an environment file or a secrets manager
 - ☐ HTTP (port 80) redirects to HTTPS (port 443)
 - ☐ Keycloak's `hostname-strict` mode is evaluated for your network topology
-

Related Pages

PAGE	DESCRIPTION
Authentication	Native session + API Key authentication (default mode)
REST API Reference	API endpoints and authentication examples
Configuration Reference	JVM properties and <code>application.yaml</code> settings
Installation Guide	Initial Turing ES installation and database setup
